The Nuprl Proof Development System, Version 5

Reference Manual and User's Guide

Christoph Kreitz

Department of Computer Science, Cornell-University
Ithaca, NY 14853-7501, U.S.A.

kreitz@cs.cornell.edu

Preface

This manual is a reference manual for version 5 of the NUPRL proof development system. As the NUPRL system is constantly under development, this manual will always be incomplete. In particular, it is missing information about recent advanced features of the system and about certain extensions of NUPRL's type theory that are currently being added to the system. More recent information and the system itself can be found at the NUPRL web pages `http://www.nuprl.org`.

From its beginnings in the 1980s, the PRL project has been guided by Robert Constable. Over the years, many researchers and students at Cornell have contributed to the theoretical foundations, the design, and the implementation of the NUPRL system.

The major developers of NUPRL 5 are Stuart Allen, Rich Eaton and Lori Lorigo. They have provided explanations for many of the new system features and added new ones that are helpful for novice users while this manual was written. Mark Bickford has used and tested the system extensively and designed many extensions that are currently being added to the system.

Although the architecture of the NUPRL system has significantly changed in release 5, the basic structure of the proof and term editors, of abstractions and display forms, of the metalanguage ML, and of rules and tactics is largely compatible with previous releases. The NUPRL 4 manuals [Jac93a, Jac93b], written by Paul Jackson, served as a foundation for the corresponding chapters.

Ithaca, December 17, 2002

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The NUPRL proof development system is a framework for the development of formalized mathematical knowledge as well as for the synthesis, verification, and optimization of software. It is based on a significant extension of Martin-Löf's intuitionistic Type Theory [ML84], which includes formalizations of the fundamental concepts of mathematics, data types, and programming. The system itself supports interactive and tactic-based reasoning, decision procedures, evaluation of programs, language extensions through user-defined concepts, and an extendable library of verified knowledge from various domains.

Since its first release in 1984 [CAB+86], the system has been undergone several significant modifications to meet the growing demands for formal knowledge and tools in programming and mathematics, the most recent being a complete redesign of its architecture, discussed in [ACE+00]. The NUPRL 5 system, which is documented in this manual, features an open, distributed architecture that integrates all its key subsystems as independent components and uses a flexible knowledge base as its central component.

1.1 The NUPRL 5 Architecture

Figure 1.1 illustrates the architecture of NUPRL 5. The system is organized as a collection of communicating processes that are centered around a common knowledge base, called the *library*. The

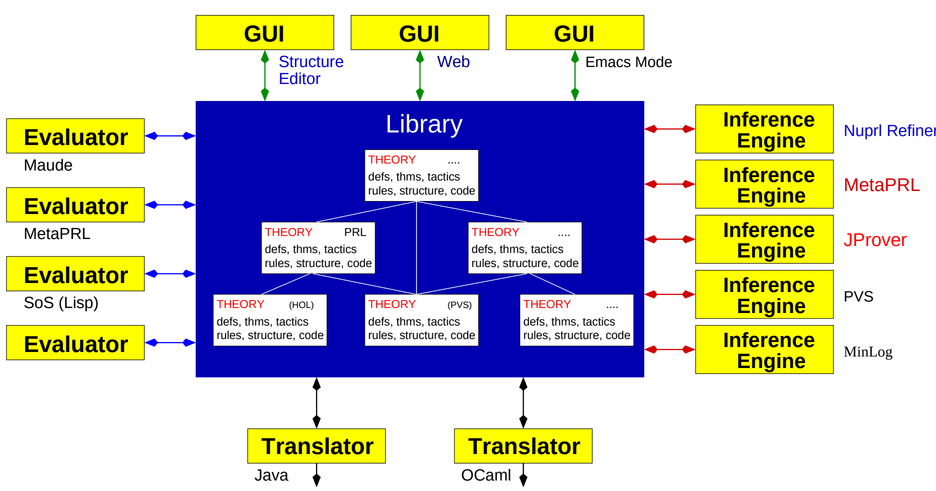


Figure 1.1: NUPRL 5 distributed open architecture

library contains definitions, theorems, inference rules, meta-level code (e.g. tactics), and structure objects that can be used to provide a modular structure for the library's contents. Inference engines (*refiners*), user interfaces (*editors*), rewrite engines (*evaluators*), and *translators* are started as independent processes that can connect to the library at any time.

The library can communicate with arbitrarily many other processes. This allows the user to connect several refiners and evaluators simultaneously, e.g. the Nuprl and MetaPRL [Met] refiners, proof systems like HOL [GM93] or PVS [ORR+96], first-order provers like JProver [SLKN01], Otter [WWM+90], EQP [McC97], or Setheo [LSBB92], proof-based program generators like MinLog [BBS+98], rewrite engines like Maude [CDE+99a], computer algebra systems like Mathematica [Wol88] or Maple [Map], decision procedures [NO79, Sho84, SVC], and model checkers [McM93, Dil96, Hol97], and even to make them cooperate. It is also possible to run different refiners in parallel on the same proof goal or several instances of the same refiner on different proof goals.

Providing several editors enables several users to work in parallel on the same formal theory while using their favorite interface. At the same time external users can access the system through the Web without having to start the whole system themselves.

Translators between the formal knowledge stored in the library and, for instance, programming languages like Java or Ocaml [Kre97, KHH98, Kre03] allow the formal reasoning tools to supplement real-world software from various domains.

Nuprl 5 is highly configurable. In its current standard configuration, which is described in this manual, the system essentially provides an extended functionality of the Nuprl 4 system [Jac94]. It consists of the library, the Nuprl 5 editor, and the Nuprl 5 refiner.

The Knowledge Base

The knowledge base is based on a transaction model for entering and modifying objects. All changes to objects, e.g. the effects of editor commands or inference steps, are immediately committed to the persistent library. The knowledge base also provides the option to undo changes, redo transactions, or to have several processes view or work on the same object – essentially following the same protocols as databases. However, changes do not overwrite an object but instead create a new version. The previous version is preserved until it is explicitly destroyed in a garbage collection process. A *version control mechanism* allows the user to recover previous versions of an object.

To account for the validity of library objects, the knowledge base supports *dependency tracking*, which will enable a user to check if theorems are valid wrt. a specific set of rules, axioms, and proof procedures.

In principle, the library does not impose any predefined structure. All visible structure, e.g. the directory structure as observed by the Nuprl 5 navigator (See Chapter 4), is generated by structure objects that are explicitly present in the library and can be customized by the user.

To prevent name clashes, the library distinguishes between objects and the names that users choose to denote them. The latter are just display versions of internal names. The can be changed without affecting the object itself.

User Interfaces

The main user interface of Nuprl 5 is the *navigator*. It communicates with the knowledge base by sending and receiving abstract terms. While displaying and editing these terms it presents them as directories, theorems, definitions, proofs, or mathematical expressions – depending on structural information found in the library. For the user, it provides the functionality of a *structure editor*: the user can mark subterms and edit slots in the displayed term and then cause the navigator to send the result back to the library, which processes the result while the user may continue to work with the editor. Again, structural information in the library determines whether the abstract terms received by the knowledge base are interpreted as a commands to store or retrieve data, as tactics calling a refiner, or as utility functions.

In addition to the navigator, NUPRL 5 provides emulations of the editors used in the NUPRL 4 system, as well as valuable extensions for facilitating proof browsing, merging, replaying and accounting. There is also a web front end [Nau98] that allows external users to browse the NUPRL library remotely.

Inference Engines

The NUPRL 5 inference engine refines proof goals by executing ML code that may include references to library objects, particularly to the inference rules and tactics stored in the knowledge base. It applies the code to a given proof goal that it receives as an abstract term and returns the resulting list of subgoals back to the library. Based on the validations given in the rule objects it can also extract programs from proofs and evaluate them. The inference mechanism is fairly straightforward and compatible with the one in NUPRL 4.

As an alternative one may invoke the MetaPRL refiner [Met], a modularized version of NUPRL's inference engine implemented in OCaml, which is significantly faster due to improvements in rewriting and evaluation. The communication between NUPRL 5 and MetaPRL utilizes the MathBus design [Mat].

We are in the process of connecting a variety of external refiners such as a constructive first-order theorem prover [KOSP00], the HOL system (via Maude [CDE$^+$99b]), Mathematica, and Isabelle [Nau99]. We will also emulate the refiner of NUPRL 3 in order to be able to restore older theories that did not survive the transition from NUPRL 3 to NUPRL 4.

1.2 Purpose of this Manual

This manual is a reference manual for version 5 of the NUPRL system. It is aimed at beginning and intermediate users of the system. NUPRL 5 is written mostly in Common Lisp, but uses some extensions that require Lucid or Allegro Lisp. It runs on Unix-based workstations that use the X window system.

Note, that this manual is still under development and incomplete. Additional online documentation can be found on the Web at the URL `http://www.nuprl.org/html/nuprldocs.html`, in the directory `/home/nuprl/nuprl5/doc/` of the installed system, and in objects such as `doc: ref editor` and `doc: navigator use`, while the system is running.

The original NUPRL book *Implementing Mathematics with the Nuprl Proof Development System* [CAB$^+$86], also available on the Web at the URL `http://www.nuprl.org/book/doc.html` is still a good background reference. However, one has to keep in mind that the system itself has been changed and extended substantially since the book was published. None of the tutorials given in the book will work in NUPRL 5. The "reference" portion of the book is superseded by this reference manual, but 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.

1.3 Tips for Beginning NUPRL 5 Users

We recommend that you run through the brief tutorial in chapter 2 before trying to do anything else with the system.

In learning to use the navigator as well as the proof and term editors, check out all the mouse commands and the buttons that are provided. Many editing operations can be done most easily with the mouse and the buttons. Familiarize yourself with the standard NUPRL theories that are already present in the library. Existing theories are an excellent resource for learning about how to structure theories and how to write proofs.

The NUPRL ML manual in Appendix B contains a tutorial in the use of ML (Appendix B.2). Use this as an introduction to ML. Daring users may also take a look at the `.ml`-files in the directory `/home/nuprl/nuprl5/lib/ml/standard`, which contain the implementation of the existing tactics collection and can be used as examples for writing tactics.

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

1.4 Conventions

We give the conventions we use in this manual for presenting user input and NUPRL output. Input which you should type is presented typewriter font. For example `this is in typewriter font`. The following symbols are also used:

- SPC for the space-bar.
- ↵ for the Return key (sometimes marked as Enter).
- LFD for the linefeed key.
- ⇆ for the Tab key.
- DEL for the delete key (sometimes marked as Rubout). On some keyboards the BACKSPACE has the same effect.
- LEFT, MIDDLE, and RIGHT for the left, middle, and right mouse button.

Modified keys are presented as follows:

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

Note that x can be either a keyboard key *or* a mouse button; for example both ⟨C-`a`⟩ and ⟨M- RIGHT ⟩ are valid modified keys. On some keyboard's (for example, those of Sparc-stations) the usual meta keys are the keys marked ◇ either side of the space-bar, while on PC keyboards this key is often marked as Alt. The ⟨S-x⟩ modifier is only used with non-printing characters (for example, ↵).

When we say "click LEFT" on some part of a window, we mean that the mouse cursor should be pointed at that part, and then the LEFT button should be pressed.

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

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

1.5 Structure of this Manual

Chapter 2 gives a tutorial-like overview of the essential features of the Nuprl 5 system. Novice users should run through this tutorial before trying to do anything else.

Chapter 3 describes how to install, start, and exit the system. It also give a few hints on customization and basic troubleshooting.

Nuprl's windows are at the "top-level" in the X environment and come with their own context-specific editors. The two main windows – the *navigator* and the ML *top loop* – are described in Chapter 4 while Chapters 5 and 6 describe the editors for *term* and for *proof windows*.

The following chapters describe Nuprl's support for the formalization of mathematical concepts and proofs. In chapter 7 we explain how to extend the logical language of Nuprl by abstract definitions and how to modify the visual presentation of formal material through display forms. Chapter 8 describes the structure of basic inferences in Nuprl as well as the most important *tactics* for automated reasoning.

The appendices describe important background information such as the type theory of Nuprl (Appendix A) and Nuprl's meta-language ML (Appendix B).

Chapter 2

A Quick Overview

The NUPRL 5 system is organized as a collection of communicating processes that are centered around a *library*, which contains definitions, theorems, inference rules, tactics, structure objects, comments, etc. Inference engines (*refiners*), user interfaces (*editors*), rewrite engines (*evaluators*), and *translators* are started as independent processes that can connect to the library at any time.

2.1 Preparation

We assume that the NUPRL 5 system has already been installed (see Section 3.2.1) and can be found in the directory `/home/nuprl/nuprl5`. We also assume that the NUPRL 5 binaries can be found in `/home/nuprl/bin/` and that the NUPRL fonts are installed in the directory `/home/nuprl/fonts/bdf`.

Make sure that your Unix path includes `/home/nuprl/bin/` and that the X-server has NUPRL's fonts loaded. Create a file `.nuprl.config` in your home directory with the following entries:

```
(libhost "HOSTNAME")
(dbpath "/home/nuprl/nuprl5/NuprlDB")
(libenv "standard")
```

The HOSTNAME for the `libhost` configuration should be the name of the host running the library process. The values for `dbpath` and the `libenv` describe the physical and logical location of the standard library. Optional settings like specific colors and fonts for the NUPRL windows may also be given in that file.

Copy the file `/home/nuprl/nuprl5/mykeys.macro` to your home directory. NUPRL reads the file `~/mykeys.macro` to determine the key bindings that will be used in various windows. You need it to initialize the key combinations described in this manual and to customize them according to your own preferences later.

2.2 Running NUPRL 5

For the basic NUPRL 5 configuration you need to run three processes: a library, an editor, and a refiner. The library (`nulib`) should be started first. The editor (`nuedd`) and the refiner (`nuref`) can then be started in any order. Generally it is a good idea to run these processes in separate emacs frames: there will be interactive top loops, so editing capabilities are sometimes useful.

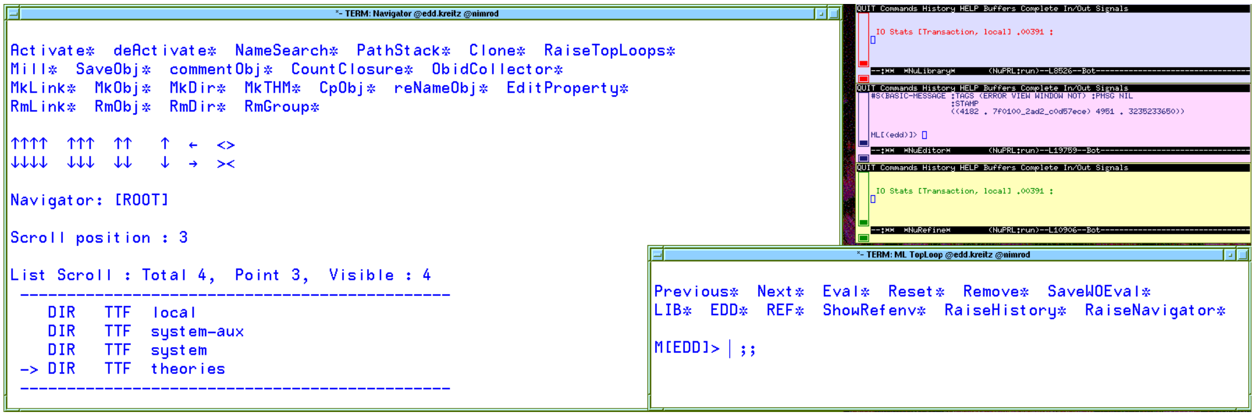


Figure 2.1: Initial NUPRL 5 screen

Once the processes have started, entering `(top)` at the Lisp prompt will start the ML system.

```
USER(1): (top)
```

Enter `go.` at the prompt to initialize the corresponding NUPRL process.

```
ML[(ORB)]> go.
```

It is important to initialize the library before the editor and the refiner. The editor process will take few minutes and then pop up two windows: a *navigator* and a *top loop*.

Figure 2.1 shows a typical initial NUPRL 5 screen. The window on the left is the NUPRL 5 navigator. The three emacs windows on the upper right run interactive top loops for the library, editor and the refiner. The NUPRL 5 top loop is shown in the window below. In contrast to the corresponding emacs top loops, the NUPRL 5 top loop incorporates the NUPRL 5 term editor. It is better suited for editing object-level terms but do not support the full editing capabilities of emacs. Unless there is a need to interact with the top ML level, one usually iconifies these four windows to create some space for the windows that will pop up while working with the system.

2.3 Using the Navigator

The navigator is the main user interface of NUPRL 5. It can be used to browse the library and to create, delete, or edit objects by initiating the appropriate editors. Figure 2.2 shows navigator window shown in its initial state. As in most NUPRL 5 windows, the upper part of the navigator window contains several *buttons*, which are indicated by a * at the end of a word. Clicking a button with the left mouse will trigger some action or pop up a template to be filled in. The lower part of the navigator window shows the *current directory* (here `ROOT`) and a listing of the *type*, *status*, and *name* of some of the objects in the directory. There is also a distinguished object, the *nav point*), which is marked by an arrow (the *navigation pointer*). When the *edit point* is in the `Scroll position` field (use the left mouse), the arrow keys on the keyboard can be used to move the through the directory tree.

$\uparrow$	UP	move navigation pointer one step up
$\downarrow$	DOWN	move navigation pointer one step down
$\leftarrow$	LEFT	move navigation pointer to next higher directory
$\rightarrow$	RIGHT	open object at navigation pointer in a new window (enter sub-directory if object is of type `DIR`)

```
- TERM: Navigator
Activate*  deActivate*  NameSearch*   PathStack*  Clone*  RaiseTopLoops
Mill*  SaveObj*  commentObj*  CountClosure*  ObidCollector*
MkLink*  MkObj*  MkDir*  mkTHM*  CpObj*  reNameObj*  EditProperty*
RmLink*  RmObj*  RmDir*  RmGroup*

↑↑↑↑  ↑↑↑  ↑↑   ↑  ←  <>
↓↓↓↓  ↓↓↓  ↓↓   ↓  →  ><

Navigator: [ROOT]

Scroll position : 0

List Scroll : Total 4,  Point 0,  Visible : 4
 ---------------------------------------------
 -> DIR   TTF  theories
    DIR   TTF  system-aux
    DIR   TTF  local
    DIR   TTF  system
 ---------------------------------------------
```

Figure 2.2: NUPRL 5 Navigator

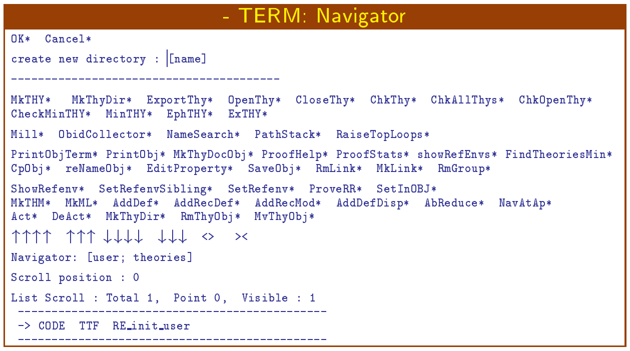
The navigator window also contains arrow buttons for faster navigation through a directory. ↑↑ and ↓↓ scroll half a screen, ↑↑↑ and ↓↓↓ scroll a full screen, and ↑↑↑↑ and ↓↓↓↓ move to the top and bottom of the directory. In addition to buttons and arrow keys, there are also a variety of special key combinations that can be used to manipulate objects in the library. These are described in Chapter 4.

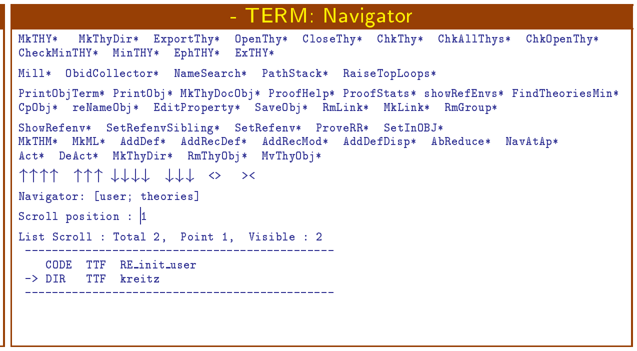
To begin working with the NUPRL system, one will usually move into the `theories` directory. Leaving the initial state will cause additional buttons to become visible.

```
- TERM: Navigator
MkTHY*   MkThyDir*  ExportThy*  OpenThy*  CloseThy*  ChkThy*  ChkAllThys*  ChkOpenThy*
CheckMinTHY*  MinTHY*  EphTHY*  ExTHY*

Mill*  ObidCollector*  NameSearch*  PathStack*  RaiseTopLoops*

PrintObjTerm* PrintObj* MkThyDocObj* ProofHelp* ProofStats* showRefEnvs* FindTheoriesMin*
CpObj*  reNameObj*  EditProperty*  SaveObj*  RmLink*  MkLink*  RmGroup*

ShowRefenv*  SetRefenvSibling*  SetRefenv*  ProveRR*  SetInOBJ*
Activate*  DeActivate*   MkObj*

↑↑↑↑  ↑↑↑ ↓↓↓↓  ↓↓↓  <>   ><

Navigator: [theories]

Scroll position : 0

List Scroll : Total 13,  Point 0,  Visible : 10
 ---------------------------------------------
 -> DIR   TTF  General
    DIR   TTF  pre-utils
    DIR   TTF  initial reference environment
    DIR   TTF  standard
    TERM  TTF  check theories_control
    DIR   TTF  Obvious
    DIR   TTF  user
    DIR   TTF  detritus
    DIR   TTF  utils
    DIR   TTF  .help
 ---------------------------------------------
```

Usually, NUPRL users will work within their own sub-directory within the directory `user` and occasionally browse the `standard` sub-directory, which contains the NUPRL type theory and a few standard libraries of formalized mathematical knowledge.

A new user-directory can be created by clicking the MkThyDir* button. This will open a template for entering the name of the new directory and move the edit point to the [name] slot.

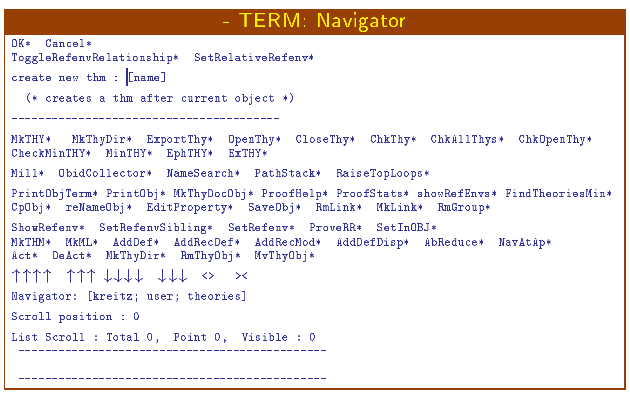


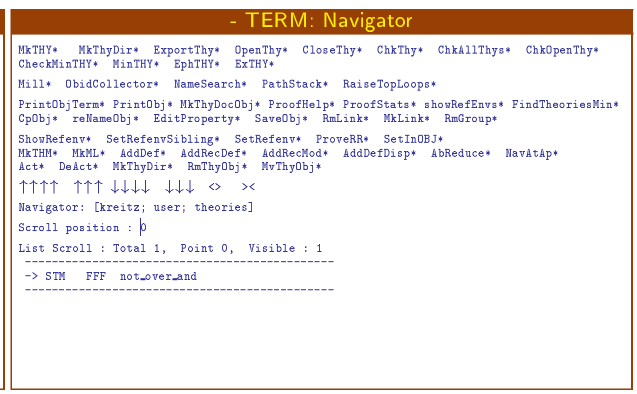


Enter the name of the directory and click OK* (or press ↵ twice). This will create the new directory object, place it immediately below the previous nav point, and move the navigation pointer to it, as shown in the right window.

2.4 Creating Theorem Objects

Before one can prove a theorem in NUPRL one has to create an object that contains it. Clicking the MkTHM* button (after moving into the user directory) will open a template for entering the name and kind of a new library object.[1] The edit point will be in the [name] slot.





As example theorem we take ⌞∀A,B:ℙ. (¬A) ∨ (¬B) ⇒ ¬(A ∧ B)⌟, one of the DeMorgan laws for propositional logic. Enter the name of the theorem not_over_and and click OK*, or press ↵ twice. This will create a new statement object named not_over_and.

2.5 Proving Theorems

To state and prove a theorem, one has to open the corresponding object. Pressing the right arrow key when the nav point is a statement object pops up a new window that shows the contents of this object. If the theorem has not been stated yet, there will be a [goal] slot in the upper part of the window and the *rule slot* next to the keyword BY below will be empty. The # in the upper left corner means that the theorem is not complete yet, while the top next to it indicates that the top node of the proof tree is being displayed.

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

```
- TERM: Navigator
MkTHY*   MkThyDir*  ExportThy*  OpenThy*  CloseThy*  ChkThy*  ChkAllThys*  ChkOpenThy*
CheckMinTHY*  MinTHY*  EphTHY*  ExTHY*

Mill*  ObidCollector*  NameSearch*  PathStack*  RaiseTopLoops*

PrintObjTerm* PrintObj* MkThyDocObj* ProofHelp* ProofStats* showRefEnvs* FindTheoriesMin*
CpObj*  reNameObj*  EditProperty*  SaveObj*  RmLink*  MkLink*  RmGroup*

ShowRefenv*  SetRefenvSibling*  SetRefenv*  ProveRR*  SetInOBJ*
MkTHM*  MkML*  AddDef*  AddRecDef*  AddRecMod*  AddDefDisp*  AbReduce*  NavAtAp*
Act*  DeAct*  MkThyDir*  RmThyObj*  MvThyObj*

↑↑↑↑  ↑↑↑ ↓↓↓↓  ↓↓↓  <>   ><

Navigator: [kreitz; user; theories]

Scroll position : 0

List Scroll : Total 1,  Point 0,  Visible : 1
 --------------------------------------------
 -> STM   FFF  not_over_and
 --------------------------------------------
```

```
- PRF: not_over_and
# top
[goal]
BY
```

Move the mouse cursor into the new window and click left in the `[goal]` slot. This initializes the Nuprl 5 *term editor* in this slot. The goal is now entered in a structural top-down fashion. Entering `all ↵` creates the template for the universal quantifier.

```
- PRF: not_over_and
# top
∀[var]:[type]. [prop]
BY
```

The `[var]` slot of the `all` template is a *text slot* and will not be interpreted. Entering `A ↵` inserts the character '`A`' in the variable slot and moves the edit point to the next slot.

The `[type]` and `[prop]` slots are *term slots*, which means that input will be interpreted and may open new templates. Entering `prop ↵ i ↵ all ↵` inserts the *propositional universe* term (whose name `prop` has nothing to do with the `prop` in the `[prop]` place-holder) into the `[type]` slot, and another template for the universal quantifier in the `[prop]` slot.

```
- PRF: not_over_and
# top
∀A:ℙ. ∀[var]:[type]. [prop]
BY
```

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

```
- PRF: not_over_and
# top
∀A,B:ℙ. [prop]
BY
```

Entering the rest of the theorem is straightforward. Typing `implies ↵` creates the implication template. Entering `or ↵ not ↵ A ↵ not ↵ B ↵` generates `(¬A) ∨ (¬B)` and `¬(A ∧ B)` is generated by `not ↵ and ↵ A ↵ B ↵`.

Before a statement can be used in a proof it must be *committed to the permanent library*. This can be done by using the key combination ⟨C-M-g⟩.[2]

```
- PRF: not_over_and
# top
∀A,B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY
```

[2]Actually, pressing ⟨C-M-g⟩ is not necessary when entering a statement for the first time, since it will be committed after the first execution of a proof tactic. However, subsequent modifications of the statement will *not* be committed without pressing ⟨C-M-g⟩. Thus it is better to make using this key combination a habit.

To begin with the proof, press the down arrow key once or use the left mouse to move the edit point into the empty rule slot next to the BY.

```
- PRF: not_over_and
# top
∀A,B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY |
```

The most common proof tactic is the *single step decomposition* tactic D (see chapter 8 for details). It requires as argument the index of the proof hypothesis to which it shall be applied or a zero if it shall be applied to the conclusion. To enter this tactic, type ⌞D 0 ⟨C-↵⟩⌟.

Pressing ⟨C-↵⟩ when in a rule slot refines the goal at current node with the corresponding tactic. In this (synchronous) mode you have to wait for the refinement process to be complete. Pressing ⟨C-M-↵⟩ instead initializes *asynchronous refinement*, which allows you to continue working while the proof goal is being refined. Once a refinement is completed the proof window gets updated and shows the subgoals that were generated by applying the tactic.

```
- PRF: not_over_and
# top
∀A,B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY D 0
# 1
1. A:ℙ
⊢ ∀B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY
# 2
.....wf.....
ℙ ∈ 𝕌'
BY
```

To prove the first subgoal, press the down arrow key. This will move into the first sub-node of the theorem, indicated by a top 1.

```
- PRF: not_over_and
# top 1
1. A:ℙ
⊢ ∀B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY |
```

To move into the second subgoal, press the right arrow key. As indicated by the .....wf..... annotation, the second subgoal is a *well-formedness goal*, stating that ℙ is a well-formed type theoretical expression. Most goals of this kind can be dealt with automatically. Typing ⌞Auto ⟨C-↵⟩⌟ will complete this subproof: no subgoals are generated and the status marker changes into a *.

```
- PRF: not_over_and
* top 2
.....wf.....
ℙ ∈ 𝕌'
BY Auto
```

Pressing the left arrow key will bring you back into the first subgoal. Alternatively you can press ⟨C-M-j⟩, which causes the editor to jump to the next unproven subgoal.

Proving the first subgoal requires more efforts. Typing ⌞Auto ⟨C-↵⟩⌟ will decompose the universal quantifier, the implication, and deal with the corresponding well-formedness subgoals. The result will be a subgoal with two additional hypotheses: a declaration of the variable B and the assumption (¬A) ∨ (¬B).

To prove this goal both the conclusion (`D 0`) and the third hypothesis (`D 3`) need to be decomposed. `Auto` does neither of these two steps automatically but can deal with the resulting subgoals. These steps can be combined into a single one by using the tactical `THEN`.

```
- PRF: not_over_and
# top 1
1. A:ℙ
⊢ ∀B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY Auto
* 1 1
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)
BY
```

Clicking the left mouse next to the `BY` allows you to enter the tactic into an empty rule slot without having to move into the corresponding sub-node. Typing ⌞`D 0 THEN D 3 THEN Auto` ⟨C-↵⟩⌟ results in a complete proof of the subgoal.

```
- PRF: not_over_and
* top 1
1. A:ℙ
⊢ ∀B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY Auto
* 1 1
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)
BY D 0 THEN D 3 THEN Auto
```

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

```
- PRF: not_over_and
* top
∀A,B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY D 0
* 1
1. A:ℙ
⊢ ∀B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY Auto
* 1 1
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)
BY D 0 THEN D 3 THEN Auto
* 2
.....wf.....
ℙ ∈ 𝕌'
BY Auto
```

The proof has already been saved in the library. To close the proof window press ⟨C-`Q`⟩. This key combination will always close the current window.

If a theorem shall be used as lemma in other proofs, it has to be *activated*. Many tactics use a list of active theorems which are searched through when attempting to prove a theorem automatically. For this purpose you have to click the `Act*` button, which changes the object status of `not_over_and` from `FFF` to `TFF`.

```
- TERM: Navigator
MkTHY*   MkThyDir*  ExportThy*  OpenThy*  CloseThy*  ChkThy*  ChkAllThys*  ChkOpenThy*
CheckMinTHY*  MinTHY*  EphTHY*  ExTHY*

Mill*  ObidCollector*  NameSearch*  PathStack*  RaiseTopLoops*

PrintObjTerm* PrintObj* MkThyDocObj* ProofHelp* ProofStats* showRefEnvs* FindTheoriesMin*
CpObj*  reNameObj*  EditProperty*  SaveObj*  RmLink*  MkLink*  RmGroup*

ShowRefenv*  SetRefenvSibling*  SetRefenv*  ProveRR*  SetInOBJ*
MkTHM*  MkML*  AddDef*  AddRecDef*  AddRecMod*  AddDefDisp*  AbReduce*  NavAtAp*
Act*  DeAct*  MkThyDir*  RmThyObj*  MvThyObj*

↑↑↑↑  ↑↑↑ ↓↓↓↓  ↓↓↓  <>   ><

Navigator: [kreitz; user; theories]

Scroll position : 0

List Scroll : Total 1,  Point 0,  Visible : 1
 ---------------------------------------------
 -> STM   TFF  not_over_and
 ---------------------------------------------
```

A theorem can also be activated by closing the proof window with ⟨C-Z⟩ instead of ⟨C-Q⟩. This will cause NUPRL to create the *extract term* of the proof (a λ-term describing its computational content) and to store it along with the theorem object. The status of that theorem will then be `TTF`.

2.6 Adding Definitions

Besides proving theorems, the most common activity in mathematics is introducing new concepts, which are defined in terms of already existing ones. This makes the formulation of theorems crisper and easier to comprehend. NUPRL supports such an enhancement of the formal language through a definition mechanism. This mechanism allows a user to introduce new terms that are definitionally equal to other terms.

As an example consider the `∃!` quantifier, which states the existence of a unique element `x∈T` that satisfies a property `P`. A typical definition for this quantifier is the following:

`∃!x:T. P[x]   ≡   ∃x:T. P[x] ∧ (∀y:T. P[y] ⇒ y=x∈T).`

This definition actually presents two aspects of a newly defined term. It first states that a new abstract term, say **`exists_uni`** is to be introduced, which has two subterms (`T` and `P`) and binds occurrences of `x` in `P`. Secondly, it states that the term is to be presented as `∃!x:T. P`.

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

To initialize this mechanism, click the `AddDef*` button with the left mouse. This will open a template for defining the abstract term.

```
- TERM: Navigator
OK*  Cancel*

add def : [lhs] ==
          [rhs]
----------------------------------------

MkTHY*   MkThyDir*  ExportThy*  OpenThy*  CloseThy*  ChkThy*  ChkAllThys*  ChkOpenThy*
CheckMinTHY*  MinTHY*  EphTHY*  ExTHY*

Mill*  ObidCollector*  NameSearch*  PathStack*  RaiseTopLoops*

PrintObjTerm* PrintObj* MkThyDocObj* ProofHelp* ProofStats* showRefEnvs* FindTheoriesMin*
CpObj*  reNameObj*  EditProperty*  SaveObj*  RmLink*  MkLink*  RmGroup*

ShowRefenv*  SetRefenvSibling*  SetRefenv*  ProveRR*  SetInOBJ*
MkTHM*  MkML*  AddDef*  AddRecDef*  AddRecMod*  AddDefDisp*  AbReduce*  NavAtAp*
Act*  DeAct*  MkThyDir*  RmThyObj*  MvThyObj*

↑↑↑↑  ↑↑↑ ↓↓↓↓  ↓↓↓  <>   ><

Navigator: [kreitz; user; theories]

Scroll position : 0

List Scroll : Total 1,  Point 0,  Visible : 0
 ---------------------------------------------
 -> STM   TFF  not_over_and
 ---------------------------------------------
```

To enter the new term on the left hand side of the definition, you have to provide its name (or *object identifier*) and a list of subterms. The ∃! has two subterms, T and P, and binds one variable in the second. To create a template for entering the details, type ⌞exists_uni(0;1)⌟.

```
- TERM: Navigator
OK*  Cancel*

add def : exists_uni\(0;1
          [rhs]
```

```
- TERM: Navigator
OK*  Cancel*

add def : exists_uni([term]; [binding].[term]) ==
          [rhs]
```

This tells the system to create a term called exists_uni, whose first term has no bound variables and whose second term has one bound variable The template, shown on the right, appears as soon as you have entered the right parenthesis that closes the subterm list. Pressing ↵ then moves the edit point into the first term slot.

```
- TERM: Navigator
OK*  Cancel*

add def : exists_uni([term]; [binding].[term]) ==
          [rhs]
```

Enter ⌞T ↵ x ↵ so_var1 ↵⌟. This puts T into the first term slot, makes x the binding variable in the second, and states that the second term will be a *second order variable* of arity 1 (see Chapter 7.1).

```
- TERM: Navigator
OK*  Cancel*

add def : exists_uni(T; x.|[variable-id][[term]]) ==
          [rhs]
```

Note that NUPRL's term editor treats any unknown name as variable name, while names that can be linked to (active) object identifiers (and display forms) will cause the corresponding template to appear. Thus T will be inserted as variable name, while so_var1 creates a new template. x had been entered into a binding slot and is thus viewed as variable.

Should you mistype so_var1 and actually enter an identifier that is unknown to NUPRL, say sovar1, the identifier will appear as variable name in the term slot.

```
- TERM: Navigator
OK*  Cancel*

add def : exists_uni(T; x.sovar1) ==
          [rhs]
```

There are two ways to correct that mistake. You may *delete* the term ⌞sovar1⌟ by clicking LEFT over it, pressing ⟨M-P⟩ to mark the full term and then ⟨C-K⟩ to cut it. Afterwards you enter so_var1 ↵ to get the correct template. Note that ⟨C-k⟩ saves the term in a cut buffer and that you can paste this term with ⟨C-Y⟩. To delete a term without saving it, you need to press ⟨C-C⟩.

Alternatively, you may use NUPRL's generic *undo* command ⟨C-_⟩, which will restore the empty term slot. Move the edit cursor into that slot either by pressing ↵ or by clicking LEFT over it and then enter enter so_var1 ↵.

Entering ⌞P ↵ x ↵⌟ next generates P[x] and moves the edit point into the right hand side of the definition.

```
- TERM: Navigator
OK*  Cancel*

add def : exists_uni(T; x.P[x]) ==
          [rhs]
```

To enter the right hand side of the definition, you have to proceed in a structural top-down fashion. Type ⌞exists ↵ x ↵ T ↵ and ↵ so_var1 ↵ P ↵ x ↵⌟

```
                    - TERM: Navigator
OK*  Cancel*

add def : exists_uni(T; x.P[x]) ==
          ∃x:T. (P[x]  ∧ [prop]
```

and then ⌞all ↵ y ↵ T ↵ implies ↵ so_var1 ↵ P ↵ y ↵ equal ↵ x ↵ y ↵ T ↵⌟.

```
                    - TERM: Navigator
OK*  Cancel*

add def : exists_uni(T; x.P[x]) ==
          ∃x:T. (P[x]  ∧  ∧ (∀y:T. (P[y]  ⇒  y=x ∈ T)))
```

The definition is now complete. To save it to the library click OK* or press ↵ again. This closes the AddDef template and creates a display form **exists_uni_df** of kind DISP, an abstraction object **exists_uni** of kind ABS, and a well-formedness theorem **exists_uni_wf** of kind STM. The navigation pointer is still where it has been before.

```
                    - TERM: Navigator
MkTHY*   MkThyDir*  ExportThy*  OpenThy*  CloseThy*  ChkThy*  ChkAllThys*  ChkOpenThy*
CheckMinTHY*  MinTHY*  EphTHY*  ExTHY*

Mill*  ObidCollector*  NameSearch*  PathStack*  RaiseTopLoops*

PrintObjTerm* PrintObj* MkThyDocObj* ProofHelp* ProofStats* showRefEnvs* FindTheoriesMin*
CpObj*  reNameObj*  EditProperty*  SaveObj*  RmLink*  MkLink*  RmGroup*

ShowRefenv*  SetRefenvSibling*  SetRefenv*  ProveRR*  SetInOBJ*
MkTHM*  MkML*  AddDef*  AddRecDef*  AddRecMod*  AddDefDisp*  AbReduce*  NavAtAp*
Act*  DeAct*  MkThyDir*  RmThyObj*  MvThyObj*

↑↑↑↑  ↑↑↑ ↓↓↓↓  ↓↓↓  <>   ><

Navigator: [kreitz; user; theories]

Scroll position : 0

List Scroll : Total 4,  Point 0,  Visible : 4
 ---------------------------------------------
 -> STM   TFF  not_over_and
    DISP  TTF  exists_uni_df
    ABS   TTF  exists_uni
    STM   TFF  exists_uni_wf
 ---------------------------------------------
```

The abstraction object contains exactly what has been typed into the **AddDef** template and usually does not have to be edited anymore. The display form has been generated from the left hand side of the **AddDef** template and currently causes the term **exists_uni** to be displayed in exactly this way. The well-formedness theorem is empty but already activated, which enables the general tactics to access it whenever they have to deal with **exists_uni**.

To view the abstraction, move the navigation pointer down 2 steps then open the object **exists_uni** by pressing the right arrow key.

```
                    - TERM: Navigator
MkTHY*   MkThyDir*  ExportThy*  OpenThy*  CloseThy*  ChkThy*  ChkAllThys*  ChkOpenThy*
CheckMinTHY*  MinTHY*  EphTHY*  ExTHY*

Mill*  ObidCollector*  NameSearch*  PathStack*  RaiseTopLoops*

PrintObjTerm* PrintObj* MkThyDocObj* ProofHelp* ProofStats* showRefEnvs* FindTheoriesMin*
CpObj*  reNameObj*  EditProperty*  SaveObj*  RmLink*  MkLink*  RmGroup*

ShowRefenv*  SetRefenvSibling*  SetRefenv*  ProveRR*  SetInOBJ*
MkTHM*  MkML*  AddDef*  AddRecDef*  AddRecMod*  AddDefDisp*  AbReduce*  NavAtAp*
Act*  DeAct*  MkThyDir*  RmThyObj*  MvThyObj*

↑↑↑↑  ↑↑↑ ↓↓↓↓  ↓↓↓  <>   ><

Navigator: [kreitz; user; theories]

Scroll position : 2

List Scroll : Total 4,  Point 2,  Visible : 4
 ---------------------------------------------
    STM   TFF  not_over_and
    DISP  TTF  exists_uni_df
 -> ABS   TTF  exists_uni
    STM   TFF  exists_uni_wf
 ---------------------------------------------
```

```
                    - ABS: exists_uni

exists_uni(T; x.P[x]) ==
  ∃x:T. (P[x]  ∧  (∀y:T. (P[y]  ⇒  y=x ∈ T)))
```

The abstraction object shows on the right hand side the term that defines the meaning of **exists_uni(T;x.P[x])** and on the left hand side the form in which **exists_uni(T;x.P[x])** is currently displayed. Right now, this is identical to the abstract term form. To close the abstraction object again press ⟨C-q⟩.

To change the appearance of the term `exists_uni(T;x.P[x])` to `∃!x:T. P[x]` you have to edit the accompanying display form. For this purpose, move the nav point one step up and open the object `exists_uni_df`.

```
- TERM: Navigator
MkTHY*   MkThyDir*  ExportThy*  OpenThy*  CloseThy*  ChkThy*  ChkAllThys*  ChkOpenThy*
CheckMinTHY*  MinTHY*  EphTHY*  ExTHY*

Mill*  ObidCollector*  NameSearch*  PathStack*  RaiseTopLoops*

PrintObjTerm* PrintObj* MkThyDocObj* ProofHelp* ProofStats* showRefEnvs* FindTheoriesMin*
CpObj*  reNameObj*  EditProperty*  SaveObj*  RmLink*  MkLink*  RmGroup*

ShowRefenv*  SetRefenvSibling*  SetRefenv*  ProveRR*  SetInOBJ*
MkTHM*  MkML*  AddDef*  AddRecDef*  AddRecMod*  AddDefDisp*  AbReduce*  NavAtAp*
Act*  DeAct*  MkThyDir*  RmThyObj*  MvThyObj*

↑↑↑↑  ↑↑↑ ↓↓↓↓  ↓↓↓  <>   ><

Navigator: [kreitz; user; theories]

Scroll position : 1

List Scroll : Total 4,  Point 1,  Visible : 4
 ---------------------------------------------
    STM   TFF  not_over_and
 -> DISP  TTF  exists_uni_df
    ABS   TTF  exists_uni
    STM   TFF  exists_uni_wf
 ---------------------------------------------
```

```
- DISP: exists_uni_df
EdAlias exists_uni ::
exists_uni(<T:T:*>;<x:var>.<P:P:*>) ==
  exists_uni(<T>; <x>.<P>)
```

The display form consists of a list of attributes (in this case only the name that can be used to open the template), a template that determines the outer appearance of a term, and the term that is to be represented by that template. Both the template and the term contain slots that are marked with `<..>` and describe the name of a placeholder, a description that will appear whenever the template is initiated, and information about parenthesizing. Chapter 7.2 describes how to create display forms from scratch, but usually copying and pasting is sufficient.

To edit the template click mouse over the `exist_uni` in the second line and erase the text `exist_uni(` using backspace and delete keys. Notice that you can't delete the slot `<T:T:*>` with these keys. Type ⌞⟨C-#⟩163!⌟ to generate the ∃ symbol (see table 5.1 on page 73 for a list of all special characters) and the exclamation mark. Delete the semicolon and the dot between the term slots and also the right parenthesis.

```
- DISP: exists_uni_df
EdAlias exists_uni ::
∃!<T:T:*><x:var><P:P:*> ==
  exists_uni(<T>; <x>.<P>)
```

To rearrange the order of the slots click left over `<T:T:*>` press ⟨M-p⟩ to mark the full slot, and ⟨C-k⟩ to cut it. Then move the mouse to the immediate right of `<x:var:*>` and press ⟨C-y⟩. Add a colon between `<x:var:*>` and `<T:T:*>`, a dot and a space between `<T:T:*>` and `<P:P:*>`.

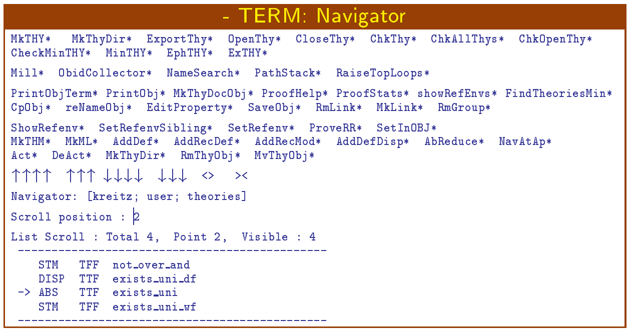
```
- DISP: exists_uni_df
EdAlias exists_uni ::
∃!<x:var>:<T:T:*>. <P:P:*> ==
  exists_uni(<T>; <x>.<P>)
```

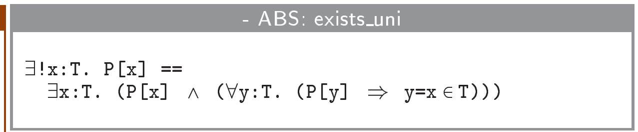
In principle, the display form is now complete. However it is advisable to edit the slot description of `T` and `P`. Click right of the second `T` in `<T:T:*>`, remove it and enter ⌞type⌟ instead. In the same way change the second `P` in `<P:P:*>` to `prop`. This makes sure that meaningful descriptions for theses slots will show up whenever the template for `exists_uni` is opened.

```
- DISP: exists_uni_df
EdAlias exists_uni ::
∃!<x:var>:<T:type:*>. <P:prop:*> ==
  exists_uni(<T>; <x>.<P>)
```

To commit the completed display form to the library, press ⟨C-Z⟩. This will also close the window again. *Do not use* ⟨C-Q⟩ unless you want all your changes to be ignored. ⟨C-Q⟩ will always close a window without saving its contents to the library.

To check the display form, open the abstraction **exists_uni** again. You will notice that the display of the abstract term `exists_uni(T;x.P[x])` has now been replaced by `∃!x:T. P[x]`.





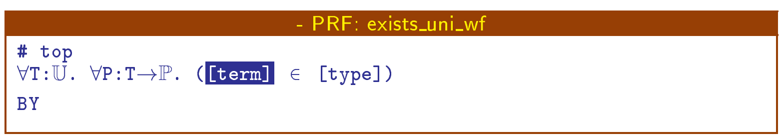
The abstraction and the display form are sufficient for using a newly defined term in formal theorems and other abstractions. However, many proofs involving an instance of `exists_uni` will involve checking the type of this term. This can be done automatically if *well-formedness theorem* is provided describes the type of the newly defined term.

A unique existence quantifier `∃!x:T. P[x]` is a proposition, provided that `T` is a type and `P` is a predicate on `T`. In other words, the type of `∃!x:T. P[x]` is ℙ if `T` is an element of the universe 𝕌 of types and `P ∈ T→ℙ`. Formally, you need to prove `∀T:𝕌. ∀P:T→ℙ. ∃!x:T.P[x] ∈ ℙ`.

To state this theorem, open the object `exists_uni_wf` and enter

⌞all ↵ T ↵ univ ↵ i ↵ all ↵ P ↵ fun ↵ T ↵ prop ↵ member ↵⌟

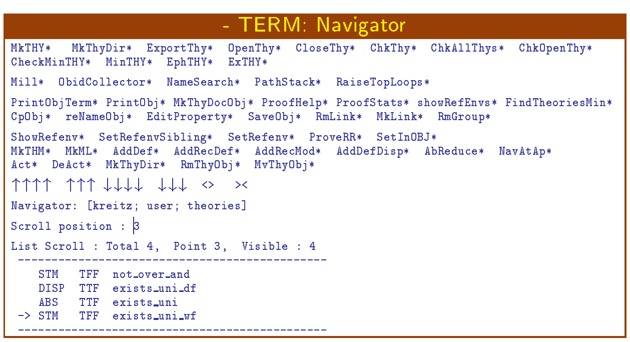
into the goal template.

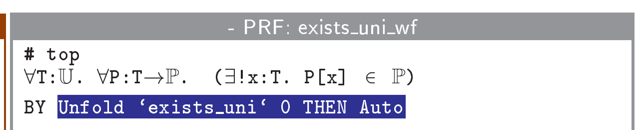


Continue with ⌞exists_uni ↵ x ↵ T ↵ so_apply1 ↵ P ↵ x ↵ prop ↵ ⟨C-M-g⟩⌟ to complete the goal and commit it to the library.

The proof of this theorem is fairly simple, since it can be handled almost completely by NUPRL's tactic `Auto`. However, since `Auto` does not unfold a definition unless it is explicitly declared as automatically unfoldable, you need to unfold the definition of `exists_uni` in the first step.

Typing ⌞Unfold ‘exists_uni‘ 0 THEN Auto⌟ into the rule slot will result in a proof of the well-formedness theorem for `exists_uni`.





This completes all necessary steps for adding a unique existence quantifier to the formal language of `Nuprl`. Close the well-formedness theorem with ⟨C-Q⟩.

2.7 Printing Snapshots

To print a snapshot of a particular object, simply click the PrintObj* button. This will create a print representation of the object at the nav point and write it into a file ~/nuprlprint/OBJECT-NAME.prl. This file can be inspected with any 8bit capable editor that has the NUPRL fonts loaded. It will also create a LaTeX-version and write it to ~/nuprlprint/OBJECT-NAME.tex. The directory ~/nuprlprint must already exist. Otherwise clicking the PrintObj* button will result in an error.

The button PrintCollection will create a print representation of a whole range of objects, while the buttons PrintThyLong and PrintThyShort will create a print representation of the directory at the nav point. The long version shows the complete proofs of theorems while the short version only prints the theorem statement and the extract of the proof, that is a term representing its computational content.

2.8 Troubleshooting

Since the NUPRL library never destroys information, typos and almost all commands can be undone by entering the key combination ⟨C-_⟩. The undo history, however, is limited and recovering from an error that was made many steps ago is more difficult.

Should you accidentally close the navigator window you may open it again by typing the command win. into the editor process top loop. This will open all the windows of the initial screen.

If the editor hangs for an unusually long time, one of the three main processes (editor, refiner, or library) may be broken. In this case there will be an error message in the corresponding (emacs) top loop. Often, evaluating the expression (fooe) will recover the process. Sometimes, typing the Lisp command :cont will help as well. In the worst case, kill the process and then restart it. The library process will detect the link going dead and clean it up automatically.

2.9 Shutting NUPRL down

In principle, there is no need to shutdown NUPRL, as all data are saved immediately and updates may be integrated by by loading patches into the running process. However the NUPRL 5 are quite demanding as far as cpu and memory are concerned.

To shutdown gracefully you should first close the refiner and the editor and then the library. Enter stop. at the corresponding ML prompt.

```
ML[(ORB)]> stop.
```

As a result, the editor and refiner will communicate to the library that they will disconnect now and then stop the respective ML and Lisp processes. The library process will cleanly shut down the knowledge base and then stop as well.

Instead of shutting down gracefully you may also simply kill all three processes to stop NUPRL 5, although this is not recommended.

Chapter 3

Running Nuprl 5

3.1 System Requirements

Nuprl 5 is written mostly in Common Lisp, but uses some extensions that require Lucid, Allegro or LCMU Common Lisp and a Unix-based X window system. The implementation of CMU Common Lisp is freely available. Other Lisp versions require a license. CMUCL is faster with smaller memory footprint but currently Allegro is a bit more stable.

The Nuprl homepage provides an executable copies of the CMUCL version of Nuprl 5 running under Linux. Executable copies for Allegro can be provided upon request. The source code as well as instructions for installing Nuprl 5 to run under other Lisp and Unix versions will be made available as soon as the system has stabilized.

The Linux release of Nuprl 5 contains 3 binary executables (the library, the editor, and the refiner) that altogether require 120 MegaBytes of disk space for the CMUCL version and 40 MegaBytes for the Allegro Version. The standard library initially requires an additional 120 MegaBytes of disk space and quickly grows to 500 MegaBytes or more.

It is recommended to run Nuprl 5 on systems that have at least 256MB of RAM, 512MB of swap space, and 800MB of disk space available. Due to its implementation in Lisp, Nuprl runs more efficiently if more memory is available. In large applications it can utilize several GigaBytes of RAM. Nuprl 5 can also profit from multiple processors or a network of computers, because the library, editor, and refiner run as independent processes,

3.2 Preparation

Before Nuprl 5 can be started for the first time, it needs to be installed properly and certain configuration files must be set up for each user.

3.2.1 Retrieving and Installing Nuprl 5

The executable copy of Nuprl 5 running under Linux can be found by going to the Nuprl 5 web page `http://www.nuprl.org/nuprl5/index.html` and following the link to the download pages for the actual Nuprl 5 release.[1] To retrieve Nuprl 5, read the instructions in the various README files for up-to-date information.

[1] Currently, the release can be found at the URL `ftp://ftp.cs.cornell.edu/pub/nuprl/nuprl5` but this directory may be moved in the future.

Currently you have to download the following files.

```
install.tgz
nupr1top.tgz
nupr15.tgz
standard-db.tgz
nupr15-linux-cmucl-run.tgz
```

If you run Solaris instead of Linux, download the file `nuprl5-solaris-cmucl-run.tgz` instead of `nuprl5-linux-cmucl-run.tgz`. Experienced users who want to build their own NUPRL binaries or modify system tactics may also want to download the file `nuprl5-source.tgz`.

For a single-user installation, it is best to build the NUPRL system within a subdirectory of the user's home directory. For a default installation, this subdirectory should be called `nuprl`. For a multi-user installation, it is recommended to build NUPRL in a directory `/home/nuprl` and to create a user `nuprl` and a group `nuprl` that gives users controlled access to this directory.

In the following we describe the default single-user installation for Linux. Instructions for a custom installation can be found in the file `README.install`.

Move all the downloaded .tgz files into your home directory and then untar the file `install.tgz` using the commands

```
SHELL-PROMPT> cd ~
SHELL-PROMPT> tar -xzf install.tgz
```

This will build a directory `/nuprl` in which you will find an installation script `install-nuprl5.pl` as well as a file `README.install` with instructions for building NUPRL 5. As the installation procedure may change in the future, it is advisable to read these instructions before starting the actual installation.

Enter the `nuprl` subdirectory and execute the installation script.

```
SHELL-PROMPT> cd ~/nuprl
SHELL-PROMPT> ./install-nuprl5.pl
```

This will untar the other .tgz files in your home directory and build the NUPRL system within the current directory. The NUPRL knowledge base will be installed at `~/nuprl/nuprl5/NuPrlDB`.

The library, editor, and refiner processes as well as several utilities can be found in `~/nuprl/bin`. To run NUPRL 5, the directory for NUPRL binaries must be included in the user's Unix load path. To do so, add the following line to your .chsrc or .login file

```
set path = (  /nuprl/bin $path)
```

The syntax is slightly different for other Unix shells.

To support the use of special mathematical symbols in formal theorems, the NUPRL system requires some special font files to be installed. These files are contained in the directory `~/nuprl/fonts/bdf` and must be included in the font search path of the X server controlling your display.[2] Add the following lines to your .xinitrc file

```
xset fp+  /nuprl/fonts/bdf
xset fp rehash
```

These commands tell X the font path to the NUPRL fonts when the X server is first started. One may also run them interactively in some shell to add the font path to the *current* X environment.

[2]It is not sufficient to have the NUPRL-fonts available on the system that runs NUPRL 5. This may cause some complications when running NUPRL 5 remotely, particularly when the Exceed X server under Windows (NT/98/2000) is used as terminal.

Most X windows systems understand bdf font files. However, one may also compile the font files into machine-specific fonts to allow faster reading. To compile fonts, one has to use the command **bdftosnf** on Sparc stations and **bdftopcf** on Linux-PCs. After the fonts have been compiled,, one has to execute the command **mkfontdir** to make the font directory.

Sometimes the X-server has limited access to the file system. In this case the fonts files may need to be placed in a public directory. Your system administrator should be able to advise you in this case.

3.2.2 User-specific Configuration

Upon startup NUPRL 5 expects to find a file `~/.nuprl.config` in the user's home directory to determine how the individual NUPRL processes will communicate. If this file is missing, NUPRL 5 will use the configuration that was present at compile time and may not be able to establish the communication. The structure of a typical `.nuprl.config` file is as follows.

```
;; ---------------------------------------------------------------
;; Which server runs the library process?
;; ---------------------------------------------------------------
(libhost "HOSTNAME")

;; ---------------------------------------------------------------
;; Where is the location of the Nuprl Library?
;; ---------------------------------------------------------------
(dbpath "DATABASE-PATH")

;; ---------------------------------------------------------------
;; What environment within the Nuprl Library shall be used?
;; ---------------------------------------------------------------
(libenv "STANDARD-LIBRARY")

;; ---------------------------------------------------------------
;; Optional settings
;; ---------------------------------------------------------------
;; Which X-server shall be used to display the Nuprl windows?
;; ---------------------------------------------------------------
;; (eddhost "localhost" 0)

;; ---------------------------------------------------------------
;; What sockets shall be used for communication?
;; ---------------------------------------------------------------
;; (sockets SOCKET1 SOCKET2)

;; ---------------------------------------------------------------
;; How to identify the user?
;; ---------------------------------------------------------------
;; (iam "USER-NAME")

;; ---------------------------------------------------------------
;; Color and size of displayed windows
;; ---------------------------------------------------------------
;; (foreground "FOREGROUND-COLOR'')
;; (background "BACKGROUND-COLOR")
;; (font "FONT-NAME")
```

The HOSTNAME for the **libhost** configuration should be the name of the host running the library process. Currently even a stand-alone machine needs a host name.

The values for **dbpath** and the **libenv** describe the physical and logical location of the standard library. In the default single-user installation, it should be `~/nuprl/nuprl5/NuPrlDB`. In a multi-user installation, it is usually `/home/nuprl/nuprl5/NuPrlDB`. The value of STANDARD-LIBRARY is usually `standard`.

All other entries of in the `.nuprl.config` file are optional. Users may redirect the Nuprl windows to a specific X server or choose a specific set of sockets for communication between the Nuprl processes. If several users shall work with the same library, the socket numbers should be identical to the ones used by the joint library process. Specific users can be identified by the system, which will be necessary for granting controlled access to certain parts of the library.

The `foreground` and `background` colors set the colors for the Nuprl windows and can be chosen according to personal preferences. Users may also chose a font for the Nuprl windows. By default, `nuprl-13` is being used but users may also select any other Nuprl fonts or other fonts that are consistent with them.[3]

The Nuprl 5 editor will read the file `~/mykeys.macro` to determine any user key bindings for motion and macro commands. If this file is missing, the key bindings that were present at compile time will be used. The standard bindings of the Nuprl 5 system are listed in the file `~/nuprl/nuprl5/sys/macro/keys.macro`. Users who wish to customize their Nuprl 5 key bindings may copy this file into the file `~/mykeys.macro` and modify it according to their needs.

It is helpful for the user to become familiar with an editor like emacs (version 19 and higher) that supports 8-bit fonts and has a capability for starting sub-shells. The editor should be run with one of the nuprl fonts. This is not strictly necessary, but is a good idea for several reasons:

- Each Nuprl process runs a "top loop" in the same window as the one from which it was started up. It accepts input from that window and frequently writes output to it. If Nuprl is started up from an editor sub-shell, it becomes easy to review this output and save portions of it to files. Editing capabilities for the input are sometimes useful as well.
- Some of Nuprl's output is in Nuprl's 8-bit font.
- Listings of theory files use Nuprl's 8-bit font. These files contain definitions, theorems and proofs, and it is often useful to be able to browse them.

3.3 Starting Nuprl 5

The basic Nuprl 5 configuration consists of three separate processes: a library, an editor, and a refiner. In single-user mode you have to start all three processes. In multi-user mode you will connect to an already running library process and only have to start an editor and an optional refiner[4] after setting up your `.nuprl.config` file accordingly. It is important to initialize the library before the editor and the refiner.

Generally it is a good idea to run the Nuprl 5 processes in separate emacs frames. In addition to editor support for the corresponding top loops this also allows you to define an interactive emacs command that starts all three processes and initializes them correctly. The Nuprl distribution provides a few utilities for starting the Nuprl 5 processes from within emacs buffers. Consult the file `README.running-nuprl` for instructions how to use them.

The next three subsections describe three interactive emacs commands `nulib`, `nuedit`, and `nurefine`. If you add these and the following definition of the emacs command `nupr15` to your `.emacs` file, you may start Nuprl 5 from emacs by typing ⌞⟨M-x⟩nuprl5 ↵⌟.

[3]The table of Nuprl's special characters (Table 5.1 on page 73) is actually based on a font `nuprl-8x13`, which extends `nuprl-13` by a few special characters with code 204 and higher.

[4]Actually, it may not even be necessary to start a new Nuprl 5 refiner, as the editor will connect to refiners that are already running. However, this would mean that you need to share that refiner with other users, which may cause unnecessary delays if the refiner is busy. Generally, it is recommended that you start your own refiner unless you only intend to browse the library.

```
(defun nuprl5 ()
  (interactive)
  (message "Starting NuPRL 5 Library, Editor, and Refiner ...")
  (nulib)
  (sleep-for 5)
  (nuedit)
  (nurefine)
)
```

It will take several minutes until all the NUPRL editor windows will begin to pop up, because initially there is a lot of communication between the editor and the library.

3.3.1 Starting the Library

The library process should be started first because both the editor and the refiner rely on information that is explicitly stored in the knowledge base (to simplify customization of these processes). In a shell, enter the command ⌞nulib ↵⌟.

SHELL-PROMPT> nulib

A Lisp session will start, followed by system messages. At the Lisp USER prompt enter ⌞(top) ↵⌟.

USER(1): (top)

This will start an ML *top loop* with some library-specific commands preloaded. At the ML prompt, enter ⌞go. ↵⌟ to initialize the NUPRL 5 library.

CURRENT TIME : TIME AND DATE

ML[(ORB)]> go.

The library process will now use the information in the file `.nuprl.config` to load the desired library environment and to open the sockets for communication. It will write some system messages to the process window and then wait for other processes to connect.

The following emacs script defines an interactive function `nulib` that performs all the above steps in a new emacs shell. Using the function `nuprl-frame` it pops up a new frame at a specific position on the screen, opens a shell process `*NuLibrary*` in it, and then subsequently sends the above command to that process.

```
(defun nuprl-frame (bufname height top-corner cmd)
  (save-excursion
    (set-buffer (make-comint bufname "/bin/csh" nil "-v"))
    (switch-to-buffer-other-frame (concat "*" bufname "*" ))
    (let ((NuPRLframe (car (cadr (current-frame-configuration)))))
      (set-frame-size     NuPRLframe 81  height)
      (set-frame-position NuPRLframe 515 top-corner)
    )
    (set-default-font "6x10")
    (while (= (buffer-size) 0) (sleep-for 1))
    (comint-send-string bufname "limit coredumpsize 0\n")
    (comint-send-string bufname cmd)
) )

(defun nulib ()
  (interactive)
  (nuprl-frame "NuLibrary" 10 76 "nulib\n")
  (message "Starting Library ...")
  (set-foreground-color "Red")
  (set-background-color "#ddddff")
  (comint-send-string "NuLibrary" "(top)\n")
  (comint-send-string "NuLibrary" "go.\n")
)
```

The shell script is designed for an XGA (1024x786) display and uses a fairly small font. You need to adjust the frame position on larger displays and if a larger font is chosen. The colors are chosen to distinguish the library frame from the other windows.

Experienced users can make further use of the function `comint-send-string` to send additional commands to the NUPRL 5 process at their convenience.

3.3.2 Starting the Refiner

Starting the refiner is similar to starting the library. In a shell, enter the command ⌞`nuref` ↵⌟.

```
SHELL-PROMPT> nuref
```

At the Lisp `USER` prompt enter ⌞`(top)` ↵⌟.

```
USER(1): (top)
```

At the ML prompt, enter ⌞`go.` ↵⌟ to initialize the NUPRL 5 refiner.

```
ML[(ORB)]> go.
```

Although it is not necessary to wait for the library process before starting the refiner it is important to enter the refiner's `go.` command *after* the library's `go.`. The library contains informations about tactics and rules that the refiner needs in order to operate properly. Initially there will be only little exchange between the library and the refiner. The refiner process will return quickly with another prompt `ML[(ORB)]>`.

The following `emacs` script defines an interactive function `nurefine` that performs the above steps in a new `emacs` shell. It pops up a `*NuRefine*` window immediately below the editor window and starts the NUPRL 5 refiner in it.

```
(defun nurefine ()
  (interactive)
  (nuprl-frame "NuRefine" 10 280 "nuref\n")
  (message "Starting Refiner ...")
  (set-foreground-color "Green4")
  (set-background-color "#ffffbb")
  (comint-send-string "NuRefine" "(top)\n")
  (comint-send-string "NuRefine" "go.\n")
)
```

It should be noted that the `emacs` functions `nulib`, `nuedit`, and `nurefine` can be invoked independently from each other. This may be helpful if one of the processes breaks and has to be completely restarted or if several refiners and/or editors shall be started.

3.3.3 Starting the Editor

To start the editor, enter the command ⌞`nuedd` ↵⌟ into a shell and then proceed as before.

```
SHELL-PROMPT> nuedd

USER(1): (top)

ML[(ORB)]> go.
```

Again, the library's `go.` command must precede that of the editor. The library contains a variety of explicit set up information that the editor needs to receive in order to determine how to display data, e.g. how to present the directory structure of the knowledge base, when and how to pop up windows, the location and meaning of editor buttons, etc.

Because of the amount of communication between the editor and the library it takes several minutes until the editor process is set up correctly. When it is ready, it will return with another prompt `ML[(ORB)]>`.
Enter ␣win. ↵␣ to have the editor pop up NUPRL 5 windows.

```
ML[(ORB)]> win.
```

The editor will establish a connection to the X-windows system and then pop up two windows: a *navigator* and a *top loop*, which will be explained in detail in Chapter 4. Afterwards it will return with another prompt `ML[(ORB)]>`.

The NUPRL *top loop* can be used for issuing commands to the library, refiner, and editor processes and provides support for editing NUPRL 5 terms (see Section 5.3). Users may safely iconify the emacs windows of the library, refiner, and editor processes at this point. A typical NUPRL 5 session will have many NUPRL windows open at the same time, so it is advisable to create some space for this, particularly when using medium-sized or larger fonts. The library, refiner, and editor process windows will only be needed for issuing low-level system commands and dealing with error situations that cause one of the NUPRL processes to break.

The following emacs script defines an interactive function `nuedit` that performs all the above steps in a new emacs shell. It pops up a `*NuEditor*` window immediately below the library window and starts the NUPRL 5 editor in it.

```
(defun nuedit ()
  (interactive)
  (nuprl-frame "NuEditor" 10 178 "nuedd\n")
  (message "Starting Editor ... please be patient")
  (set-foreground-color "midnightblue")
  (set-background-color "#ffd8ff")
  (comint-send-string "NuEditor" "(top)\n")
  (comint-send-string "NuEditor" "go.\n")
  (comint-send-string "NuEditor" "win.\n")
)
```

If you run the NUPRL 5 editor on a remote machine, make sure that its display is directed to your local machine *before* `nuedd` is entered. This is usually done by setting the environment variable `DISPLAY`. In a cshell you can do this with the following command

```
CSHELL-PROMPT> setenv DISPLAY LOCAL-HOST-NAME:0.0
```

In the emacs script you would have to insert the line

```
(comint-send-string bufname "setenv DISPLAY LOCAL-HOST-NAME:0.0\n'")
```

into the definition of `nuprl-frame`, immediately before `(comint-send-string bufname cmd)`.

Alternatively you may change the `eddhost` setting in you `.nuprl.config` file to redirect the NUPRL windows to your local machine's X server.

3.4 Exiting NUPRL 5

When you are ready to stop, first stop the editor and refiner process, and lastly the library. To shutdown gracefully, enter ␣stop. ↵␣ at the ML prompts of the three processes.

```
ML[(ORB)]> stop.
```

As a result, the editor and refiner will communicate to the library that they will disconnect now and then stop the respective ML and Lisp processes. The library process will cleanly shut down the knowledge base and then stop as well. Depending on the size of the knowledge base this may take between a few seconds and several minutes.

It is important that you explicitly terminate the three NUPRL processes rather than just quitting out of the editor NUPRL 5 is running under. In the latter case, the Lisp process can be left floating around in a hung state, hogging memory resources. This could also happen if your editor crashes or if you kill the shell (or emacs buffer) in which NUPRL 5 runs. You can use the Unix command `ps` to check for a hung Lisp process and the command `kill` to kill it.

The interactive emacs command `nuxit`, described below, is a safe way to terminate NUPRL 5.

```
(defun nuxit ()
  (interactive)
   (message "Shutting Down NuPRL 5 Library, Editor, and Refiner ...")
   (comint-send-string "NuEditor"  "stop.\n")
   (comint-send-string "NuRefine"  "stop.\n")
   (sleep-for 5)
)
```

Advanced emacs users may want to add to this script commands that kill the respective buffers and emacs frames after the library process has terminated.

Instead of shutting down gracefully you may also simply kill all three processes to stop NUPRL 5. In this case the library process will clean up the knowledge base when it is started the next time. This method for exiting NUPRL, however, is is not recommended.

3.5 Hints on Using the System

NUPRL's windows are at the "top-level" in the X environment. The windows can be managed (positioned, sized, etc.) in the same way as other top-level applications such as X-terminals. Creation and destruction of NUPRL windows, and manipulation of window contents, is done solely via commands interpreted by NUPRL.

NUPRL will receive mouse clicks and keyboard strokes whenever the input focus is on any of its windows. Any input event will make this window *active*, which is identified by the presence of NUPRL's *cursor*. This cursor appears either as a thin vertical bar between characters or as a highlighted (reverse video) region. The specific location of the cursor determines the semantics of keyboard strokes and mouse clicks, and is – like in most editors – independent of the current location of the mouse cursor.

The two main windows – the navigator window and the NUPRL 5 top loop – are intended to remain throughout the session. You may kill and reopen them at any time although it is not recommended to do so. You may create multiple clones of the navigator but not of the top loops. Chapter 4 describes the use of these windows as well as the kinds of objects that can be found in the library.

There are two other kinds of windows; *term editor* windows and *proof editor* windows. Both are used for editing objects in the library. The structure of NUPRL terms and the term editor is described in Chapter 5. The proof editor is described in Chapter 6.

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

Most Lisp versions allow computations to be interrupted. This is usually done by sending ⟨C-C⟩ to the Lisp process, or ⟨C-C⟩⟨C-C⟩ if Lisp is started up from an emacs sub-shell. (Sometimes Lisp catches the first two or three interrupt requests.) This will cause Lisp to enter its *debugger*, from which the computation can be resumed or aborted. Section 3.6 below describes how to use the Lisp debugger, and in particular, what to do if a NUPRL 5 process crashes. NUPRL is a continually-evolving experimental research system, and it is inevitable that it will contain bugs.

Aborting either of the three NUPRL 5 processes is always safe, since changes to objects, e.g. the effects of editor commands or inference steps, are immediately committed to the persistent library. When a NUPRL 5 process is restarted, the state should be exactly as it was before the process was killed.

Please report any behavior you think is due to a bug, or inconsistencies between the operation of the system and the documentation. Also report any break-points that you hit; they have either been left in the code accidentally, or they are there to help track down the source of bugs. We welcome suggestions for improvement. Send e-mail to nuprlbugs@cs.cornell.edu.

3.6 Troubleshooting

In this section we discuss problem situations that need to be resolved on the system level. Recovering from errors *within* either of the editors or the navigator is discusses in the respective chapters.

All system and error messages are directed to the emacs windows containing the three NUPRL 5 processes. It is recommended to check these windows if the system seems to be stuck or if other problems occur.

In most cases where the system appears to be stuck, one of the three Lisp processes is garbage-collecting. Depending on processor speed and available memory this may take a few minutes.

If the editor hangs for an unusually long time, one of the three main processes may have been thrown into the Lisp debugger. This may happen if a breakpoint was mistakenly left in the NUPRL code or if you hit a bug. You may also have accidentally interrupted Lisp. In either there will be an error message in the corresponding (Lisp) top loop. The particular debugger appearance and commands given below are for Allegro Common Lisp. Other Lisps should be similar.

The initial message put out by the debugger should tell you what caused it to be invoked. The following message, for instance appears after a keyboard interrupt

```
Error: Received signal number 2 (Keyboard interrupt)
  [condition type: INTERRUPT-SIGNAL]

Restart actions (select using :continue):
 0: continue computation

[changing package from "COMMON-LISP-USER" to "NUPRL5"]
[1c] NUPRL5(2):
```

To resume after an interrupt or breakpoint, enter ⌞:cont ↵⌟.

```
[1c] NUPRL5(2): :cont

ML[(edd)]>
```

If the ML prompt appears again, the process has successfully resumed. In some cases, Lisp cannot simply resume and will print another error message, as in the following case

```
Error: Non-structure argument NIL passed to structure-ref
[1] NUPRL5(3): :cont
Error: Can't continue and no restarts.
[2] NUPRL5(4):
```

In most of these cases, entering the expression ⌞(fooe) ↵⌟ will reset and restart the process.

```
[2] NUPRL5(4): (fooe)
ML[(edd)]>
```

In the worst case, kill the process by entering ␣:exit↵␣ and then restart it from scratch. The other processes will detect the link going dead and clean it up automatically.

If you kill a NUPRL window using window manager commands instead of the appropriate NUPRL editor commands, you will break the X connection and crash the editor. Future releases of NUPRL 5 will automatically repair the editor process but until then you have to recover at the lisp debug prompt using the following command sequence.

```
[2] NUPRL5(4): (ml-text "nuprl_oed_reset()")
[2] NUPRL5(5): (fooe)
ML[(edd)]> win.
```

If you kill the library process while it is shutting down the knowledge base, the knowledge base may end up in an unstable state and you may not be able to restart the system anymore. In this case you have to enter the directory containing the knowledge base (e.g. `~/nuprl/nuprl5/NuPrlDB`) and move the most recent subdirectories out of it, preserving them in some temporary directory.

The system will usually come up properly afterwards and return the knowledge base into a well-defined state but you will lose all the modifications recorded in the subdirectories that you moved out of the main directory, so it may be useful to move them out one at a time, until the NUPRL system starts again.

If you run into the same type of unrecoverable error twice, you may want to send a bug report to `nuprlbugs@cs.cornell.edu`. In this case type ␣:zoom↵␣ into the Lisp process before killing it and copy the output, together with the initial error messages into you bug report. Also, mention briefly what you were doing at the time of the crash. This will help the NUPRL programmers to identify the cause for the problem and fix it.

3.7 Customization

Experienced users will probably want to create their own initialization procedures for NUPRL. These could allow customizations such as:

- Changing key-bindings for the term and proof editors.
- Loading more / different tactics.

Currently, these initialization can only be run after starting up the pre-prepared disksaves for the NUPRL 5 library, editor, or refiner. You probably will want to put all your initialization commands into a Lisp file that is automatically loaded whenever a disksave is started up.

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

The key bindings for the navigator and the term and proof editors can be altered by creating your own key macro files. The NUPRL 5 editor will look for a file `~/mykeys.macro` to determine any user-defined key bindings.

Chapter 4

The Navigator and the Top Loops

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

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

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

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

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

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

4.1 The Library

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

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

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

4.1.1 Library Objects

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.1.2 The Library Table

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

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

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

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

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

4.1.3 User Interaction with the Library

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

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

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

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

4.2 Nuprl Windows

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

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

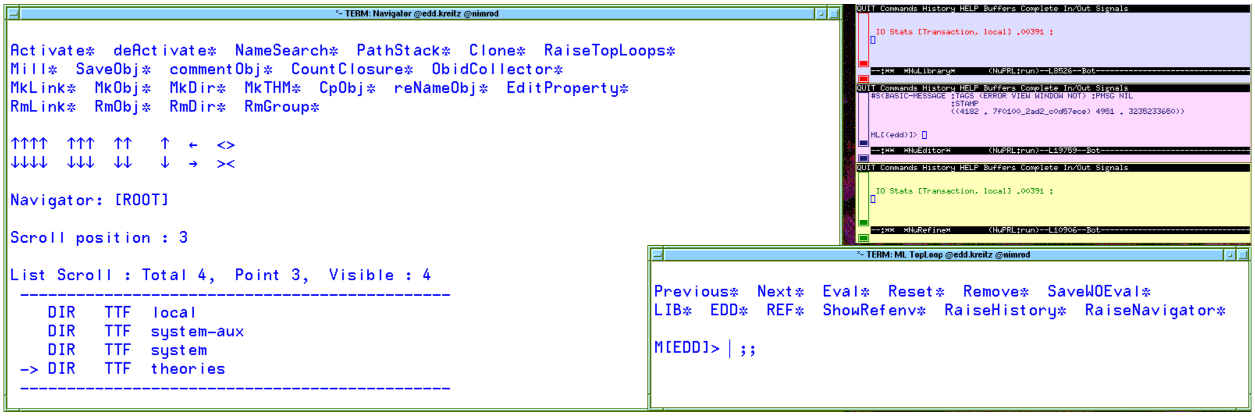


Figure 4.1: Initial NUPRL 5 screen

4.2.1 The Navigator Window

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

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

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

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

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

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

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

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

> `STM` stands for *statement* objects, `PRF` for *proof* objects, `INF` for *inference* objects, `ABS` for *abstractions*, `DISP` for *display forms*, `PRC` for *precedence* objects, `CODE` for *ML* code, `RULE` for inference *rules*, `COM` for *comments*, `DIR` for *directories*, and `TERM` for *objects* of unspecified kind.
>
> Proof and inference objects are usually not listed in the directory but can be accessed only through the proof editor (Chapter 6.2).

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

> The first character describes whether the object has been *activated* and is `T` in most cases.
>
> The second character is reserved for theorem objects and describes if the theorem has a complete proof and an extract term. For all other objects it is `T`.
>
> The third character describes the status of the sticky bit. It is `F` for most objects.

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

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

4.2.2 The ML Top Loop Window

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

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

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

4.2.3 The Process Top Loop Windows

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

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

4.3 Library Commands

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

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

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

These bindings are also valid in many other editing contexts.

4.3.1 Browsing the Library

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

4.3.1.1 Viewing and Editing Objects

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

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

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

Table 4.1: Navigator Motion Commands

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

4.3.1.2 Searching for Objects

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

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

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

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

```
- TERM: Navigator
Hide*  Backward*  Forward*  Global*  Tree*  List*
PreviousPattern*  NextPattern*  Reset*  Cancel*
(* jumps to next matching name in library *)
Search Forward : Pattern" [pattern]
   Searching entire lib.
   !Void()
----------------------------------------
MkTHY*  OpenThy*  CloseThy*  ExportThy*  ChkThy*  ChkAllThys*  ChkOpenThy*
CheckMinTHY*  MinTHY*  EphTHY*  ExTHY*

Mill*  ObidCollector*  NameSearch*  PathStack*  RaiseTopLoops*
PrintObjTerm*  PrintObj*  MkThyDocObj*  ProofHelp*  ProofStats*  showRefEnvs*  FixRefEnvs*
CpObj*  reNameObj*  EditProperty*  SaveObj*  RmLink*  MkLink*  RmGroup*

ShowRefenv*  SetRefenvSibling*  SetRefenvUsing*  SetRefenv*  ProveRR*  SetInOBJ*
MkTHM*  MkML*  AddDef*  AddRecDef*  AddRecMod*  AddDefDisp*  AbReduce*  NavAtAp*
Act*  DeAct*  MkThyDir*  RmThyObj*  MvThyObj*

↑↑↑↑  ↑↑↑  ↑↑   ↑  ←  <>
↓↓↓↓  ↓↓↓  ↓↓   ↓  →  ><

Navigator: [num_thy_1; standard; theories]

Scroll position : 0

List Scroll : Total 159,  Point 0,  Visible : 01
 ---------------------------------------------
 -> CODE  TTF  RE_init_num_thy_1
    COM   TTF  num_thy_1_begin
    COM   TTF  num_thy_1_summary
    COM   TTF  num_thy_1_intro
    DISP  TTF  divides_df
    ABS   TTF  divides
    STM   TTF  divides_wf
    STM   TTF  comb_for_divides_wf
    STM   TTF  zero_divs_only_zero
    STM   TTF  one_divs_any
 ---------------------------------------------
```

```
- TERM: Navigator
Hide*  Backward*  Forward*  Global*  Tree*  List*
PreviousPattern*  NextPattern*  Reset*  Cancel*
(* jumps to next matching name in library *)
Search Forward : Pattern" divides$|
   Searching entire lib.
   4218   divides
----------------------------------------
MkTHY*  OpenThy*  CloseThy*  ExportThy*  ChkThy*  ChkAllThys*  ChkOpenThy*
CheckMinTHY*  MinTHY*  EphTHY*  ExTHY*

Mill*  ObidCollector*  NameSearch*  PathStack*  RaiseTopLoops*
PrintObjTerm*  PrintObj*  MkThyDocObj*  ProofHelp*  ProofStats*  showRefEnvs*  FixRefEnvs*
CpObj*  reNameObj*  EditProperty*  SaveObj*  RmLink*  MkLink*  RmGroup*

ShowRefenv*  SetRefenvSibling*  SetRefenvUsing*  SetRefenv*  ProveRR*  SetInOBJ*
MkTHM*  MkML*  AddDef*  AddRecDef*  AddRecMod*  AddDefDisp*  AbReduce*  NavAtAp*
Act*  DeAct*  MkThyDir*  RmThyObj*  MvThyObj*

↑↑↑↑  ↑↑↑  ↑↑   ↑  ←  <>
↓↓↓↓  ↓↓↓  ↓↓   ↓  →  ><

Navigator: [num_thy_1; standard; theories]

Scroll position : 5

List Scroll : Total 159,  Point 5,  Visible : 10
 ---------------------------------------------
    CODE  TTF  RE_init_num_thy_1
    COM   TTF  num_thy_1_begin
    COM   TTF  num_thy_1_summary
    COM   TTF  num_thy_1_intro
    DISP  TTF  divides_df
 -> ABS   TTF  divides
    STM   TTF  divides_wf
    STM   TTF  comb_for_divides_wf
    STM   TTF  zero_divs_only_zero
    STM   TTF  one_divs_any
 ---------------------------------------------
```

Figure 4.2: Pattern-based name search

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

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

4.3.1.3 Advanced Motion: Using Path Stacks

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

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

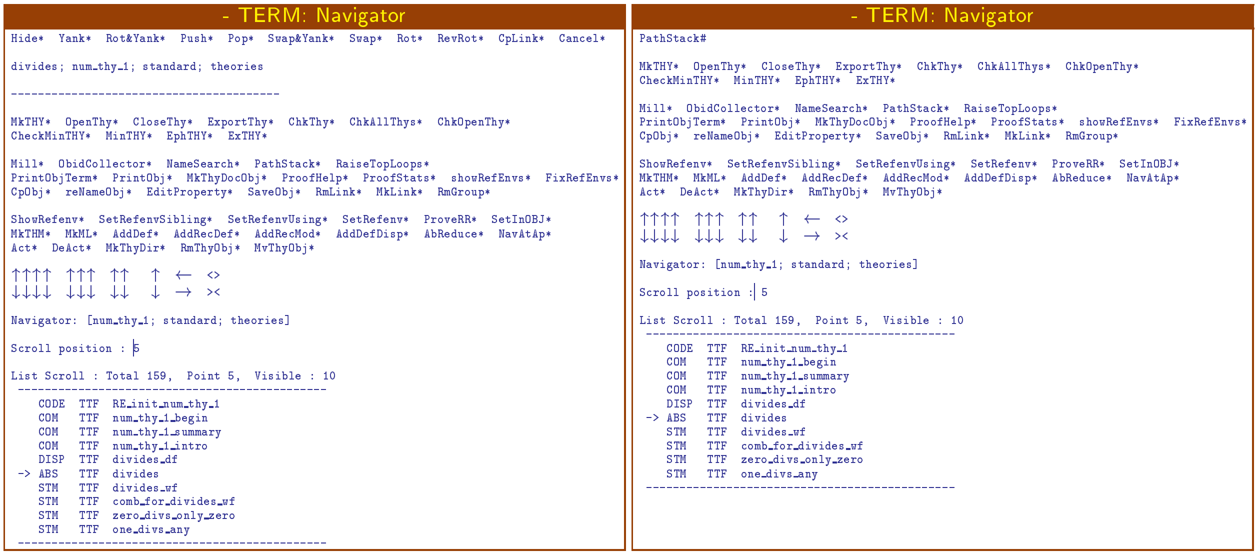


Figure 4.3: Path stack command zone

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

4.3.2 Operations on objects

4.3.2.1 Creating Objects

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

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

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

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

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

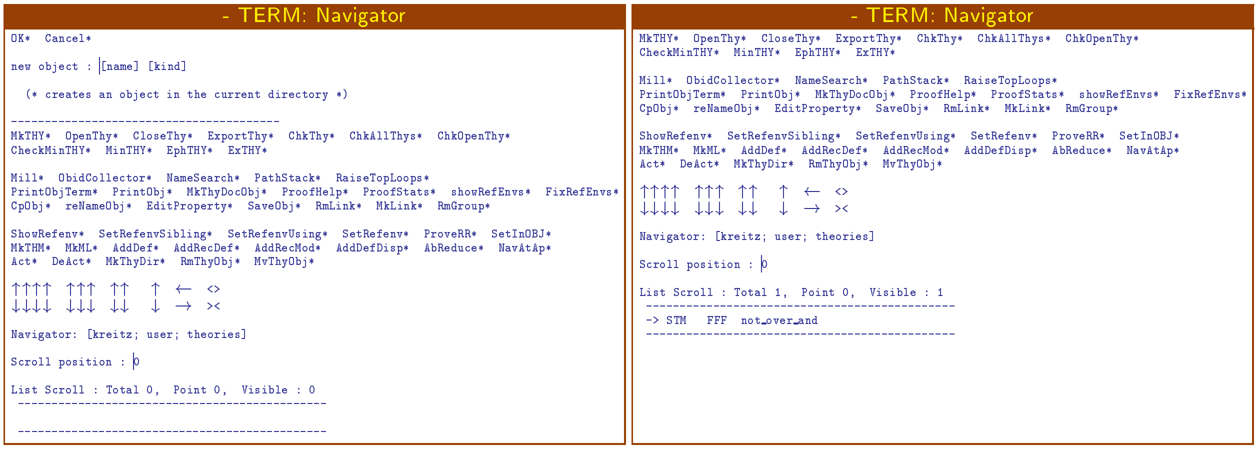


Figure 4.4: Creating Objects: Initial template and resulting update to the library

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

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

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

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

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

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

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

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

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

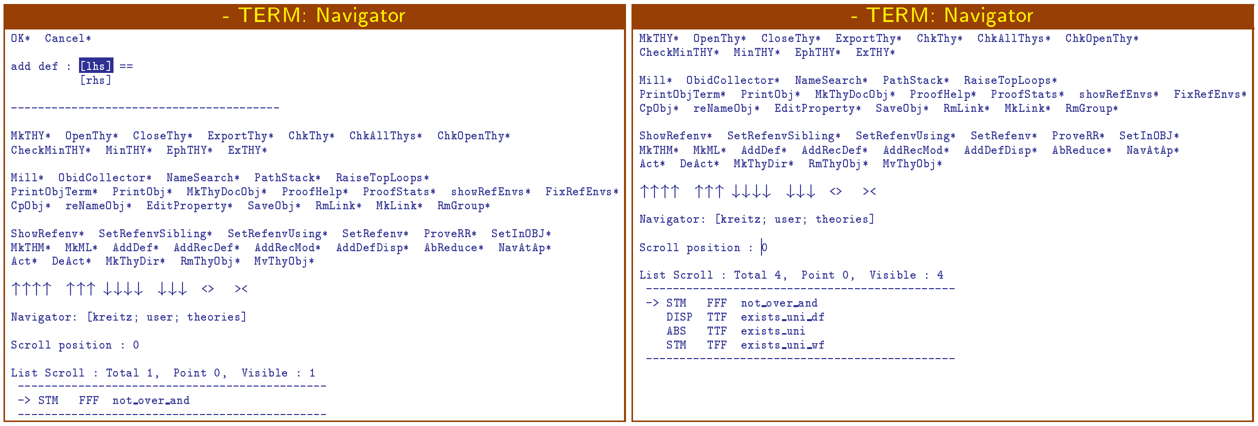


Figure 4.5: Creating Definitions: Initial template and resulting update to the library

4.3.2.2 Creating Definitions

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

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

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

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

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

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

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

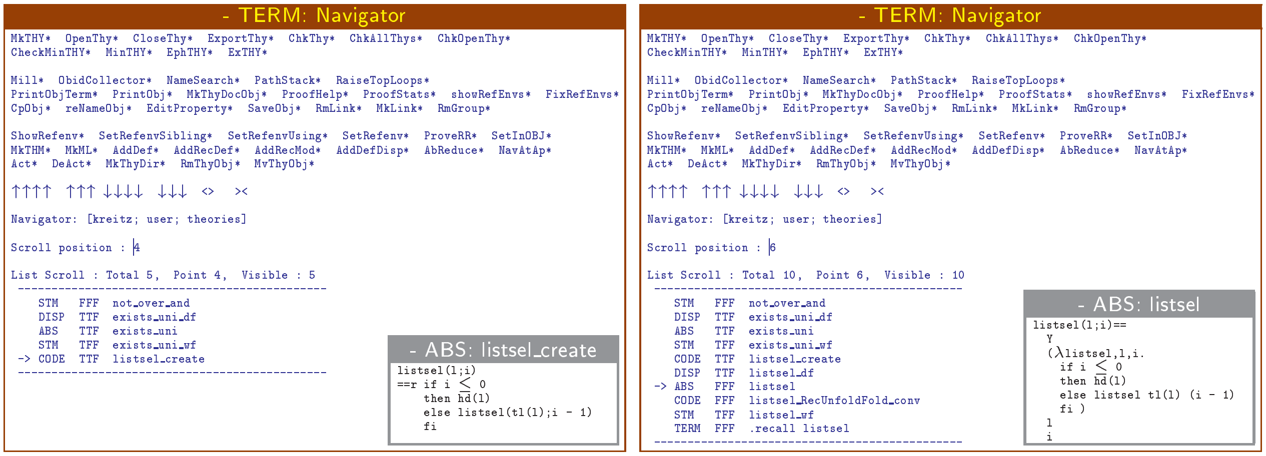


Figure 4.6: Creating Recursive Definitions: code object and created definition objects

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

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

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

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

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

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

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

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

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

Recursive definitions can also be created with the command
⌞`add_rec_def_at` *lhs rhs* `(inr(`*directory, position*`))`⌟,
where *lhs* is the left hand side of the definition, *rhs* its right hand side, *directory* the object identifier of the directory in which the new object shall be placed, and *position* the name of the object after which the definition shall be inserted. Thus, to create the above five objects within the ML top loop one could type

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

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

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

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

4.3.2.3 Creating Modules

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

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

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

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

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

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

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

4.3.2.4 Copying Objects

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

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

4.3.2.5 Links

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

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

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

4.3.2.6 Removing Objects and Links

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

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

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

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

4.3.2.7 Moving Objects

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

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

4.3.2.8 Renaming Objects

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

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

4.3.2.9 Activating and Deactivating Objects

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

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

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

4.3.2.10 Editing Object Properties

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

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

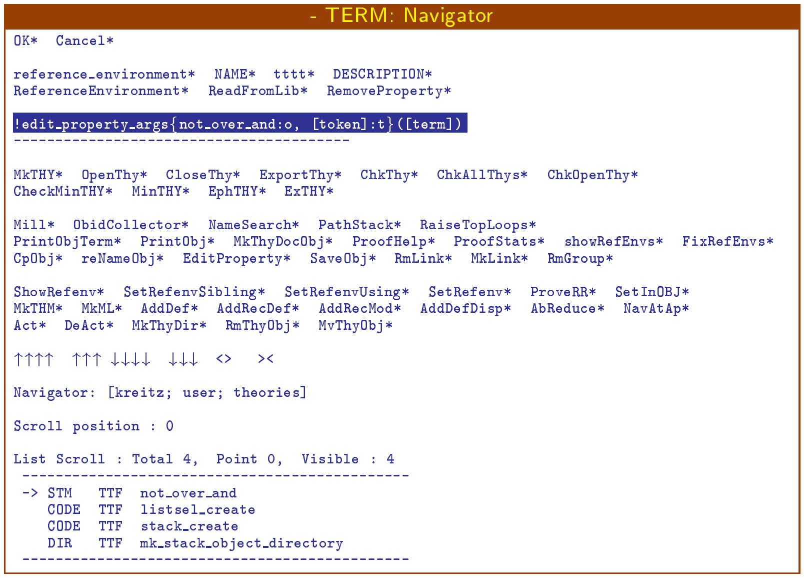


Figure 4.8: Editing Object Properties

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

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

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

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

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

4.3.2.11 Saving Objects

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

4.3.2.12 Printing Objects

Often users like to create print representations of objects in order to document their formal theories on paper or on the web. Clicking the `PrintObj*` command button will create a print representation of the object at the navigation pointer and write it into a file `~/nuprlprint/`*name*`_obj.prl`. This file can be inspected with any 8bit capable editor that has the NUPRL fonts loaded. It will also create a LaTeX-version and write it to `~/nuprlprint/`*name*`_obj.tex`. The directory `~/nuprlprint` must already exist. Otherwise clicking the `PrintObj*` button will result in an error. The same effect can be achieved by typing the command ⌞`print_an_object` *object*⌟ into the editor ML top loop.

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

4.3.2.13 Commenting Objects

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

Figure 4.9: Commenting an object

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

4.3.2.14 Proof Help

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

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

4.3.3 Theory Operations

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

4.3.3.1 Object Dependencies and Reference Environments

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

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

The specification of a reference environment consists of:

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

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

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

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

  minimal: minimal relative to the empty environment.

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

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

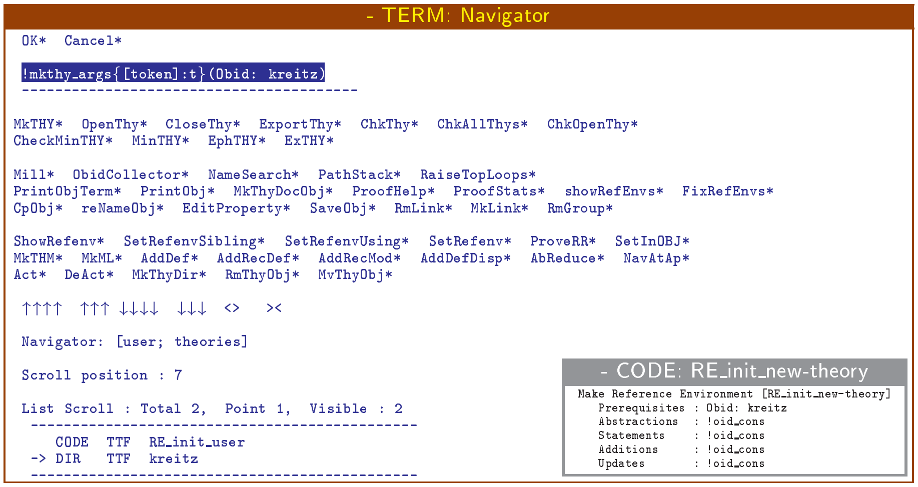


Figure 4.10: Creating Theories: initial template and generated static reference environment

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

4.3.3.2 Creating Theories and Sub-Theories

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

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

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

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

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

4.3.3.3 Changing Theory Modes

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

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

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

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

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

The above commands can also be executed by typing
⌞`close_theory` *theory*⌟,
⌞`open_theory` *theory*⌟,
⌞`set_theory_relative_minimal` *theory*⌟,
⌞`set_theory_ephemeral` *theory*⌟, or
⌞`set_theory_explicit` *theory*⌟
into the *library* ML top loop, where *theory* is the object identifier of the current theory.

4.3.3.4 Examining and Modifying Reference Environments

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

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

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

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

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

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

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

4.3.3.5 The Reference Environment Register

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

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

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

The above commands can also be executed by typing

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

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

4.3.3.6 Checking Theories

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

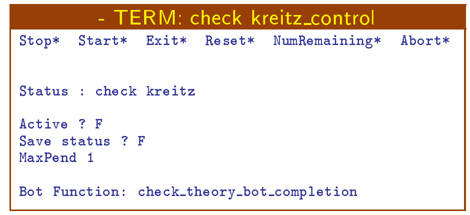


Figure 4.11: Checking a theory

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

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

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

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

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

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

4.3.3.7 Exporting and Importing Theories

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

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

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

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

4.3.3.8 Printing Theories

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

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

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

4.3.3.9 Creating Theory Documentation

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

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

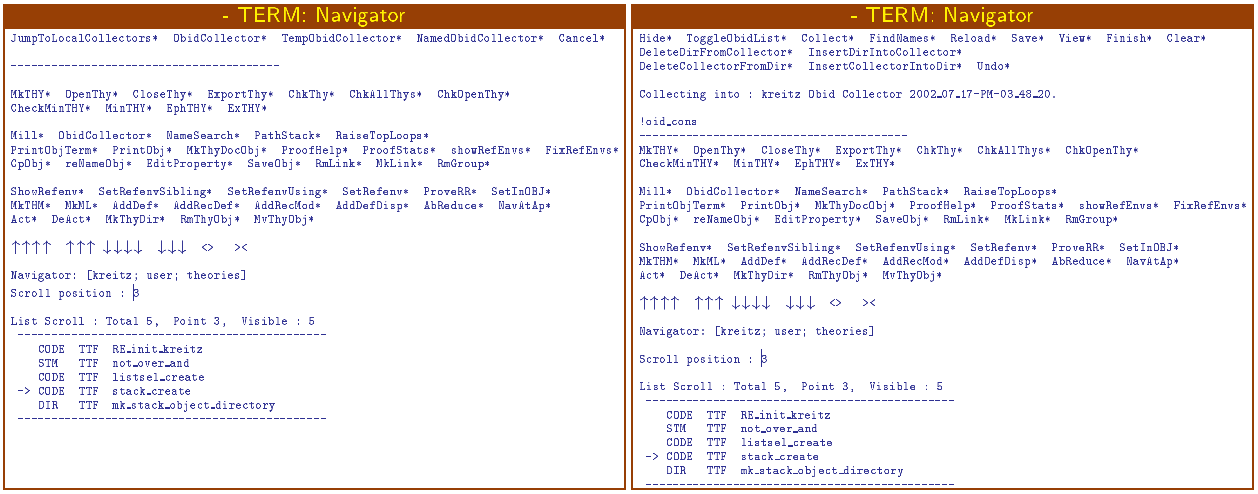


Figure 4.12: Creating Object Collections

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

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

4.3.4 Miscellaneous Operations

4.3.4.1 Creating Object Collections

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

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

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

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

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

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

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

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

Printing Object Collections

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

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

4.3.4.2 Milling

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

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

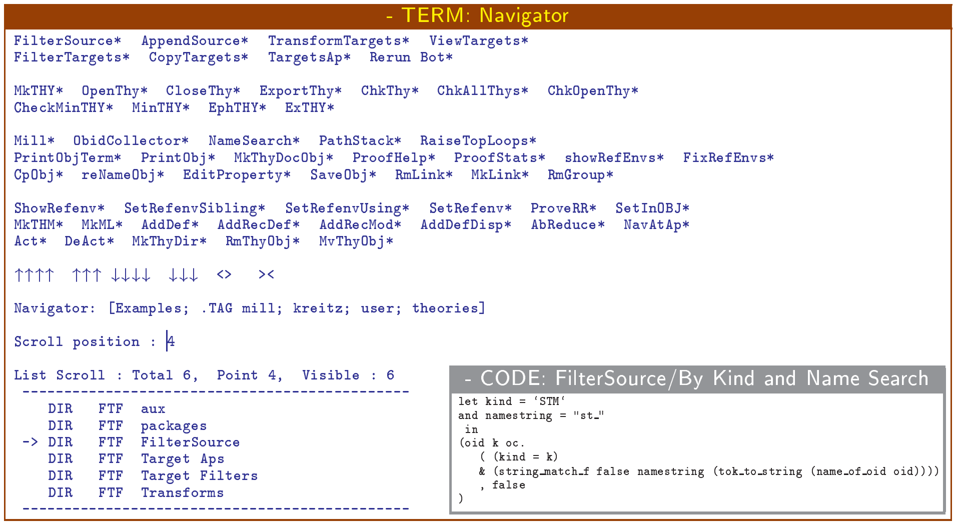


Figure 4.13: Standard Milling Directory

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

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

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

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

4.3.4.3 NavAtAp

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

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

4.3.4.4 Cloning the Navigator

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

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

4.3.4.5 Raising the ML top loop window

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

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

4.4 The ML Top Loop

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

4.4.1 Top loop command buttons

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

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

Table 4.2: Navigator command buttons

```
TERM: ML TopLoop

Previous*  Next*  Eval*  Reset*  Remove*  SaveWOEval*
LIB*  EDD*  REF*  ShowRefenv*  RaiseHistory*  RaiseNavigator*

M[REF]> FwdThruLemma|  ;;
```

```
TERM: EvaluatorHistory

Previous*  Next*  RaiseEvaluator*

M[REF]> FwdThruLemma ;;

    - : (tok -> int list -> tactic)
```

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

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

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

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

Reset*: Reset the command line zone.

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

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

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

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

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

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

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

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

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

4.4.2 The command line zone editor

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

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

Table 4.3: Command line zone editor commands and bindings

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

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

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

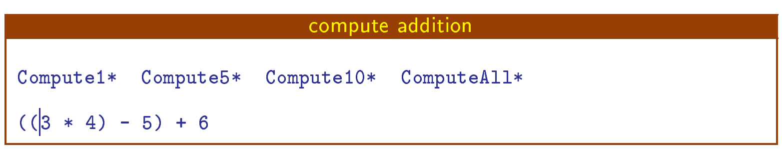
4.4.3 Top Loop Commands

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

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

4.4.3.1 Invoking the Nuprl term evaluator

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



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

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

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

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

4.4.3.2 Loading and compiling ML code

NUPRL allows users to load files containing ML code into the library, editor, or refiner processes. To do so, a user has to type the command ⌞`loadt_system` *root-dir* `[`*path*$_1$*, files*$_1$ `; ...;` *path*$_n$*, files*$_n$`]`⌟ into the respective ML top loop. *root-dir* is a string describing the root directory of the user's NUPRL files. *path* is a list of strings describing the sub-directory in which the specific files reside, and *files* is a list of file names (without the `.ml` extension) in that directory. The command will compile all the named files and load the compiled code into the current process. For instance,

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

will compile and load files in the following order

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

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

4.4.3.3 Importing Text

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

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

4.5 Process Top Loops

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

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

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

- Editor Commands:
  - ⌞`nuprl_oed_suspend ();;`⌟ closes all NUPRL windows.
  - ⌞`nuprl_oed_resume ();;`⌟ reopens the NUPRL windows.
  - ⌞`nuprl_oed_reset ();;`⌟ : kills all NUPRL windows.
  - ⌞`win.`⌟ opens the navigator and ML top loop windows.
  - ⌞`set_xhost` "*hostname*" *display_index*`;;`⌟ redirects all NUPRL windows to the specified display after the next suspend/resume cycle. *hostname* must be a string describing the display host and *display_index* a number (usually 0) specifying the display terminal on that host.
  - ⌞`nuprl_oed_rehash ();;`⌟ rehashes the macros and bindings in the user's `mykeys.macro` file (see Section 3.2.2).
- Commands for the Editor or Refiner:
  - ⌞`setup_connect` *socket*$_1$ *socket*$_2$ "*hostname*"`;;`⌟ sets up a connection to the library at host *hostname* using the indicated socket numbers.
  - ⌞`dc ();;`⌟ try to establish the connection that was set up.
  - ⌞`dd ();;`⌟ disconnect the process.
  - ⌞`open_lib` ‘*lib-memnonic*‘`;;`⌟ opens the connection to the library environment called *lib-memnonic*.
  - ⌞`close_lib` ‘*lib-memnonic*‘`;;`⌟ closes the connection to the library environment *lib-memnonic*.

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

  The above commands are implicitly executed when the editor and refiner processes are started, using the data contained in the user's `.nuprl.config` file (see Section 3.2.2).
- Library commands:
  - ⌞`nosa` *socket1*`;;`⌟ Opens a connection to a client using the indicated socket number.

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

Library Lisp commands:

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

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

4.6 Recovering from Errors

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

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

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

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

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

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

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

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

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

Chapter 5

Editing Terms

Terms in NUPRL are a general-purpose, uniform data-structure that serves two different purposes.

- Terms are used to represent NUPRL's *object logic*, that is the expressions and types of its type theory together with user-defined extensions as well as all NUPRL propositions. Sometimes we refer to terms of this kind as *object-language terms*.
- Nearly all library objects, that is proofs, abstractions, display forms, and even the descriptions of the NUPRL editors are represented as terms to which we refer as *system-language terms*.

Terms are either *primitive* or *abstract*. Primitive terms have fixed pre-defined meanings and are used to describe the primitives of NUPRL's type theory as well as the foundation of the NUPRL system. Abstract terms or *abstractions* (see Chapter 7.1) are defined as being equal to other, more primitive terms. The definitions of abstractions are stored in abstraction objects of NUPRL's library (see Section 4.1.1). The visual appearance of a term is governed by its *display forms* (see Chapter 7.2), which are defined in the display-form objects of NUPRL's library.

This chapter describes the internal structure of terms, the interplay between NUPRL's structured editor and display forms, and the commands for interactively viewing and editing terms in NUPRL.

5.1 Uniform Term Structure

NUPRL terms have the form $\boldsymbol{opid}\{p_1{:}F_1,\ldots,p_k{:}F_k\}(x_1^1,\ldots,x_{m_1}^1.t_1;\ \ldots;\ x_1^n,\ldots,x_{m_n}^n.t_n)$, where $\boldsymbol{opid}$ is the *operator identifier*. The $p_i{:}F_i$ are *parameters* and the $x_1^i,\ldots,x_{m_i}^i.t_i$ are the *bound subterms*, expressing that the variables $x_1^i,..,x_{m_i}^i$ become bound in the term t_i. The tuple $(m_1, \ldots, m_n)$, where $m_j \geq 0$ is the *arity* of the term. Appendix A.1 describes the current parameter types and the acceptable strings for opids, parameters, and variables. The primitive operators of NUPRL's type theory are listed in Table A.1 on page 148.

When writing terms, we sometimes omit the {} brackets around the parameter list if it is empty. Note that parameters are separated by commas while subterms are separated by semicolons.

Terms are implemented as tree data structures in NUPRL's meta-language ML (see Appendices B). Most users will rarely have to work with terms on the ML level but use the term editor to view and edit terms.

5.2 Structured Editing

In mathematics one distinguishes between logical structure of objects, and the notation in which they are presented. When we talk of the *logical structure* of a term, we are thinking of the abstract mathematical object that it represents. NUPRL's *uniform term syntax*, described in the previous section, is meant to reflect the regularity of these objects.

In contrast to that, *notation* aims at a visual presentation of abstract objects on the printed page or on the computer screen. Familiar notation for mathematical expressions helps human readers to construct the described abstract object in their minds, but should not be confused with the abstract structure itself.

In mathematics notation is crucial issue. Many mathematical developments depend heavily on the adoption of some clear notation, which makes mathematics much easier to read. However, mathematical notation can be rather complex and ambiguous if one is not aware of the immediate context it is presented in. Juxtaposition of symbols, for instance, can mean function application in one place and multiplication in another.

Notation understandable by machines, on the other hand, is often restricted to source texts in ASCII files that can be easily be parsed by a computer. Programming languages allow *overloading* of operators and *implicit coercions* and resolve these ambiguities by type-checking and similar methods. For interactive theorem proving, however, parseable ASCII notation is far from being sufficient, as it is not anywhere as easy to read as mathematics in books and papers.

The NUPRL system provides an interactive editing mechanism that presents terms in mathematical notation and groups the notation in chunks that correspond to parts of the internal tree structures. Users edit the tree structure directly, so there is no need for a parser.[1] Such editors are often called *structured editors*. The advantages of structured editors are:

- Structured editors allow using a notation that is ambiguous to a machine but unambiguous to a human who is aware of its full context.
- Formal notation is not limited to ASCII characters. NUPRL uses a single 8-bit font of up to 256 characters for displaying formal text on the screen while it is being edited and provides mechanisms for integrating LaTeX and Display PostScript-like technology to generate almost text-book quality displays. Currently the latter is only used for printing formal mathematical text but not for editing it.
- Formal notation may become context dependent. Theorems, definitions and proofs may contain local abbreviations and implicit information.
- Notation can be freely changed without altering the underlying logical structure of terms.
- Structured editors link abstract terms to notation. Users who find a particular notation confusing only need to point and click the mouse on the notation in question in order to receive a formal definition and possibly additional explanations.

The main disadvantage of structured editors is that they are quite different from conventional text editors like Emacs. Users have to get used to entering and editing terms in accordance with their tree structure instead of their textual appearance. They also have less control over formatting, since all display formatting is done automatically and may only be influenced by display forms (see Chapter 7.2). It may take a while to learn how to use a structured editor and to take advantage of its capabilities. Suggestions for improvements from users of the NUPRL editor are always welcome.

5.2.1 Term Display

To associate a chunk of notation with a term, NUPRL's term editor relies on *display form definitions*. Binary addition, for instance, is represented by the term `add`$(x;y)$, but conventionally written with

[1] An alternative approach, taken for instance by MATHEMATICA [Wol88], is to use a rich parseable ASCII syntax for input and then to process the input with pretty-printing routines for formatted output like Display PostScript.

the symbol + in infix notation. We could write the notation chunk as $\boxed{\square \texttt{ + } \square}$, where the $\square$'s are holes for the two subterms, and the outer box shows the extent of the chunk. We call these holes *term slots*, because they can be filled by terms. In a display form definition we usually label term slots with variables to indicate how term slots correspond to the logical structure of a term. The display form for addition, for instance, can be defined as

$$\boxed{\boxed{x} \texttt{ + } \boxed{y}} \quad \equiv_{\text{df}} \quad \texttt{add}(x\texttt{; } y)$$

where $a \; =_{\text{dform}} b$ means that a is defined as the *display form* for b. Term slots stretch to accommodate the terms inserted in them. For instance, the term `add(mul(1;2);3)` will be shown as

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

NUPRL automatically adds parentheses according to display form *precedences*. When a display form of lower precedence is inserted into the slot of display form with higher precedence, parentheses are automatically inserted to delimit the slot. For instance, if we assign the display form for `mul(`x`;`y a higher precedence than the one for `add(`x`;`y`)` then the term `add(mul(1;2);3)` is displayed without parentheses, while `mul(1;add(2;3))` is displayed as

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

Note that parentheses are inserted merely to disambiguate the notation for a human reader, who cannot see the term slots but only ⌞`1 * (2 + 3)`⌟.

In addition to term slots and the fixed components of a notation chunk, some display forms contain *text slots*. These are slots in a definition that are filled with text strings like values for term parameters and names of binding variables. A universally quantified formula, for instance, is represented by the term ⌞`all(`T`;`x`.`P`)`⌟, meaning that "for all x of type T, the proposition P is true" (note that free occurrences of the variable x become bound in P). To generate the usual notation ⌞$\forall x\!:\!T.\ P$⌟ for this formula, we use the display form definition

$$\boxed{\forall\, ⌜x⌟ : \boxed{T}.\ \boxed{P}} \quad \equiv_{\text{df}} \quad \texttt{all}(T\texttt{; } x\texttt{.}P)$$

where ⌜ ⌟ is used to indicate a text slot.

Two display form definitions in NUPRL are rather special: the display form definition for variable terms, and the display form definition for natural numbers. Both display forms have only a single text slot, and no other printing or whitespace characters.

$$\boxed{⌜x⌟} \quad \equiv_{\text{df}} \quad \texttt{variable}\{x\texttt{:v}\}$$

$$\boxed{⌜i⌟} \quad \equiv_{\text{df}} \quad \texttt{natural\_number}\{i\texttt{:n}\}$$

In general, a display form for a term is made up of text and term slots, interspersed with printing and space characters. We can annotate display forms with *formatting commands* that specify where line breaks can be inserted, and how to control indentation. The structure and appearance of display form definitions, which are contained in *display* objects in NUPRL theories, as well as the effect of precedences on parenthesization is described in detail in Chapter 7.2.

NUPRL's term editor builds the notation for a term from the display forms associated with each node of the term tree. Thus the structure of the notation mirrors the tree structure of the term. The *display form tree* of a term is the tree structure that one actually edits with NUPRL's term editor. In a display form tree, each display form and each slot is considered a node of the tree. If a slot is not empty, it is identified with the display form tree or the text string filling it. All the slots of a display form are considered to be the immediate children of the display form and the editor considers slots ordered in the order they appear, left to right, in display form definitions.

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

5.2.2 Editor Modes

Users can navigate through a term by moving a cursor (sometimes called the *point*). Depending on the position of the cursor, the term editor will be in one of three modes, which are indicated by the shape of the cursor.

In *term mode* the cursor is positioned at some node of the term tree. The term node is indicated by highlighting its notation and the notation for all its subtrees. The highlighting is usually achieved by using reverse video; swapping foreground and background colors. For example,

    ∀i:ℤ. ∃j:ℤ. (j = (i + 1))

indicates that a *term cursor* is at the subterm ⌞`j = i + 1`⌟. Occasionally a term has no width, and a term cursor on such a term is displayed as a thin vertical line. In this document, we indicate such a cursor by ‖. In term mode, keystrokes corresponding to printing characters form (parts of) editor commands.

In *text mode* the cursor is positioned within a text slot and displayed as ⌞|⌟. Keystrokes corresponding to printing characters cause those characters to be inserted at the position of the cursor. The *text cursor* is significantly thinner than the term cursor on a no-width term, so it should be easy to distinguish the two. It may be positioned either between two adjacent characters, before the first character of a text slot, or after the last. Valid text cursors for the text string ⌞`abcdef`⌟, for instance include

    |abcdef   abc|def   abcdef|

There is a potential ambiguity as to *which* text slot a text cursor is at. Consider, for instance, two adjacent text slots containing the strings ⌞`aaa`⌟ and ⌞`zzz`⌟ and the following text cursor:

    aaa|zzz

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

Certain cursor motion commands are designed for moving around a term's display character-by-character as with a conventional text editor. In this case the cursor occupies a single character position on the screen. If possible, the editor uses a text cursor. Otherwise it uses a *screen cursor*, which is displayed by outlining the character. For instance, if we had the following text cursor in a term:

    ∀|i:ℤ. ∃j:ℤ. (j = (i + 1))

then a 'move-left-one-character' command would leave a screen cursor over the character ⌞∀⌟.

    ∀i:ℤ. ∃j:ℤ. (j = (i + 1))

In screen mode, keystrokes corresponding to printing characters form parts of editor commands. In the rest of this document we will never have to explicitly represent a screen cursor, so all outlined terms should be interpreted as term cursors.

5.2.3 Term and Text Sequences

The term editor has special features for handling certain kinds of sequences of terms, which makes them appear much like terms with variable numbers of subterms. A *term sequence* is constructed by iteratively pairing term slots in a right-associative way and displayed as a linear sequence. A sequence containing 4 empty term slots, for instance, might be displayed as:

```
(□, □, □, □)
```

Different kinds of term sequences have different pairing terms, a special term to represent the empty sequence, different left and right delimiters, (the ⌞`(`⌟ and ⌞`)`⌟ respectively in the example), and different element separators (the ⌞`,`⌟ in the example). Delimiters and separators in term sequences always consist of at least one character.

The editor considers all the term slots of the sequence as siblings in the display form tree, and the whole sequence as their immediate parent. Often, the editor does not distinguish between a term and a one-element sequence containing that term but treats a term as a one element sequence. Thus for nearly all purposes the internal structure of sequences can safely be ignored.

A *text sequence* is a text string in which characters may be replaced by terms. Text sequences are primarily used for proof tactics and other ML code, for comments, and for the left-hand sides of display forms. The editor presents a text sequence as a display form with alternating text and term slots. In contrast to term sequences, text sequences normally have no delimiters or element separators. They are however easily identified because they usually occur in well-defined contexts. An example of a text sequence is the ML expression:

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

This text sequence consists of 3 term slots filled with the terms ⌞`'n + 1'`⌟, ⌞◊⌟, and ⌞◊⌟, and 4 text slots filled with the text strings ⌞`With `⌟, ⌞` (D 0)`⌟, ⌞`THENW TypeCheck`⌟, and ⌞⌟ (the null or empty text string). The ⌞◊⌟'s are new-line terms, which are usually kept invisible and only shown with a printing character for illustration purposes. New-line characters in text should be avoided as much as possible, as this simplifies the display formatting algorithm.

5.3 Term Editor Windows

Term editor windows are used for viewing and editing terms. Except for the navigator and proof editor windows, most windows opened by NUPRL are term editor windows. Each window displays a single display form tree representing a single term. All editing operations can be carried out from the keyboard alone, but the editor accepts input from the keypad and the mouse as well.

Input characters typed at the keyboard in multi-character commands are echoed as highlighted text near the position of the cursor, and can be corrected by using DEL. Certain key combinations are bound to editor commands, which also may be invoked explicitly by typing ⌞⟨C-M-x⟩*command-name*⌟. The default key bindings are intended to be reminiscent of Emacs's key bindings. A summary of all the key bindings that we will describe below can be found in Table 5.8 at the end of this chapter. Users who wish to use alternative key bindings may customize the term editor as described in Section 5.8.

The editor adjusts the display of an object in a window to the size of the window. If the window is too small, not all the object can be displayed at once. In this event, one can resize the window, or scroll the window up and down. Sometimes, if the window is to narrow, some subterms are *elided*.

The display form tree for an elided subterm is replaced by ⌞...⌟. Currently, the only way to to un-elide a subterm is to widen the window as much as possible. Eventually, one will be able to examine elided subterms by moving the root display form of an editor window to some term tree position other than the term root.

Term editor windows are opened when a user access a library object through the navigator. Opening a proof object opens a *proof editor window*, which in turn opens a term slot for entering the goal sequence and text slots for entering refinement rules (see Chapter 6 for details). To close and change term editor windows, one may use the following commands and key sequences.

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

EXIT first saves a copy of the object. It then checks the object before closing the window. This *checking* has the same effect on library objects as using the ML `check` command. If the check fails, then the window is left open. If you still want to close the window, use QUIT instead. Separate save and check commands are described in Section 5.7.

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

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

5.4 Entering Information

Term editor windows for ML and comment objects initially open in text mode while abstractions and display forms usually open in term mode.[2] Special display forms allow opening text slots in term windows while special key sequences are used to open term slots within text sequences (see Section 5.4.2 below).

5.4.1 Inserting Text

The following commands are for inserting text whenever the editor is in text mode.

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

Standard ASCII printing characters (including space) self insert in text mode. Non-standard characters can be inserted using INSERT-SPEC-CHAR-*num*, where *num* is the decimal code for the character. Table 5.1 lists the currently available special character codes.[3] Alternatively, special characters can be copied from the object `FontTest` in the library theory `core-1`.

The INSERT-NEWLINE command is only appropriate in text sequences. Within terms the newline character is actually a term whose display is controlled by the display layout algorithm.

[2]Currently, newly created objects contain an empty term slot. Removing this slot in ML and comment objects with ⟨C-C⟩ puts the editor into text mode. The term slot in abstractions and display forms cannot be removed.

[3]The NUPRL fonts may be extended in the future. Executing the UNIX command `xfd -font nuprl-13 &` pops up a display of the actual standard NUPRL font. Clicking LEFT on a character results in its decimal code being displayed.

	0	1	2	3	4	5	6	7	8	9
120									$\mathbb{P}$	$\mathbb{R}$
130	$\mathbb{N}$	$\mathbb{C}$	$\mathbb{Q}$	$\mathbb{Z}$	$\mathbb{U}$	$\Leftarrow$		$\Rightarrow$	$\lceil$	$\rceil$
140	$\vdash$	$\int$	$\cdot$	$\downarrow$	α	β	$\wedge$	$\neg$	$\in$	π
150	λ	γ	δ	$\uparrow$	$\pm$	$\oplus$	∞	∂	$\subset$	$\supset$
160	$\cap$	$\cup$	$\forall$	$\exists$	$\otimes$	$\leftrightarrow$	$\leftarrow$		$\neq$	$\diamond$
170	$\leq$	$\geq$	$\equiv$	$\vee$		$\_$	$\rightarrow$	Σ	Δ	Π
180	$\times$	$\div$	$^{+}$	$^{-}$	^{0}	$\subseteq$	$\supseteq$	$_{0}$	$_{1}$	$_{2}$
190	$_{3}$	$\circ$	$\mathbb{B}$	ρ	$_{a}$	$_{q}$	$_{b}$	$_{d}$	$_{c}$	$\vert$
200					^{1}	^{2}	^{3}	μ	ϵ	θ
210	$\bigcap$	$\bigcup$	$\hat{=}$	$\doteq$	$\Leftrightarrow$	$\not\rightarrow$	$\sqsubseteq$		$\top$	$\bot$
220			$\mapsto$		ä	ö	ü	ß	Ä	Ö
230	Ü	T	F			$\emptyset$	$\notin$	$\not\subseteq$		$\not\sqsubseteq$
240	Γ	Λ	$\backslash$	Θ	σ				$_{i}$	$_{j}$
250	$_{k}$	$_{l}$	$_{m}$	$_{n}$						

Table 5.1: NUPRL special character codes

5.4.2 Adding and Removing Slots

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

NUPRL's term editor makes it possible to combine informal, semi-formal, and formal knowledge by inserting terms into text sequences. These terms are displayed according to their display forms and surrounded by ordinary text.

To make this possible, a term slot must be opened and then initialized. The latter inserts a *term holder* into the term slot, which initially looks like ⌞`'[term]'`⌟. The commands OPEN-LIST-LEFT-AND-INIT and OPEN-LIST-RIGHT-AND-INIT combine these two commands and position the cursor at the empty slot of the term holder. Terms may now be inserted into the slot as usual (see Section 5.4.3 below).

The commands for opening, initializing, and removing slots apply both in text and in term mode and thus have a slightly more general meaning than just described.

In term mode, OPEN-LIST-TO-LEFT and OPEN-LIST-TO-RIGHT only apply if the term cursor is an element of a (term or text) sequence. They add a new empty slot to the left (or right) of the cursor and move the cursor to the new empty slot. On an empty term sequence, both commands have the same effect; they simply delete the nil sequence term. In text mode, both commands open up an empty term slot at the text cursor, and leave the cursor at the new slot.

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

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

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

5.4.3 Inserting Terms

In structured editing, one usually enters terms in a top-down fashion, starting with the root of the term tree and working on down to the leaves. This means that one has to work with *incomplete* terms. For example, at an intermediate stage of entering the term ⌞∀i:ℤ. ∃j:ℤ. j = i + 1⌟ one might be presented with the term:

∀i:ℤ. ∃|[var]:[type]. [prop]

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

name	INSERT-TERM-*name*	insert *name* into empty slot
⟨C-I⟩*name*	INSERT-TERM-LEFT-*name*	insert *name*, using existing term as left subterm
⟨M-I⟩*name*	INSERT-TERM-RIGHT-*name*	insert *name*, using existing term as right subterm
⟨C-S⟩*name*	SUBSTITUTE-TERM-*name*	replace existing term with *name*
⟨C-M-I⟩	INIT-TERM	initialize term slot
⟨C-M-S⟩	SELECT-DFORM-OPTION	selects display form variations

To insert terms into term slots one may use the editor commands listed above. In these commands, *name* is a string of characters naming a new term to be inserted. The interpreter for *name* strings checks each of the following conditions until it finds one which applies.[4]

1. *name* is an editor command enabled in a particular context.
2. *name* is an *alias* for some display form, defined in in the library object for that display form.
3. *name* is the name of a display form object. In this case it refers to the first display form defined in that object.
4. *name* is of the form *ni* where n is the name of a display form object and i is a natural number. *ni* refers to the i-th display form definition in the object named n. Definitions in objects are numbered starting from 1.

[4]Note that this order gives display form names and aliases preference over abstraction names. The operator identifier of a term can *not* be used to identify a term, if it is neither of these three. This is particularly important when referring to the elementary terms of type theory. To find out how to refer to a particular term, mark the term and enter ⟨C-X⟩`df` to see its display form or ⟨C-X⟩`ab` to see the abstraction that defines it, if there is one.

5. *name* is the name of an abstraction object. In this case it refers to the earliest display form in the library for that abstraction.
6. *name* is all numerals, then the term referred to is the term ⌞`natural-number`{*name*:`n`}`()`⌟ term of NUPRL's object language.
7. If none of the above applies, *name* is assumed to refer to the term ⌞`variable`{*name*:`v`}`()`⌟.

Since names always have acceptable extensions as variable names, the editor does not interpret *name* until some explicit terminator such as $\boxed{\text{SPC}}$ (NO-OP) or a cursor motion command is typed. ↵ (RIGHT-EMPTY-SLOT, see Section 5.5) is a particularly useful terminator.

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

INSERT-TERM-LEFT-*name* is intended for use at a filled term slot. Its behavior is to:
1. save the existing term in the slot, leaving the slot empty,
2. insert the new display form referred to by *name* into the slot,
3. paste the saved term into the leftmost term slot of the new display form. If the new display form has no term slots, then the saved term is lost.

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

SUBSTITUTE-TERM-*name* replaces one display form with another that has the same sequence of child text and term slots. The children of the old display form become the children of the new one. If the new display form has a different sequence of children SUBSTITUTE-TERM-*name* tries something sensible, but in these cases it is safer to explicitly cut and paste the children.

INIT-TERM initializes a term slot to some default term. INITIALIZE-TERM is automatically invoked by NUPRL to initialize new windows. To re-initialize a window, place a term cursor at the root of the term in the window, delete the term and then give the INITIALIZE-TERM command. The default terms for particular contexts are described in various sections of this document. If no default has been designated, INITIALIZE-TERM does nothing.

SELECT-DFORM-OPTION selects an alternative display form for the term where the term cursor is positioned. For instance, if term cursor is positioned at an independent function type, it selects the more general dependent-function display form.

5.4.4 Adding New Terms

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

- Position a term cursor at an empty slot and enter the letters of the new term's opid
- Enter a (possibly empty) list of parameter types for the new term (see Section A.1.2), abbreviated by single letters. The list has to be delimited by ⌞{⌟ and ⌞}⌟ characters, and elements must be separated by ⌞,⌟ characters. Empty lists of parameter types are optional.
- Enter a list of subterm arities, i.e. numbers designating the number of binding variables for each subterm. The list must be delimited by ⌞`(`⌟ and ⌞`)`⌟ characters, and elements must be separated by ⌞`;`⌟ characters. This list has to be entered even when it is empty.

Upon receiving the opid, the list of parameter types, and the list of subterm arities the editor creates a new term in uniform syntax with appropriate place-holders for parameters and subterms. For instance, entering `myid{n,t}(0;1)` creates the term

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

5.4.5 Exploded Terms

Exploded terms provide access to the internal structure of a term, allowing users to change its opid, the number and kind of its parameters, or its arity. They are most commonly used for creating new terms in an abstraction or for changing the definition of an abstraction.

A term constructor is *exploded* by replacing it by a special collection of terms that make it possible to edit its structure. Exploded terms may be generated from scratch by typing `exterm` into an empty term slot, or by positioning the term cursor over a term and typing ⟨C-X⟩`ex`. Exploded terms may be imploded again into the term which the exploded term represents by typing ⟨C-X⟩`im`. The commands for editing exploded terms are summarized in the table below.

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

The editor is somewhat intelligent when new slots with OPEN-LIST-TO-LEFT and OPEN-LIST-TO-RIGHT. Depending on the context, the new slot will be a placeholder for a bound term (`[bterm]`), bound variable (`[bvar]`), or a parameter (`[parm]`).

To show how exploded terms are used, we walk through the entry of the term ⌞`foo{bar:s}(A;x.B)`⌟. Create an empty term slot and enter ⌞`exterm`SPC⌟. The highlighted term should look like:

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

Enter the opid ⌞`foo`⌟ to get:

```
EXPLODED<<foo| {}([bterm])>>
```

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

```
EXPLODED<<foo {|}([bterm])>>
```

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

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

Insert the string parameter with text `bar` by typing ⌞`sparm`↵`bar`⌟:

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

Click LEFT on the `[bterm]` and enter ⟨C-O⟩⟨C-O⟩ to make a two element sequence for bound terms, leaving the cursor on the left-most element.

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

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

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

You could now go ahead and fill in the binding variable, and subterm slots by typing ⌞A ↵ x ↵ B ↵⌟.

EXPLODED<<foo {bar:s}(.A;x,.B;)>>

Finally, click LEFT on any part of EXPLODED and then enter ⌞⟨C-X⟩im⌟ to implode the exploded terms. You should now have the term:

foo{bar:s}(.A;x.B)

In general, when imploding and exploding terms the parameter values, binding variable names, and subterms stay the same, so entering and/or editing them when a term is exploded has the same effect as when the term is imploded.

5.5 Cursor and Window Motion

Nuprl's editor supports two basic forms of cursor motion. *Screen oriented* cursor motion commands ignore the structure of the term in the window and allow one to quickly navigate to parts of the screen. In contrast to that *tree oriented* cursor motion commands follow the structure of the term tree. In addition to key sequences *mouse commands* allow easy jumping around terms and *search commands* allow moving the cursor to a particular substring.

5.5.1 Screen Oriented Motion

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

Screen oriented cursor motion commands are listed in the table above. After a screen cursor command the cursor is always either in text mode or screen mode. To switch to term mode, one may use the SWITCH-TO-TERM command if the cursor is over the printing character of a display form. If the cursor is moved over the top or bottom of the display, the window scrolls appropriately. There are also explicit window scrolling commands.

5.5.2 Tree Oriented Motion

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

UP, LEFT, RIGHT, DOWN-LEFT, DOWN-RIGHT are the basic walking commands. These commands recognize text and term sequences, and skip over their internal structure. Within text slots, LEFT and RIGHT stop at each word.

RIGHT-LEAF, LEFT-LEAF, RIGHT-EMPTY-SLOT, LEFT-EMPTY-SLOT are particularly good for rapidly moving around terms, since you can often get where you want to go by just repeatedly using one of them. Note that ↵ is not bound to RIGHT-EMPTY-SLOT within text sequences. In that case, you need to use ⟨C- ↵ ⟩.

5.5.3 Mouse Commands

LEFT	MOUSE-MARK-THEN-SET-POINT	set mark then point
⟨C- LEFT ⟩	MOUSE-MARK-THEN-SET-POINT-TO-TERM	set mark then point to term

MOUSE-MARK-THEN-SET-POINT first sets the *mark*, an auxiliary cursor for marking regions (see Section 5.6.2) at the current position of the *editor cursor* and *then* sets the editor's cursor, the *point*, to where the mouse is pointing. MOUSE-SET-POINT results in a text cursor if one is valid between the character pointed to and the character to the immediate left. If there is a null width term to the immediate left of the mouse, it results in a term cursor pointing to that term. Otherwise, the editor cursor is set to the most smallest term that contains the character being pointed to. MOUSE-MARK-THEN-SET-POINT-TO-TERM is like MOUSE-MARK-THEN-SET-POINT except that point is always set to the term immediately surrounding the character being pointed to.

5.5.4 Search for Subterms

⟨M-s⟩	SET-SEARCH-MODE	initialize substring search
⟨C-s⟩	VIEW-SEARCH-FORWARDS	search forward
⟨C-r⟩	VIEW-SEARCH-BACKWARDS	search backward

SET-SEARCH-MODE initializes the search for a substring. It expects a substring to be be entered and then sets the cursor to the next text or term slot that contains this substring. Thus a user has to enter ⌞⟨M-s⟩ *substring* SPC⌟ to search for the first occurrence of *substring*. Currently, *substring* cannot contain whitespace.

As long as the editor is in search mode VIEW-SEARCH-FORWARDS move the cursor to the next slot containing *substring*. VIEW-SEARCH-BACKWARDS does the same moving backwards.

5.6 Cutting and Pasting

Cut-and-paste commands work on terms, segments of text slots, and segments of text and term sequences. In this section we refer to these collectively as *items*. Items can be saved on a *save stack*, in which they are represented as terms. Often it is possible to cut one kind of item and then paste it into another kind of context. For example, one can cut a term and paste into text sequence, or cut a segment of text from a text slot and paste into a term sequence. Within the context of NUPRL's editor, cut-and-paste commands have the following meaning:

SAVE: push a copy of an item onto the save stack, leaving the item in place. Similar to *copy-as-kill* in Emacs.

DELETE: remove an item from a buffer without saving it.

CUT: (= SAVE + DELETE) remove an item from a buffer and push it onto the top of the *save stack*. Similar to *kill* in Emacs, although NUPRL does *not* append together items that were cut immediately one after the other.

PASTE: insert the item on top of the stack back into a buffer, removing it from the stack. Successive pastes thus retrieve items that were saved earlier.

PASTE-COPY: insert the item on top of the stack back into a buffer without removing it from the stack. This is useful for making several copies of an item. Similar to *yank* in Emacs.

PASTE-NEXT: remove the item just pasted from the buffer and paste the item that is now on top of the stack. Can only be used immediately after a PASTE. By repeating PASTE-NEXT one may back through the save stack for some desired item. Similar to *yank-next* in Emacs.

5.6.1 Basic Commands

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

DELETE-CHAR-TO-LEFT and DELETE-CHAR-TO-RIGHT are conventional character deletion commands. They only remove the character without saving it on the save stack. They can be used in any text slot of a term or in a text sequence and also work on newline terms in text sequences.

CUT-WORD-TO-RIGHT cuts the word to the right of a text cursor. If a term is to the immediate right of a text cursor in a text sequence, then that term is cut. CUT, SAVE, and DELETE work on a term underneath a term cursor as described above. These commands work fine on terms in text and term sequences. Note that deleting a term leaves an empty term slot.

When a term cursor is at an empty term slot, the PASTE and PASTE-COPY commands paste the term on top of the stack into the slot. PASTE-NEXT replaces the last term pasted with the term on top of the paste stack. It should only be used immediately after a PASTE or a previous PASTE-NEXT.

5.6.2 Cutting and Pasting Regions

A *region* is a segment of any text slot, or a segment of a text or term sequence. A region is delimited by the editor's term or text cursor and an auxiliary text or term cursor position. Following Emacs's terminology, we call the cursor's position the *point* and the auxiliary cursor position the *mark*. It does not matter whether mark is to the left or the right of point when selecting a region. In what follows, we call the left-most of point and mark the *left delimiter*, and the right-most the *right delimiter*. If a term is used as a region delimiter, the term is included in the region.

Various regions are acceptable. For selecting a text string in a text slot, both delimiters must be text cursor positions. For selecting a segment of a term sequence, both delimiters must be term cursor positions. For selecting a segment of a text sequence, text or term cursor positions may be used for each delimiter. The commands for cutting and pasting regions are shown below.

⟨C- SPC⟩	SET-MARK	set mark at point
⟨C-X⟩⟨C-X⟩	SWAP-POINT-MARK	swap point and mark
⟨C-W⟩	CUT-REGION	cut region
⟨M-W⟩	SAVE-REGION	save of region
⟨C-M-W⟩	DELETE-REGION	delete region

SET-MARK sets the mark to the current cursor position while SWAP-POINT-MARK swaps the mark and the editor cursor. This command is often used to check the mark's position.

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

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

5.6.3 Mouse Commands

LEFT	MOUSE-MARK-THEN-SET-POINT	set mark then point
⟨C- LEFT⟩	MOUSE-MARK-THEN-SET-POINT-TO-TERM	set mark then point to term
⟨C- MIDDLE⟩	MOUSE-PASTE	paste region
⟨M- MIDDLE⟩	MOUSE-PASTE-NEXT	replace last paste with new paste
⟨C-M- MIDDLE⟩	MOUSE-PASTE-COPY	paste copy of region on stack
⟨C- RIGHT⟩	MOUSE-CUT	cut term or region
⟨M- RIGHT⟩	MOUSE-SAVE	save term or region
⟨C-M- RIGHT⟩	MOUSE-DELETE	delete term or region

MOUSE-SET-POINT and MOUSE-SET-TERM-POINT first set the mark at the current editor cursor position and then set the point to where the mouse is pointing, as described in Section 5.5.3. These commands are set up so that a region can be selected by using them at both ends; after the second MOUSE-SET-POINT the mark will be at one end of the region and point will be at the other.

MOUSE-PASTE, MOUSE-PASTE-NEXT, and MOUSE-PASTE-COPY are the same as PASTE, PASTE-NEXT, and PASTE-COPY. MOUSE-CUT is the same as CUT-REGION in text sequences and text slots and the same as CUT otherwise. MOUSE-SAVE and MOUSE-DELETE behave similarly. Note all that these commands do *not* move the point before cutting and pasting.

5.7 Utilities

NUPRL's term editor provides various utility commands that are shown in the table below. The IDENTIFY-TERM, SUPPRESS-DFORM and UNSUPPRESS-DFORM commands assist in interpreting unfamiliar or ambiguous display forms. Exploding a term reveals its internal structure. Viewing abstraction and display form definitions of a term help understanding the formalization of a user-defined concept and its notation. The latter two commands can also issued via mouse commands.

⟨C-X⟩id	IDENTIFY	gives info on term at cursor
⟨C-X⟩su	SUPPRESS	suppress display form at cursor
⟨C-X⟩un	UNSUPPRESS	unsuppress display form at cursor
⟨C-X⟩ns	TERM-INSERT-NULL	insert empty string in text slot
⟨C-X⟩df	VIEW-DISP	view display form def for term
⟨C-X⟩ab	VIEW-ABS	view abstraction def of term
MIDDLE	VIEW-DISP	view display form of term
RIGHT	VIEW-ABS	view abstraction definition of term

IDENTIFY will print out in the ML Top-Loop window information on the term and display form at the current cursor position.

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

UNSUPPRESS restores a suppressed display form if the editor cursor is at a term to which that suppressed display form belongs. Display forms remain suppressed until explicitly unsuppressed or until the editor window is closed.

TERM-INSERT-NULL is useful for inserting empty text strings into text slots. Normally, when all the characters in a text slot that is outside of a text sequence are deleted, a text slot placeholder is left to indicate what kind of item should be inserted into the slot. Use this command if an empty string is what is really wanted.

VIEW-DFORM and MOUSE-VIEW-DISP open the display form object that defines the display form of the term pointed to by the editor cursor (or the mouse).

VIEW-ABSTRACTION and MOUSE-VIEW-AB open the abstraction object that defines the type theoretical meaning of the term pointed to by the editor cursor (or the mouse).

(c-U)	open term slot	$\boxed{\text{DEL}}$	delete char to left of text cursor
(cm-I)	init term slot with prl term	⟨C-D⟩	delete char to right of text cursor
(c-O)	open term slot and init	⟨M-D⟩	cut word to right of text cursor
⟨C-Q⟩	close window without saving	⟨C-K⟩	cut term
⟨C-Z⟩	save, check, and close window	⟨M-K⟩	save term
⟨C-J⟩	jump to next window	⟨C-M-K⟩	delete term
⇆	jump to ML top loop	⟨C-Y⟩	paste item
x	insert char *x*	⟨M-Y⟩	delete item then paste next item
⟨C-#⟩*num*	insert special char *x*	⟨C-M-Y⟩	paste copy of item
↵	insert newline	⟨C- $\boxed{\text{SPC}}$ ⟩	set mark at point
name	insert *name*	⟨C-X⟩⟨C-X⟩	swap point and mark
⟨C-I⟩*name*	insert *name*	⟨C-W⟩	cut region
⟨M-I⟩*name*	insert *name*	⟨M-W⟩	save of region
⟨C-S⟩*name*	replace with *name*	⟨C-M-W⟩	delete region
⟨C-M-I⟩	initialize term slot	⟨C-(Y⟩	paste region
⟨C-M-S⟩	selects dform variations	⟨M-Y⟩	replace last paste with new paste
⟨C-U⟩	open slot to left of cursor	⟨C-M-Y⟩	paste copy of region on save-stack top
⟨M-U⟩	open slot to right of cursor	$\boxed{\text{LEFT}}$	set mark then point
⟨C-O⟩	open slot to left and init	⟨C- $\boxed{\text{LEFT}}$ ⟩	set mark then point to term
⟨M-O⟩	open slot to right and init	$\boxed{\text{MIDDLE}}$	view display form of term
⟨C-C⟩	close slot and move left	⟨C- $\boxed{\text{MIDDLE}}$ ⟩	as PASTE
⟨M-C⟩	close slot and move right	⟨M- $\boxed{\text{MIDDLE}}$ ⟩	as PASTE-NEXT
⟨C-P⟩	move cursor up 1 character	⟨C-M- $\boxed{\text{MIDDLE}}$ ⟩	as PASTE-COPY
⟨C-N⟩	move cursor down 1 character	$\boxed{\text{RIGHT}}$	view abstraction definition of term
⟨C-B⟩	move cursor left 1 character	⟨C- $\boxed{\text{RIGHT}}$ ⟩	cut term or region
⟨C-F⟩	move cursor right 1 character	⟨M- $\boxed{\text{RIGHT}}$ ⟩	save term or region
⟨C-A⟩	move to left side of screen	⟨C-M- $\boxed{\text{RIGHT}}$ ⟩	delete term or region
⟨C-E⟩	move to right side of screen	⟨C-X⟩id	gives info on term at cursor
⟨C-L⟩	scroll window up 1 line	⟨C-X⟩su	suppress display form at cursor
⟨M-L⟩	scroll window down 1 line	⟨C-X⟩un	unsuppress display form at cursor
⟨C-V⟩	move window down 1 page	⟨C-X⟩ex	explode term at cursor
⟨M-V⟩	move window up 1 page	⟨C-X⟩im	implode term at cursor
⟨C-T⟩	switch to term mode	⟨C-X⟩ch	check object
⟨M-P⟩	move up to parent	⟨C-X⟩sa	save object
⟨M-B⟩	structured move left	⟨C-X⟩ab	view abstraction def of term
⟨M-F⟩	structured move right	⟨C-X⟩df	view display form def for term
⟨M-N⟩	move to leftmost child	⟨C-X⟩ns	insert empty string in text slot
⟨M-M⟩	move to rightmost child	**exterm**	insert new exploded term
⟨M-A⟩	move to left-most sibling	**lparm**	insert level exp parm
⟨M-E⟩	move to right-most sibling	**vparm**	insert variable parm
⟨M-<⟩	move up top of term	**tparm**	insert token parm
⟨C- $\boxed{\text{LFD}}$ ⟩	next leaf to right	**sparm**	insert string parm
⟨M- $\boxed{\text{LFD}}$ ⟩	next leaf to left	**nparm**	insert natural number parm
↵	next empty slot to right	⟨C-O⟩	open bterm / parm / bvar slot to left
⟨C- ↵ ⟩	next empty slot to right	⟨M-O⟩	open bterm / parm / bvar slot to right
⟨M- ↵ ⟩	next empty slot to left		

Table 5.2: All key and mouse commands

```
% -------------------------------------------------------------------
% Arrow keys
%
% Default :
%
(UP)==(m-p)
(LEFT)==(m-b)
(RIGHT)==(m-f)
(DOWN)==(m-n)
%
% Text :
%
(RIGHT)==(-text)(m-X)screen-right
(LEFT)==(-text)(m-X)screen-left
% -------------------------------------------------------------------
% Mouse keys:
%
(MouseLeft)==(cm-X)mouse-mark-then-set-point
(c-(MouseLeft))==(cm-X)mouse-mark-then-set-point-to-term
%
(MouseMiddle)==(m-X)view-disp
(c-(MouseMiddle))==(cm-X)mouse-paste
(m-(MouseMiddle))==(cm-X)mouse-paste-next
(cm-(MouseMiddle))==(cm-X)mouse-paste-copy
%
(MouseRight)==(m-X)view-abs
(c-(MouseRight))==(cm-X)mouse-cut
(m-(MouseRight))==(cm-X)mouse-save(m-X)blink
(cm-(MouseRight))==(cm-X)mouse-delete
%
% -------------------------------------------------------------------
```

Table 5.3: Term-editor fragment of the standard `mykeys.macro` file

5.8 Customizing the Editor

The current key bindings of NUPRL's term editor, which are summarized in Table 5.8, are intended to be reminiscent of Emacs's key bindings and compatible with previous releases of NUPRL.

Users who wish to change the default bindings may do so by putting a file called `mykeys.macro` into their home directory. In this file one may define bindings of keys or key combinations to commands of NUPRL's term editor (as well as to navigator and proof editor commands). Bindings can be made globally or depending on the context of cursor. Table 5.3 lists the term-editor fragment of a standard `mykeys.macro` file with the default bindings. Users should exercise caution when changing global bindings, as these may have unwanted effects on the navigator and the proof editor.

Chapter 6

Interactive Proof Development

Whenever an object of kind `STM` is opened, NUPRL's proof editor will be invoked on it. The proof editor provides an interactive method for constructing and modifying proofs in a highly visual fashion. Users enter a goal statement and develop its proof in a *top-down* fashion: proof goals will be *refined* into "smaller" subgoals until all subgoals represent basic axioms or already proven facts.

In this chapter we will first describe the general structure of NUPRL proofs and then discuss the features and usage of the proof editor.

6.1 Proof Structure

NUPRL's inference system is based on the notion of *sequents*. These are objects of the form

$$H_1, \ldots, H_n \vdash C \quad \lfloor\text{ext } t\rfloor$$

which are read as "*under the assumptions H_i we can prove that the type C is inhabited by some member t*". C is called the *conclusion* of the sequent and the H_i the *hypotheses*.[1] A hypothesis is either an *assumption* A_i or a *type declaration* $x_i{:}T_i$. A type declaration $x_i{:}T_i$ is considered to bind free occurrences of the variable x_i in the terms to the right of it, that is the hypotheses $H_{i+1}, \ldots, H_n$ and in the conclusion C. The expression t is called the *extract term* of the sequent. It will be constructed *during* the proof and remains unknown up to its completion.

Sometimes we refer collectively to the hypotheses and the conclusion of the sequent as *clauses*. The word *goal* is used either to refer to a whole sequent or to just the conclusion. Which should be clear from context.

NUPRL's inference rules refine sequents, obtaining subgoal sequents whose proofs would suffice to validate the original goal. *Primitive rules* (see Section 6.1.3.1 below) are usually characterized by rule schemata that match placeholders for the hypotheses and conclusion against a goal sequent and instantiate subgoal sequents accordingly. They also describe how to construct the extract term of the goal sequent from extract terms of the subgoal sequents (see Section 8.1 for details). *Tactic rules* (Section 6.1.3.2) combine several primitive rules into a single inference rule.

Proofs are trees whose nodes contain a *goal* sequent and a *refinement* slot. The refinement slot usually contains an inference rule and the goal sequents of the *children* of the node are the subgoals resulting from applying this rule to the node's goal. If the refinement slot is empty, the node has no children and is considered *unrefined*.

[1] The older NUPRL literature uses >> instead of the turnstyle symbol ⊢ to separate hypotheses from the conclusion.

A proof is *complete* if it has no unrefined nodes, which means that the goals of the leaf nodes are completely proven by their inference rules. In this case, the *top goal* of the proof, i.e. the goal of the root node, is called a *theorem*. The extract term of a theorem can be constructed bottom-up, using the instructions contained in each of the inference rules occurring in the proof.

In NUPRL, sequents, rules, and proofs are abstract data structures that are accessible from ML. Like all system components, they are implemented in the form of abstract terms and in principle, all term editing features described in Chapter 5 can be applied to them. However, all modifications to a proof have to pass through the proof editor before they are saved to the library, which ensures that all the proofs in the library are correct.

In the sections below we will briefly describe the essential aspects of these data structures.

6.1.1 Sequents

The data structure of sequents consists of a list of hypotheses $H_1, \ldots, H_n$ and a conclusion C. The conclusion is a proposition of NUPRL's logic, while each hypothesis may either be an assumption, i.e. a proposition, or a type declaration. For the sake of uniformity, assumptions are considered type declarations[2] with *invisible variables*. Furthermore, a sequent may contain *hidden hypotheses*. These are hypotheses that cannot be used for constructive reasoning but become accessible in parts of the proof that do not contribute to the extract term.

Sequents must be *closed*. Free variables in the conclusion must be declared in one of the hypotheses while free variables occurring in hypothesis H_i must be declared in on of the hypotheses $H_1 \ldots, H_{i-1}$. Obviously, all variables declared in the hypotheses have to be distinct.

Sequents do not explicitly contain extract terms, since extract terms are only constructed for complete proofs.

In NUPRL, sequents occur only within the nodes of a proof and are therefore considered identical to proof nodes with empty refinement slots. Usually NUPRL displays sequents vertically and explicitly numbers the hypotheses, so a sequent $H_1, \ldots, H_n \vdash C$ is displayed as:

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

Variables, whose name starts with a % character are considered invisible and will not be displayed.

The system provides a few ML functions for accessing the components of a sequent.

```
var_of_declaration:     assumption -> var
type_of_declaration:    assumption -> term
is_hidden_declaration:  assumption -> bool
mk_declaration:         (var # term # bool) -> assumption

conclusion:             proof -> term
hypotheses:             proof -> assumption list
mk_sequent:             (var # term # bool) list -> term -> proof
```

Advanced users may take advantage of these functions when developing proof tactics that analyze the contents of a sequent in order to determine appropriate inferences.

[2]Since NUPRL's type theory incorporates the *propositions-as-types* principle, which associates logical propositions with the type of all their proofs, all the clauses of a sequent contain types.

6.1.2 Proof Objects

Proof trees are implemented in NUPRL as *recursive data structure*, consisting of either

- an *unrefined* goal sequent g, or
- a goal sequent g, an inference rule r, and a list $p_1, \ldots, p_n$ of proofs.

Thus each sub-tree of a proof is considered a proof as well, which makes it possible to reason locally. The NUPRL proof editor (see Section 6.2 below) enables users to focus on any node of a proof tree and to view the corresponding sub-proof as a full proof object.

The sequent g in a proof object is referred to as the (*root*) *goal* of the proof and the goals of the proofs $p_1; \ldots; p_n$ in a refined proof are referred to as its *subgoals*. A proof is *good* if

1. every sequent in the proof is closed,
2. in every sequent all variables declared in the hypotheses are distinct, and
3. at every refined node of the proof tree, the subgoals are the result of applying the rule r to the root goal g.

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

Each statement object in NUPRL's library is associated with a list of proof objects with the same root goal, sometimes referred to as the *main goal* of the statement object. The main goal must be an *initial sequent*, i.e. a sequent with an empty hypotheses list. The main goal is a *theorem* if at least one of its proofs is complete.

If a statement object is linked to more than one proof object, one of them is considered the actual proof. The proof editor enables users to switch between proofs for the same statement, which allows them to formalize different approaches to solving the same problem or to work on a "better" proof for a theorem while preserving the existing ones.

The system provides a few ML functions for accessing the components of a proof.

```
var_of_hyp:     int -> proof -> var
type_of_hyp:    int -> proof -> term

conclusion:     proof -> term
hypotheses:     proof -> assumption list
refinement:     proof -> rule
children:       proof -> proof list

mk_sequent:     (var # term # bool) list -> term -> proof
refine:         rule -> tactic
```

The function `mk_sequent` is the only way to build unrefined proofs in NUPRL, while `refine` is the only way for constructing functions that modify proofs from scratch. The proof editor implicitly makes use of these functions when a user initiates and refines a proof.

6.1.3 Refinement Rules

Refinement rules in NUPRL serve two purposes. They *decompose* a goal sequent into a list of subgoal sequents and they provide a *validation*, which transforms evidence for the validity of the subgoals into evidence for the validity of the original goal. Refinement rules are therefore implemented as functions that transform a proof into a list of (unrefined) proofs and a validation v. The validation in turn is a function transforms a list of proofs into a proof.

A refinement rule is *correct*, if the validity of the generated subgoals $g_1; \ldots; g_n$ implies the validity of the root goal g, and if the validation v transforms complete proofs $p_1; \ldots; p_n$ for the subgoals into a complete proof p for the root goal. Logically, validations only prove for the correctness of the rule applications. Computationally, however, they provide evidence for the validity of the main theorem and can therefore be used for building the extract term of a theorem.

Nuprl distinguishes two kinds of rules. *Primitive refinement rules* are the basic rules of inference that constitute the formal theory on which Nuprl is founded. *Tactic rules* are the basis for automating the application of primitive rules. Tactics can only be constructed by combining (converted) primitive rules and other tactics into a new refinement rule. This guarantees that the correctness of proofs generated by tactics only depends on the correctness of the primitive inference rules, which in turn are justified semantically.

6.1.3.1 Primitive Refinement Rules

Nuprl's primitive refinement rules are all introduced by rule objects (see Section 8.1.1) in the system's library. The current system has primitive rules for a constructive *type theory* that is closely related to Martin-Löf type-theory (see Appendix A.3 for a complete list of rules). All Nuprl proofs are eventually justified by these primitive rules.[3]

The correctness of a Nuprl proof depends only on the correctness of these rules and of Nuprl's *refiner*. The *refiner* is a fixed piece of Lisp that applies primitive rules to unrefined leaves of proofs.

In contrast to previous releases of Nuprl, users of Nuprl 5 cannot invoke primitive rules directly. The proof editor expects users to enter tactics when refining a proof, which means that primitive rules have to be converted into tactics before they can be applied (see Section 8.1.3). Furthermore, using primitive rules would require users to understand how mathematical concepts are coded within type-theory. Tactics operate at a higher level of reasoning and are much easier to deal with.

6.1.3.2 Tactic Rules

As explained in detail in Chapter 8, tactics are ML functions that enable one to automate application of primitive rules. If one applies a tactic to a proof and the tactic does not fail, then the tactic returns a proof built entirely from primitive rules. The Nuprl proof editor treats a tactic like a single inference rule and only displays the unrefined leaves of the generated proof tree. Users may view the generated primitive proof on demand.

More precisely, if a tactic rule is applied to a proof node, the Nuprl proof editor will perform the following steps.

1. The ML text of the tactic is interpreted by the ML system and applied to the current proof node, resulting in a proof tree p with unrefined leaves $p_1, \ldots, p_n$. Note that the root goal of p is always identical to the goal sequent of the current proof node and that p also includes validations.

2. Instead of simply replacing the proof node by the proof p, the editor stores p together with the tactic text in a *tactic rule*.

3. The tactic rule is inserted as refinement of the proof node and the leaves $p_1, \ldots, p_n$ become the new children of the node.

[3] This may change in the future, as Nuprl may also accept proofs provided by external proof engines without transforming them into type-theoretical proofs

The display of the tactic rule hides the proof tree p. When one views a tactic rule refinement, one only ever sees the text of the tactic. From a logical point of view, it is not strictly necessary to keep p around at all, after the tactic has executed. However, it is necessary to access the validation contained in the proof when constructing the extract term of the main theorem.

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

Note that applying a tactic rule to an already refined proof node overwrites the existing proof tree. The editor first discards the existing refinement of the node and then proceeds as described above. The previous refinement, however, remains stored in the library and can still be accessed and reinserted into the proof.

6.2 The Proof Editor

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

The proof editor generates windows onto sections of proofs. One can have windows open on different proofs at the same time, and even view multiple proofs of the same theorem. In the latter event, one proof is the *main proof* while the other ones are *backup proofs*.

6.2.1 Proof Window Format

Each proof window is associated with a node of a proof. It shows the goal sequent at that node, the refinement rule (if any) at that node, the immediate subgoals, and the proofs (if any) of these subgoals, as long as they fit into the window. Figure 6.1 shows an example of a window onto a refined node of a proof and an example of a window onto an unrefined node of a proof.

```
- PRF: intsqrt
(1) # top 1
(2)
(3) 1. x:ℕ
    ⊢ ∃y:ℕ. y^2≤x  ∧  x<(y+1)^2
(4) BY NatInd 1
(5) # 1 1
    .....basecase.....
    ⊢ ∃y:ℕ. y^2≤0  ∧  0<(y+1)^2
(6) BY exR ⌜0⌝
    There is 1 hidden subgoal
(5) # 1 2
    .....upcase.....
    1. x:ℤ
    2. 0<x
    3. ∃y:ℕ. y^2≤x-1  ∧  x-1<(y+1)^2
    ⊢ ∃y:ℕ. y^2≤x  ∧  x<(y+1)^2
(5) BY
```

```
- PRF: intsqrt
(1) # top 1 2
(2) .....upcase.....
(3) 1. x:ℤ
    2. 0<x
    3. ∃y:ℕ. y^2≤x-1  ∧  x-1<(y+1)^2
    ⊢ ∃y:ℕ. y^2≤x  ∧  x<(y+1)^2
(4) BY
```

Figure 6.1: Proof window on refined and unrefined proof node

The title of each window contains the indicator PRF, denoting the kind of the object being viewed, and the name of the statement object associated with the proof. The numbered parts of these windows are as follows:

① The # indicates that this proof node is considered *incomplete.* Other symbols used here are * for *complete*, and - for *bad* proofs.
The top 1 and top 1 2 are tree addresses of the nodes being viewed. The left window shows the first child of the root of the proof and the right window shows the second child of that proof node.

② Some nodes are annotated to indicate the nature of the proof goal. Typical annotations are wf for well-formedness subgoals or upcase and downcase in inductive proofs. These annotations may be used by tactics but are mainly intended to assist the users.

③ This is the goal sequent of the proof node. Hypotheses are numbered and listed vertically. The conclusion is at the bottom after the turnstyle.

④ This is a tactic which was executed on the goal ③ above in order to generate the subgoals ⑤ below. The BY is part of the proof node display, and is not part of the tactic.
In an unrefined proof, there is an empty text slot after the BY.

⑤ These are the subgoals of the proof node. Each subgoal comes with a status (*, #, or -), an address, and the subgoal sequent. For brevity, only hypotheses that have changed or been added are displayed in the subgoal sequents.

⑥ If a subgoal has a proof, it is being displayed immediately below the subgoal as long as it fits into the window. If a proof is too large to be shown, the editor will only display its top-level tactic and indicate that its subgoals are hidden. In this case users have to move into the corresponding node to see further details or to continue editing the proof.

Sometimes the proof window is too short to display all the goal, rule, and subgoals. In this case the cursor motion commands described in Section 5.5 will automatically scroll the window. One can of course also resize the window.

6.2.2 Proof Motion Commands

←	move to sibling to immediate left
→	move to sibling to immediate right
⟨M-a⟩	move to left-most sibling
⟨M-e⟩	move to right-most sibling
↑	move up to parent node
↓	move down to selected subgoal
⟨M-z⟩	zoom in on current node
⟨C-↑⟩	move up to top of proof
⟨C-M-j⟩	jump to next unrefined node

The keyboard commands for navigating through a proof tree are summarized in the table below. In addition to these, most of the motion commands for terms described in Section 5.5 can be used for navigating through (the term-tree of) a proof window.[4]

[4]In contrast to previous releases, Nuprl 5 only uses arrow keys for proof-tree navigation. The emacs-like keyboard and mouse commands used in Nuprl 4 are now captured by the term editor, which has priority over the proof editor, and are thus reserved for navigating through the term tree of the proof window as described in Section 5.5.2.

The left-, right-, up-, and down-arrow keys move one node left, right, down, or up in the proof tree. If a node has many siblings, users may also use the key combinations ⟨M-a⟩ and ⟨M-e⟩ for larger jumps. In each case the proof window will *focus* on the new node, i.e. make it the root of the currently displayed proof tree. The cursor will be positioned in the refinement slot of that node. The proof window on the right of Figure 6.1, for instance, results from the window on the left by pressing ↓ first and then →. Pressing the ↑ key in the right window will again produce the window on the left.

Proof tree motion is relative to the node where the cursor is positioned. If the cursor is at the current root node, then pressing ←, →, or up ↑ will move the focus to a sibling or parent node that is currently not visible, while ↓ will focus on the first visible subgoal. If the cursor is at a subgoal of the current root node, then pressing an arrow key will move the focus relatively to that node, while pressing ⟨M-z⟩ will cause the proof window to focus on that node.

⟨C-↑⟩ causes the editor to jump to the root of the proof tree that is currently being edited, while ⟨C-M-j⟩ causes it to jump to the next unproven subgoal in that tree, using a preorder traversal of the proof tree. If there are none, ⟨C-M-j⟩ shifts the focus to the root of the proof tree.

6.3 Stating and Proving Theorems

The proof editor is invoked whenever a statement object is opened for viewing. Usually, this is done through the navigator by using either the right arrow key or the middle mouse button (Section 4.3.1.1). As proofs associated with statement objects are *not* copied when they are viewed with the proof editor, all changes made to proofs are immediately committed to the library. This is in contrast to editing objects of other kinds (abstractions, display forms, code objects, . . .) where changes are only committed when one exits the object or explicitly asks for changes to be saved. Users who want to make tentative changes to a section of a proof should first create a backup proof (see Section 6.4.3) and then work on either of the two versions.

6.3.1 Editing The Main Goal

```
PRF not_over_and
# top
[goal]

BY
```

```
PRF not_over_and
# top
∀A,B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))

BY
```

When a new proof window is opened, the window appears as depicted on the left above. A term cursor is positioned on an empty `[goal]` slot immediately below the root address. A user may now enter the main goal using the structured term editor (Chapter 5).[5] The window on the right, for instance, is the result of entering

␣all ↵ A ↵ prop ↵ i ↵ all ↵ B ↵ prop ↵ i ↵ implies ↵ or ↵ not ↵ A ↵ not ↵ B ↵ not ↵ and ↵ A ↵ B ↵␣

into the initial proof window. Pressing ↓ then moves the cursor into the text slot for proof tactics.

It should be noted that the proof goal is *not* committed to the library until the first refinement step has been executed. If a user closes the proof window after entering the main goal, the input will be discarded and the statement object remains empty. To commit a proof goal explicitly to the library, one may use the key combination ⟨C-M-g⟩. This will save the main goal into a proof object, which will be linked to the statement object that is currently being viewed.

[5]Currently, all input until the first ↵ will be ignored. This is an interface bug that will be corrected in the future.

Using ⟨C-M-g⟩ is required if one wants to change an already existing main goal. Simply editing the goal is not sufficient and an error message will be produced if a user tries to refine a modified proof goal that has not yet been committed.

Changing a proof by modifying either its main goal or one of the refinement steps will remove it from the proof window but not from the library. Previous versions of a proof may be recovered by walking through the proof editor history (see Section 6.4.2 for details).

6.3.2 Refining Proof Goals

To refine a proof goal, one uses the arrow keys or the mouse to move the cursor into the text slot for entering proof tactics and then types the name of the tactic to be applied. Tactics are ML functions that may have Nuprl terms and other parameters as arguments (see Chapter 8). The structure and special editing commands for tactics are the same as for `CODE` objects. Nuprl terms may be entered using the term editor after opening a term slot with ⟨C-o⟩ (Section 5.4.2). Other tactic arguments have to follow the syntax of the corresponding ML data types. Tactics may include ML comments (using `%` both as left and right delimiter) and newline characters, which will be inserted when a user types the ↵ key.

There are two possible modes for executing a tactic. In *synchronous mode*, initiated by pressing ⟨C-↵⟩, the tactic is sent to the refiner and the editor waits for the refinement process to be complete before allowing the user to continue. Pressing ⟨C-↵⟩ instead initiates *asynchronous refinement*, which allows the user to work on other proof goals while the proof goal is being refined. This is useful when executing complex tactics that may take a long time to complete.

Once a (synchronous or asynchronous) refinement is successfully completed, the proof is committed to the library and the proof window gets updated, showing the subgoals that were generated by applying the tactic. If the refinement fails, an error message describing the nature of the error and some debugging information will be inserted after the tactic and the proof goal will be marked as *bad*. The proof itself will not be changed.

Entering ⌞D 0 ⟨C-M-↵⟩⌟, for instance, surrounds the tactic `D 0` by markers to indicate that it is currently being processed (see the window on the left below). Upon completion of the refinement, the editor removes the markers and inserts the resulting subgoals as shown in the window on the right below. The latter is also the result of entering ⌞D 0 ⟨C-↵⟩⌟.

```
PRF not_over_and
# top
∀A,B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY << D 0 >>
```

```
PRF not_over_and
# top
∀A,B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY D 0
# 1
1. A:ℙ
⊢ ∀B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY
# 2
.....wf.....
ℙ ∈ 𝕌'
BY
```

A refinement is always initiated for the proof goal at the current location of the cursor. Users are free to refine subgoals in any order and to modify already existing refinements. The latter will erase the proof tree at the current proof node and insert the result of the modified refinement into the proof window (see Section 6.4.1 for details). The old proof may, however, be recovered by walking through the proof editor history (see Section 6.4.2).

6.3.3 Generating Extract Terms

Proof steps are committed to the library as soon as a refinement step has been executed successfully. Users may therefore simply close the proof editor once a proof is complete. Like all other windows, the proof editor window will be closed by ⟨C-q⟩.

Pressing ⟨C-z⟩ instead will cause the proof editor to build the extract term of the proof before closing the proof window. As the extract term is constructed by assembling the validations contained in the refinements of each proof node, it can only be built if the proof is complete. Using ⟨C-z⟩ on an incomplete proof will close the proof window but pop up a window with an error message.

Term extraction is NUPRL's mechanism for supporting the *proofs-as-programs* principle. Extract terms describe the computational content of a proof and the algorithms that are synthesized while formally solving a program specification problem. These algorithms are proven to satisfy the specification and can be executed by the NUPRL term evaluator as described in Section 4.4.3.1.

Once created, extract terms and the corresponding proof extract tree can also be viewed from within the proof editor. Typing ⟨C-M-v⟩`pex` pops up the extract term of the saved proof, while ⟨C-M-v⟩`pet` shows the proof extract tree. The windows below, for instance, show a complete proof of the statement ⌞`∀A,B:ℙ. (¬A ∨ ¬B) ⇒ ¬(A ∧ B)`⌟ and the term extracted from that proof.

```
PRF not_over_and
* top
∀A,B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY D 0
* 1
1. A:ℙ
⊢ ∀B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY Auto
* 1 1
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)
BY D 0 THEN D 3 THEN Auto
* 2
.....wf.....
ℙ ∈ 𝕌'
BY Auto
```

```
extract: not_over_and
Refresh*  Quit*
 (get_prf_extract Obid: not_over_and)
  @ 2:15PM 9/10/2002

 λA,B,%,%1.case % of inl(%2) => Ax | inr(%3) => Ax
```

6.4 Advanced Editing Features

6.4.1 Modifying existing refinements

In previous releases of NUPRL, modifications to an existing refinement of a proof node caused the entire proof tree below that node to be lost. Often however, some of the subgoals generated by the new refinement could be solved by replaying certain parts of that proof tree. Since NUPRL 5 immediately commits all successful refinement steps to the library, it is possible to reuse these steps when a proof is modified.

The default refinement, initiated by pressing ⟨C-↵⟩ (or ⟨C-M-↵⟩) refines the goal at the current proof node and reuses the proofs of subgoals that were already refined in the previous proof. This is useful, when some of the newly created subgoals are identical to subgoals of the previous refinement of that node. In the example below, for instance, replacing the tactic ⌞`D 0`⌟ by ⌞`D 0 THENW Auto`⌟ generates the same "main" subgoal and initiating the refinement with ⟨C-↵⟩ preserves the proof of that subgoal (center window). In previous releases, that proof would have been lost and the whole proof would have become incomplete (right window).

```
PRF not_over_and
* top 1

1. A:ℙ
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)

BY D 0

* 1 1

4. A ∧ B
⊢ false

BY D 3 THEN Auto

* 1 2
.....wf.....
A ∧ B ∈ ℙ

BY Auto
```

```
PRF not_over_and
* top 1

1. A:ℙ
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)

BY D 0 THENW Auto

* 1 1

4. A ∧ B
⊢ false

BY D 3 THEN Auto
```

```
PRF not_over_and
# top 1

1. A:ℙ
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)

BY D 0 THENW Auto

#  1 1

4. A ∧ B
⊢ false

BY
```

Instead of reusing the proof of a specific subgoal, users may also want to reuse the tactics that had been applied to the subgoals of the previous refinements. Using ⟨M- ↵⟩ instead of ⟨C- ↵⟩ causes NUPRL 5 to reuse the *tactic tree* of the previous proof for subsequent refinements. This is useful when the new refinement generates subgoals that are different from the previous ones but that can be solved in the same way.

If users want to discard the original proof and just execute a single new refinement step, they may enter the keyboard macro ⟨C-M-r⟩`st`. "`step`" refinement emulates the behavior of previous NUPRL releases, which is meaningful if reusing the previous proof would be time-consuming and of little use for the new proof.

Often, users want to rearrange a proof in a way that each refinement step corresponds to an argument a human would make. Usually this means assembling several refinements steps into one and adding a comment that describes the logical meaning of that step. NUPRL offers some support for this technique.[6] Pressing ⟨C-M-r⟩`kr` will collect *all* the inference steps of the proof tree starting at the current node into a single refinement step. This step consists of a tactic that combines the individual steps using the tacticals `THEN` and `THENL`. The window on the right below shows the result of applying this command to the proof on the left. Extensions of this command (like accumulating only a certain amount of steps or inserting comments) will be added in the future.

```
PRF not_over_and
* top 1

1. A:ℙ
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)

BY D 0

* 1 1

4. A ∧ B
⊢ false

BY D 3 THEN Auto

* 1 2
.....wf.....
A ∧ B ∈ ℙ

BY Auto
```

```
PRF not_over_and
* top 1

1. A:ℙ
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)

BY D 0 THENL [D 3 THEN Auto; Auto]
```

Pressing ⟨C-M-r⟩`dk` turns a "kreitzed" proof back into its original form. It has no effect on refinements that were not previously generated by ⟨C-M-r⟩`kr`.

[6]At some time in the past, the name *kreitzing* was introduced for this technique. For lack of a better name, it is still called that way.

Similarly to the path stack of the navigator (Section 4.3.1.3) the proof editor enable users to jump between commonly used positions in the proof tree. Pressing ⟨C-`h`⟩ marks the current proof address and stores it in an address stack. Pressing ⟨M-`h`⟩ jumps back to that position and removes the address from the stack.

NUPRL also enables users to copy a pattern of reasoning used in one proof to another proof. Pressing ⟨M-`k`⟩ copies the proof at the current node to a proof stack. ⟨C-`y`⟩ pastes the proof on top of the proof stack into the current proof node and removes it from the stack. Pasting a proof means re-executing the tactics of its tactic tree until one of the tactics fails or the complete tactic tree has been reused.

6.4.2 Proof History

In the course of proof development, NUPRL's proof editor stores each refinement as a separate object in the library. As a consequence, users may walk backward and forward through the proof history and continue the refinement of previous versions of the proof.

Pressing ⟨C-←⟩ reverts the proof window to the previous proof in the proof history while ⟨C-→⟩ moves to the next proof, if there is one. If a user refines one of the previous proofs in the history, all subsequent proofs in that history will be discarded the new proof will be the last proof in the new history. Users who want to save an older version of a proof to the library without modifying it, can do so by pressing ⟨C-`x`⟩⟨C-`s`⟩ (just moving through the history does not change the stored proof). Pressing ⟨C-M-`e`⟩ reverts the proof window to the proof that was last stored in the library.

The proof history is only preserved through a proof editing session. Once the proof window is closed the history is discarded.

6.4.3 Backup Proofs

In addition to using a temporary proof history NUPRL 5 allows users to create and edit backup proofs that are linked permanently to a statement object. This makes it possible to elaborate and keep different proofs of the same statement and to preserve several interim versions of a proof attempt until they are not needed anymore.

Pressing ⟨C-M-`c`⟩ will create a backup copy of current proof and save it to the library. This proof will usually remain invisible. Pressing ⟨M-→⟩ pops up a proof editor window for each backup proof of the current statement.

Users may create multiple backup copies, including backups of backup proofs. The proof from which all these backups were created will remain to be the *main proof* unless the user explicitly changes that by pressing ⟨C-M-`f`⟩. This will declare the current proof to be the main proof from now on. To remove a specific proof, users may type ⟨C-M-`d`⟩ into the proof window of that proof. Typing ⟨C-`x`⟩⟨C-`b`⟩ deletes all backup proofs.

It should be noted that deleted backup proofs and the temporary proof history are still contained in the library but not linked to the statement object anymore. Expert users may still be able to retrieve them if that should be necessary.

6.4.4 Views of Proofs and Refinements

NUPRL enables users to look at proof trees in different ways. In the default view, the proof window displays the addresses, goals, and refinements of each visible node. Pressing ⟨C-M-`t`⟩ will show the tactic tree instead, ⟨C-M-`a`⟩`a` the proof structure (i.e. the address tree), and ⟨C-M-`v`⟩`v` the goal tree. ⟨C-M-d⟩ will revert to the default view. Examples of these four views are displayed in the windows below.

```
PRF not_over_and
* top 1

1. A:ℙ
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)

BY D 0 THENW Auto

* 1 1

4. A ∧ B
⊢ false

BY D 3 THEN Auto
```

```
PRF not_over_and
* top 1
BY D 0 THENW Auto

1 1
BY D 3 THEN Auto
```

```
PRF not_over_and
* top 1

* 1 1
```

```
PRF not_over_and
* top 1

1. A:ℙ
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)

* 1 1

4. A ∧ B
⊢ false
```

Users may also want to create views on specific proof parts. Pressing ⟨C-M-p⟩ pops up a new proof window whose main proof is the current node. This window is considered a scratch window. Users may execute refinements in this window but these refinements will not affect the original proof. If a statement has several proofs, pressing ⟨C-M-i⟩ will pop up a window that contains pointers to other proofs of this same statement.

```
PRF not_over_and
# top 1 1

1. A:ℙ
2. B:ℙ
3. (¬A) ∨ (¬B)
4. A ∧ B
⊢ false

BY D 3

#  1 1 1

3. ¬A
4. A ∧ B
⊢ false

#  1 1 2

3. ¬B
4. A ∧ B
⊢ false
```

```
PRF not_over_and
# top
1. A : ℙ
2. B : ℙ
3. (¬A) ∨ (¬B)
4. A ∧ B
⊢ False
BY !rule_instance{direct_computation_hypothesis:o}(#3 [1:(¬A) ∨ (¬B)] ())

# 1
3. ¬A + (¬B)
4. A ∧ B
⊢ False
BY !rule_instance{unionElimination:o}(#3 %2 %3 ())

# 1 1
4. ¬A
5. A ∧ B
⊢ False
BY !rule_instance{thin:o}(#3 ())

# 1 1 1
3. ¬A
4. A ∧ B
⊢ False
BY

# 1 2
4. ¬B
5. A ∧ B
⊢ False
BY !rule_instance{thin:o}(#3 ())

# 1 2 1
3. ¬B
4. A ∧ B
⊢ False
BY
```

The keyboard macros ⟨C-M-h⟩ and ⟨C-M-l⟩ are used to show internal details of a refinement step. ⟨C-M-h⟩ pops up a window that shows the the actual steps performed by the refinement tactic, while ⟨C-M-p⟩ shows the proof on the level of primitive inferences. An example of a primitive proof tree corresponding to simple refinement step is shown above.

6.4.5 Miscellaneous Features

Users may *print* proofs by typing ⟨C-M-m⟩ into the editor window. This will print a snapshot of the current proof to a file nuprlprint/*stm-name* in the user's home directory, where *stm-name* is the name of the corresponding statement object. The directory nuprlprint must already exist.

The generic commands for closing and saving editor windows have a slightly different meaning in the context of proof editing. As usual, ⟨C-q⟩ *closes* a proof window without initiating any modifications. However, since proofs are saved after each refinement step, all editing steps that were performed after opening the proof editor are already committed to the library and will not be discarded. *Saving* a proof window with ⟨C-z⟩ therefore has a somewhat stronger meaning than just saving its visible contents. In addition, an extract term will be created and saved, provided the proof is complete.

6.5 Customizing the Proof Editor

Table 6.5 summarizes the current commands of NUPRL's proof editor. Users may change the keyboard macros that initiate these commands by editing the file mykeys.macro, which is also used for modifying the key bindings of the term editor (Section 5.8). Modifications should be done carefully to avoid that they affect the navigator and term editor as well.

6.6 Troubleshooting

Most proof editing mistakes can easily be corrected by typing the *undo* command ⟨C-_⟩ or by walking backwards through the proof history with ⟨C-←⟩.

A common problem occurs when users try to delete a refinement rule as a whole with ⟨C-k⟩ or ⟨C-c⟩. This will delete the text slot for entering refinement tactics and leave a term slot, displayed as [left]. Usually the *undo* command ⟨C-_⟩ will bring back the original rule, which can then be modified using *text* editing commands. If the rule cannot be removed by text editing, for instance if it was created by ⟨C-M-r⟩kr, users should instead enter ⌞itext_df⌟ into the [left] slot to get an empty text slot.

Movement	
←	move to sibling to immediate left
→	move to sibling to immediate right
⟨M-a⟩	move to left-most sibling
⟨M-e⟩	move to right-most sibling
↑	move up to parent node
↓	move down to selected subgoal
⟨M-z⟩	zoom in on current node
⟨C-↑⟩	move up to top of proof
⟨C-M-j⟩	jump to next unrefined node
Inference	
⟨C-↵⟩	refine goal at current node
⟨M-↵⟩	refine goal reusing the tactic tree below
⟨C-M-↵⟩	asynchronously refine goal
⟨C-M-r⟩st	"step" refine
⟨C-M-r⟩kr	kreitz this subtree
⟨C-M-r⟩dk	de-kreitz this node
Copy/Paste:	
⟨C-h⟩	mark proof address
⟨M-h⟩	goto proof at address on stack
⟨M-k⟩	copy proof when selected
⟨C-y⟩	paste proof on stack into current proof node
Proof Editor History	
⟨C-←⟩	reverts window to previous proof in history walk
⟨C-→⟩	reverts window to next proof in history walk
⟨C-x⟩⟨C-s⟩	save proof in window to library
⟨C-M-e⟩	update the proof window with the current proof in the library
Saving and Deleting Proofs	
⟨C-M-g⟩	save goal to library
⟨C-M-c⟩	make backup copy of current proof
⟨C-M-f⟩	set current proof to be the main proof
⟨M-→⟩	bring up backup proofs
⟨C-M-d⟩	delete current proof
⟨C-x⟩⟨C-b⟩	delete all backup proofs
Alternative Views	
⟨C-M-p⟩	create scratch window containing the current sub-proof
⟨C-M-i⟩	pop up pointers to other proofs of the current statement
⟨C-M-d⟩	select default view mode
⟨C-M-t⟩	view the tactic tree
⟨C-M-a⟩a	view the address tree
⟨C-M-a⟩v	view the goal tree
Proofs Details	
⟨C-M-h⟩	show interior proof
⟨C-M-l⟩	show primitive proof
⟨C-M-v⟩pet	show extract tree
⟨C-M-v⟩pex	show extract term
Miscellaneous	
⟨C-M-m⟩	print current proof
⟨C-q⟩	close proof window
⟨C-z⟩	generate extract term and close proof window

Table 6.1: Proof Editor Keyboard Macros

Chapter 7

Definition and Presentation of Terms

Nuprl offers users an opportunity to extend the basic language of type theory by introducing new, abstract terms whose meaning is defined in terms of existing language constructs. In addition to that users may also modify the visual appearance of abstract terms[1] and adjust the presentation of formal material without changing the formal content itself, which makes it possible to use familiar notation or to present the same material differently to different target groups.

Users can create several kinds of library objects for this purpose. *Abstractions* are used to introduce the *abstract definition* of a new term, *display forms* define the *textual presentation* of abstract terms, and *precedence objects* assign *precedences* for terms to control automatic parenthesization. In Section 4.3.2.2 we briefly described how to create abstractions and display forms using the navigator's `AddDef*` button. In this chapter we will describe the contents and features of abstractions, display forms, and precendences as well as editor support for defining them.

7.1 Abstractions

Abstractions are terms that are definitionally equal to other terms. In Nuprl they may be defined in terms of language primitives and other abstractions, but the dependency graph for abstractions should be acyclic. In particular, an abstraction not depend on itself. Recursive definitions can be introduced using the `AddRecDef` button as described in Section 4.3.2.2.

Abstraction definitions have form

lhs `==` *rhs*

where *lhs* and *rhs* are *pattern* terms that may contain free variables. The latter are implicitly universally quantified. When Nuprl *unfolds* some *lhs-inst* instance of *lhs*, it first matches this instance against the pattern *lhs* and generates bindings for the free variables of *lhs* accordingly. It then applies these bindings to the free variables in *rhs* to calculate the term *rhs-inst* into which *lhs-inst* unfolds. Therefore, all free variables of *rhs* must also occur free in since otherwise unfolding a definition would yield a term with unbound variables. An example of an abstraction object is given below.

- ABS: int_seg

```
{i..j⁻} == { k:ℤ | i ≤ k < j }
```

[1] This includes user defined terms, primitive terms of Nuprl's type theory, and even the terms used for describing Nuprl editing features such as the navigator, proof terms, or the appearance of abstractions and display forms.

The abstraction defines a type of segments of integers. The abstraction object `int_seg` already consults a display form in the presentation of the left hand side of the definition. The structure of the left hand side becomes more readily apparent if we write it in uniform syntax (which can be made visible by exploding the term as described in Section 5.4.5). ⌞`{i..j⁻}`⌟ is ⌞`int_seg(i;j)`⌟, a term with opid `int_seg`, no parameters, and two subterms. An instance of the left hand side is ⌞`{0..10⁻}`⌟, which would unfold to ⌞`{ k:ℤ | 0 ≤ k < 10 }`⌟.

Just as abstractions can be unfolded by applying their definition left-to-right, so instances of their right hand sides can be *folded* up to be instances of their left hand sides. Folding, however does not always work, as information can be lost in the unfolding process. For instance, an abstraction can have variables and parameters that are not used in its definition but are only used for "book-keeping purposes". In this case the variables and parameters only occur on the left hand side of the definition and would have to be inferred when folding up a specific instance of the right hand side.

7.1.1 Bindings in Abstractions

In additions to ordinary variables, abstractions can have binding structure. Consider, for instance, the definition of the *unique existence* quantifier below.

- ABS: exists_uni

```
∃!x:T. P[x] == ∃x:T. P[x] ∧ (∀y:T. P[y] ⇒ y=x∈T)
```

Here `x` represents a variable that becomes bound in the term `P[x]` and this binding structure must be mapped from the abstract term ⌞`exists_uni(T; x.P[x])`⌟ to its definition

First-order matching and substitution are inadequate for handling terms with binding structure, since they consider variables to be independent from each other and thus cannot express the dependency between `x` and `P[x]`. NUPRL's therefore uses *second-order* matching and substitution functions to handle abstractions with binding variables in a systematic way.

A *second-order binding* is a binding $v \mapsto x_1,\ldots,x_{a_n}.t$ of a second-order variable v to a second-order term $x_1,\ldots,x_{a_n}.t$. A *second-order variable* is essentially an identifier as with normal variables, but it also has an associated *arity* $n{\geq}0$. *Second-order terms* are a generalization of terms that can be thought of as 'terms with holes', i.e. as terms with missing subtrees.[2] They can be represented by bound-terms such as $x_1,\ldots,x_{a_n}.t$, where the binding variables are place-holders for the missing subtrees. In a second-order binding $v \mapsto x_1,\ldots,x_{a_n}.t$, the arity of v must be equal to n.

An *instance* of a second-order variable v with arity n is a term $v[a_1;\ldots;a_n]$, where $a_1,\ldots,a_n$ are terms, also called the *arguments* of v. A *second-order substitution* is a list of second-order bindings. The result of applying the binding $[v \mapsto w_1,\ldots,w_n.t_{w_1,\ldots,w_n}]$ to the variable instance $v[a_1;\ldots;a_n]$, is the term $t_{a_1,\ldots,a_n}$ – the arguments of the instance of the second-order variable fill the holes of the second-order term.

In our above example, `P` is a second order variable with arity 1, and the terms `P[x]` and `P[y]` are second-order-variable instances. Consider unfolding an instance of the left-hand side, say the term ⌞`∃!i:ℤ. i=0∈ℤ`⌟. The substitution generated by matching this against ⌞`∃!x:T. P[x]`⌟ would be

```
[ P ↦ i. i=0∈ℤ ; T ↦ ℤ ]
```

and the result of applying this to the right hand side of the definition would be

```
∃x:T. x=0∈ℤ ∧ (∀y:ℤ. y=0∈ℤ ⇒ y=x∈ℤ)
```

[2] Roughly, second-order terms are like functions on terms but there are subtle differences between the two concepts.

Actually, the matching and substitution functions used by NUPRL are a little smarter than shown above, as they try to maintain names of binding variables. The result one would get in NUPRL would be ⌞∃i:T. i=0 ∈ ℤ ∧ (∀y:ℤ. y=0 ∈ ℤ ⇒ y=i ∈ ℤ)⌟.

NUPRL does not allow nested bindings on the left-hand side of abstraction definitions. All variables must either be first-order or second-order variables with first-order variable arguments.

7.1.2 Parameters in Abstractions

Abstractions can also contain *meta-parameters*, i.e. placeholders for parameters that matching and substitution treat as variables. We usually indicate that a parameter is meta by prefixing it with a $ sign. For example, we might define an abstraction `label`{x:`t`,i:`n`} as shown below

```
- ABS: label

label{$tok:t,$nat:n}() == <"$tok",$nat>
```

Meta-parameters make it possible to map parameters in newly defined abstractions onto parameters of existing terms. In the above example labels are defined as pairs of tokens and natural numbers and the parameter `$tok` is mapped onto the parameter of the term `token` while the parameter `$nat` is mapped onto the parameter of the term `natural_number`, which is revealed when the right hand side of the definition is exploded into ⌞`pair(token{$tok:t};natural{$nat:n})`⌟.

Level-expression variables occurring in level-expression parameters of abstraction definitions are always considered meta-parameters, so there is no need to designate them explicitly. However, indicating meta-parameters explicitly makes it easier to identify them as such.

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

7.1.3 Attributed Abstractions

A recently added feature of abstraction definitions is an optional list of *attributes* or *conditions*. An attribute is simply an alpha-numeric label associated with the abstraction and the general form of an abstraction with conditions $c_1, \ldots, c_n$ is:

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

Abstraction conditions can be used to hold information about abstractions that may be useful to tactics and other parts of the NUPRL system. They could, for instance, be used to group abstractions into categories, and when doing a proof, one could ask for all abstractions in a given category to be treated in a particular way, e.g. to unfold all abstractions of a category "`notational abbreviations`".

7.1.4 Editor Support

In this section, we describe the editor support for abstraction objects. An abstraction can be viewed by opening it with the navigator (Section 4.3.1.1) or by using the VIEW-ABSTRACTION command (⟨C-X⟩`ab`) on a term containing an instance of it (Section 5.7). The following commands and key sequences may be used for editing abstractions.

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

Since most abstraction objects are created using the `AddDef*` mechanism described in Section 4.3.2.2, the left and right hand side of the abstraction is already present when the object is opened. In the rare case that the abstraction object was created with the `MkObj*` command button (Section 4.3.2.1) it will contain an empty term slot when it is first visited, which must be initialized before a definition can be entered.

The INITIALIZE command will create an uninstantiated abstraction definition term, which looks like:

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

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

so_var$n$ has to be used to enter second order variable instances on the left and right hand sides of the definition. This will insert a term of the form ⌞variable{x:v}(a_1;...;a_n)⌟ where x is the variable's name, and $n > 0$ is a natural number. The library display form object for this term is named so_var$n$ so this family of names can be used to reference them.

Note that abstraction objects are the *only* places where these second-order variable instances are used. When writing propositions, second-order variable instances are simulated using the so_apply$n$ abstraction.

CYCLE-META-STATUS converts a parameter into a meta-parameter if the text cursor is in the parameter's text slot. If the parameter is already meta, using this twice will cycle its status back to being a normal parameter.

SELECT-TERM-OPTION enables a user to add conditions to an abstraction. By default, an abstraction definition term has an empty condition sequence as a subterm, which is hidden by the display form for abstractions. Moving the term cursor over the whole abstraction term and using SELECT-TERM-OPTION will add an empty term slot for a condition. The condition term is much like the term for variables; it has a single text slot, and otherwise no other display characters. To get additional slots for condition terms one may use OPEN-SEQ-TO-LEFT or OPEN-SEQ-TO-RIGHT.

7.2 Term Display

Display form objects are used to control the visual presentation of formal mathematical concepts. They define how a term shall appear when it is being displayed on the screen or printed on paper. This enables users to present formal content within a variety of notations without having to change the internal logical representation of these terms. Display forms are commonly created whenever a definition is introduced using the `AddDef*` mechanisms (Section 4.3.2.2), but they may also be added for existing abstractions as well as for the primitive terms of the library.

def-seq	::=	*definition* `;;`
	|	*definition* `;;` *def-seq*
definition	::=	*format-seq* `==` *term*
	|	*attr-seq* `::` *format-seq*`==` *term*
format-seq	::=	*format*
	|	*format format-seq*
attr-seq	::=	*attribute*
	|	*attribute* `::` *attr-seq*

Figure 7.1: Display Object Structure

In NUPRL the presentation of all formal content, including the appearance of the navigator, the editor, sequents, and proofs is controlled by display forms and may be adjusted according to the preferences of a user. Even the mechanisms for editing and presenting display forms themselves may be modified. The display forms for the quantifiers (`all_df`, `exists_df`) and the logical connectives (`and_df`, `or_df`, ...), for instance, appear as "display form generators".

The top level structure of a display form object is summarized by the grammar shown in Figure 7.1. An object contains one or more display form *definitions*. Each definition has a right-hand-side *term* to which the display form applies, and a sequence of *formats* that specify how to display the term. A definition also has an optional sequence of *attributes* that specify extra information about the definition.

Usually, all the definitions in one object refer to a closely related set of terms. When choosing a display form to use for a term, the layout algorithm tries definitions in a backward order, so definitions are usually ordered from more general to more specific.

7.2.1 Editing Display Form Objects

Since most display forms are created using the `AddDef*` mechanism described in Section 4.3.2.2, a right-hand-side term, a standard sequence of formats, and an alias attribute (see Section 7.2.4 below) are already present when the object is opened. If the object was created with the `MkObj*` command (Section 4.3.2.1) it will contain an empty term slot, which must be initialized before a display form definition can be entered.
The command ⟨C-M-`I`⟩ will create an initial display form definition, which looks like:

```
== [rhs]
```

To get additional slots for display form definitions one may use the commands ⟨C-`O`⟩ and ⟨M-`O`⟩.

An initial display form definition has an empty attribute sequence as a subterm, which is hidden by the display form for display form definitions. Moving the term cursor over the whole term and using ⟨C-M-`S`⟩ will add an empty term slot for an attribute.

7.2.2 Right-hand-side Terms

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

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

The right-hand-side term is restricted to being a term whose subterms are either constant terms, i.e. terms with no meta-variables, or meta-terms. To enter a meta-term into a term slot one has to use the name `mterm`. To turn parameters and variable into meta-parameters or meta-bound-variables, position a text cursor in the appropriate parameter or bound variable slot and give the CYCLE-META-STATUS command ⟨C-M⟩ (twice). Display-meta-variables are readily recognized because they have `<>` as delimiters.

The rhs right-hand-side term may also contain normal parameters, bound variables and variable terms. These are treated like constants: for a definition to be applicable, they must match exactly.

7.2.3 Format Sequences

Format sequences are text sequences that may contain slots for meta-variables and commands for controlling the layout of formal material through insertion of optional spaces, line breaking, and indentation. Except for text strings, all formats must be entered into term slots, which may be created as described in Section 5.4.2.

The various kinds of formats are summarized in the table below. The *Name* column gives the name that has to be entered into a term slot to create the format, while the *Display* column describes how the format will be presented within a display form definition.

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

7.2.3.1 Slot Formats

Slot formats are placeholders for the children of a display form instance. Text slots are generally used for meta-parameters and meta-bound-variables, while term slot formats contain meta-terms.

The *id* in a slot format is the name of the slot. The slot corresponds to the meta-variable of the right-hand-side term with the same name. *ph* is place-holder text, which will appear (enclosed within `[]`'s) in the slot whenever it is uninstantiated in some instance of the display form. The `L,E` and `*` options on the term slot formats control parenthesization of the slot and are discussed in Section 7.2.4.2.

7.2.3.2 Margins

The margin control format {$\rightarrow i$}, where $i \geq 0$, pushes a new left margin i characters to the right of the format position onto the *margin stack*. The layout algorithm uses the top of the margin stack to decide the column to start laying out at after a line break. To create the format, enter `pushm` into a term slot and edit the number 0 in the format `{→0}` accordingly.

The margin control format `{←}` (`popm`) pops the current margin off the top of the margin stack and restores the left margin to a previous margin. Usually display forms should have matching `pushm`'s and `popm`'s.

7.2.3.3 Line Breaking

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

- In a *hard* zone, `{\\`a`}` always causes a line break.
- In a *soft* zone, either none or all of the `{\\`a`}` are taken.
- In a *linear* zone, `{\\`a`}` never causes a line break. Instead, its position is filled by the text string a, which usually is a sequence of blank characters.

Break zones are started and ended by zone delimiters. Display form format sequences should usually include matching start and end zone formats. There is one end delimiter `{]}` (`ezone`) for all kinds of zones. Each kind of zone has its own start delimiter:

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

A linear zone is special in that all zones nested inside are also forced to be linear. Therefore a linear zone contains no line-breaks and always is laid out on a single line. If a linear zone doesn't fit on a single line, the layout algorithm chooses subterms to elide (see Section 5.3) to try and make it fit.

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

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

7.2.3.4 Optional Spaces

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

7.2.4 Attributes

Attributes specify extra information about display form definitions. By default, display form definitions are created with a right-hand-side term, a standard sequence of formats, and a single alias attribute. Moving the cursor over the whole attribute term and using ⟨C-O⟩ or ⟨M-O⟩ will create additional attribute slots to the left or right of this attribute.

Possible display form attributes are summarized in the table below. The *Name* column gives the name that has to be entered into a term slot to create the attribute, while the *Display* column describes how it will be presented within a display form definition.

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

The `alias` attribute provides an alternate name which the input editor recognizes as referring to the definition. Alternate names are often convenient abbreviations for the full names of definitions. The iteration attributes `ithd` and `ittl` control selection of a definition by the display layout algorithm. They are used to come up with convenient notations for iterated structures, which are discussed in Section 7.2.4.1. The `parens` and `prec` attributes affect automatic parenthesization, described in Section 7.2.4.2. The `index` attribute together with the name of the object containing a definition give a unique name for the definition. Conditions specify requirements for using a display form definition. Each condition $c_1, \ldots, c_n$ in the `conds` term is a term with a alpha-numeric label associated with the display form definition.

7.2.4.1 Iteration

The iteration attributes control choice of display form definition based on immediately-nested occurrences of the same term. The idea is to group occurrences into *iteration families*. An iteration family has a *head* display form definition and one or more tail definitions. A tail definition can only be used as an immediate subterm of a head in the same family or another tail in the same family. Choice of display form is also affected by the use of the *iterate variable* `#` as the *id* of a term slot format (Section 7.2.3.1). If `#` is used in some term slot of a definition, then the definition is only usable if the same term occurs in the subterm slot that uses the `#`.

The following set of display forms for λ abstraction terms, for instance, makes sure that the λ character is suppressed on nested occurrences:

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

Using these, the term `lambda(x.lambda(y.lambda(z.x)))` will be displayed as ⌞`λx,y,z.x`⌟ instead of ⌞`λx.λy.λz.x`⌟.

7.2.4.2 Parenthesization

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

Precedence Objects collectively introduce a set of precedence elements, and define a partial order on them. The components of a precedence object and the names used to enter them by are summarized in the table below. The `prser`, `preq`, and `prpar` terms are sequence constructors that may be nested. The standard editor commands described in Section 5.4.2 work on the sequences built with these terms.

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

Object names and object pointers are the primitive elements in a precedence order. Serial precedence terms impose a linear order on a set of precedences $p_1 \ldots p_n$. Equal precedence term declare all precedences p_i to be equal in the precedence order. Parallel precedence terms declare all precedences p_i to have the same "rank" in the precedence order while being unrelated to each other.

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

Examples of a base set of precedences set up for the current NUPRL theories can be found in the standard theory `core_1`.

Automatic Parenthesis Selection. The parenthesization of a term slot of a display form is controlled by the *parenthesis slot-option*, i.e. the third field of the term slot in the display form definition (see Section 7.2.3.1), by the `parens` attribute of the display form filling that term slot, and by the relative precedences of the term slot and the term filling it. The precedence of a term slot is usually that of the display form containing it, although it is possible to assign precedences to individual slots. The parenthesis control works as follows:

- Term slots are parenthesized only if the filling display form has a `parens` attribute. If this attribute is absent, the slot is never parenthesized. The `parens` attribute must be explicitly added to a display form definition for that definition to ever be parenthesized.
- Term slots are parenthesized unless parenthesization is suppressed by the parenthesis slot-options. These options have the following meanings:

  `L` : Suppress parentheses if the display-form precedence is *less than* the display-form precedence of the term filling the slot.

  `E` : Suppress parentheses if the display-form precedence is *less than or equal to* the display-form precedence of the term filling the slot.

  `*` : Always suppress parentheses.

The `L` and `E` options make it possible to represent the conventional precedences and associativity laws of standard infix operators. If they are used in the definitions of display forms for the arithmetic terms `plus(`a`;`b`)`, and `times(`a`;`b`)`, then the term `plus(`a`;times(`b`;`c`))` is displayed as $\llcorner a + b * c \lrcorner$, but `times(`$a$`;plus(`$b$`;`$c$`))` is displayed as $\llcorner a * (b + c) \lrcorner$. Similarly, `function(`$A$`;function(`$B$`;`$C$`))` is displayed as $\llcorner A \rightarrow B \rightarrow C \lrcorner$, but `function(function(`$A$`;`$B$`);`$C$`)` is displayed as $\llcorner (A \rightarrow B) \rightarrow C \lrcorner$

The `L`, `E` and `*` characters in the display of term slot formats are display forms for parenthesization control terms. To change the parenthesis slot-options, one may delete the term and enter the new option using the names shown in the table below.

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

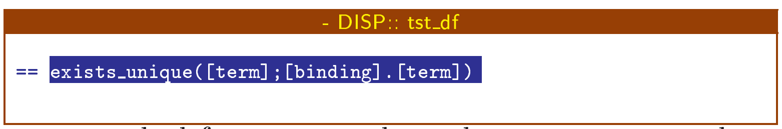
The parenthesization control terms also allow the specification of the delimiter characters used for parenthesization, and a precedence for the individual slot. No specific editor support has yet been provided for these features.

7.2.5 Examples

As an example, we walk through the entry of a display form definition from scratch. We start by creating a new display form object and viewing it. Click the `MkObj*` button, enter ⌞`tst_df`⌟ as name and ⌞`disp`⌟ as kind, and click the `OK*` button.[4]

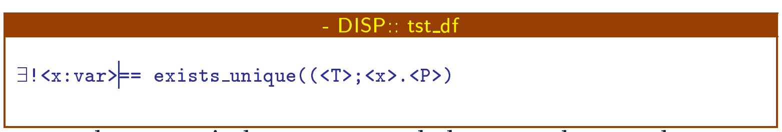
Open the object by pressing the right arrow or clicking on it with the mouse. This will pop up a window, containing a highlighted empty term slot. Initialize the display form definition by entering ⟨C-M-`I`⟩. The window now looks like:

We begin by entering the right-hand side of the display form. Click LEFT on the `[rhs]` placeholder, and enter ⌞`exists_unique(0;1)`⌟ to create a new term (see Section 5.4.4). Do not fill in the variable slot or either of the subterm slots. The definition should now look like:



Enter ↵ or click LEFT on the left-most term slot and enter `<T>`, `<x>`, and `<P>` as meta-terms and meta-variable, respectively, by typing ⌞`mterm`↵`T`↵`x`⟨C-M⟩↵`mterm`↵`P`⌟. As a result you get:

To create a display form for the term on the right hand side, click LEFT on the first `=` to get a text cursor in the empty format sequence on the left-hand side of the definition. Type ⌞⟨C-`#`⟩`163!`⟨C-`O`⟩`slot`↵`x`↵`var`⟨C-F⟩⌟ to generate an ∃ symbol and an exclamation mark as initial text and a slot for the variable `x`. The definition should now look like:



Enter a colon, the type slot, a period, a space, and the second term slot:

⌞`:`⟨C-`O`⟩`sslot`↵`T`↵`type`⟨C-F⟩⟨C-F⟩⟨C-F⟩`.` ⟨C-`O`⟩`eslot`↵`P`↵`prop`⌟

The definition should now look like:

[4]Note that the `MkObj*` button is not present in user theories. The only way to create display forms is through the `AddDef*` and `AddDefDisp*` buttons, which already generate a right-hand-side term, a standard sequence of formats, and an alias attribute.

```
- DISP:: tst_df

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

The display form definition is now complete and may be saved by closing the display form object with ⟨C-Z⟩. However, the definition does not yet include line breaking or parenthesization information. In particular, it does not contain any visible delimiter for the end of the `prop` slot.

We therefore want the layout algorithm to automatically parenthesize the display form. To add parenthesizing attributes, click LEFT on the second `=` character to get a term cursor over the whole definition, and then enter ⌞⟨C-M-S⟩ ↵ ⟨C-0⟩⌟ to get two empty attribute slots, with a term cursor over the first:

```
- DISP:: tst_df

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

To instantiate the attribute slots enter ⌞`parens` ↵ `prec` ↵ `exists`⌟ To get:

```
- DISP:: tst_df

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

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

We may also add a soft-break format such that the period separating the `type` slot from the `prop` slot does not appear if a break is taken. Click LEFT on the ⌞.⌟ character and delete it using ⟨C-D⟩. Enter ⌞⟨C-0⟩`sbreak` SPC⌟, click LEFT on the } after the `?` character in the soft break display form, and enter ⌞.⌟.

```
- DISP:: tst_df

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

Finally, we add a second display form for the term $\exists!x{:}T.P_x$, which drops the type information from the display whenever the type is $\mathbb{N}$.

Click LEFT on the second `=` character to get a term cursor over the whole definition, and then enter ⌞⟨M-0⟩⌟ to get a second initial display form definition *after* it.

```
- DISP:: tst_df

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

== [rhs]
```

Copy the first definition into the second as follows: enter ⟨C-K⟩ to replace the initial display form by an empty term slot, move the term cursor over the whole first definition, copy it with ⟨M-K⟩, move the term cursor back over the empty term slot, and paste the first definition with ⟨C-Y⟩.

```
- DISP:: tst_df

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

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

On the right hand side of the definition, replace the meta-term `<T>` by the type constant ℕ. Click LEFT on the `<` character, enter ⟨C-K⟩ and then type ⌞`nat`↵⌟ to enter the type ℕ.

On the left hand side, remove the term slot for `T` by clicking LEFT on the `>` character (!) and entering ⟨C-K⟩. Remove the colon with BACKSPACE.

```
- DISP:: tst_df

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

Parens::Prec(exists)::∃!<x:var>{\?.} <P:prop:E>
 == exists_unique((ℕ;<x>.<P>)
```

In a similar way, one may add further display form definitions with iteration families to suppress the ∃! string and duplicate type information in nested occurrences of the term $\exists!x{:}T.P_x$. The display form object `exists_df` in the standard theory `core_1` can be used as an example for doing that.

Chapter 8

Rules and Tactics

Logically, all formal reasoning in NUPRL is based on the basic inference rules of *intuitionistic type theory* [ML84, CAB+86]. In practice, however, reasoning with basic inference rules can become very tedious. Therefore NUPRL provides the concepts of proof *tactics*, i.e. programmed applications of inference rules. Proof tactics range from straightforward applications of inference rules to fully automated proof search mechanisms for certain domains. Most tactics are deal with intellectually trivial, but formally lengthy proof details, enabling the user to concentrate on the more interesting aspects of a proof. Others mimic the particular style of reasoning in a certain application domain and can only be applied in this context. Technically, all refinement steps in NUPRL are executions of tactics. Basic inference rules, although explicitly present in the NUPRL library, cannot be invoked directly but only through conversion into tactics.

In the rest of this chapter we will describe the structure of basic inference rules and tactics as well as the most important groups of tactics that are currently implemented. NUPRL's type theory, its semantics, and the guiding principles for its development is explained in [CAB+86]. It should be noted, however, that NUPRL's type theory is *open-ended* and that fundamentally new concepts and their inference rules are added whenever this is turns out to be necessary.

A complete list of the current set of inference rules can be found in Appendix A.3.

8.1 Rules

Inference rules characterize the semantics of all formal expressions in NUPRL. For each type construct they describe how to form types and the conditions for two types to be equal, how to form members of types and the prerequisites for two members of a type to be equal in that type.

NUPRL's proof calculus is based on the notion of *sequents*. These are objects of the form $x_1{:}T_1, \ldots, x_n{:}T_n \vdash C$, which should be read as "Under the *hypotheses* that the x_i are variables of type T_i a member of the *conclusion* C can be constructed" (see Section 6.1.1 for details).

In NUPRL a proof for such a sequent is developed in a *top-down* fashion. Proof rules *refine* a goal sequent obtaining subgoal sequents whose proofs would suffice to validate the original goal. They are described by rule schemata with placeholders for lists of hypotheses and conclusions. A rule

$\Gamma \vdash C \quad \lfloor$**ext** $m\rfloor$
  by *rule-name (& optional arguments)*
  $\Gamma_1 \vdash C_1 \quad \lfloor$**ext** $m_1\rfloor$
  $\ldots$
  $\Gamma_n \vdash C_n \quad \lfloor$**ext** $m_n\rfloor$

expresses the fact that the goal sequent $\Gamma \vdash C$ is provable if all the subgoals $\Gamma_i \vdash C_i$ are. Furthermore, it also describes how to construct an (implicitly present) member m of C from members m_i of the subgoal conclusions C_i.

Rules may operate both on the conclusion or the hypotheses of a proof goal. In most cases they simply decompose a type or a member of a type into smaller components. The rule for the formation of pairs, for instance, states that in order to form a member $\langle s, t\rangle$ of a product type $S\times T$ it suffices to form a member s of the type S and a member t of T.

$\Gamma \vdash S\times T$ $\lfloor$**ext** $\langle s, t\rangle\rfloor$
  by independent_pairFormation
  $\Gamma \vdash S$ $\lfloor$**ext** $s\rfloor$
  $\Gamma \vdash T$ $\lfloor$**ext** $t\rfloor$

This rule can be applied to any goal whose conclusion is a product type. In this case S is matched against the left, and T against the right component of that product, while Γ is matched against the complete list of hypotheses. As a result, Nuprl generates two new proof goals, the first with the left component as proof goal and the second one with the right component. The list of hypotheses remains unchanged in both cases.

8.1.1 Representation of Inference Rules

Nuprl's inference rules are not hard-wired into the code of the system but explicitly represented in the library as objects of kind *rule*. Rule definitions are 'terms' in the sense described in Chapter 5 and they can be edited like any other term. The rule for the formation of pairs, for instance, is represented by an object called `independent_pairFormation`.

- RULE: independent_pairFormation
```
H  ⊢ A × B ext <a, b>
  BY independent_pairFormation ()

  H  ⊢ A ext a
  H  ⊢ B ext b
```

The fact that the rule object is almost identical to the rule described on paper, makes it very easy to verify the implementation of intuitionistic type theory in Nuprl. The explicit representation of inference rules in Nuprl also allows a user to modify the logic represented in system by adding new rules or deactivating existing ones. Thus Nuprl supports any logic that can be represented as a top-down sequent calculus.

Nuprl provides the basic mechanisms for applying rule objects to proofs. In most cases this means matching the rule's top goal against the current proof goal and extending the proof tree by the instantiated sub-goals of the rule. Some rules, such as the decision procedure `arith` provide explicit calls to special purpose algorithms that will be executed when Nuprl applies the rules.

- RULE: arith
```
H  ⊢ C ext t
  BY arith 𝕌

     Let SubGoals t = CallLisp(ARITH)
  SubGoals
```

8.1.2 Rule Arguments

In most cases, the application of an inference rule to a proof goal requires more information than just the name of the rule. For instance, if a rule shall be applied to one of the hypotheses, it is necessary to identify this hypothesis; or if a rule creates new variables in the subgoals, it is necessary to give names to these variables. Because inference rules are supposed to be applied schematically, this information will not be determined automatically but has to provided as additional arguments. Inference rules may require the following arguments.

Hypothesis index. If a rule shall be applied to a particular hypothesis of the proof goal, the corresponding index i of that hypothesis in the hypotheses list of the goal must be provided. The rule `hypothesis`, for instance, can be used to prove a goal if the conclusion is identical (α-equal) to one of the assumptions, but the index of that assumption must be given.

$\Gamma, x{:}T, \Delta \vdash T$ $\lfloor$**ext** $x\rfloor$
  by `hypothesis` $\boxed{i}$

- RULE: hypothesis

```
H x:A, J ⊢ A ext x
  BY hypothesis #$i
  No Subgoals
```

Variable names. Many inferences require the creation of new variables when refining a proof goal. The corresponding proof rules do not choose the names for these variables automatically but expect them to be provided explicitly. The rule `functionEquality`, for instance, is used to prove two dependent function types $x_1{:}S_1{\to}T_1$ and $x_2{:}S_2{\to}T_2$ equal. It creates two subgoals: the two domains S_1 and S_2 must be equal, and the two ranges $T_1[x/x_1]$ and $T_2[x/x_2]$ must be equal for each argument $x \in S_1$. The second subgoal contains a new variable, whose name must be given.

$\Gamma \vdash x_1{:}S_1{\to}T_1 = x_2{:}S_2{\to}T_2 \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
  by `functionEquality` $\boxed{x}$
    $\Gamma \vdash S_1 = S_2 \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
    $\Gamma, x{:}S_1 \vdash T_1[x/x_1] = T_2[x/x_2] \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$

- RULE: functionEquality

```
H  ⊢ (x1:a1 → b1) = (x2:a2 → b2)
  BY functionEquality y

  H  ⊢ a1 = a2
  H y:a1 ⊢ !subst(b1; x1.y) = !subst(b2; x2.y)
```

Universe level. Because of the expressiveness of type theory, the well-formedness of a type expression cannot be decided automatically, but must be established in the course of a proof development. Whenever a rule creates a new declaration in one of the its subgoals, the well-formedness of the corresponding type must be established. If this cannot not guaranteed by the proof of the other subgoals (as in the case of `functionEquality` it must be proven as a separate subgoal. The rule `lambdaEquality`, for instance, proves the equality of two λ-terms in a function type $x{:}S{\to}T$. For this purpose, it declares a new variable X' of type S and adds a subgoal stating that S belongs to some universe $\mathbb{U}_j$. The level j of that universe must be given.

$\Gamma \vdash \lambda x_1.t_1 = \lambda x_2.t_2 \in x{:}S{\to}T$ $\lfloor$Ax$\rfloor$
  by `lambdaEquality` $\boxed{j}$ x'
    $\Gamma, x'{:}S \vdash t_1[x'/x_1] = t_2[x'/x_2] \in T[x'/x]$ $\lfloor$Ax$\rfloor$
    $\Gamma \vdash S \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$

- RULE: lambdaEquality

```
H  ⊢ (λx1.b1) = (λx2.b2)
  BY lambdaEquality !parameter{i:l} z ()

  H z:A ⊢ !subst(b1; x1.z) = !subst(b2; x2.z)
  H  ⊢ A = A
```

Values for variables. Some proof goals can only be decomposed if it is known how to instantiate a certain variable. To prove $\exists x{:}S.T[x]$ – which is the same as the dependent product $x{:}S{\times}T[x]$), for instance, one has to provide a value s for the existentially quantified variable x and can then go on to prove $T[s]$. The corresponding rule `dependent_pairFormation` requires this term as one of its arguments.

$\Gamma \vdash x{:}S{\times}T$ $\lfloor$**ext** $\langle s,t\rangle\rfloor$
  by `dependent_pairFormation` j $\boxed{s}$ x'
    $\Gamma \vdash s \in S$ $\lfloor$Ax$\rfloor$
    $\Gamma \vdash T[s/x]$ $\lfloor$**ext** $t\rfloor$
    $\Gamma, x'{:}S \vdash T[x'/x] \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$

- RULE: dependent_pairFormation

```
H  ⊢ x:A × B ext <a, b>
  BY dependent_pairFormation !parameter{i:l} a y ()

  H  ⊢ a = a
  H  ⊢ !subst(B; x.a) ext b
  H y:A ⊢ !subst(B; x.y) = !subst(B; x.y)
```

Type of a subterm. Sometimes decomposing a proof goal into smaller components involves proving that certain subterms belong to a type that cannot immediately constructed from the types mentioned in the goal. Proving the equality of two function applications $f_1 t_1$ and $f_2 t_2$ in a type T, for instance, requires proving f_1 and f_2 equal in some type $x{:}S{\rightarrow}T$. As this type cannot be derived from T it must be given explicitly.

$\Gamma \vdash f_1 t_1 = f_2 t_2 \in T[t_1/x]$ ⌊Ax⌋
by `applyEquality` $\boxed{x{:}S{\rightarrow}T}$
$\Gamma \vdash f_1 = f_2 \in x{:}S{\rightarrow}T$ ⌊Ax⌋
$\Gamma \vdash t_1 = t_2 \in S$ ⌊Ax⌋

- RULE: applyEquality

```
H  ⊢ (f1 a1) = (f2 a2)
  BY applyEquality x:A → B

  H  ⊢ f1 = f2
  H  ⊢ a1 = a2
```

Term dependency. Applying a proof rule may sometimes involve replacing an sub-expression e in the goal by some other expression. If e occurs several times in the goal, one must indicate which of the occurrences of e shall be replaced and which one shall not. Technically, this is done by providing a term substitution: all occurrences of e in the term C that shall be affect by the rule application are replaced by a new variable z. The rule will then substitute every occurrence of z by the new expression. Both z and the modified term $C[z]$ must be given as arguments. As an example, consider the rule `decideEquality` for proving two case analyses equal.

$\Gamma \vdash \text{case } e_1 \text{ of inl}(x_1) \mapsto u_1 \mid \text{inr}(y_1) \mapsto v_1 = \text{case } e_2 \text{ of inl}(x_2) \mapsto u_2 \mid \text{inr}(y_2) \mapsto v_2 \in C[e_1/z]$ ⌊Ax⌋
by `decideEquality` $\boxed{z\ C}$ $S{+}T$ s t y
$\Gamma \vdash e_1 = e_2 \in S{+}T$ ⌊Ax⌋
$\Gamma,\ s{:}S,\ y{:}\, e_1{=}\text{inl}(s) \in S{+}T \vdash u_1[s/x_1] = u_2[s/x_2] \in C[\text{inl}(s)/z]$ ⌊Ax⌋
$\Gamma,\ t{:}T,\ y{:}\, e_1{=}\text{inr}(t) \in S{+}T \vdash v_1[t/y_1] = v_2[t/y_2] \in C[\text{inr}(t)/z]$ ⌊Ax⌋

- RULE: decideEquality

```
H  ⊢ case e1 of inl(x1) => l1 | inr(y1) => r1 = case e2 of inl(x2) => l2 | inr(y2) => r2
  BY decideEquality z T (A + B) u v w

  H  ⊢ e1 = e2
  H u:A, w:(e1 = (inl u )) ⊢ !subst(l1; x1.u) = !subst(l2; x2.u)
  H v:B, w:(e1 = (inr v )) ⊢ !subst(r1; y1.v) = !subst(r2; y2.v)
```

In most cases, the arguments of an inference rule can easily be determined from the proof goal. NUPRL's tactic collection therefore provides a set of single-step inference tactics that try to compute the arguments required by the corresponding inference rule from the actual proof context before executing the rule (see Section 8.1.3 below). In some situations, however, the relation between the proof goal and the arguments of inference rule is not so obvious and it is necessary that the user provides the rule arguments explicitly in order to be able to complete the proof.

8.1.3 Converting rules into tactics

NUPRL's basic mechanism for creating and modifying proofs is the application of proof *tactics*. These are functions that take a sequent (i.e. the goal) and generate a list of sequents (i.e. the subgoals) as well as a *validation*, which shows that a proof for the main goal can be constructed from proofs for all the subgoals.

Basic inference rules cannot be applied directly to a proof goal, but first have to be converted into a tactic. Technically, this is done by calling a meta-level function `refine` that takes a primitive rule, that is the name of a rule object and the corresponding rule arguments, and generates a tactic that executes the rule. The function `refine` encodes NUPRL's mechanism for applying rule objects to proofs (c.f. Section 8.1.1 above) and is the only method for creating proof tactics from scratch.

This guarantees that all proof steps are eventually based on primitive inference rules and thus correct with respect to the implemented proof calculus.

In most cases, the applicable inference rule and its arguments can easily be determined from the proof context. NUPRL's library therefore contains a small collection of single-step inference tactics that subsumes the complete set of elementary inference rules. Almost all primitive inferences can be expressed the single-step decomposition tactics `D`, `MemCD`, `EqCD`, `MemHD`, `EqHD`, `MemTypeCD`, `EqTypeCD`, `MemTypeHD`, `EqTypeHD`, and `NthHyp`. These tactics, which are described in detail in Section 8.3.1, uniformly apply to both the hypotheses and the conclusion of a proof goal. Often a user only has to give the index of the *goal clause* to which they shall be applied. The tactic then analyzes the syntactical structure of the indicated clause, identifies the applicable rule, and tries to determine its arguments from the proof context.

For the sake of efficient interaction, the heuristics built into these tactics only determine variable names, universe levels, types of subterms, and term dependencies, while the user always has to provide the clause index and values to be substituted for variables.[1] A user may also want to override the choices made by the single-step decomposition tactics. For this purpose NUPRL provides a collection of *tacticals* (i.e. functions that take tactics as arguments and generate tactics) that enable a user to supply rule arguments explicitly (see Section 8.2.2 for details). Arguments may be supplied to tactics as follows.

Clause index. Most single-step tactics require a clause index to be given as explicit argument. By convention (see Section 8.2.1.1 for a discussion) the index `0` stands for the conclusion of the proof goal, while positive numbers indicate hypotheses. As a convenience, negative indices can be used to count backwards from the end of the hypothesis list. The index `(-1)` indicates the last hypothesis in the goal, `(-2)` the second to last, etc.

The rule `independent_pairFormation`, for instance, operates on the conclusion and is represented by the single-step tactic `D 0`. The rule `hypothesis` i is represented by `NthHyp` i.

Variable names. All tactics in NUPRL's library automatically assign names to new variables. In rare cases a user may want to select different names. The tactical `New` takes as an input a list of variables and a tactic and generates a tactic that uses the provided variables instead of the automatically chosen ones.

The the tactic `D 0`, which usually suffices to represent the rule `functionEquality` x, can be forced to choose the name x for the new variable by writing `New` $[x]$ (D 0).

Universe level. Universe levels can usually be determined from the immediate proof context and the well-formedness theorems of user-defined concepts. In some cases, the heuristics choose a rather arbitrary, high universe level. To enforce a particular level, one may use the tactical `At`.

The the tactic `D 0`, which usually suffices to represent the rule `lambdaEquality` $j\ x'$, can be forced to choose the universe level j by writing `At` $\mathbb{U}_j$ `(D 0)`. In addition, one could also enforce the use of x' as new variable name as in `At` $\mathbb{U}_j$ `(New` $[x']$`(D 0))`

Values for variables. The `With` tactical can be used to supply terms that are needed for the instantiation of variables. The rule `dependent_pairFormation` $j\ s\ x'$, for instance, requires the term s to be provided explicitly. It is represented by the tactic `With` s `(D 0)`. A universe level and a variable name may also be provided by writing `At` $\mathbb{U}_j$ `(With` s `(New` $[x']$ `(D 0)))`.

[1]NUPRL's library contains proof tactics that attempt to determine clause index and values for variables automatically. But since the underlying heuristics are often time consuming and sometimes choose values leading to subgoals that cannot be proven, they are not suited for representing single-step inferences.

Type of a subterm Although type checking in general is undecidable for intuitionistic type theory, the types of subterms occurring in a proof goal can almost always be determined automatically. A user may override the heuristically chosen type if it is too coarse for completing the proof or provide a type explicitly to improve the efficiency of proof checking. The tactical `With` can be used for this purpose as well.

The tactic MemCD, which usually suffices to represent the rule `applyEquality` $x{:}S{\to}T$, can be forced to use a particular function type $x{:}S{\to}T$ by writing `With` $x{:}S{\to}T$ `MemCD`.

Term dependency. Term dependencies can be supplied to a tactic with the `Using` tactical. The tactic `D 0`, which usually suffices to represent the rule `decideEquality` z C $S{+}T$ s t y, can be forced to use a particular dependency $C[z]$ with a new variable z by writing `Using [`z`,`C`] (D 0)`. Note that a disjunctive type and up to three new variables could be provided as well, as in `Using [`z`,`C`] (With` $S{+}T$ `(New [`s`;`t`;`y`] (D 0)))`.

Rule selection. In some rare cases it is impossible to determine the exact rule that shall be applied to a particular proof clause. For instance, there are two different rules can be applied to a disjoint union $S{+}T$ in the conclusion of a goal.

$\Gamma \vdash S{+}T$ $\lfloor$**ext** $\mathsf{inl}(s)\rfloor$	$\Gamma \vdash S{+}T$ $\lfloor$**ext** $\mathsf{inr}(t)\rfloor$
by `inlFormation` j	**by** `inrFormation` j
$\Gamma \vdash S$ $\lfloor$**ext** $s\rfloor$	$\Gamma \vdash T$ $\lfloor$**ext** $t\rfloor$
$\Gamma \vdash T \in \mathbb{U}_j$ $\lfloor\mathsf{Ax}\rfloor$	$\Gamma \vdash S \in \mathbb{U}_j$ $\lfloor\mathsf{Ax}\rfloor$

There is no simple heuristic for determining from the proof context which of the two must be applied (although this is possible in limited application domains). Instead, a user has to *select* between the two rules, using the tactical `Sel`. The rule `inlFormation` j will be executed by writing `Sel 1 (D 0)`, while `Sel 2 (D 0)` leads to the execution of `inrFormation` j. In both cases a universe level can also be supplied.

A complete description of all the inference rules currently present in the Nuprl system as well as the corresponding single-step tactics can be found in Appendix A.3.

8.2 Introduction to Tactics

The creation and modification of proofs in Nuprl is based on the concept of *tactics*. Logically, tactics are functions that represent valid refinements. They convert a proof goal into a list of subgoals such that the validity of all the subgoals implies the validity of the initial proof goal. They range from elementary inference steps to sophisticated proof strategies that can solve major proof problems automatically.

Technically, tactics are functional programs written in the ML programming language (see Appendix B). They take as input a proof goal and return a list of proof goals and a *validation*. The validation is a function that constructs a proof for the initial goal from proofs for the subgoals and thus validates the refinement step performed by the tactic.

Tactics are usually initiated from within the proof editor. After the user types in the tactic the proof editor applies it to the current proof goal. The proof goals generated by the tactic will be added to the proof tree as *children* of the current proof node and displayed as subgoal sequents still to be proven. The validation is stored in the proof node as well but remains invisible. Upon completion of the proof the validations of all proof nodes are composed into a proof for the initial proof goal, which provides computational evidence for the validity of the initial theorem.

Tactics can either be built from elementary inference rules using the function `refine` (see Section 8.1.3), or by composing existing tactics into new ones using tacticals (see Section 8.8) or more sophisticated ML programs that analyze the proof context before initiating a refinement step. This makes sure that all refinements performed by tactics are eventually based on elementary inference rules and thus correct with respect to the underlying logic.[2] Applying a tactic to a proof goal may produce incomplete proofs or not terminate, but it *always results in a valid proof*.

NUPRL's standard library contains a large collection of useful tactics that have been developed over the past 15 years. Users may extend that collection by adding their own tactics as code-objects to their personal directories. But in most cases it is sufficient to use the existing tactics and to combine them through tacticals.

In the rest of this chapter we will describe the most important standard tactics contained in the library. Further tactics may be found by inspecting NUPRL's library of standard theories.

8.2.1 Tactic Arguments

Invoking a tactic often requires certain arguments to be supplied. These may be indices of hypotheses to which the tactic shall be applied (type `int`), NUPRL terms that instantiate variables (type `term`), new variables to be generated (type `var`), universes to be used in well-formedness goals (type `term`), names of library objects to be consulted (type `token`), substitutions to be applied (type `(var#term) list`), sub-tactics to be used during refinement (type `tactic`), and others.

The proof goal (type `proof`) to which the tactic shall be applied is always the last argument of a tactic. If the tactic is invoked in the proof editor, the user does not have to supply this argument, as the proof editor will automatically insert it.

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

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

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

8.2.1.1 Referring to hypotheses in a sequent

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

[2] An experienced NUPRL hacker will, of course, find ways to bypass these mechanisms and modify proofs without using elementary inference rules. However, the dependency tracking mechanisms of NUPRL's library are able to detect theorems whose proofs were constructed that way and to identify all library objects that depend on such theorems. Thus it is possible to account for the validity of theorems even in the presence of hacks.

There are tactics which work in similar ways on both hypotheses and the conclusion. In this case, we call the hypothesis and conclusion collectively *clauses*, refer to the conclusion as *clause 0*, and to hypothesis i ($i \neq 0$) as *clause* i.

As a convention in this manual, we prefix a hypothesis with a number followed by a period if we want to indicate explicitly the number of a hypothesis in a schematic sequent. For example, if hypothesis i is proposition P, we write the hypothesis as $i.\,P$.

8.2.1.2 Universes and Level Expressions

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

Since propositions are encoded as types, propositions reside in universes too. In keeping with the propositions-as-types encoding, we define a family of propositional universe abstractions $\mathbb{P}_1$, $\mathbb{P}_2$, $\ldots$, which unfold to the corresponding primitive type universe terms $\mathbb{U}_1$, $\mathbb{U}_2$, $\ldots$.

If one is only allowed to use constant levels for universes, one often has to choose arbitrarily levels for theorems. One would then find that one needed theorems that were stated at a higher level, and would have to reprove those theorems. This was the case in NUPRL 3 and earlier releases.

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

$$L \;::=\; v \;\mid\; k \;\mid\; L\;i \;\mid\; L' \;\mid\; [L \,|\cdots|\, L]$$

The v are level-expression variables, which can be arbitrary alphanumeric strings. They are implicitly quantified over all positive integer levels. The k are level expression constants, which can be arbitrary positive integers. The i are level expression increments and must be non-negative integers. The expression $L\;i$ is interpreted as standing for levels $L{+}i$. L' is an abbreviation for L `1`. The expression $[L_1 |\cdots| L_n]$ is interpreted as being the maximum of the expressions $L_1 \cdots L_n$.

Usually when stating theorems, only level expressions of the form v and v' need be used. Other expressions get automatically created by tactics. Further, it is sufficient to use a single level-expression variable throughout a theorem statement, as two occurrences of the same level-expression variable will not be related. For example, we normally prove the theorem $\forall A, B{:}\mathbb{P}_i.\; A \Rightarrow (B \Rightarrow A)$ rather than $\forall A{:}\mathbb{P}_i.\forall B{:}\mathbb{P}_j.\; A \Rightarrow (B \Rightarrow A)$.

8.2.2 Optional Arguments

Unlike Lisp functions, ML functions cannot take optional arguments, although it is natural to want to write tactics which do take optional arguments. One approach is to provide a set of variants of each tactic for the most common combinations of arguments. This can be confusing, and places an extra burden on the user who has to keep track of these variants.

NUPRL allows optional arguments to be passed to tactics by providing tacticals (see Section 8.8 below) that attach these arguments to the proof argument (i.e. the last argument) of a tactic (c.f. Section 8.2.1). Currently, this is supported for arguments of type `int`, `tactic`, `term`, `tok`, `var` and `(var#term) list`. Each argument is also given a token label, and arguments are looked up by these labels. Sets of arguments are maintained on a stack, which enables nesting of tactics that use optional arguments. Tacticals for manipulating these arguments are:

`New` `([`v_1`;...;`v_n`] :var list)` T

Runs tactic T with variables v_1 to v_n as optional arguments, Typically, `New` is used to supply a tactic with names for newly created variables. The argument labels of the v_i are `‘v1‘` to `‘vn‘`.

`With` (t `:term`) T

Runs tactic T with term t as optional argument, which may, for instance, be a term to be instantiated for a variable or the type of some subgoal. The argument label of t will be `‘t1‘`.

`At` (U `:term`) T

Runs tactic T with term U as a optional `‘universe‘` argument. U should either be either a type universe or a propositional universe term.

`Using` (sub `:(var#term) list`) T

Runs tactic T with the substitution sub as optional `‘sub‘` argument. The substitution sub may be applied to instantiate variables or to indicate dependencies in some term.

`Sel` (k `:int`) T

Runs tactic T with the integer k as optional argument. `Sel` is used for selecting a simple component of a formula or a subterm of a term. The argument label of k will be `‘n‘`

`Thinning` T

Runs tactic T with the token `‘yes‘` as optional `‘thinning‘` argument. Some tactics may optionally remove (or *thin*) hypotheses that are considered superfluous after a refinement step. `Thinning` causes T's default behavior to be to thin.

`NotThinning` T

Runs tactic T with the token `‘no‘` as optional `‘thinning‘` argument. This causes T's default behavior to be to not thin.

`WithArgs` ($args$ `:(tok#arg) list`) T

Run tactic T with the arguments in $args$ on the top of the stack. `arg` is an ML abstract data type, defined as the disjoint union of the types `int`, `tactic`, `term`, `tok`, `var` and `(var#term) list`. There are injection and projection functions for each of these types, such as `int_to_arg` and `arg_to_int`. All the above tacticals are special cases of `WithArgs`

Each tactic description in this chapter includes information on the optional arguments (if any) that it takes. Note that some tactics do useful preprocessing on some of their arguments. In these cases there would be a performance penalty if such arguments were supplied.

8.2.3 Proof Annotations

Nuprl *proof* terms can be annotated with extra information that is not relevant to the logical correctness of a proof but may assist in structuring proofs and identifying applicable tactics.

Goal Labels. Nuprl tactics generate various kinds of subgoals. Some express the main proof idea, while others are only auxiliary or well-formedness goals. In inductive proofs one distinguishes the base case from the step cases. Usually, different kinds of goals have to be treated differently, so one would like subsequent tactics to discriminate on subgoal kind. Sometimes a subgoal's kind can be deduced directly from its structure, but this can be a error-prone process. Thus the tactics that generate the subgoals often attach explicit labels to them indicating their kind. Labels consist of an ML token and an optional number. Typical examples of labels are **main**, **upcase**, **aux**, and **wf**. Sometimes tactics generate a set of subgoals of the same kind, where the order of the subgoals is important. The optional numbers are used to discriminate between these subgoals. Most descriptions of tactics include information on subgoal labeling. It is also a simple matter to find out what labels are generated by experimentation.

```
- PRF: int_mul
# top
∀x:ℤ. x * x ≥ 0
BY D 0
# 1
1. x:ℤ
⊢ x * x ≥ 0
BY
# 2
.....wf.....
ℤ ∈ 𝕌1
BY
```

For historical reasons, goal labels are sometimes known as *hidden labels*.[3] Labels may be added explicitly using the following tactics.

`AddHiddenLabel` *lab*

Add label *lab* to the current goal.

`AddHiddenLabelAndNumber` *lab i*

Add label *lab* to the current goal along with the integer label i.

`RemoveHiddenLabel`

Remove the label from the current goal.

Note that the goal label created by `AddHiddenLabelAndNumber 'x' 1` is ⌞x⌟ and not ⌞x 1⌟, although it will be displayed as the latter. The number is considered as strictly separate from the label and will only be considered by tactics that discriminate on the number as well. Removing a label is equivalent to adding the label `main`. This label will not be displayed, since unlabeled goals are usually considered main goals.

To make subsequent tactics discriminate on labels one usually applies the tacticals described in Section 8.8.2 below. For convenience, labels are divided into the classes *main* and *aux*. The discriminating tacticals allow one to select either subgoals with a particular label, or subgoals of one of these two classes.

8.2.4 Soft Abstractions

In most proofs tactics do not deal with basic expressions of NUPRL's logic but with user-defined concepts. Before applying generic tactics to these, the corresponding abstractions need to be unfolded first. To prevent unwanted unfolding of abstractions, tactics usually do not unfold abstractions unless they are designated as *soft*.

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

Specific tactics that unfold soft abstractions are `MemCD`, `EqCD`, `NthHyp`, `NthDecl` (Section 8.3.1), `Eq` (Section 8.3.3), `Inclusion`. (Section 8.7), the forward and backward chaining tactics (Section 8.4), and the atomic rewrite conversions based on lemmata and hypotheses (Section 8.9.2).

Table 8.1 lists the most important soft abstractions in NUPRL's standard libraries. The logic abstractions (`and`, `or`, `implies`, `exists`, `all`) are made soft because the well formedness rule for the underlying primitive term is simpler and more efficient than the well formedness lemma would

[3] In NUPRL 3 labels used not to be visible when editing proofs with the proof editor.

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

Table 8.1: Soft abstractions in NUPRL's basic libraries

be. The softness is also useful when one wishes to blur the distinction between propositions and types, for example when reasoning explicitly about the inhabitants of propositions. `member`, `nequal`, `rev_implies`, `ge` and `gt` are soft principally because it can simplify matching.

Abstractions are not soft by default. They can be declared soft or hard by supplying their opids to the functions

`add_soft_abs` *abs*

Declare the abstractions in the token list *abs* as soft.

`remove_soft_abs` *abs*

Declare the abstractions in the token list *abs* as hard.

Instances of these functions are usually kept in ML objects in close proximity to the abstraction definitions that they are declaring soft. For an example use of `add_soft_abs`, see the object `soft_ab_decls` in the `core_2` theory.

8.2.5 Universal Formulas

Many of NUPRL's tactics work on a specific subclass of logical formulas generated by the grammar

$$P \quad ::= \quad \forall x{:}T.\ P \mid P_a \Rightarrow P \mid P \Leftarrow P_a \mid P \wedge P \mid P \Leftrightarrow P \mid C$$

where T is a type, and C is a propositional term not of the above form. We call these *universal formulas*, *positive definite formulas*, or *Horn clauses*. Formulas generates without the $\wedge$ and $\Leftrightarrow$ connectives are also called *simple universal formulas*. We call the proposition C a *consequent* and each P_a an *antecedent*. Occasionally we refer to the types T as *type antecedents*.

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

Such formulas are the standard way of describing *derived rules of inference*, and are used as such by the forward and backward chaining tactics (see Section 8.4). Often, a consequent C of a formula will be an equivalence relation, in which case the formula can be used as a rewrite rule by the rewrite package (see Section 8.6).

Occasionally, one has a universal formula where the outermost constructor of C is also one of the constructors that make up the universal formula. In this case, one can surround C by a *guard* abstraction to designate it as consequent of the formula. A guard abstraction takes a single subterm as argument and unfolds to this subterm. The tactics that take apart universal formulas recognize and automatically remove guard abstractions, so the user rarely has to explicitly unfold them.

8.3 Basic Tactics

8.3.1 Single-Step Decomposition

Single-step decomposition tactics invoke the primitive *formation*, *equality*, and *elimination* rules of Nuprl's logic described in Appendix A.3. These rules describe how types and their members can be formed and when two types or members are equal by decomposing the corresponding terms in the hypotheses or the conclusion of a goal into smaller fragments. Structurally the effect is always the same, as the top-level terms are analyzed and their sub-terms will occur in the subgoals.

Each single-step decomposition tactic covers a large collection of primitive inference rules using a single name. The tactic name indicates which part of a hypothesis or conclusion will be decomposed.

`D` *c*

Decompose the *outermost connective* of clause *c*. `D` can take several optional arguments:

- A `‘universe‘` argument, usually supplied using the `At` tactical.
- A `‘t1‘` argument for a term (using `With`). This argument is necessary when decomposing a hypothesis with an outermost universal quantifier, or a conclusion with an outermost existential quantifier.
- `‘v1‘` and `‘v2‘` arguments for new variable names (using `New`. These are useful if one is not satisfied with the system supplied variable names.
- An `‘n‘` argument to select a subterm (using `Sel`). This is necessary when applying `D` to a disjunct in the conclusion.

Usually `D` unfolds all top level abstractions and applies the appropriate primitive (formation or elimination) rule. It is somewhat intelligent with instances of *set* and *squash* terms.

`ID` *c*

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

`MemCD`

Decompose the immediate subterm of a *membership term* in the *conclusion*.

```
- PRF: pairtest
# top
<3, 4> ∈ ℕ × ℤ

BY MemCD

# 1
.....subterm..... T:t
⊢ 3 ∈ ℕ

BY

# 2
.....subterm..... T:t
⊢ 4 ∈ ℤ

BY
```

For primitive terms `MemCD` uses the appropriate primitive equality rule. For abstractions, it tries to use an appropriate well-formedness lemma. Soft abstractions will be unfolded if there is no appropriate well-formedness lemma. A subgoal corresponding to the n-th subterm will be labeled with label `subterm` and number n. Other subgoals are labeled `wf`.
An example application of `MemCD` is shown on the right.

Well-formedness lemmata for a term t with operator identifier *opid* should have the name *opid*`_wf` and consist of a simple universal formula with consequent $\llcorner t \in T \lrcorner$. Usually, the subterms of t should be all variables, but constants are acceptable too. If more than one lemma is needed, the lemma names should be distinguished by suffices to the *opid*`_wf` root.

Usually `MemCD` attempts to use lemmata in *reverse dependency order* (i.e. the later ones do not refer to the former and are often more general) but different orders may be specified if needed. If the conclusion is $a \in A$ where a is an instance of t and A does not match any of the T of the lemmata, then `MemCD` tries matching a against the term t of the last lemma. If this succeeds and

Tactic	In the hypotheses	In the conclusion
`UnivCD`		$\forall \ \Rightarrow \qquad \rightarrow$
`GenUnivCD`		$\forall \ \Rightarrow \ \wedge \ \Leftrightarrow \ \rightarrow \ \times$
`RepD`	$\wedge$	$\forall \ \Rightarrow \qquad \rightarrow$
`GenRepD`	$\wedge$	$\forall \ \Rightarrow \ \wedge \ \Leftrightarrow \ \rightarrow \ \times$
`ExRepD`	$\exists \ \wedge$	$\forall \ \Rightarrow \qquad \rightarrow$
`GenExRepD`	$\exists \ \wedge \ \vee$	$\forall \ \Rightarrow \ \wedge \ \Leftrightarrow \ \rightarrow \ \times$

Table 8.2: Iterated decomposition tactics and the connectives they decompose

generates some substitution σ, `MemCD` produces a subgoal $\sigma T \subseteq A$ and tries to use the `Inclusion` tactic (Section 8.7) to prove it.

`MemHD` i

Decompose the immediate subterm of a *membership term* in *hypothesis* i.

Since there are no primitive rules for decomposing equalities in hypotheses, `MemHD` only works on hypotheses when the type of the membership term is a `product` or `function` type. For products, the decomposition generates the first and second projection of the term. For functions, it creates a function application. A `‘t1‘` argument is required in this case.

`MemberEqD` c

Decompose the immediate subterm of a *membership term* in clause c. Works like `MemCD` on the conclusion and like `MemHD` on a hypothesis.[4]

`EqD` c

Decomposes terms which are the immediate subterms of an *equality term* in clause c.

`EqD` is like `MemberEqD`, except that it expects and generates equality terms rather than membership terms. It is good for congruence reasoning and is used extensively by the rewrite package (see Section 8.6).
Commonly used variants are the tactics `EqCD` and `EqHD` i, which work identical to `EqD` on the conclusion and a hypothesis, respectively.

`EqTypeD` c

Decompose just the *type subterm* of an *equality* term in clause c. Only works when the type is a `set` type or is an abstraction that eventually unfolds to a set type.
Commonly used variants are `EqTypeCD` and `EqTypeHD` i

`MemTypeD` c

Decompose just the *type subterm* of a *membership* term in clause c. Only works when the type is a `set` type or is an abstraction that eventually unfolds to a set type.
Commonly used variants are `MemTypeCD` and `MemTypeHD` i

The above tactics cover almost all of the primitive inference rules of NUPRL's logic. Appendix A.3 gives a complete description of all the inference rules and the corresponding tactics.

Several tactics perform iterated decomposition of clauses. Table 8.2 lists the most commonly used ones, together with the logical connectives they decompose. These tactics do not decompose guarded terms. If a guard is encountered in the process of decomposing the conclusion, the guard is removed and decomposition of the conclusion stops.

[4] A better name would have been `MemD`

8.3.2 Structural

Structural tactics invoke inferences that depend only on the syntactical structure of the proof goal but are independent of a particular logic.

`Hypothesis`
: Prove goals of form $\ldots A \ldots \vdash A'$ where the conclusion A' is α-equal to A.

`NthHyp` i
: Prove goals of form $\ldots A \ldots \vdash A'$ where A is the i-th hypothesis and A' is α-equal to A.

`Declaration`
: Prove goals of form $\ldots x{:}T \ldots \vdash x \in T'$ or $\ldots x{:}T \ldots \vdash x{=}x \in T'$ where T' is α-equal to T.

`NthDecl` i
: Prove goals of form $\ldots x{:}T \ldots \vdash x \in T'$ or $\ldots x{:}T \ldots \vdash x{=}x \in T'$ where $x{:}T$ is the i-th declaration and T' is α-equal to T.

`Contradiction`
: Prove goals where both P and $\neg P$ occur in the hypotheses list.

`Assert` t
: Assert term t as last hypothesis. Generates a `main` subgoal with t asserted, and an `assertion` subgoal to prove t. Logically this inference is often called a *cut*.
A more general variant is AssertAt i t, which asserts t before hypothesis i.

`Thin` i
: Delete hypothesis i.

`MoveToHyp` i j
: Move hypothesis i to before hypothesis j.

`MoveToConcl` i
: Move hypothesis i into the conclusion. If the goal has the form $\ldots A \ldots \vdash C$, where A is the i-th hypothesis, then `MoveToConcl` generates the `main` subgoal $\ldots \vdash A \Rightarrow C$. If the i-th hypothesis is a declaration $x{:}T$, it generates the `main` subgoal $\ldots \vdash \forall x{:}T.C$.
`MoveToConcl` first invokes itself recursively on any hypothesis that might depend on hypothesis i.

`MoveDepHypsToConcl` i
: Use `MoveToConcl` to move all hypotheses that use the variable declared by hyp i into the conclusion. Hypothesis i itself will not be moved.

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

`RenameBVars (`σ`:(var#var) list)` c
: Rename all occurrences of bound variables in clause c using the substitution σ.

The following two tactics are usually used inside other tactics.

`Id`
: The identity tactic, which does not change the proof goal

`Fail`
: The tactic that always fails.

8.3.3 Decision procedures

Decision procedures use special-purpose reasoning to decide problems within a specific limited application domain. In many cases, they analyze the problem by translating it into a different problem domain (e.g. the problem of finding cycles in a directed graph) for which there are well-known decision algorithms. To ensure consistency with type theory, some of these procedures have to generate proof tactics that validate the result of the analysis and create appropriate subgoals if necessary. For `Arith` and `Eq`, the consistency of the decision procedure has been proven externally and the procedures are implemented as elementary inference rules.

Decision procedures require the well-formedness of the components used during the analysis to be proven as separate subgoals, as this is a purely type-theoretical issue that cannot be addressed by the decision algorithm.

8.3.3.1 Logical reasoning

`ProveProp`

Prove a goal that involves only simple propositional reasoning.
The proof strategy is basically a classical tableau prover for propositional logic, which exhaustively decomposes propositions and seeks for applications of the `Hypothesis` tactic. Because NUPRL sequents only allow one conclusion (rather than many as in tableau calculi) the tactic has to do 'or' branching and backtracking when it tackles an $\vee$ conclusion or an $\Rightarrow$ or $\neg$ hypothesis. It fails if not all main goals can be solved.

The tactic is not complete for intuitionistic propositional logic, because it always thins $\Rightarrow$ and $\neg$ hypotheses that are decomposed. Common variants of `ProveProp` are

- `ProvePropWith` T, which tries running the tactic T before abandoning a search path in an 'or' branch and continues the search on any `main` subgoal that T creates.
- `ProveProp1`, which leaves main subgoals at 'or' branching points of the search for a solution when the search down every branch fails.

`JProver`

Prove a goal that involves only first-order reasoning.
`JProver` [SLKN01] is a complete theorem prover for first-order intuitionistic logic that is based on a strategy called the connection method [KO99]. Upon success it generates a sequent proof for the proof goal [KS00] that may be inspected by the user. Since first-order logic is undecidable, `JProver` will not terminate if the goal cannot be proven and must be interrupted.
`JProver` is run as an *external* prover, which means that the METAPRL proof engine must be connected to NUPRL before invoking `JProver`.

8.3.3.2 Exploiting properties of relations

`Eq`

Prove goals of form $H \vdash s = t \in T$ using hypotheses that are equalities over T and the laws of reflexivity, commutativity and transitivity.
`Eq` also uses hypotheses that are equalities over a type T' when $T = T'$ can be deduced from other hypotheses using reflexivity, commutativity and transitivity.

`RelRST`

Prove goals by exploiting common properties of binary relations, including reflexivity, symmetry, transitivity, irreflexivity, antisymmetry, and linearity.
The heart of `RelRST` is a routine that builds a directed graph based on the binary relations in a sequent and finds shortest paths in the graph. It can also handle strict order relations and relations with differing strengths.

RelRST uses the the same database on relations and some of the same lemmata as the rewrite package (see Section 8.6). In addition, it relies on library lemmata of the following forms.

- *Irreflexivity* lemmata, which should be named *opid-lt*`_irreflexivity` and have form
  $\forall x_1{:}T_1 \ldots x_m{:}T_m.\forall y{:}S.A_1 \Rightarrow \ \ldots \ \Rightarrow A_n \Rightarrow \neg(y < y)$
- *Antisymmetry* lemmata, which should be named *opid-le*`_antisymmetry` and have form
  $\forall x_1{:}T_1 \ldots x_m{:}T_m.\forall y, y'{:}S.A_1 \Rightarrow \ \ldots \ \Rightarrow A_n \Rightarrow y{\leq}y' \Rightarrow y'{\leq}y \Rightarrow y = y' \in S$
- *Complementing* lemmata, which should be named *opid-le*`_complement` and have form
  $\forall x_1{:}T_1 \ldots x_m{:}T_m.\forall y, y'{:}S.A_1 \Rightarrow \ \ldots \ \Rightarrow A_n \Rightarrow \ \Rightarrow \neg(y{\leq}y') \Rightarrow y' < y$

  or be named *opid-lt*`_complement` and have form
  $\forall x_1{:}T_1 \ldots x_m{:}T_m.\forall y, y'{:}S.A_1 \Rightarrow \ \ldots \ \Rightarrow A_n \Rightarrow \ \Rightarrow \neg(y < y') \Rightarrow y'{\leq}y$

where *opid-lt* and *opid-le* are the operator identifiers of the relations $<$ and $\leq$.

RelRST generalizes the equality decision procedure used in previous versions of NUPRL that could only handle such reasoning with the equality relation. Examples of its use can be found in the theory of integer divisibility within the theory `num_thy_1`.

8.3.3.3 Integer arithmetic

`Arith`

Prove goals of the form $H \vdash C_1 \vee \ \ldots \ \vee C_m$ by a restricted form of arithmetic reasoning. Each C_i must be an *arithmetic relation* over the integers built from $<$, $\leq$, $>$, $\geq$, $=$, $\neq$, and negation.
`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. 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.
The heart of `Arith` is a procedure that translates the sequent into a directed graph whose egdes are labelled with natural numbers and finds positive cycles in that graph [Cha82].

`RepeatEqCDForArith`

Apply arithmetic equality reasoning if the conclusion is $a{=}b \in T$ and a and b arithmetically simplify to same expression.[5] `RepeatEqCDForArith` first decomposes a and b using `EqCD` and then applies `Arith` to subgoals containing integer equalities.

`SupInf`

Solve linear inequalities over integers and subtypes of integers.
The algorithm used in `Arith` cannot solve general sets of linear inequalities over the integers, though such problems are abundant. `SupInf` uses an adaptation of Bledsoe's *Sup-Inf* method [Ble75] for solving integer inequalities. While this method is only complete for the rationals, it is sound for the integers and does work well in practice.
`SupInf` converts the sequent into a conjunction of terms of the form $0{\leq}e_i$ where each e_i is a linear expression over the rationals in variables $x_1 \ldots x_n$ and determines whether or not there exists an assignment of values to the x_j that satisfies the conjunction. The algorithm works by determining upper and lower bounds for each of the variables in turn.

`SupInf` identifies counter-examples if it fails. These can be viewed by looking at value of the ML variable `supinf_info`. The value gives a list of bindings of variables in the goal for the counter-

[5] Arithmetic simplification applies only subterms that involve the basic arithmetic operators `+`, `-`, `*`, `/` and `rem`. Currently simplification involving `/` and `rem` does not work and has been disabled.

example. If `SupInf` finds an integer counterexample, then the goal is definitely unprovable. If a rational counter-example is given, then `SupInf` is unsure whether the goal is true or not.

`SupInf'`

Like `SupInf`, but tries inferring additional arithmetic information about the non-linear terms in arithmetic expressions. The information on a non-linear term t is gathered in two ways:

1. If standard type-inference returns a subtype of $\mathbb{Z}$ for t, then the predicate information from the subtype is added.
2. Information may be gathered from *arithmetic property lemmata*, i.e. lemmata of the form

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

   where C is constructed from $\wedge$, $\vee$, and standard arithmetic relations over integer subtypes built from $<$, $\leq$, $>$, $\geq$, $=$, $\neq$, and their negations. To apply the lemma, *match handles* are selected from C, i.e. terms occuring in arithmetic relations in C that contain all the free variables contained in C and the A_i. If, after matching a match handle with t, all the instantiated A_i are equal to hypotheses of the sequent, then the instantiated clause C will be added as information.[6]

   Arithmetic property lemmata are identified by invoking the ML function

   `add_arith_lemma` *lemma-name*.

`SupInf'` is still under development and should be used with caution when combined with type-checking tactics: it may get invoked unendingly on subgoals derived from ones that it created.

8.3.4 Autotactics

Autotactics are used primarily for typechecking and well-formedness goals. They should not be used for other non-trivial proof goals, as their behavior on such goals is somewhat unpredictable.

`Trivial`

Completely prove a goal by applying various steps of trivial reasoning.
Trivial reasoning includes `Hypothesis`, `Declaration`, `Contradiction`, and `Eq`. `Trivial` also proves goals of the form $H \vdash$ `True`, $H_1 \ldots$ `False` $\ldots H_n \vdash C$, and $H_1 \ldots$ `Void` $\ldots H_n \vdash C$.

`Auto`

Repeatedly apply the following tactics until no further progress is made.

- `Trivial`
- `GenExRepD`
- `MemCD` for (non-recursive) member conclusions and `EqCD` on reflexive equality conclusions.
- `Arith`
- `RepeatEqCDForArith`
- `EqTypeCD` if the conclusion is $a \in T$ or $a{=}b \in T$ and T is a subset of $\mathbb{Z}$.

`Auto` and its variants frequently encounter the same goals over and over again, so solved proof goals will be cached. Common variants of `Auto` are

- `StrongAuto`, which also tries `MemCD` and `EqCD` on *recursive* primitive terms;
- `SIAuto`, which also tries using the `SupInf` tactic; and
- `Auto'`, which uses `SupInf'` instead of `Arith`.

[6]The condition that A_i must be equal to some hypothesis is too strict and may be relaxed in the future.

8.4 Forward and Backward Chaining

Forward and backward chaining means treating a component of a universal formula (see Section 8.2.5) as derived inference rule. *Backward chaining* involves matching the conclusion of the goal against the consequent of a universal formula, which leaves the instantiated antecedents of the universal formula as new subgoals. *Forward chaining* involves matching hypotheses of the goal against antecedents of a universal formula and asserting the instantiated consequent of the universal formula as a new hypothesis. A simplified version of forward chaining is *instantiating* the universal formula by explicitly providing a list of terms to be substituted for the quantified variables.

Chaining tactics consult universal formulas can either be found in the hypotheses of the current proof goal or as lemma in the library. Therefore each tactic comes in two versions.

`InstHyp [`t_1`;..;`t_n`]` *i*
`InstLemma` *name* `[`t_1`;..;`t_n`]`

Instantiate hypothesis *i* (or lemma *name*) with terms $t_1 \ldots t_n$. If the lemma has m distinct level expressions, the first m terms should be level expressions to substitute for these.[7]

`InstConcl [`t_1`;..;`t_n`]`

Instantiate existential quantifiers in the conclusion with terms $t_1 \ldots t_n$.

`FHyp` *i* `[`h_1`;..;`h_n`]`
`FLemma` *name* `[`h_1`;..;`h_n`]`

Forward chain through hypothesis *i* (or lemma *name*) matching its antecedents against any of the hypotheses $h_1 \ldots h_n$. The order of the h_j is immaterial: the tactics try all possible pairings of hypotheses with antecedents. If there are more antecedents that hypotheses listed, the antecedents not matched will manifest themselves as new subgoals to be proved.
The main subgoal with the consequent asserted is labelled `main`. Unmatched antecedents are labelled `antecedent` and the rest are labelled `wf`. Aliases are `FwdThruLemma` and `FwdThruHyp`.

Forward chaining can take an optional argument (c.f. Section 8.2.2), supplied by using the `Sel` tactical, to select a specific component of a conjunction or equivalence in a universal formula. An argument of `-1` forces the tactic to treat the whole subformula as simple.

`BHyp` *i*
`BLemma` *name*

Backward chain through hypothesis *i* (or lemma *name*) matching its consequent against the conclusion of the goal. Subgoals corresponding to antecents of the lemma (hyp) are labelled with `antecedent`. The rest are labelled `wf`. Aliases are `BackThruLemma` and `BackThruHyp`.

An explicit list of variable bindings can be supplied to backward chaining as optional argument by using the `Using` tactical. This argument is necessary if some of the variable bindings cannot be inferred by matching. For instance

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

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

`BHypWithUnfolds` *i as*
`BLemmaWithUnfolds` *name as*

Backward chain through hypothesis *i* (or lemma *name*) while unfolding the abstractions in *as*.

[7]To inject a level expression L into a term list one has to invoke the special term `parameter` in the term editor.

`Backchain` *bc_names*
`CompleteBackchain` *bc_names*

Repeatedly try backchaining using lemmata named in *bc_names* in the order given. `Backchain` leaves alone any subgoals which do not match the consequent of any of the lemmata while `CompleteBackchain` backtracks in the event of any such subgoal coming up.

In addition to lemma names, *bc_names* may contain a few special names:

- An positive integer i, indicating that hypothesis i shall be consulted.
- `‘hyps‘`, indicating that all hypothesis should be consulted in the order in which they occur. Hypotheses that declare variables will be ignored.
- `‘rev_hyps‘`, indicating that all hypothesis should be consulted in reverse order.
- `‘new_hyps‘`, indicating that all new hypotheses introduced by backchaining should be consulted, beginning with the least recent.
- `‘rev new_hyps‘`, indicating that all new hypotheses introduced by backchaining should be consulted, beginning with the most recent.

Common variants of backchaining are `HypBackchain` and `CompleteHypBackchain` which perform backchaining with the arguments `[‘rev_new_hyps‘; ‘rev_hyps‘]`.

8.5 Case Splits and Induction

Case split and induction tactics analyze conclusions that depend on finite or enumerable alternatives. The general way to do this is to backchain through an appropriate lemma (see e.g. the lemmata `int_upper_ind` and `int_seg_ind` at the end of the `int_2` theory). To use these lemmata, one must ensure that the outermost universal quantifier in the conclusion corresponds to the type of the induction.

The following tactics are good for a few common cases. They expect the variable the induction / case-split is being done over to be declared in some hypothesis.

`BoolCases` i

Do a case split over a boolean variable declared in hypothesis i. Generates two subgoals, labelled `truecase` and `falsecase`, where the boolean variable is replaced by `tt` or `ff`.

`Cases` $[t_1; ..; t_n]$

Perform an n-way case split over the terms t_i as follows:

$$
\begin{array}{l}
H \vdash C \\
\quad \textbf{by}\ \texttt{Cases}\ [t_1; ..; t_n] \\
\qquad \texttt{...assertion...}\ H \vdash t_1 \vee .. \vee t_n \\
\qquad\qquad\qquad\qquad H,\ t_1 \vdash C \\
\qquad\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\qquad H,\ t_n \vdash C
\end{array}
$$

`Decide` P

Perform a case split over a *decidable* proposition P and its negation. Like `Cases` $[P; \neg P]$, but immediately runs the tactic `ProveDecidable` tactic on the first subgoal $\vdash P \vee \neg P$, which may generate wellformedness subgoals with labels in the `aux` class. If `ProveDecidable` fails then `Decide` fails too.

Because of the constructive nature of Nuprl's type theory $P \vee \neg P$ is not true for every proposition P.

IntInd i

Perform integer induction on hypothesis i, generating three subgoals labelled `upcase`, `basecase` and `downcase`. `IntInd` is a little smarter than the primitive rule `intElimination` (see Appendix A.3.8) in that it first moves any hypotheses that depend on the induction variable to the conclusion and maintains the name of the induction variable.

NatInd i

Perform natural-number induction on hypothesis i, which must contain a declaration of type $\mathbb{N}$, generating two subgoals labelled `upcase` and `basecase`. `NatInd` first moves any depending hypotheses to the conclusion and maintains the name of the induction variable.

NSubsetInd i

Perform induction on a subrange of the natural numbers. Hypothesis i must contain a declaration of type $\mathbb{N}$, $\mathbb{N}^+$, $\{i\ldots\}$, or $\{i\ldots j^-\}$. `NSubsetInd` generates two main subgoals labelled `upcase` and `basecase` an many `aux` subgoals which should always be easily solvable by `Auto`.

CompNatInd i

Perform *complete* induction on hypothesis i, which must contain a declaration of type $\mathbb{N}$.

ListInd i

Perform list induction on hypothesis i, which must contain a declaration of type T `list`, generating two subgoals labelled `upcase` and `basecase`. `ListInd` is a little smarter than the primitive rule `listElimination` (see Appendix A.3.10) in that it first moves any depending hypotheses to the conclusion and maintains the name of the induction variable.

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

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

8.6 Simple Rewriting

Rewriting is the process of transforming goals and hypotheses into equivalent ones. In NUPRL, rewrite tactics are based on unfolding and folding abstractions, applying primitive reductions, and using equivalences in lemmata and hypotheses of the form $\forall x_1{:}T_1 \ldots x_i{:}T_i.\ a = b$.

NUPRL's rewrite package (see Section 8.9) provides a collection of ML functions for creating rewrite rules and applying them in various fashions. The rewriting tactics in this section are sufficient in many situations while hiding the complex conversion language of the rewrite package.

8.6.1 Folding and Unfolding Abstractions

Unfolds as c

Unfold a c $\equiv$ `Unfolds [`a`]` c

Unfold all visible occurrences of abstractions listed in the token list as in clause c. `Unfolds` fails if clause c contains none of the abstractions listed in as.

RepUnfolds as c

Repeatedly try unfolding any occurrences of abstractions listed in the token list as in clause `c`.

RecUnfold a c

Unfold all visible occurrences of the recursive definition of a in clause c.

Folds *as* *c*
Fold *a* *c* $\equiv$ `Folds [`*a*`]` *c*

Fold all visible occurrences of abstractions listed in the token list *as* in clause `c`. `Folds` fails none of the subterms of clause *c* matches the right hand side of an abstraction listed in *as*.

RecFold *a* *c*

Try to fold an instance of the recursively defined term *a* in clause *c*.

8.6.2 Evaluating Subexpressions

Reduce *c* $\equiv$ `AbReduce` *c*

Repeatedly contract all primitive and abstract redices in clause *c*.

The `Reduce` tactic can take an optional *force* argument:

With *force* **(Reduce** *c***)**

only reduces those redices with strength less than or equal to *force*. Details about defining abstract redices and setting the strength of redices can be found in Section 34.

To reduce a specific subterm of a clause, one may apply the tactic **ReduceAtAddr** *address* *c*, where *address* is a list of integers describing the exact address of the subterm in the term tree of clause *c*. Applying this rule is only recommended for advanced users who are very familiar with the term structure of NUPRL expressions.

ArithSimp *c*

Arithmetically simplify clause *c*.
This creates a `main` subgoal with clause *c* rewritten in arithmetical canonical form and an `aux` subgoal stating the equivalence between the original clause and the rewritten one.

8.6.3 Substitution

NUPRL's logic contains a few rules for carrying out simple kinds of substitutions. These rules often generate fewer and easier-to-solve well-formedness goals than the rewrite package and are accessible through the tactics described here.

Subst *eq* *c*

If the term *eq* has the form $t_1 = t_2 \in T$ then replace all occurrences of t_1 in clause *c* by t_2.
Three subgoals are generated: an `equality` subgoal to prove that $t_1 = t_2 \in T$, a `main` subgoal with the substitution carried out, and a `wf` subgoal to prove functionality of the clause (see the rule **substitution** in Section A.3.5).

HypSubst *i* *c*
RevHypSubst *i* *c*

Run the `Subst` tactic using the equality proposition in hypothesis *i* (in reversed order). This generates only a `main` and a `wf` subgoal.

SubstClause *t* *c*

Replace clause *c* with term *t*. This generates a `main` subgoal and an `equality` subgoal (see the rule **hyp_replacement** in Section A.3.16).

Token	Rule
i	Use hypothesis *i* as an left-to-right rule
i`<`	Use hypothesis *i* as an right-to-left rule
name	Use lemma *name* as an left-to-right rule
name`<`	Use lemma *name* as an right-to-left rule
`r:`*id*	Reduce redex with operator identifier *id*
`r*`	Reduce any redex
`r*`*force*	Reduce any redex with force *force*
`u:`*id*	Unfold abstraction with operator identifier *id*
`f:`*id*	Fold abstraction with operator identifier *id*

Table 8.3: Format of Tokens in Rewrite Control Strings

8.6.4 Generic Rewrite Tactics

The tactics `RWW` and `RWO` subsume the above tactics by providing uniform access to all kinds of rewrite rules. They take a control string to specify the rewrite rules to use. The control string should be a whitespace-separated list of tokens as specified in Table 8.3.

`RWW "`*ctl-str*`"` *c*

Repeatedly apply rewrite rules specified by *ctl-str* to all subterms of clause *c* until no further progress is made.

`RWO "`*ctl-str*`"` *c*

Apply rewrite rules specified by *ctl-str* in one top-down pass over clause *c*. `RWO` does not go into subterms of terms that result from rewriting a subterm of *c*.

In some cases, `RWW` and `RWO` generate more subgoals than the more specific tactics, as they are implemented in a different fashion.

8.7 Miscellaneous Tactics

Type Inclusion

`Inclusion` *i*

Prove goals of the form $\ldots,\ i.\ x{:}T, \ldots \vdash x \in T'$ or $\ldots,\ i.\ t \in T, \ldots \vdash t \in T'$, where either types T and T' are equivalent or T is a proper subtype of T'. `Inclusion` also solves similar goals where one or both of the membership terms are replaced by equality terms.

The specific kinds of relations between T and T' that `Inclusion` currently handles are roughly:

- T and T' are the same once all soft abstractions are unfolded.
- T and T' are both universe or prop terms and the level of T is no greater than the level of T' for any instantiation of level variables.
- T and T' are each formed by using subset types, and both have some common superset type. In this case `Inclusion` tries to show that the subset predicates of T' are implied by the subset predicates of T together with other hypotheses.
- T and T' have the same outermost type constructor. In this case, the inclusion goal is reduced to one or more inclusion goals involving the immediate subterms of T and T'. Currently works for function, product, union and list types.
- T is a subtype of T' according to a lemma in the library.

For the inclusion reasoning involving subset types to work, one has to supply information about abstractions involving subset types using the function `add_set_inclusion_info`. The theory `int_1` contains several examples of the use of this function.

Squash Stability and Hidden Hypotheses

The constructivity of NUPRL's logic manifests itself in the fact that the decomposition of sets in hypotheses (rule `setElimination` in Section A.3.12) results in a subgoal containing the predicate part of the set term as a *hidden* hypothesis (the rule `quotient_equalityElimination` in Section A.3.14 has a similar effect). A hidden hypothesis does not contribute to the computational content of a proof (i.e. the term inhabiting its conclusion) and is therefore not immediately usable. However, there are ways in which it might become usable later.

A hidden hypothesis P in a proof goal can be unhidden if either P or the conclusion of the goal is *squash stable*, which roughly means that it is possible to determine the computational content if one knows that there is one (in the classical sense). Squash stability is defined in the theory `core_2` as $\downarrow P \Rightarrow P$, where the proposition $\downarrow P \equiv \{x{:}\mathsf{Unit}|P\}$ (read 'squash P') is true exactly when P is true, but has no computational content.

Squash stability can be inferred for many predicates using the tactic `ProveSqStable`, which is not called by the `D` tactic because it can be rather slow. Instead, hypothesis can be unhidden by applying one of the following tactics.

`Unhide`

Try to unhide hidden hypotheses, first by checking whether the conclusion is squash stable and then, if this fails, by checking each hidden hypothesis separately for squash stability.

`Unhide` applies the tactics `UnhideAllHypsSinceSqStableConcl`, which tries to prove the conclusion squash stable using `ProveSqStable` and unhides all hidden hypotheses if this succeeds, and UnhideSqStableHyp i, which tries to prove the hidden hypothesis i squash stable.

`AddProperties` i

Add the predicate part of the set type underlying an abstraction A in hypothesis i as a new hypothesis immediately after i.
Hypothesis i should be declaration of form A or a proposition of form $t \in A$ or $t = t' \in A$, where A is an abstraction with an associated *property lemma* of the form $\vdash \forall x{:}T.\ \forall y{:}A.P[x,y]$.

`GenConcl` $\llcorner t\texttt{=}v \in T\lrcorner$

Generalize occurrences of t as subterms of the conclusion to the variable v. This adds new hypotheses declaring v to be of type T and stating $t\texttt{=}v \in T$.

`Fiat`

If you about to give up hope on a theorem, this tactic is guaranteed to provide satisfaction.
`Fiat` uses NUPRL's `because` rule, which should be used for experimental purposes only. NUPRL's library has a mechanism to detect uses of this rule while checking a theory for consistency.

8.8 Tacticals

Tacticals are functions for converting tactics into new one. Apart from injecting optional arguments as described in Section 8.2.2, they are most commonly used for composing tactics. 2-ary tacticals are often written in infix form and distinguished from others by having the first part of their name in all capitals. Infix tacticals always associate to the left.

8.8.1 Basic Tacticals

T_1 `THEN` T_2
: Apply T_1 and then run T_2 on all the subgoals generated by T_1.

T `THENL [`T_1`;...;`T_n`]`
: Apply T and then run T_i on the i-th subgoal generated by T. T must create exactly n subgoals.

T_1 `ORELSE` T_2
: Apply T_1. If it fails, run T_2 instead.

`Try` T $\equiv$ T `ORELSE Id`
: Apply T. If it fails, leave the proof unchanged.

`Complete` T
: Apply T but fail if T generates subgoals (i.e. does not complete the proof).

`Progress` T
: Apply T but fail if T does not change the goal (i.e. makes no progress).

`Repeat` T
: Repeat running T on subgoals created by previous applications until no further progress is made.

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

`If` e T_1 T_2
: Apply T_1 if e `pf` evaluates to `true`, where `pf` is the current proof goal. Otherwise, run T_2.

8.8.2 Label Sensitive Tacticals

Label sensitive tacticals tacticals allow one to apply a tactic only to goals with a particular label, or to goals of one of the classes *main*, *aux*, and *predicate* (c.f. Section 8.2.3). For the former, the label associated with the tactic has to match exactly the label of the goal. For the latter, one may use the class names `main`, `aux`, and `predicate` as wild cards.[8]

`IfLab` lab T_1 T_2
: If lab matches the label of `pf`, run T_1. Otherwise, run T_2.

`IfLabL [`l_1`,`T_1`;` l_2`,`T_2`; ...;` l_n`,`T_n`]`
: Run the first tactic T_i for which l_i matches the label of `pf`. If none of the labels match, leave the proof unchanged.

T_1 `THENM` T_2 $\equiv$ T_1 `THEN IfLab ‘main‘` T_2 `Id`
T_1 `THENA` T_2 $\equiv$ T_1 `THEN IfLab ‘aux‘` T_2 `Id`
T_1 `THENW` T_2 $\equiv$ T_1 `THEN IfLab ‘wf‘` T_2 `Id`
: Apply T_1 and then run T_2 on all `main`/`aux`/`wf` subgoals.

T `THENLL [`l_1`,`Ts_1`;` l_2`,`Ts_2`;...;` l_n`,`Ts_n`]`
: Apply T and then do the following on each subgoal. Select the first list of tactics Ts_i for which l_i matches the label of the subgoal. If the subgoal also has a number label j, run the jth tactic from Ts_i on it. If it has no number label, run the first tactic listed in Ts_i.
`THENLL` fails if there are not sufficiently many tactics in Ts_i. It runs the `Id` tactic if a subgoal label does not match any of the l_i.

`SeqOnM [`T_1`;...;`T_n`]`
: Run the tactics T_1 to T_n on successive `main` subgoals.

[8]Unfortunately `main` is currently used as both a class name and a particular label name, which means there is no way to select only subgoals in with the particular label `main`.

RepeatM T
RepeatMFor n T
Repeat the tactic T on `main` subgoals (exactly n times).

8.8.3 Multiple Clause Tacticals

Multiple clause tacticals allow to apply a tactic T `: int -> tactic` to several clauses of a goal with `n` hypotheses.

`On [`c_1`;...;`c_n`]` T $\equiv$ T c_1 `THENM ... THENM` T c_n
Run T on the clauses $c_1 \ldots c_n$. If T succeeds on some clause, then `On` only continues on subgoals created by T that are labelled `main`.

`AllHyps` T $\equiv$ `On [n;n-1;...;1] (ı. Try (`T `i))`
Try running T on all hypotheses starting with the end of the hypothesis list and working backwards. If T succeeds on some hypothesis, then `AllHyps` only continues on subgoals created by T that are labelled `main`.

`All` T $\equiv$ `On [n;n-1;...;1;0] (ı. Try (`T `i))`
Try running T on all hypotheses and then on the conclusion.

`OnSomeHyp` T $\equiv$ T `n ORELSE ... ORELSE` T `1`
Try running T on one of the hypotheses of the goal, starting with the end of the hypothesis list and working backwards.

The library contains a few abstractions that provide a more elegant notation for the most common combinations of tactics and tacticals.

`auto`	T `...`	$\equiv$	T `THEN Auto`
`aux_auto`	T `...a`	$\equiv$	T `THENA Auto`
`siauto`	T `...`	$\equiv$	T `THEN SIAuto`
`autop`	T `...'`	$\equiv$	T `THEN Auto'`
`aux_autop`	T `...a'`	$\equiv$	T `THENA Auto'`
`aux_siauto`	T `...as`	$\equiv$	T `THENA SIAuto`

8.9 The Rewrite Package

NUPRL's rewrite package is a collection of ML functions for creating rules for rewriting terms into equivalent ones and applying them in various fashions to clauses of a sequent. The package supports rewrite rules involving various equivalence relations – such as the 3-place equality-in-a-type relation, logical bi-implication, the permutation relation on lists, abstractions, and computational equivalence – and takes care of automating proofs that these equivalence relations are respected by rewriting. It also supports rewriting rules involving arbitrary transitive relations such as logical implication and takes care of checking that relevant terms are appropriately monotonic.

The package is based around ML objects called *conversions*, similar to those found in other tactic-based theorem provers such as LCF, HOL, and Isabelle, provide a language for systematically building up rewrite rules in a fashion similar to the way tactics are assembled using tacticals.

8.9.1 Introduction to Conversions

A conversion is a function that transforms a term t into a new term t' that is equivalent to t with respect to some *relation* r. The conversion also produces a *justification* j that describes how to prove that $t\ r\ t'$ holds. The transformation takes place in an *environment* e, which specifies amongst other things the types of the variables that might be free in t. Conversions fail if they are not appropriate for the term they are applied to.

In NUPRL, the type `convn` of conversions is an ML concrete type abbreviation for the type

`env -> term -> (term # reln # just)`,

where `env`, `reln` and `just` are abstract types for *environments* (Section 8.9.1.1), *relations* (Section 8.9.1.2), and *justifications* (Section 8.9.1.3). The language of conversions provides a small set of *atomic conversions* that may be assembled into more advanced conversions using higher order functions called *conversionals*.

Atomic conversions (Section 8.9.2) are either based on direct computation rules (which includes folding and unfolding abstractions) or can be created from lemmata and hypotheses that contain universally quantified formulas with consequent of the form $a\ r\ b$, where the free variables of b are a subset of those in a. Applying a conversion to a term t means either executing the corresponding computation or matching the term t against a and replacing it by an appropriate instance of b. That is, if σ is a substitution of the free variables in a such that $\sigma a = t$, then t will be replaced by $t' = \sigma b$. For instance, rewriting the term $(2{\times}3) + 0$ with the relation $x + 0 = x$ means matching $(2{\times}3) + 0$ against $x + 0$, which yields a substitution σ that binds x to $2{\times}3$. The result of rewriting is the term $\sigma x = 2{\times}3$.

Atomic conversions cannot by themselves rewrite subterms of a given term. For instance, applying an atomic conversion to rewrite $(1 + 0){\times}3$ with the relation $x + 0 = x$ fails. *Conversionals* (Section 8.9.4) provide the means for applying conversions to subterms of a term and help controlling the sequence in which atomic conversions are applied to these subterms.

An example of a conversional is `SweepUpC`, which attempts to apply a conversion c to each subterm of a term t, working from the leaves of term t up to its root. Another example is `ORELSEC`, which first tries to apply a conversion c_1 to a term and, if that fails, applies a conversion c_2.

Conversionals rely on a variety of lemmata, which we will describe in Section 8.9.1.4. These lemmata have to state reflexivity, transitivity and symmetry properties of the relation r and congruence properties of the terms making up the clauses that are being rewritten.

A tactic `Rewrite : convn -> int -> tactic` is used for making a conversion applicable to some clause of a proof goal. It takes care of executing the justifications generated by conversions. Section 8.9.1.5 lists common variations on this tactic.

8.9.1.1 Environments

An *environment* is a list of propositions and declarations of variable types that are being assumed. The environment of the conclusion of a sequent is the list of all the hypotheses. The environment of a hypothesis is the list of hypotheses to the left of it. We can also talk about *local environments* of subterms of sequent clauses. For example, in the sequent $x_1{:}H_1, \ldots, x_n{:}H_n \vdash \forall y{:}T.\ B \Rightarrow C$ the local environment for the subterm C in the conclusion is $x_1{:}H_1, \ldots, x_n{:}H_n,\ y{:}T,\ B$. The rewrite conversionals keep track of the local environment each conversion is being applied in, and every conversion takes as its first argument an expression e of type `env` which supplies this local information. The environment information is used by lemma and hypothesis conversions in three ways.

- Declarations of variables in the environment are used to infer types that help to complete matches.
- Propositions in the environment state the assumptions that are necessary for conditional rewrites to go through. For example, if the subterm C in $x_1{:}H_1, \ldots, x_n{:}H_n \vdash \forall y{:}T.\ B \Rightarrow C$ is rewritten by a rewrite rule based on the lemma $\forall z{:}T.\ A[z] \Rightarrow t[z] = t'[z]$ and matching the term C against t results in binding the variable z to a term s, then the subgoal that has to be proven for the rewrite rule to be valid is $x_1{:}H_1,\ \ldots, x_n{:}H_n,\ y{:}T, B, \vdash A[s]$.
- The hypothesis conversions access the hypothesis list via environment terms.

Currently, NUPRL only knows how to extend the environment when descending to the subterms of a $\forall$-, $\exists$-, and $\Rightarrow$-term. For other terms, it does not modify the environment, unless explicitly told so by the user, who may extend the list of environment update functions by applying the function `add_env_update_fun`. Further details can be found in the system file `env.ml`.

8.9.1.2 Relations

The rewrite package supports rewriting with respect to both primitive and user-defined equivalence relations. Some examples are:

- $t \sim t'$, the computational equality relation,
- $t = t' \in T$, the primitive equality relation of NUPRL's type theory,
- $P \Leftrightarrow Q$, the logical bi-implication,
- $i = j \text{ mod } n$, the equality on the integers modulo a positive natural,
- $t =_q t'$, equality of rationals (represented as pairs of integers),
- $l \equiv l'$, the permutation relation on lists.

The package also supports 'rewriting' with respect to any relation that is transitive but not necessarily symmetric or reflexive, such as logical implication[9] $\Rightarrow$ and order relations, because proofs involving transitive relations and monotonicity properties of terms can be made very similar in structure to those involving equivalence relations and congruence properties.

For each user-defined relation, the user has to provide the rewrite package with lemmata about transitivity, symmetry, reflexivity and *strength*, where a binary relation r over a type T is stronger than a relation r' $a\ r\ b$ implies $a\ r'\ b$ for all $a, b \in T$. These lemmata are used by the package for the justification of rewrites (see Section 8.9.1.4). The user may also provide a declaration that identifies *relation families* and extra properties of relations.

Rewrite relations should be defined as first-order terms with two principal arguments supplied as subterms. Additional parameters should always be positioned before the principal argument subterms. For example, the equality relation $t = t' \in T$ has the type T as additional parameter and the internal structure `equal(`T`;`t`;`t'`)`. Relations are most commonly represented as logical propositions (i.e. are of type $\mathbb{P}$). Boolean-valued relations are also accepted, but they have to be wrapped in the `assert` abstraction if used in a context where a logical proposition is expected.

Equivalence relations should be declared by an invocation of

`declare_equiv_rel` *rnam* *stronger-rnam*

This declares the term with operator identifier *rnam* to be an equivalence relation, and the term with operator identifier *stronger-rnam* to be an immediately stronger equivalence relation. Commonly, there will be only one such declaration for each *rnam* and *stronger-rnam* will be `` `equal` ``. However, multiple declarations for a single equivalence relation are sometimes needed.

[9] Note that treating implication as a rewrite relation leads to a generalization of forward and backward chaining.

Order relations are grouped into *relation families*, i.e. lattices of order and equivalence relations of the form:

$$
\begin{tikzcd}
 & \leq \arrow[dl, dash] \arrow[dr, dash] & & \geq \arrow[dl, dash] \arrow[dr, dash] & \\
< & & = & & >
\end{tikzcd}
$$

where weaker relations are higher in the lattice, and relations within a family satisfy

$$
\begin{array}{lclclcl}
a{<}b & \Leftrightarrow & b{>}a & \qquad & a{=}b & \Leftrightarrow & a{\leq}b \wedge b{\leq}a \\
a{\leq}b & \Leftrightarrow & b{\geq}a & \qquad & a{<}b & \Leftrightarrow & a{\leq}b \wedge \neg(b{\leq}a)
\end{array}
$$

The converse of an order relation r should always be defined directly in terms of r. For example, the definition of the abstraction `rev_implies` is $P \Leftarrow Q \;\equiv\; Q \Rightarrow P$. The rewrite package assumes that order relations can be inverted by folding and unfolding such definitions. A relation family should be declared using an invocation of

`declare_rel_family` *lt le eq ge gt*

where dummy terms (i.e. the abstraction `dummy()`, which displays as `?`) should be used as placeholders when a member of a family is missing. To simplify such invocations, a user may enter the name `relfam` into a term slot, which will display the template:

Relation Family	Relation Family
$<$: [lt]	$<$: i $<$ j
$\leq$: [le]	$\leq$: i $\leq$ j
$\equiv$: [eq]	$\equiv$: i $=$ j
$\geq$: [ge]	$\geq$: i $\geq$ j
$>$: [gt]	$>$: i $>$ j

As an example, the invocation to the right shows the declaration of the standard order relations on the integers as relation family. Frequently several order relation families share the same equivalence relation, but there may also be equivalence relations that are not be associated with any order relation family.

The partial order of strengths of relations is the reflexive transitive closure of the strength relation for each family and the equivalence relation declarations. Additional relations between order relations can be declared using

`declare_order_rel_pair` *stronger-rtm weaker-rtm*

For the sake of clarity, all relation declarations should be inserted in ML objects that are positioned after the referred-to relations but before any lemmata that might be accessed by the rewrite package.

8.9.1.3 Justifications

The justification produced by a rewrite rule describes how to prove that the origin and the result of rewriting stand in the relation r. This information is used by the rewrite tactic to generate the corresponding NUPRL proof. There are two types of justifications.

Computational Justifications are lists of precise applications of the forward and reverse direct computation rules. As these are comparatively very fast and generate no well-formedness subgoals, the rewrite package uses these whenever possible.

Tactic Justifications are more generally applicable, but make extensive use of lemmata (see Section 8.9.1.4 below) and often generate many well-formedness subgoals.

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

8.9.1.4 Lemma Support

The rewrite package must have access to several kinds of lemmata in order to construct justifications for rewrites. This section describes those lemmata.

Functionality Lemmata give congruence and monotonicity properties of terms. They are required by conversionals like `SubC` to construct justifications for rewriting terms based on the justifications for rewriting the immediate subterms of those terms. A functionality lemma for a term with operator *op* should have the form

$$\forall z_1{:}S_1..z_k{:}S_k.\forall x_1,y_1{:}T_1..x_n,y_n{:}T_n.\ A_1 \Rightarrow \ldots \Rightarrow A_m \\ \Rightarrow x_1 r_1 y_1 \Rightarrow \ldots \Rightarrow x_n r_n y_n \Rightarrow op(x_1;..;x_n)\ r\ op(y_1;..;y_n)$$

where $k,m \geq 0$. The universal quantifiers and A's can be intermixed, but the antecedents containing the r_i must come afterward and be in the same order as the subterms of *op*.

If *op* binds variables in its subterms, then these variables should be bound by universal quantifiers wrapped around the appropriate r_i antecedents. For example, the lemma for functionality of $\exists$ with respect to the $\Leftrightarrow$ relation is:

$$\forall A_1,A_2{:}\mathbb{U}_i.\forall P_1{:}A_1 \Rightarrow \mathbb{P}_i.\forall P_2{:}A_2 \Rightarrow \mathbb{P}_i.A_1 = A_2 \in \mathbb{U}_i \\ \Rightarrow (\forall x{:}A_1.\ P_1[x] \Leftrightarrow P_2[x])\ \Rightarrow\ \exists x{:}A_1.P_1[x]\ \Leftrightarrow\ \exists x{:}A_2.P_2[x]$$

To allow conversionals to find functionality lemmata in the library, they should be named ***opid*** `_functionality`[***_index***] where *opid* is the operator identifier of *op* and ***_index*** is an optional suffix. When more than one functionality lemma is created for a given operator, they must be ordered with the most specific $r_1 \ldots r_n$ first, as conversionals search for functionality lemmata in the order in which they appear.

Functionality lemmata are not needed when all the r_i and r are primitive equalities. In this case functionality information can be derived from the well-formedness lemma for *op*.

Transitivity Lemmata give transitivity information for rewrite relations. They are used to construct the justification in sequencing conversionals like `ANDTHENC` and should have the form:

$$\forall z_1{:}S_1..z_k{:}S_k.\forall x_1,x_2,x_3{:}T.\ A_1 \Rightarrow \ldots \Rightarrow A_m \Rightarrow x_1\ r_a\ x_2 \Rightarrow x_2\ r_b\ x_3 \Rightarrow x_1\ r_c\ x_3$$

where $k,m \geq 0$ and r_c should be the weaker of r_a and r_b. Transitivity lemmata should be named ***opid-of-***r_c`_transitivity`[***_index***]. Transitivity lemmata are not needed for primitive equality relations.

Weakening Lemmata extend the usefulness of the transitivity and functionality lemmata. They should have the form

$$\forall z_1{:}S_1..z_k{:}S_k.\forall x_1,x_2{:}T.\ A_1 \Rightarrow \ldots \Rightarrow A_m \Rightarrow x_1\ r_a\ x_2 \Rightarrow x_1\ r_b\ x_2,$$

where $k,m \geq 0$ and r_b is weaker than r_a, and be named ***opid-of-***r_b`_weakening`[***_index***]. Weakening lemmata are required for all reflexive relations r_b with r_a being equality.

Inversion Lemmata are used by the `Rev*` atomic conversions and in conjunction with weakening, transitivity, and functionality lemmata when these mix order and equivalence relations. They are required for equivalence relations, but not for equality or order relations. Inversion lemmata should have the form

$$\forall z_1{:}S_1..z_k{:}S_k.\forall x_1,x_2{:}T.\ A_1 \Rightarrow \ldots \Rightarrow A_m \Rightarrow x_1\ r\ x_2 \Rightarrow x_2\ r\ x_1$$

where $k,m \geq 0$ and should be named ***opid-of-***r`_inversion`.

Note that for order relations one only needs lemmata for one direction as the other can be derived from them. For example, one does not require both the lemma $\forall a, b, c.\ a{\le}b \Rightarrow b{\le}c \Rightarrow a{\le}c$ and $\forall a, b, c.\ a{\ge}b \Rightarrow b{\ge}c \Rightarrow a{\ge}c$.

If NUPRL finds a lemma missing in the course of constructing a rewrite justification it prints out an error message suggesting the kind and structure of the missing lemma. After adding an appropriate lemma to the library, you need to evaluate the function `initialize_rw_lemma_caches ()` to make it accessible to the rewrite package.

8.9.1.5 Applying Conversions

The following tactics can be used to make conversions applicable to some clause of a proof goal.

`Rewrite` *c i*

Apply conversion *c* to clause *i*. The subgoal with the result of the conversion is labelled `main` while the labels of the other subgoals fall into the `aux` class.

A shorthand notation for `Rewrite` is `RW`. The following variants of `Rewrite` provide a controlled application of the conversion *c* to the subterms of clause *i* by combining `Rewrite` with conversionals (see Section 8.9.4) in various fashions.

`RWH` *c i* ≡ `RW (HigherC` *c*`)` *i* :
: Apply *c* to the first possible subterm of clause *i*, starting from the root.

`RWU` *c i* ≡ `RW (SweepUpC` *c*`)` *i* :
: Apply *c* to all subterms of clause *i*, starting from the leaves.

`RWD` *c i* ≡ `RW (SweepDnC` *c*`)` *i* :
: Apply *c* to all subterms of clause *i*, starting from the root.

`RWN` *n c i* ≡ `RW (NthC` *n c*`)` *i* :
: Apply *c* to the *n*-th immediate subterm of clause *i*.

`RWAddr` *addr c i* ≡ `RW (AddrC` *addr c*`)` *i* :
: Apply *c* to the subterm of clause *i* whose address in *addr*

`RewriteType` *c i*

Apply conversion *c* to the type of a member or equality term in clause *i*.
The advantage of this tactic over `Rewrite` is that this generates simpler well-formedness goals. In particular, it generates no well-formedness goals involving the equands of the equality or the element of the member term. A shorthand notation for `RewriteType` is `RWT`

For testing the effect of a conversion *c* on a term *t* with an empty environment one may evaluate the expression `apply_conv` *c t* in the refiner top loop.

8.9.1.6 Conversion Arguments

The descriptions of conversions in the sections below assume that the conversions have been applied to an environment *e* and a term *t*. Types of arguments to conversions are:

c∗	: `convn`	*type of conversions*
e∗	: `env`	*environment*
i j	: `int`	*hypothesis or clause indices*
addr	: `int list`	*subterm address*
a	: `tok`	*name of abstraction*
name	: `tok`	*name of lemma or cached conversion*
t∗	: `term`	

A suffix *s* on the name of an argument indicates that it is a list. For example *cs* is considered to have type `conv list`.

8.9.2 Atomic Conversions

Atomic conversions are the basic building blocks for constructing conversions. They may rewrite terms according to given lemmata and hypotheses, fold and unfold abstractions, or evaluate primitive and abstract redices. For the sake of completeness, there are also two trivial conversions.

`IdC`
: The identity conversion, which does not change a term

`FailC`
: The conversion that always fails

8.9.2.1 Lemma and Hypothesis Conversions

Lemma and hypothesis conversions derive rewrite rules from lemmata and hypotheses that contain either simple or general universal formulae (see Section 8.2.5). The consequents of these formulae must be of form $a\ r\ b$, where r is a relation as described in Section 8.9.1.2.

Usually we describe these conversions as rewriting in a left-to-right direction: they replace instances of a's by instances of b's. Each conversion also has a twin conversion that works right-to-left, which is indicated by a prefix `Rev` to their names.

`LemmaC` *name*
`RevLemmaC` *name*
: If lemma *name* contains a simple universal formula with consequent $a\ r\ b$, rewrite instances of a to instances of b (or instances of b to instances of a, respectively).

`HypC` *i*
`RevHypC` *i*
: If hypothesis *i* contains a simple universal formula with consequent $a\ r\ b$, rewrite instances of a to instances of b (or instances of b to instances of a, respectively).

The above conversions are instances of two more general conversions

```
GenLemmaWithThenLC (n:int) (hints:(var#term) list) (Tacs:tactic list) (name:tok)
GenHypWithThenLC   (n:int) (hints:(var#term) list) (Tacs:tactic list) (i:int)
```

which rewrite according to *general* universal formulae in lemmata and hypotheses. The meaning of their arguments is as follows:

n indicates the *n*-th consequent of a general universal formula. If `-1` is used then the formula is always treated as simple. In particular a $\Leftrightarrow$ relation will be considered the relation in the consequent rather than a part of the structure of the general universal formula.

hints supply bindings for variables in the formula that NUPRL's matching routines cannot guess.

Tacs is used for *conditional* rewriting, i.e. when the antecedents of a formula have to be checked for validity before the rewrite rule is used.
The tactics in *Tacs* are paired up with the subgoals formed from instantiated antecedents. The rewrite goes through only if each tactic completely proves its corresponding subgoal. If there are fewer tactics than antecedents, an appropriate number of copies of the head of *Tacs* will be added to left of *Tacs*. If *Tacs* is empty, then rewriting must go through unconditionally.

name is the name of the lemma.

i is the number of a hypothesis. Negative numbers are allowed (c.f. Section 8.2.1.1).

Other useful specializations of `GenLemmaWithThenLC` and `GenHypWithThenLC` are:

`GenLemmaC` *n name*	≡	`GenLemmaWithThenLC` *n* `[] []` *name*
`LemmaWithC` *hints name*	≡	`GenLemmaWithThenLC (-1)` *hints* `[]` *name*
`LemmaThenLC` *Tacs name*	≡	`GenLemmaWithThenLC (-1) []` *Tacs name*
`GenHypC` *n i*	≡	`GenHypWithThenLC` *n* `[] []` *i*
`HypWithC` *hints i*	≡	`GenHypWithThenLC (-1)` *hints* `[]` *i*
`HypThenLC` *Tacs i*	≡	`GenHypWithThenLC (-1) []` *Tacs i*

The hypothesis conversions described here derive their rewrite rules from *local* environments (Section 8.9.1.1) that they are presented with on their first applications. If the conversions are applied with conversionals such as `HigherC` or `NthC` that start applying a conversion at the top of a term, then the environment is always the same as the environment of the clause being rewritten.

8.9.2.2 Atomic Direct-Computation Conversions

Low level direct-computation conversions are not usually invoked directly by the user, but useful for controlling the evaluation of redices and the folding/unfolding of abstractions in advanced rewrite strategies. Computation conversions for interactive invocation are described in Section 8.9.3.

`UnfoldTopAbC`
`UnfoldsTopC` *as*
`UnfoldTopC` *a* ≡ `UnfoldsTopC [`*a*`]`
  Unfold t if it is an abstraction (with operator identifier listed in *as*).

`AUnfoldsTopC` *attrs*
  Unfold t if it is an abstraction that has any of the attributes in *attrs* (see Section 7.1.3).

`RecUnfoldTopC` *a*
  Unfold the recursive definition of *a* if it occurs on the top level of t.

`FoldsTopC` *as*
`FoldTopC` *a* ≡ `FoldsTopC [`*a*`]`
  Try to fold an instance of an abstraction whose operator identifier is listed in *as*

`RecFoldTopC` *a*
  Try to fold an instance of the recursively defined term *a* on the top level of t.

`RecUnfoldTopC` and `RecFoldTopC` work only with recursive definitions that have been introduced with the `AddRecDef*` button (see Section 4.3.2.2).

`TagC` *tagger*
  Do forward computations on the term t as indicated by the tags in `(`*tagger* t`)`.

`RedexC`
  Contract t if it is a primitive redex.

`AbRedexC`
`ForceRedexC` *force*
  Contract t if it is a primitive or abstract redex (of strength less or equal to *force*).

`AnyExtractC`
`ExtractC` *names*
  Expand t if it is an extract term (of a theorem listed in *names*).

Abstract redices. Primitive redices that are buried under abstractions are called *abstract redices*. For example, the first and second projection functions for pairs are abstractions:

```
*A pi1     t.1 ≡ let <x,y> = t in x
*A pi2     t.2 ≡ let <x,y> = t in y
```

and the term `⌞<a,b>.1⌟` is an abstract redex, which contracts to the term `⌞a⌟`.

To contract abstract redices, the conversion `AbRedexC` consults an abstract redex table, whose entries are created using the function

`add_AbReduce_conv` *opid c*

where *opid* is the operator identifier of the outermost term of the redex[10] and *c* is a conversion for contracting instances of the redex. Instances of `add_AbReduce_conv` are usually included in ML objects positioned immediately after the definitions of non-canonical abstractions.

An alternative method for indicating an abstract redex is to associate a 'reducible' attribute (see Section 7.1.3) with a non-canonical abstraction using the function

`add_reducible_ab` *opid*

The `AbRedexC` conversion first unfolds all reducible abstractions at the top level of the term before further analyzing it to see if it is a redex. When this method is applicable, it is more concise than using `add_AbReduce_conv`.

Reduction Strengths and Forces. In some situations it is desirable to have some redices contracted but not others. To this end, one may specify the *strength* of a redex and provide an optional *force* argument to tactics invoking direct computation conversions. Strengths and forces are arranged in a partial order on ML tokens. A redex is contracted only if the reduction force applied to it is greater or equal to its strength. A strength is associated with a redex in two ways.

- The strength is directly associated with the reduction rule for the redex.
- The strength is associated with the canonical term that is the principal argument of the redex. Strengths can be associated with canonical terms using the function

  `note_reduction_strength` *opid strength*

The currently supported strengths, in increasing order, are

`‘1‘` beta redices
`‘2‘` other primitive redices
`‘3‘` abstract redices recursive
`‘4‘` abstract redices non-recursive
`‘6‘` module projection functions with coercion arguments
`‘7‘` functions creating module elements from concrete parts.
`‘8‘` quasi-canonical redices.
`‘9‘` irreducible terms.

Contraction conversions that should be sensitive to the force of reduction can be added to the abstract redex table using the function

`add_ForceReduce_conv` *opid c*

where *opid* is the operator identifier of the outermost term of the redex and *c* is a conversion that takes a token argument for the force with which contraction of the redex is being attempted.

[10]If the outermost term is an (iterated) `apply` term, then *opid* refers to the operator identifier of the term at the head of the application.

8.9.3 Composite Direct Computation Conversions

The following conversions are commonly used for folding and unfolding abstractions and for the evaluation of primitive and abstract redices in a term.

`UnfoldsC` *as*
`UnfoldC` *a*
Unfold all occurrences of abstractions listed in *as*, starting at the leaves of t.

`RecUnfoldC` *a*
Unfold all occurrences of the recursive definition of *a* in t.

`FoldsC` *as*
`FoldC` *a*
Fold all instances of abstractions whose operator identifiers are listed in *as*.

`RecFoldC` *a*
Fold all instances of the recursively defined term *a* in t.

`ReduceC` ≡ `AbReduceC` ≡ `PrimReduceC`
`ForceReduceC` *force*
Repeatedly contract all redices in t (with maximal strength *force*) starting from the root.

`EvalC` *as*
Repeatedly unfold abstractions listed in *as* starting at the leaves, and then contract all redices.

`NormalizeC`
`SemiNormC` *as*
Repeatedly unfold abstractions (listed in *as*), starting at the root, and then contract redices.

`IntSimpC`
Rewrite t into arithmetical canonical form.

8.9.4 Conversionals

Conversionals provide the means for sequencing conversions in a way similar to the basic tacticals described in Section 8.8.1 and to apply them to subterms in a controlled fashion.

c_1 `ANDTHENC` c_2
Apply c_2 to the result of c_1. Fail if either c_1 or c_2 fails.

c_1 `ORTHENC` c_2
Apply c_2 to the result of c_1, or to t if c_1 fails

c_1 `ORELSEC` c_2
Apply c_1. If this fails, apply c_2.

`RepeatC` *c*
`RepeatForC` *n c*
Repeat *c* until it fails (exactly *n* times).

`ProgressC` *c*
Apply *c*, but fail if *c* does not change the term.

`TryC` *c*
Apply *c*. If this fails, leave the term unchanged.

`AllC` *cs*
Iteratively apply all conversions from *cs*. Fail if one of them fails.

`SomeC` *cs*
Iteratively apply all applicable conversions from *cs*.

FirstC *cs*
: Apply the first applicable conversion from *cs*.

SubC *c*
: Apply *c* to all immediate subterms of t (in left-to-right order).

SubIfC *p c*
: Apply *c* to all immediate subterms of t that satisfy the proposition *p*

NthSubC *n c*
: Apply *c* to the *n*-th immediate subterm of t.

AddrC *addr c*
: Apply *c* to the subterm of t with term address *addr*.

HigherC *c*

LowerC *c*
: Apply *c* the first applicable subterm of t, starting from the root (leaves).

NthC *n c*
: Execute the *n*-th successful application of *c* to the subterms of t, starting from the root.

SweepDnC *c*

SweepUpC *c*
: Apply *c* to all subterms of t, starting from the root (leaves).

TopC *c*

DepthC *c*
: Repeatedly apply *c* to the first applicable subterm of t, starting from the root (leaves).

8.9.5 Macro Conversions

Macro conversions allow to express rewrite rules via pattern terms that describe the left and the right hand side of a transformation.

MacroC *name* c_1 t_1 c_2 t_2
: Rewrite an instance of t_1 to the corresponding instance of t_2 using forward and reverse computation steps.
c_1 and c_2 must be direct computation conversions that rewrite the pattern terms t_1 and t_2 to the same term. `MacroC` uses second-order matching when matching instance terms against t_1. *name* is a failure token, which will be returned if rewriting fails.

SimpleMacroC *name* t_1 t_2 *as*
: Rewrite an instance of t_1 to an instance of t_2 by unfolding abstractions from *as* and contracting redices.

FwdMacroC *name c t*
: Rewrite an instance of t to an instance of the term resulting from applying *c* to t.

Using macro conversions enables a user to express rewrite steps in a user-defined theory in a way that does not reveal the underlying type theory. An example for the use of `MacroC` is the conversion for unrolling the Y combinator, defined in the theory `core_2`:

```
*A ycomb          Y ==  λf.(λx.f (x x)) (λx.f (x x))
*M ycomb_unroll  let YUnrollC =
                    MacroC ‘YUnrollC‘
                     (AllC [UnfoldC ‘ycomb‘; RedexC; RedexC])     ⌜Y F⌝
                     (AllC [UnfoldC ‘ycomb‘; AddrC [2] RedexC])  ⌜F (Y F)⌝
                  ;;
```

For another example, look at the `length_unroll` object in the `list_1` theory. `SimpleMacroC` can be used if the rewrite steps involved in justifying the equivalence of t_1 and t_2 are simpler, as in the case of the abstract redex for the projection functions on products

```
let pi1_evalC =
  SimpleMacroC `pi1_evalC` ⌜<a, b>.1⌝ ⌜a⌝ [`pi1`]
;;
```

Appendix A

The Basic Nuprl Type Theory

A.1 Syntax

Nuprl terms have the form $\texttt{opid}\{p_1{:}F_1,..,p_k{:}F_k\}(x_1^1,..,x_{m_1}^1.t_1;\ \ldots;\ x_1^n,..,x_{m_n}^n.t_n)$. We name the parts of a term as follows:

- $\texttt{opid}\{p_1{:}F_1,..,p_k{:}F_k\}$ is the *operator*.

  The parts of the operator are:

  - **opid** is the *operator identifier*.
  - $p_j{:}F_j$ is the j-th *parameter*. p_j is the *parameter value* and F_j is the *parameter type*.

- The tuple $(m_1, \ldots, m_n)$, where $m_j \geq 0$ is the *arity* of the term.
- $s_i = x_1^i,..,x_{m_i}^i.t_i$ is the i-th *bound-term* of the term, which binds free occurrences of the *variables* $x_1^i,\ldots,x_{m_i}^i$ in t_i.

When writing terms, we sometimes omit the {} brackets around the parameter list if it is empty. Note that parameters are separated by commas while subterms are separated by semicolons.

A.1.1 Operator Identifiers

Operator identifiers are character strings drawn from the alphabet[1] `a–z A–Z 0–9 _ - !`. An `!` at the start of a character string indicates that the term does not belong to Nuprl's object language. Operator identifiers are implemented using ML type `tok`. Valid operators are listed in Nuprl's *operator table*, which contains the basic operators[2] given in Table A.1 as well as conservative language extensions defined by abstractions in the library.

[1] We distinguish between the ASCII character `-` and the character range $x-y$, indicating the characters from x to y.

[2] Note that the operator identifier of a term is not always identical to the name a user has to type into Nuprl's term editor in order to generate the corresponding display template. The latter depends only on the information provided in the display form while the abstract name can only be used to enter terms in the abstract (expanded) mode. Different names are, for instance, used for the simple inductive types (simplerec instead of rec) and the less_than predicate (lt instead of `less_than`). A complete list of display forms for the basic terms can be found in the system's `core_1` theory.

Canonical (Types)	(Members)	**noncanonical**
	var$\{x{:}\mathtt{v}\}()$ x	
function$\{\}(S;\, x.T)$ $x{:}S\rightarrow T$, $S\rightarrow T$	**lambda**$\{\}(x.t)$ $\lambda x.t$	**apply**$\{\}(\boxed{f};t)$ $\boxed{f}\ t$
product$\{\}(S;\, x.T)$ $x{:}S\times T$, $S\times T$	**pair**$\{\}(s;t)$ $\langle s,t\rangle$	**spread**$\{\}(\boxed{e};\, x,y.u)$ let $\langle x,y\rangle = \boxed{e}$ in u
union$\{\}(S;T)$ $S{+}T$	**inl**$\{\}(s)$, **inr**$\{\}(t)$ inl(s), inr(t)	**decide**$\{\}(\boxed{e};\, x.u;\, y.v)$ case $\boxed{e}$ of inl$(x) \mapsto u \mid$ inr$(y) \mapsto v$
universe$\{j{:}\mathtt{l}\}()$ $\mathbb{U}_j$	*– All types of level j –*	
equal$\{\}(s;t;T)$ $s = t \in T$	**Axiom**$\{\}()$ Ax	
void$\{\}()$ void	*– No canonical elements –*	**any**$\{\}(e)$ any(e)
Atom$\{\}()$ Atom	**token**$\{\mathtt{token}{:}\mathtt{t}\}()$ "token"	**atom_eq**$(\boxed{u};\, \boxed{v};\, s;\, t)$ if $\boxed{u}{=}\boxed{v}$ then s else t
int$\{\}()$ $\mathbb{Z}$	**natural_number**$\{n{:}\mathtt{n}\}()$ n	**ind**$\{\}(\boxed{u}, x.f_x;\, s;\, \mathtt{base}, y.f_y)t$ ind$(\boxed{u}, x.f_x;\, s;\, \mathtt{base}, y.f_y)t$
	minus$\{\}($**natural_number**$\{n{:}\mathtt{n}\}())$ $-n$	**minus**$\{\}(\boxed{u})$, **add**$\{\}(\boxed{u};\boxed{v})$, **sub**$\{\}(\boxed{u};\boxed{v})$ $-\boxed{u}$, $\boxed{u}{+}\boxed{v}$, $\boxed{u}{-}\boxed{v}$
		mul$\{\}(\boxed{u};\boxed{v})$, **div**$\{\}(\boxed{u};\boxed{v})$, **rem**$\{\}(\boxed{u};\boxed{v})$ $\boxed{u}{*}\boxed{v}$, $\boxed{u}{\div}\boxed{v}$, $\boxed{u}$ rem $\boxed{v}$
		int_eq$(\boxed{u};\, \boxed{v};\, s;\, t)$, **less**$(\boxed{u};\, \boxed{v};\, s;\, t)$ if $\boxed{u}{=}\boxed{v}$ then s else t, if $\boxed{u}{<}\boxed{v}$ then s else t
less_than$\{\}(u;v)$ $u{<}v$	**Axiom**$\{\}()$ Ax	
list$\{\}(T)$ T list	**nil**$\{\}()$, **cons**$\{\}(t;l)$ $[]$, $t{::}l$	**list_ind**$\{\}(\boxed{s};\, \mathtt{base};\, x,l,f_{xl}.t)$ list_ind$(\boxed{s};\, \mathtt{base};\, x,l,f_{xl}.t)$
rec$\{\}(X.T_X)$ rectype $X = T_X$	*– members defined by unrolling T_X –*	**rec_ind**$\{\}(\boxed{e};\, f,x.t)$ let* $f(x) = t$ in $f(\boxed{e})$
set$\{\}(S;\, x.T)$ $\{x{:}S \mid T\}$, $\{S \mid T\}$	*– members of S that satisfy P –*	
isect$\{\}(S;\, x.T)$ $\cap x{:}S.T$	*– Terms that belong to all $T[x]$ –*	
quotient$\{\}(T;\, x,y.E)$ $x,y : T/\!/E$	*– members of T, new equality –*	

Table A.1: Basic operators of Nuprl's Type Theory. Terms are divided into canonical and noncanonical terms. Principal arguments in noncanonical terms are marked by a $\boxed{\text{box}}$. Standard display forms of terms are written below the abstract representation. The distinction between types and members is *not* a part of Nuprl's syntax.

A.1.2 Parameters

Parameters $p\!:\!F$ consist of a parameter name p and a parameter family F. The current parameter families and associated values are:

`variable` : Names of variables, implemented using the ML data type `var`.
Acceptable names are generated by the regular expression [`a−z A−Z 0−9 _ - %`]$^+$. The `%` character has a special use.

`natural` : Natural numbers (including 0), implemented using the ML data type `int`.
Acceptable numbers are generated by the regular expression $\mathtt{0} + [\mathtt{1}-\mathtt{9}][\mathtt{0}-\mathtt{9}]^*$.

`token` : Character strings, implemented using the ML data type `tok`.
Acceptable strings can draw from any non-control characters in Nuprl's font.

`string` : Character strings, implemented using the ML data type `string`.
Acceptable strings can draw from any non-control characters in Nuprl's font.

`level-expression` : Universe level expressions, implemented using the ML data type `level_exp`.
Universe level expressions are used to index universe levels in Nuprl's type theory. Their syntax is described by the grammar $L ::= v \mid k \mid L\ i \mid L' \mid [L|\cdots|L]$ where v denotes a level-expression variable (alphanumeric string), k a level expression constant (positive integer), and i a level expression increment (non-negative integer).

Level-expression variables are implicitly quantified over all positive integer levels. The expression $L\ i$ is interpreted as standing for levels $L+i$. L' is an abbreviation for $L\ \mathtt{1}$. The expression $[L_1|\cdots|L_n]$ is interpreted as being the maximum of expressions $L_1 \cdots L_n$.

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

A.1.3 Binding Variables

Binding variables are character strings drawn from the same alphabet as variable parameters. To express terms without bindings, the empty string can be used as *null variable*. Null variables never bind. Binding variables are implemented using ML type `var`.

A.1.4 Injection of Variables and Numbers

In Nuprl, we consider variables and terms to be distinct. We have a special term kind, `variable{v}` for injecting variables into the term type. When we talk of the variable x *as a term*, we really mean the term `variable`{x:`v`}. In a similar way, when we talk of the number n *as a term*, we really mean the term `natural_number`{n:`n`}.

The injection is often made implicitly when it is clear from the context. Nuprl's editor atomatically converts variables and numbers into terms when they are typed into templates for terms.

A.1.5 Term Display

The *display* of NUPRL terms is not necessarily identical to their abstract form. Usually, they are presented in a more conventional notation, which is created by the display forms described in Chapter 7.2. In Table A.1 we present the standard display of NUPRL terms immediately below their abstract form.

Redex	Contractum
$(\boxed{\lambda x . u})\ t$	$\xrightarrow{\beta}\ u[t/x]$
let $\langle x, y\rangle = \boxed{\langle s, t\rangle}$ in u	$\xrightarrow{\beta}\ u[s,t/x,y]$
case $\boxed{\mathsf{inl}(s)}$ of inl$(x) \mapsto u$ | inr$(y) \mapsto v$	$\xrightarrow{\beta}\ u[s/x]$
case $\boxed{\mathsf{inr}(t)}$ of inl$(x) \mapsto u$ | inr$(y) \mapsto v$	$\xrightarrow{\beta}\ v[t/y]$
if $\boxed{a}$=$\boxed{b}$ then s else t	$\xrightarrow{\beta}$ *s, if a = b; t, otherwise*
ind$(\boxed{0}, x.f_x;\ s;\ \textbf{base}, y.f_y)t$	$\xrightarrow{\beta}$ **base**
ind$(\boxed{i}, x.f_x;\ s;\ \textbf{base}, y.f_y)t$	$\xrightarrow{\beta}\ t[i,\mathsf{ind}(\boxed{i\text{-}1}, x.f_x;\ s;\ \textbf{base}, y.f_y)t\ /y, f_y]$, $(i>0)$
ind$(\boxed{-i}, x.f_x;\ s;\ \textbf{base}, y.f_y)t$	$\xrightarrow{\beta}\ s[-i,\mathsf{ind}(\boxed{-i\text{+}1}, x.f_x;\ s;\ \textbf{base}, y.f_y)t\ /x, f_x]$, $(i>0)$
$-\boxed{i}$	$\xrightarrow{\beta}$ *The negation of i (as number)*
$\boxed{i}+\boxed{j}$	$\xrightarrow{\beta}$ *The sum of i and j*
$\boxed{i}-\boxed{j}$	$\xrightarrow{\beta}$ *The difference of i and j*
$\boxed{i}*\boxed{j}$	$\xrightarrow{\beta}$ *The product of i and j*
$\boxed{i}\div\boxed{j}$	$\xrightarrow{\beta}$ *0, if j=0; the integer division of i and j, otherwise*
$\boxed{i}$ rem $\boxed{j}$	$\xrightarrow{\beta}$ *0, if j=0; the division rest of i and j, otherwise*
if $\boxed{i}$=$\boxed{j}$ then s else t	$\xrightarrow{\beta}$ *s, if i = j; t, otherwise*
if $\boxed{i}$<$\boxed{j}$ then s else t	$\xrightarrow{\beta}$ *s, if i < j; t, otherwise*
list_ind$(\boxed{[]};\ base;\ x, l, f_{xl}.t)$	$\xrightarrow{\beta}$ *base*
list_ind$(\boxed{s::u};\ base;\ x, l, f_{xl}.t)$	$\xrightarrow{\beta}\ t[s,u,\mathsf{list\_ind}(u;\ base;\ x, l, f_{xl}.t)/x, l, f_{xl}]$
let* $f(x) = t$ in $f(\boxed{e})$	$\xrightarrow{\beta}\ t[\lambda y.\mathsf{let}^*\ f(x) = t\ \mathsf{in}\ f(y), e\ /\ f, x]$

Table A.2: Redex–Contracta Table for Nuprl's Type Theory: the $\boxed{\text{principal arguments}}$ must be in the corresponding canonical form

A.2 Semantics

A.2.1 Evaluation

Nuprl's semantics is based on a notion of values. Terms are divided into *canonical* forms, i.e. values, and *noncanonical* forms, i.e. terms that need to be evaluated. Evaluation in Nuprl is *lazy*: whether a term is canonical or not depends solely on its operator identifier but not on its subterms. In noncanonical forms, certain subterms are marked as *principal arguments*. If a principal argument is instantiated with a matching canonical form, the expression becomes *reducible* (i.e. a *redex*) and can be evaluated to its *contractum*, defined in a *redex–contracta table*.

Nuprl's *evaluation mechanism* first computes the values of all principal arguments of a noncanonical expression. If an argument does not have a value or if the resulting expression is not reducible, evaluation stops: the expression has no value. Otherwise the expression will be reduced according to redex–contracta table and the resulting term will be evaluated.

Canonical forms and noncanonical forms together with their principal arguments are given in Table A.1. The corresponding redex–contracta table is given in Table A.2.

$x_1{:}S_1{\to}T_1 = x_2{:}S_2{\to}T_2$	if $S_1{=}S_2$ and $T_1[s_1/x_1]{=}T_2[s_2/x_2]$ for all s_1, s_2 with $s_1{=}s_2 \in S_1$.
$T = S_2{\to}T_2$	if $T = x_2{:}S_2{\to}T_2$ for some $x_2 \in \mathcal{V}$.
$S_1{\to}T_1 = T$	if $x_1{:}S_1{\to}T_1 = T$ for some $x_1 \in \mathcal{V}$.
$x_1{:}S_1{\times}T_1 = x_2{:}S_2{\times}T_2$	if $S_1{=}S_2$ and $T_1[s_1/x_1]{=}T_2[s_2/x_2]$ for all s_1, s_2 with $s_1{=}s_2 \in S_1$.
$T = S_2{\times}T_2$	if $T = x_2{:}S_2{\times}T_2$ for some variable x_2.
$S_1{\times}T_1 = T$	if $x_1{:}S_1{\times}T_1 = T$ for some variable x_1.
$S_1{+}T_1 = S_2{+}T_2$	if $S_1{=}S_2$ and $T_1{=}T_2$.
$\mathbb{U}_{j_1} = \mathbb{U}_{j_2}$	if $j_1{=}j_2$ *(as natural number)*
$s_1{=}t_1 \in T_1 = s_2{=}t_2 \in T_2$	if $T_1{=}T_2$, $s_1{=}s_2{\in}T_1$, and $t_1{=}t_2{\in}T_1$.
void = void	
Atom = Atom	
$\mathbb{Z} = \mathbb{Z}$	
$i_1{<}j_1 = i_2{<}j_2$	if $i_1 = i_2 \in \mathbb{Z}$ and $j_1 = j_2 \in \mathbb{Z}$
T_1 list $= T_2$ list	if $T_1 = T_2$
rectype $X_1{=}T_1$ = rectype $X_2{=}T_2$	if $T_1[X/X_1] = T_2[X/X_2]$ for all types X
$\{x_1{:}S_1 \mid T_1\} = \{x_2{:}S_2 \mid T_2\}$	if $S_1{=}S_2$ and there are terms p_1, p_2 and a variable x, which occurs neither in T_1 nor in T_2, such that $p_1 \in \forall x{:}S_1.\ T_1[x/x_1] \Rightarrow T_2[x/x_2]$ and $p_2 \in \forall x{:}S_1.\ T_2[x/x_2] \Rightarrow T_1[x/x_1]$.
$T = \{S_2 \mid T_2\}$	if $T = \{x_2{:}S_2 \mid T_2\}$ for some variable x_2.
$\{S_1 \mid T_1\} = T$	if $\{x_1{:}S_1 \mid T_1\} = T$ for some variable x_1.
$\cap x_1{:}S_1.T_1 = \cap x_2{:}S_2.T_2$	if $S_1{=}S_2$ and $T_1[s_1/x_1]{=}T_2[s_2/x_2]$ for all s_1, s_2 with $s_1{=}s_2 \in S_1$.
$x_1,y_1 : T_1/\!/E_1 = x_2,y_2 : T_2/\!/E_2$	if $T_1 = T_2$ and there are terms p_1,p_2,r,s,t and variables x, y, z, which occur neither in E_1 nor in E_2, such that $p_1 \in \forall x{:}T_1.\forall y{:}T_1.\ E_1[x,y/x_1,y_1] \Rightarrow E_2[x,y/x_2,y_2]$, $p_2 \in \forall x{:}T_1.\forall y{:}T_1.\ E_2[x,y/x_2,y_2] \Rightarrow E_1[x,y/x_1,y_1]$, $r \in \forall x{:}T_1.\ E_1[x,x/x_1,y_1]$, $s \in \forall x{:}T_1.\forall y{:}T_1.\ E_1[x,y/x_1,y_1] \Rightarrow E_1[y,x/x_1,y_1]$, and $t \in \forall x{:}T_1.\forall y{:}T_1.\forall z{:}T_1.$ $E_1[x,y/x_1,y_1] \Rightarrow E_1[y,z/x_1,y_1] \Rightarrow E_1[x,z/x_1,y_1]$

Table A.3: Type semantics table for Nuprl

A.2.2 Judgments

The meaning of type theoretical expressions is given in the form of *judgments* about essential properties of the terms. Judgments are assertions of certain truths that form the foundation of type theory. We distinguish 4 types of judgments: *Typehood* (T Type), *Type Equality* ($S{=}T$), *Membership* ($t \in T$), and *Member Equality* ($s{=}t \in T$). The precise meaning of these judgments is defined as follows.

T Type if $T{=}T$.

$S{=}T$ if there are canonical terms S' and T' such that $S \xrightarrow{*} S'$, $T \xrightarrow{*} T'$ and $S'{=}T'$ follows from the *type semantics table*.

$t \in T$ if $t{=}t \in T$.

$s{=}t \in T$ if there are canonical terms s', t' and T' such that $s \xrightarrow{*} s'$, $t \xrightarrow{*} t'$, $T{=}T'$, and $s'{=}t' \in T'$ follows from the *member semantics table*.

Nuprl's type semantics table is given in Table A.3 and its member semantics table in Table A.4.

$\lambda x_1.t_1 = \lambda x_2.t_2 \in x{:}S\to T$	if $x{:}S\to T$ Type and $t_1[s_1/x_1] = t_2[s_2/x_2] \in T[s_1/x]$ for all s_1,s_2 with $s_1=s_2\in S$.
$\langle s_1,t_1\rangle = \langle s_2,t_2\rangle \in x{:}S\times T$	if $x{:}S\times T$ Type, $s_1=s_2\in S$, and $t_1=t_2\in T[s_1/x]$.
$\mathsf{inl}(s_1) = \mathsf{inl}(s_2) \in S+T$	if $S+T$ Type and $s_1=s_2\in S$.
$\mathsf{inr}(t_1) = \mathsf{inr}(t_2) \in S+T$	if $S+T$ Type and $t_1=t_2\in T$
$\mathsf{Ax} = \mathsf{Ax} \in s=t\in T$	if $s = t \in T$
$s = t \in \mathsf{void}$	***never holds !***
$\ddot{}token\ddot{} = \ddot{}token\ddot{} \in \mathsf{Atom}$	
$i = i \in \mathbb{Z}$	
$\mathsf{Ax} = \mathsf{Ax} \in s<t$	if $s \xrightarrow{*} i$ and $t \xrightarrow{*} j$ for some integers i, j with $i<j$
$[] = [] \in T\ \mathsf{list}$	if T Type
$t_1{::}l_1 = t_2{::}l_2 \in T\ \mathsf{list}$	if T Type, $t_1=t_2\in T$, and $l_1=l_2\in T\ \mathsf{list}$
$s = t \in \mathsf{rectype}\ X=T_X$	if $\mathsf{rectype}\ X=T_X$ Type and $s = t \in T_X[\mathsf{rectype}\ X=T_X/X]$
$s = t \in \{x{:}S \mid T\}$	if $\{x{:}S \mid T\}$ Type, $s=t\in S$, and there is some term $p\in T[s/x]$
$t_1 = t_2 \in \cap x{:}S.T$	if $\cap x{:}S.T$ Type and $t_1=t_2\in T[s/x]$ for all $s\in S$.
$s = t \in x,y : T//E$	if $x,y : T//E$ Type, $s\in T$, $t\in T$, and there is some term $p\in E[s,t/x,y]$
$x_1{:}S_1\to T_1 = x_2{:}S_2\to T_2 \in \mathbb{U}_j$	if $S_1=S_2\in\mathbb{U}_j$ and $T_1[s_1/x_1]=T_2[s_2/x_2]\in\mathbb{U}_j$ for all s_1,s_2 with $s_1=s_2\in S_1$
$T = S_2\to T_2 \in \mathbb{U}_j$	if $T = x_2{:}S_2\to T_2 \in \mathbb{U}_j$ for some variable x_2
$S_1\to T_1 = T \in \mathbb{U}_j$	if $x_1{:}S_1\to T_1 = T \in \mathbb{U}_j$ for some variable x_1
$x_1{:}S_1\times T_1 = x_2{:}S_2\times T_2 \in \mathbb{U}_j$	if $S_1=S_2\in\mathbb{U}_j$ and $T_1[s_1/x_1]=T_2[s_2/x_2]\in\mathbb{U}_j$ for all s_1,s_2 with $s_1=s_2\in S_1$
$T = S_2\times T_2 \in \mathbb{U}_j$	if $T = x_2{:}S_2\times T_2 \in \mathbb{U}_j$ for some variable x_2
$S_1\times T_1 = T \in \mathbb{U}_j$	if $x_1{:}S_1\times T_1 = T \in \mathbb{U}_j$ for some variable x_1
$S_1+T_1 = S_2+T_2 \in \mathbb{U}_j$	if $S_1=S_2\in\mathbb{U}_j$ and $T_1=T_2\in\mathbb{U}_j$
$s_1=t_1\in T_1 = s_2=t_2\in T_2 \in \mathbb{U}_j$	if $T_1=T_2\in\mathbb{U}_j$, $s_1=s_2\in T_1$, and $t_1=t_2\in T_1$.
$\mathbb{U}_{j_1} = \mathbb{U}_{j_2} \in \mathbb{U}_j$	if $j_1=j_2<j$ *(as natural number)*
$\mathsf{void} = \mathsf{void} \in \mathbb{U}_j$	
$\mathsf{Atom} = \mathsf{Atom} \in \mathbb{U}_j$	
$\mathbb{Z} = \mathbb{Z} \in \mathbb{U}_j$	
$i_1<j_1 = i_2<j_2 \in \mathbb{U}_j$	if $i_1 = i_2 \in \mathbb{Z}$ und $j_1 = j_2 \in \mathbb{Z}$
$T_1\ \mathsf{list} = T_2\ \mathsf{list} \in \mathbb{U}_j$	if $T_1 = T_2 \in \mathbb{U}_j$
$\mathsf{rectype}\ X_1=T_1 = \mathsf{rectype}\ X_2=T_2 \in \mathbb{U}_j$	if $T_1[X/X_1] = T_2[X/X_2] \in \mathbb{U}_j$ for all $X \in \mathbb{U}_j$
$\{x_1{:}S_1 \mid T_1\} = \{x_2{:}S_2 \mid T_2\} \in \mathbb{U}_j$	if $S_1=S_2\in\mathbb{U}_j$ and there are terms p_1, p_2 and a variable x, which occurs neither in T_1 nor in T_2, such that $p_1 \in \forall x{:}S_1.\ T_1[x/x_1]\Rightarrow T_2[x/x_2]$ and $p_2 \in \forall x{:}S_1.\ T_2[x/x_2]\Rightarrow T_1[x/x_1]$.
$T = \{S_2 \mid T_2\} \in \mathbb{U}_j$	if $T = \{x_2{:}S_2 \mid T_2\} \in \mathbb{U}_j$ for some variable x_2.
$\{S_1 \mid T_1\} = T \in \mathbb{U}_j$	if $\{x_1{:}S_1 \mid T_1\} = T \in \mathbb{U}_j$ for some variable x_1.
$\cap x_1{:}S_1.T_1 = \cap x_2{:}S_2.T_2 \in \mathbb{U}_j$	if $S_1=S_2\in\mathbb{U}_j$ and $T_1[s_1/x_1]=T_2[s_2/x_2]\in\mathbb{U}_j$ for all s_1,s_2 with $s_1=s_2\in S_1$.
$x_1,y_1 : T_1//E_1 = x_2,y_2 : T_2//E_2 \in \mathbb{U}_j$	if $T_1 = T_2 \in \mathbb{U}_j$ and there are terms p_1,p_2,r,s,t and variables x,y,z, which occur neither in E_1 nor in E_2, such that $p_1 \in \forall x{:}T_1.\forall y{:}T_1.\ E_1[x,y/x_1,y_1]\Rightarrow E_2[x,y/x_2,y_2]$, $p_2 \in \forall x{:}T_1.\forall y{:}T_1.\ E_2[x,y/x_2,y_2]\Rightarrow E_1[x,y/x_1,y_1]$, $r \in \forall x{:}T_1.\ E_1[x,x/x_1,y_1]$, $s \in \forall x{:}T_1.\forall y{:}T_1.\ E_1[x,y/x_1,y_1]\Rightarrow E_1[y,x/x_1,y_1]$, and $t \in \forall x{:}T_1.\forall y{:}T_1.\forall z{:}T_1.$ $E_1[x,y/x_1,y_1]\Rightarrow E_1[y,z/x_1,y_1]\Rightarrow E_1[x,z/x_1,y_1]$

Table A.4: Member semantics table for Nuprl

A.3 Inference rules

Nuprl's inference rules describe the top-down refinement of proof sequents (see Chapter 6) and the bottom-up construction of extract terms. Rules are written in a top-down fashion, showing the goal sequent above the rule name and the subgoal sequents below it.

For each type there are rules for type formation and type equality, formation and equality of canonical members, equality of noncanonical forms, type decomposition in hypotheses (elimination), computation rules, and possibly additional rules. In addition to the rule name, a rule may need certain arguments (see Section 8.1.2) such as

- The position of a hypothesis to be used as in `hypothesis` $\boxed{i}$
- Names for newly created variables as in `functionEquality` $\boxed{x}$
- The universe level of a type as in `lambdaEquality` $\boxed{j}$ x'
- A term that instantiates a variable as in `dependent_pairFormation` j $\boxed{s}$ x'
- The type of some subterm in the goal as in `applyEquality` $\boxed{x{:}S{\rightarrow}T}$
- The dependency of a term $C[z]$ from a variable z as in `decideEquality` $\boxed{z\ \ C}$ $S{+}T$ s t y

Most of the elementary inference rules are subsumed by the one-step decomposition tactics `D`, `MemCD`, `EqCD`, `MemHD`, `EqHD`, `MemTypeCD`, `EqTypeCD`, `MemTypeHD`, `EqTypeHD`. These tactics try to determine the parameters of the corresponding rules from the context unless they are explicitly provided with the tacticals `New`, `At`, `With`, or `Using` (see Section 8.2.2). A user may choose to use these tacticals to support the tactics in situations where appropriate parameters cannot be found automatically or in order to enforce the use of, for instance, particular names for newly created variables.

In the following we present the basic inference rules of NUPRL's type theory as well as the tactics that can be used to perform the same one-step decomposition of proof goals. For the latter we describe both the minimal form, which only lists tactics that are needed for injecting required arguments, and a maximal form with all tacticals that may have an effect on the execution of the tactic. Some rules that are now considered obsolete are not covered by tactics and have to be converted explicitly into tactics using the function `refine` (see Section 8.1.3).

For integer and list induction we use the abstract terms instead of the lengthy display forms. Goals of the form $a \in T$ always abbreviate a=$a \in T$. $\neg P$ stands for $P{\rightarrow}\mathsf{void}$, $s{\neq}t$ for $\neg(s{=}t \in \mathbb{Z})$, and $s{\leq}t$ for $\neg(t{<}s)$.

A.3.1 Functions

$\Gamma \vdash \mathbb{U}_j$ $\lfloor$**ext** $x{:}S{\to}T\rfloor$
  by dependent_functionFormation x S
  $\Gamma \vdash S \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
  $\Gamma,\ x{:}S \vdash \mathbb{U}_j$ $\lfloor$**ext** $T\rfloor$

$\Gamma \vdash \mathbb{U}_j$ $\lfloor$**ext** $S{\to}T\rfloor$
  by independent_functionFormation
  $\Gamma \vdash \mathbb{U}_j$ $\lfloor$**ext** $S\rfloor$
  $\Gamma \vdash \mathbb{U}_j$ $\lfloor$**ext** $T\rfloor$

$\Gamma \vdash x_1{:}S_1{\to}T_1 = x_2{:}S_2{\to}T_2 \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
  by functionEquality x
  $\Gamma \vdash S_1 = S_2 \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
  $\Gamma,\ x{:}S_1 \vdash T_1[x/x_1] = T_2[x/x_2] \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$

$\Gamma \vdash S_1{\to}T_1 = S_2{\to}T_2 \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
  by independent_functionEquality
  $\Gamma \vdash S_1 = S_2 \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
  $\Gamma \vdash T_1 = T_2 \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$

$\Gamma \vdash \lambda x_1.t_1 = \lambda x_2.t_2 \in x{:}S{\to}T$ $\lfloor$Ax$\rfloor$
  by lambdaEquality j x'
  $\Gamma,\ x'{:}S \vdash t_1[x'/x_1] = t_2[x'/x_2] \in T[x'/x]$ $\lfloor$Ax$\rfloor$
  $\Gamma \vdash S \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$

$\Gamma \vdash x{:}S{\to}T$ $\lfloor$**ext** $\lambda x'.t\rfloor$
  by lambdaFormation j x'
  $\Gamma,\ x'{:}S \vdash T[x'/x]$ $\lfloor$**ext** $t\rfloor$
  $\Gamma \vdash S \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$

$\Gamma \vdash f_1 t_1 = f_2 t_2 \in T[t_1/x]$ $\lfloor$Ax$\rfloor$
  by applyEquality $x{:}S{\to}T$
  $\Gamma \vdash f_1 = f_2 \in x{:}S{\to}T$ $\lfloor$Ax$\rfloor$
  $\Gamma \vdash t_1 = t_2 \in S$ $\lfloor$Ax$\rfloor$

$\Gamma,\ f{:}S{\to}T,\ \Delta \vdash C$ $\lfloor$**ext** $t[f\,s,/y]\rfloor$
  by independent_functionElimination i y
  $\Gamma,\ f{:}S{\to}T,\ \Delta \vdash S$ $\lfloor$**ext** $s\rfloor$
  $\Gamma,\ f{:}S{\to}T,\ y{:}T,\ \Delta \vdash C$ $\lfloor$**ext** $t\rfloor$

$\Gamma,\ f{:}x{:}S{\to}T,\ \Delta \vdash C$ $\lfloor$**ext** $t[f\,s,\mathsf{Ax}/y,z]\rfloor$
  by dependent_functionElimination i s y z
  $\Gamma,\ f{:}x{:}S{\to}T,\ \Delta \vdash s \in S$ $\lfloor$Ax$\rfloor$
  $\Gamma,\ f{:}x{:}S{\to}T,\ y{:}T[s/x],\ z{:}y{=}f\,s \in T[s/x],\ \Delta \vdash C$ $\lfloor$**ext** $t\rfloor$

$\Gamma \vdash (\lambda x.t)\ s = t_2 \in T$ $\lfloor$Ax$\rfloor$
  by applyReduce
  $\Gamma \vdash t[s/x] = t_2 \in T$ $\lfloor$Ax$\rfloor$

$\Gamma \vdash f_1 = f_2 \in x{:}S{\to}T$ $\lfloor$**ext** $t\rfloor$
  by functionExtensionality j $x_1{:}S_1{\to}T_1$ $x_2{:}S_2{\to}T_2$ x'
  $\Gamma,\ x'{:}S \vdash f_1 x' = f_2 x' \in T[x'/x]$ $\lfloor$**ext** $t\rfloor$
  $\Gamma \vdash S \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
  $\Gamma \vdash f_1 \in x_1{:}S_1{\to}T_1$ $\lfloor$Ax$\rfloor$
  $\Gamma \vdash f_2 \in x_2{:}S_2{\to}T_2$ $\lfloor$Ax$\rfloor$

Basic Inference Rule	*Corresponding Tactic* *with required arguments*	*with optional tacticals*
dependent_functionFormation x S	---	
independent_functionFormation	---	
functionEquality x	EqCD	New $[x]$ EqCD
independent_functionEquality	---	
lambdaEquality j x'	EqCD	At $\mathbb{U}_j$ EqCD
lambdaFormation j x'	D 0	
applyEquality $x{:}S{\to}T$	EqCD	With $x{:}S{\to}T$ EqCD
independent_functionElimination i y	D i	
dependent_functionElimination i s y z	D i	
applyReduce	ReduceEquands 0	ReduceAtAddr [2] 0
functionExtensionality j $x_1{:}S_1{\to}T_1$ $x_2{:}S_2{\to}T_2$ x'	EqExtWith	Ext

A.3.2 Products

$\Gamma \vdash \mathbb{U}_j \quad \lfloor\textbf{ext } x{:}S{\times}T\rfloor$
by dependent_productFormation x S
$\Gamma \vdash S \in \mathbb{U}_j \quad \lfloor Ax\rfloor$
$\Gamma,\ x{:}S \vdash \mathbb{U}_j \quad \lfloor\textbf{ext } T\rfloor$

$\Gamma \vdash \mathbb{U}_j \quad \lfloor\textbf{ext } S{\times}T\rfloor$
by independent_productFormation
$\Gamma \vdash \mathbb{U}_j \quad \lfloor\textbf{ext } S\rfloor$
$\Gamma \vdash \mathbb{U}_j \quad \lfloor\textbf{ext } T\rfloor$

$\Gamma \vdash x_1{:}S_1{\times}T_1 = x_2{:}S_2{\times}T_2 \in \mathbb{U}_j \quad \lfloor Ax\rfloor$
by productEquality x'
$\Gamma \vdash S_1 = S_2 \in \mathbb{U}_j \quad \lfloor Ax\rfloor$
$\Gamma,\ x'{:}S_1 \vdash T_1[x'/x_1] = T_2[x'/x_2] \in \mathbb{U}_j \quad \lfloor Ax\rfloor$

$\Gamma \vdash S_1{\times}T_1 = S_2{\times}T_2{\in}\mathbb{U}_j \quad \lfloor Ax\rfloor$
by independent_productEquality
$\Gamma \vdash S_1 = S_2 \in \mathbb{U}_j \quad \lfloor Ax\rfloor$
$\Gamma \vdash T_1 = T_2 \in \mathbb{U}_j \quad \lfloor Ax\rfloor$

$\Gamma \vdash \langle s_1,t_1\rangle = \langle s_2,t_2\rangle \in x{:}S{\times}T \quad \lfloor Ax\rfloor$
by dependent_pairEquality j x'
$\Gamma \vdash s_1 = s_2 \in S \quad \lfloor Ax\rfloor$
$\Gamma \vdash t_1 = t_2 \in T[s_1/x] \quad \lfloor Ax\rfloor$
$\Gamma,\ x'{:}S \vdash T[x'/x] \in \mathbb{U}_j \quad \lfloor Ax\rfloor$

$\Gamma \vdash x{:}S{\times}T \quad \lfloor\textbf{ext } \langle s,t\rangle\rfloor$
by dependent_pairFormation j s x'
$\Gamma \vdash s \in S \quad \lfloor Ax\rfloor$
$\Gamma \vdash T[s/x] \quad \lfloor\textbf{ext } t\rfloor$
$\Gamma,\ x'{:}S \vdash T[x'/x] \in \mathbb{U}_j \quad \lfloor Ax\rfloor$

$\Gamma \vdash \langle s_1,t_1\rangle = \langle s_1,t_1\rangle \in S{\times}T \quad \lfloor Ax\rfloor$
by independent_pairEquality
$\Gamma \vdash s_1 = s_2 \in S \quad \lfloor Ax\rfloor$
$\Gamma \vdash t_1 = t_2 \in T \quad \lfloor Ax\rfloor$

$\Gamma \vdash S{\times}T \quad \lfloor\textbf{ext } \langle s,t\rangle\rfloor$
by independent_pairFormation
$\Gamma \vdash S \quad \lfloor\textbf{ext } s\rfloor$
$\Gamma \vdash T \quad \lfloor\textbf{ext } t\rfloor$

$\Gamma \vdash \text{let } \langle x_1,y_1\rangle = e_1 \text{ in } t_1 = \text{let } \langle x_2,y_2\rangle = e_2 \text{ in } t_2 \in C[e_1/z] \quad \lfloor Ax\rfloor$
by spreadEquality z C $x{:}S{\times}T$ s t y
$\Gamma \vdash e_1 = e_2 \in x{:}S{\times}T \quad \lfloor Ax\rfloor$
$\Gamma,\ s{:}S,\ t{:}T[s/x],\ y{:}\ e_1{=}\langle s,t\rangle{\in}x{:}S{\times}T \vdash t_1[s,t/x_1,y_1] = t_2[s,t/x_2,y_2] \in C[\langle s,t\rangle/z] \quad \lfloor Ax\rfloor$

$\Gamma,\ z{:}\,x{:}S{\times}T,\ \Delta \vdash C \quad \lfloor\textbf{ext } \text{let } \langle s,t\rangle = z \text{ in } u\rfloor$
by productElimination i s t
$\Gamma,\ z{:}\,x{:}S{\times}T,\ s{:}S,\ t{:}T[s/x]\ \Delta[\langle s,t\rangle/z] \vdash C[\langle s,t\rangle/z] \quad \lfloor\textbf{ext } u\rfloor$

$\Gamma \vdash \text{let } \langle x,y\rangle = \langle s,t\rangle \text{ in } u = t_2 \in T \quad \lfloor Ax\rfloor$
by spreadReduce
$\Gamma \vdash u[s,t/x,y] = t_2 \in T \quad \lfloor Ax\rfloor$

Basic Inference Rule	*Corresponding Tactic* *with required arguments*	*with optional tacticals*
dependent_productFormation x S	---	
independent_productFormation	---	
productEquality x'	EqCD	
independent_productEquality	---	
dependent_pairEquality j x'	EqCD	
dependent_pairEquality2 j x'	EqCD	
dependent_pairFormation j s x'	With s (D 0)	At $\mathbb{U}_j$ (With s (New [x'] (D 0)))
independent_pairEquality	EqCD	
independent_pairFormation	D 0	
spreadEquality z C $x{:}S{\times}T$ s t y	EqCD	
productElimination i s t	D i	
spreadReduce	ReduceEquands 0	ReduceAtAddr [2] 0

A.3.3 Disjoint Union

$\Gamma \vdash \mathbb{U}_j \ \lfloor \textbf{ext } S{+}T \rfloor$
  by unionFormation
  $\Gamma \vdash \mathbb{U}_j \ \lfloor \textbf{ext } S \rfloor$
  $\Gamma \vdash \mathbb{U}_j \ \lfloor \textbf{ext } T \rfloor$

$\Gamma \vdash S_1{+}T_1 = S_2{+}T_2 \in \mathbb{U}_j \ \lfloor \text{Ax} \rfloor$
  by unionEquality
  $\Gamma \vdash S_1 = S_2 \in \mathbb{U}_j \ \lfloor \text{Ax} \rfloor$
  $\Gamma \vdash T_1 = T_2 \in \mathbb{U}_j \ \lfloor \text{Ax} \rfloor$

$\Gamma \vdash \mathsf{inl}(s_1) = \mathsf{inl}(s_2) \in S{+}T \ \lfloor \text{Ax} \rfloor$
  by inlEquality j
  $\Gamma \vdash s_1 = s_2 \in S \ \lfloor \text{Ax} \rfloor$
  $\Gamma \vdash T \in \mathbb{U}_j \ \lfloor \text{Ax} \rfloor$

$\Gamma \vdash S{+}T \ \lfloor \textbf{ext } \mathsf{inl}(s) \rfloor$
  by inlFormation j
  $\Gamma \vdash S \ \lfloor \textbf{ext } s \rfloor$
  $\Gamma \vdash T \in \mathbb{U}_j \ \lfloor \text{Ax} \rfloor$

$\Gamma \vdash \mathsf{inr}(t_1) = \mathsf{inr}(t_2) \in S{+}T \ \lfloor \text{Ax} \rfloor$
  by inrEquality j
  $\Gamma \vdash t_1 = t_2 \in T \ \lfloor \text{Ax} \rfloor$
  $\Gamma \vdash S \in \mathbb{U}_j \ \lfloor \text{Ax} \rfloor$

$\Gamma \vdash S{+}T \ \lfloor \textbf{ext } \mathsf{inr}(t) \rfloor$
  by inrFormation j
  $\Gamma \vdash T \ \lfloor \textbf{ext } t \rfloor$
  $\Gamma \vdash S \in \mathbb{U}_j \ \lfloor \text{Ax} \rfloor$

$\Gamma \vdash \mathsf{case}\ e_1\ \mathsf{of}\ \mathsf{inl}(x_1) \mapsto u_1 \mid \mathsf{inr}(y_1) \mapsto v_1 = \mathsf{case}\ e_2\ \mathsf{of}\ \mathsf{inl}(x_2) \mapsto u_2 \mid \mathsf{inr}(y_2) \mapsto v_2 \in C[e_1/z] \ \lfloor \text{Ax} \rfloor$
  by decideEquality z C $S{+}T$ s t y
  $\Gamma \vdash e_1 = e_2 \in S{+}T \ \lfloor \text{Ax} \rfloor$
  $\Gamma,\ s{:}S,\ y{:}\ e_1{=}\mathsf{inl}(s) \in S{+}T \vdash u_1[s/x_1] = u_2[s/x_2] \in C[\mathsf{inl}(s)/z] \ \lfloor \text{Ax} \rfloor$
  $\Gamma,\ t{:}T,\ y{:}\ e_1{=}\mathsf{inr}(t) \in S{+}T \vdash v_1[t/y_1] = v_2[t/y_2] \in C[\mathsf{inr}(t)/z] \ \lfloor \text{Ax} \rfloor$

$\Gamma,\ z{:}S{+}T,\ \Delta \vdash C \ \lfloor \textbf{ext } \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u \mid \mathsf{inr}(y) \mapsto v \rfloor$
  by unionElimination i x y
  $\Gamma,\ z{:}S{+}T,\ x{:}S,\ \Delta[\mathsf{inl}(x)/z] \vdash C[\mathsf{inl}(x)/z] \ \lfloor \textbf{ext } u \rfloor$
  $\Gamma,\ z{:}S{+}T,\ y{:}T,\ \Delta[\mathsf{inr}(y)/z] \vdash C[\mathsf{inr}(y)/z] \ \lfloor \textbf{ext } v \rfloor$

$\Gamma \vdash \mathsf{case}\ \mathsf{inl}(s)\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u \mid \mathsf{inr}(y) \mapsto v = t_2 \in T \ \lfloor \text{Ax} \rfloor$
  by decideReduceLeft
  $\Gamma \vdash u[s/x] = t_2 \in T \ \lfloor \text{Ax} \rfloor$

$\Gamma \vdash \mathsf{case}\ \mathsf{inr}(t)\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u \mid \mathsf{inr}(y) \mapsto v = t_2 \in T \ \lfloor \text{Ax} \rfloor$
  by decideReduceRight
  $\Gamma \vdash v[t/y] = t_2 \in T \ \lfloor \text{Ax} \rfloor$

Basic Inference Rule	*Corresponding Tactic* *with required arguments*	*with optional tacticals*
unionFormation	---	
unionEquality	EqCD	
inlEquality j	EqCD	
inlFormation j	Sel 1 D 0	
inrEquality j	EqCD	
inrFormation j	Sel 2 D 0	
decideEquality z C $S{+}T$ s t y	EqCD	Using [z,C] (With $S{+}T$ (New s t y EqCD))
unionElimination i x y	D i	
decideReduceLeft	ReduceEquands 0	ReduceAtAddr [2] 0
decideReduceRight	ReduceEquands 0	ReduceAtAddr [2] 0

A.3.4 Universes

$\Gamma \vdash \mathbb{U}_k$ $\lfloor$**ext** $\mathbb{U}_j\rfloor$
by `universeFormation` j *

$\Gamma \vdash \mathbb{U}_j = \mathbb{U}_j \in \mathbb{U}_k$ $\lfloor$Ax$\rfloor$
by `universeEquality` *

$\Gamma \vdash T \in \mathbb{U}_k$ $\lfloor$Ax$\rfloor$
by `cumulativity` j *
$\Gamma \vdash T \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$

*: *proviso* $j < k$

Basic Inference Rule	*Corresponding Tactic* *with required arguments with optional tacticals*
`universeFormation` j	`D 0`
`universeEquality`	`EqCD`
`cumulativity` j	`Cumulativity` j

A.3.5 Equality

$\Gamma \vdash \mathbb{U}_j$ $\lfloor$**ext** $s = t \in T\rfloor$
by equalityFormation T
  $\Gamma \vdash T \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
  $\Gamma \vdash T$ $\lfloor$**ext** $s\rfloor$
  $\Gamma \vdash T$ $\lfloor$**ext** $t\rfloor$

$\Gamma \vdash s_1 = t_1 \in T_1 = s_2 = t_2 \in T_2 \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
by equalityEquality
  $\Gamma \vdash T_1 = T_2 \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
  $\Gamma \vdash s_1 = s_2 \in T_1$ $\lfloor$Ax$\rfloor$
  $\Gamma \vdash t_1 = t_2 \in T_1$ $\lfloor$Ax$\rfloor$

$\Gamma \vdash$ Ax = Ax $\in s = t \in T$ $\lfloor$Ax$\rfloor$
by axiomEquality
  $\Gamma \vdash s = t \in T$ $\lfloor$Ax$\rfloor$

$\Gamma,\ z\colon s = t \in T,\ \Delta \vdash C$ $\lfloor$**ext** $u\rfloor$
by equalityElimination i
  $\Gamma,\ z\colon s = t \in T,\ \Delta[\mathsf{Ax}/z] \vdash C[\mathsf{Ax}/z]$ $\lfloor$**ext** $u\rfloor$

$\Gamma,\ x{:}T,\ \Delta \vdash x = x \in T$ $\lfloor$Ax$\rfloor$
by hypothesisEquality i

$\Gamma \vdash C[s/x]$ $\lfloor$**ext** $u\rfloor$
by substitution j $s{=}t \in T$ x C
  $\Gamma \vdash s{=}t \in T$ $\lfloor$Ax$\rfloor$
  $\Gamma \vdash C[t/x]$ $\lfloor$**ext** $u\rfloor$
  $\Gamma,\ x{:}T \vdash C \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$

$\Gamma \vdash s = t \in T$ $\lfloor$Ax$\rfloor$
by equality

Decision procedure for elementary equalities

Basic Inference Rule	*Corresponding Tactic* *with required arguments*	*with optional tacticals*
equalityFormation T	---	
equalityEquality	EqCD	
axiomEquality	EqCD	
equalityElimination i	D i	
hypothesisEquality i	Declaration	NthDecl i
substitution j $s{=}t \in T$ x C	Subst $s{=}t \in T$ 0	At j (BasicSubst $s{=}t \in T$ x C)
equality	Eq	Eq

A.3.6 Void

$\Gamma \vdash \mathbb{U}_j$ $\lfloor$**ext** void$\rfloor$
by voidFormation

$\Gamma \vdash$ void = void $\in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
by voidEquality

$\Gamma \vdash$ any(s) = any(t) $\in T$ $\lfloor$Ax$\rfloor$
by anyEquality
$\Gamma \vdash s = t \in$ void $\lfloor$Ax$\rfloor$

Γ, z: void, $\Delta \vdash C$ $\lfloor$**ext** any(z)$\rfloor$
by voidElimination i

Basic Inference Rule	*Corresponding Tactic* *with required arguments with optional tacticals*
voidFormation	---
voidEquality	EqCD
anyEquality	EqCD
voidElimination i	D i

A.3.7 Atom

$\Gamma \vdash \mathbb{U}_j$ $\lfloor$**ext** Atom$\rfloor$
by atomFormation

$\Gamma \vdash$ Atom = Atom $\in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
by atomEquality

$\Gamma \vdash$ "*token*" = "*token*" $\in$ Atom $\lfloor$Ax$\rfloor$
by tokenEquality

$\Gamma \vdash$ Atom $\lfloor$**ext** "*token*"$\rfloor$
by tokenFormation "*token*"

$\Gamma \vdash$ if u_1=v_1 then s_1 else t_1 = if u_2=v_2 then s_2 else t_2 $\in T$ $\lfloor$Ax$\rfloor$
by atom_eqEquality v

$\Gamma \vdash u_1 = u_2 \in$ Atom $\lfloor$Ax$\rfloor$
$\Gamma \vdash v_1 = v_2 \in$ Atom $\lfloor$Ax$\rfloor$
Γ, v: u_1=$v_1$$\in$Atom $\vdash s_1 = s_2 \in T$ $\lfloor$Ax$\rfloor$
Γ, v: $\neg$(u_1=$v_1$$\in$Atom) $\vdash t_1 = t_2 \in T$ $\lfloor$Ax$\rfloor$

$\Gamma \vdash$ if u=v then s else t = $t_2 \in T$ $\lfloor$Ax$\rfloor$
by atom_eqReduceTrue

$\Gamma \vdash s = t_2 \in T$ $\lfloor$Ax$\rfloor$
$\Gamma \vdash u = v \in$ Atom $\lfloor$Ax$\rfloor$

$\Gamma \vdash$ if u=v then s else t = $t_2 \in T$ $\lfloor$Ax$\rfloor$
by atom_eqReduceFalse

$\Gamma \vdash t = t_2 \in T$ $\lfloor$Ax$\rfloor$
$\Gamma \vdash \neg(u = v \in$ Atom) $\lfloor$Ax$\rfloor$

Basic Inference Rule	*Corresponding Tactic* *with required arguments*	*with optional tacticals*
atomFormation	---	
atomEquality	EqCD	
tokenEquality	EqCD	
tokenFormation "*token*"	D 0	
atom_eqEquality v	EqCD	
atom_eqReduceTrue	PrimReduceFirstEquand ‘true‘	PrimReduceEquands ‘true‘ [1]
atom_eqReduceFalse	PrimReduceFirstEquand ‘false‘	PrimReduceEquands ‘false‘ [1]

A.3.8 Integers

$\Gamma \vdash \mathbb{U}_j \quad \lfloor \textbf{ext } \mathbb{Z} \rfloor$
by `intFormation`

$\Gamma \vdash \mathbb{Z} \in \mathbb{U}_j \quad \lfloor \text{Ax} \rfloor$
by `intEquality`

$\Gamma \vdash n = n \in \mathbb{Z} \quad \lfloor \text{Ax} \rfloor$
by `natural_numberEquality`

$\Gamma \vdash \mathbb{Z} \quad \lfloor \textbf{ext } n \rfloor$
by `natural_numberFormation` n

$\Gamma \vdash \text{-}s_1 = \text{-}s_2 \in \mathbb{Z} \quad \lfloor \text{Ax} \rfloor$
by `minusEquality`
$\quad \Gamma \vdash s_1 = s_2 \in \mathbb{Z} \quad \lfloor \text{Ax} \rfloor$

$\Gamma \vdash s_1 + t_1 = s_2 + t_2 \in \mathbb{Z} \quad \lfloor \text{Ax} \rfloor$
by `addEquality`
$\quad \Gamma \vdash s_1 = s_2 \in \mathbb{Z} \quad \lfloor \text{Ax} \rfloor$
$\quad \Gamma \vdash t_1 = t_2 \in \mathbb{Z} \quad \lfloor \text{Ax} \rfloor$

$\Gamma \vdash \mathbb{Z} \quad \lfloor \textbf{ext } s + t \rfloor$
by `addFormation`
$\quad \Gamma \vdash \mathbb{Z} \quad \lfloor \textbf{ext } s \rfloor$
$\quad \Gamma \vdash \mathbb{Z} \quad \lfloor \textbf{ext } t \rfloor$

$\Gamma \vdash s_1 - t_1 = s_2 - t_2 \in \mathbb{Z} \quad \lfloor \text{Ax} \rfloor$
by `subtractEquality`
$\quad \Gamma \vdash s_1 = s_2 \in \mathbb{Z} \quad \lfloor \text{Ax} \rfloor$
$\quad \Gamma \vdash t_1 = t_2 \in \mathbb{Z} \quad \lfloor \text{Ax} \rfloor$

$\Gamma \vdash \mathbb{Z} \quad \lfloor \textbf{ext } s - t \rfloor$
by `subtractFormation`
$\quad \Gamma \vdash \mathbb{Z} \quad \lfloor \textbf{ext } s \rfloor$
$\quad \Gamma \vdash \mathbb{Z} \quad \lfloor \textbf{ext } t \rfloor$

$\Gamma \vdash s_1 * t_1 = s_2 * t_2 \in \mathbb{Z} \quad \lfloor \text{Ax} \rfloor$
by `multiplyEquality`
$\quad \Gamma \vdash s_1 = s_2 \in \mathbb{Z} \quad \lfloor \text{Ax} \rfloor$
$\quad \Gamma \vdash t_1 = t_2 \in \mathbb{Z} \quad \lfloor \text{Ax} \rfloor$

$\Gamma \vdash \mathbb{Z} \quad \lfloor \textbf{ext } s * t \rfloor$
by `multiplyFormation`
$\quad \Gamma \vdash \mathbb{Z} \quad \lfloor \textbf{ext } s \rfloor$
$\quad \Gamma \vdash \mathbb{Z} \quad \lfloor \textbf{ext } t \rfloor$

$\Gamma \vdash s_1 \div t_1 = s_2 \div t_2 \in \mathbb{Z} \quad \lfloor \text{Ax} \rfloor$
by `divideEquality`
$\quad \Gamma \vdash s_1 = s_2 \in \mathbb{Z} \quad \lfloor \text{Ax} \rfloor$
$\quad \Gamma \vdash t_1 = t_2 \in \mathbb{Z} \quad \lfloor \text{Ax} \rfloor$
$\quad \Gamma \vdash t_1 \neq 0 \quad \lfloor \text{Ax} \rfloor$

$\Gamma \vdash \mathbb{Z} \quad \lfloor \textbf{ext } s \div t \rfloor$
by `divideFormation`
$\quad \Gamma \vdash \mathbb{Z} \quad \lfloor \textbf{ext } s \rfloor$
$\quad \Gamma \vdash \mathbb{Z} \quad \lfloor \textbf{ext } t \rfloor$

$\Gamma \vdash s_1 \,\mathsf{rem}\, t_1 = s_2 \,\mathsf{rem}\, t_2 \in \mathbb{Z} \quad \lfloor \text{Ax} \rfloor$
by `remainderEquality`
$\quad \Gamma \vdash s_1 = s_2 \in \mathbb{Z} \quad \lfloor \text{Ax} \rfloor$
$\quad \Gamma \vdash t_1 = t_2 \in \mathbb{Z} \quad \lfloor \text{Ax} \rfloor$
$\quad \Gamma \vdash t_1 \neq 0 \quad \lfloor \text{Ax} \rfloor$

$\Gamma \vdash \mathbb{Z} \quad \lfloor \textbf{ext } s \,\mathsf{rem}\, t \rfloor$
by `remainderFormation`
$\quad \Gamma \vdash \mathbb{Z} \quad \lfloor \textbf{ext } s \rfloor$
$\quad \Gamma \vdash \mathbb{Z} \quad \lfloor \textbf{ext } t \rfloor$

$\Gamma \vdash 0 \leq s \,\mathsf{rem}\, t \wedge s \,\mathsf{rem}\, t < t \quad \lfloor \text{Ax} \rfloor$
by `remainderBounds1`
$\quad \Gamma \vdash 0 \leq s \quad \lfloor \text{Ax} \rfloor$
$\quad \Gamma \vdash 0 < t \quad \lfloor \text{Ax} \rfloor$

$\Gamma \vdash 0 \leq s \,\mathsf{rem}\, t \wedge s \,\mathsf{rem}\, t < \text{-}t \quad \lfloor \text{Ax} \rfloor$
by `remainderBounds2`
$\quad \Gamma \vdash 0 \leq s \quad \lfloor \text{Ax} \rfloor$
$\quad \Gamma \vdash t < 0 \quad \lfloor \text{Ax} \rfloor$

$\Gamma \vdash s \,\mathsf{rem}\, t \leq 0 \wedge s \,\mathsf{rem}\, t > t \quad \lfloor \text{Ax} \rfloor$
by `remainderBounds3`
$\quad \Gamma \vdash s \leq 0 \quad \lfloor \text{Ax} \rfloor$
$\quad \Gamma \vdash t < 0 \quad \lfloor \text{Ax} \rfloor$

$\Gamma \vdash s \,\mathsf{rem}\, t \leq 0 \wedge s \,\mathsf{rem}\, t > \text{-}t \quad \lfloor \text{Ax} \rfloor$
by `remainderBounds4`
$\quad \Gamma \vdash s \leq 0 \quad \lfloor \text{Ax} \rfloor$
$\quad \Gamma \vdash 0 < t \quad \lfloor \text{Ax} \rfloor$

$\Gamma \vdash s = (s \div t) * t + (s \,\mathsf{rem}\, t) \quad \lfloor \text{Ax} \rfloor$
by `divideRemainderSum`
$\quad \Gamma \vdash s = s \in \mathbb{Z} \quad \lfloor \text{Ax} \rfloor$
$\quad \Gamma \vdash t \neq 0 \quad \lfloor \text{Ax} \rfloor$

$\Gamma \vdash \mathsf{ind}(u_1; x_1, f_{x_1}.s_1; base_1; y_1, f_{y_1}.t_1) = \mathsf{ind}(u_2; x_2, f_{x_2}.s_2; base_2; y_2, f_{y_2}.t_2) \in T[u_1/z]$ ⌊Ax⌋
by indEquality z T x f_x v

$\Gamma \vdash u_1 = u_2 \in \mathbb{Z}$ ⌊Ax⌋
$\Gamma,\ x{:}\mathbb{Z},\ v{:}\ x{<}0,\ f_x{:}T[(x{+}1)/z] \vdash s_1[x, f_x/x_1, f_{x1}] = s_2[x, f_x/x_2, f_{x2}] \in T[x/z]$ ⌊Ax⌋
$\Gamma \vdash base_1 = base_2 \in T[0/z]$ ⌊Ax⌋
$\Gamma,\ x{:}\mathbb{Z},\ v{:}\ 0{<}x,\ f_x{:}T[(x{-}1)/z] \vdash t_1[x, f_x/y_1, f_{y_1}] = t_2[x, f_x/y_2, f_{y_2}] \in T[x/z]$ ⌊Ax⌋

$\Gamma,\ z{:}\mathbb{Z},\ \Delta \vdash C$ ⌊**ext** $\mathsf{ind}(z;\ x, f_x.s[\mathsf{Ax}/v];\ base;\ x, f_x.t[\mathsf{Ax}/v])$⌋
by intElimination i x f_x v

$\Gamma,\ z{:}\mathbb{Z},\ \Delta,\ x{:}\mathbb{Z},\ v{:}\ x{<}0,\ f_x{:}C[(x{+}1)/z] \vdash C[x/z]$ ⌊**ext** s⌋
$\Gamma,\ z{:}\mathbb{Z},\ \Delta \vdash C[0/z]$ ⌊**ext** $base$⌋
$\Gamma,\ z{:}\mathbb{Z},\ \Delta,\ x{:}\mathbb{Z},\ v{:}\ 0{<}x,\ f_x{:}C[(x{-}1)/z] \vdash C[x/z]$ ⌊**ext** t⌋

$\Gamma \vdash \mathsf{ind}(i; x, f_x.s; base; y, f_y.t) = t_2 \in T$ ⌊Ax⌋
by indReduceDown

$\Gamma \vdash t[i, \mathsf{ind}(i{+}1; x, f_x.s; base; y, f_y.t)/x, f_x] = t_2 \in T$ ⌊Ax⌋
$\Gamma \vdash i < 0$ ⌊Ax⌋

$\Gamma \vdash \mathsf{ind}(i; x, f_x.s; base; y, f_y.t) = t_2 \in T$ ⌊Ax⌋
by indReduceUp

$\Gamma \vdash t[i, \mathsf{ind}(i{-}1; x, f_x.s; base; y, f_y.t)/y, f_y] = t_2 \in T$ ⌊Ax⌋
$\Gamma \vdash 0 < i$ ⌊Ax⌋

$\Gamma \vdash \mathsf{ind}(i; x, f_x.s; base; y, f_y.t) = t_2 \in T$ ⌊Ax⌋
by indReduceBase

$\Gamma \vdash base = t_2 \in T$ ⌊Ax⌋
$\Gamma \vdash i = 0 \in \mathbb{Z}$ ⌊Ax⌋

$\Gamma \vdash \mathsf{if}\ u_1{=}v_1\ \mathsf{then}\ s_1\ \mathsf{else}\ t_1 = \mathsf{if}\ u_2{=}v_2\ \mathsf{then}\ s_2\ \mathsf{else}\ t_2 \in T$ ⌊Ax⌋
by int_eqEquality

$\Gamma \vdash u_1 = u_2 \in \mathbb{Z}$ ⌊Ax⌋
$\Gamma \vdash v_1 = v_2 \in \mathbb{Z}$ ⌊Ax⌋
$\Gamma,\ v{:}\ u_1{=}v_1 \vdash s_1 = s_2 \in T$ ⌊Ax⌋
$\Gamma,\ v{:}\ u_1{\neq}v_1 \vdash t_1 = t_2 \in T$ ⌊Ax⌋

$\Gamma \vdash \mathsf{if}\ u{=}v\ \mathsf{then}\ s\ \mathsf{else}\ t = t_2 \in T$ ⌊Ax⌋
by int_eqReduceTrue

$\Gamma \vdash s = t_2 \in T$ ⌊Ax⌋
$\Gamma \vdash u = v \in \mathbb{Z}$ ⌊Ax⌋

$\Gamma \vdash \mathsf{if}\ u{=}v\ \mathsf{then}\ s\ \mathsf{else}\ t = t_2 \in T$ ⌊Ax⌋
by int_eqReduceFalse

$\Gamma \vdash t = t_2 \in T$ ⌊Ax⌋
$\Gamma \vdash u \neq v$ ⌊Ax⌋

$\Gamma \vdash \mathsf{if}\ u_1{<}v_1\ \mathsf{then}\ s\ \mathsf{else}\ t = \mathsf{if}\ u_2{<}v_2\ \mathsf{then}\ s_2\ \mathsf{else}\ t_2 \in T$ ⌊Ax⌋
by lessEquality

$\Gamma \vdash u_1 = u_2 \in \mathbb{Z}$ ⌊Ax⌋
$\Gamma \vdash v_1 = v_2 \in \mathbb{Z}$ ⌊Ax⌋
$\Gamma,\ v{:}\ u_1{<}v_1 \vdash s_1 = s_2 \in T$ ⌊Ax⌋
$\Gamma,\ v{:}\ u_1{\geq}v_1 \vdash t_1 = t_2 \in T$ ⌊Ax⌋

$\Gamma \vdash \mathsf{if}\ u{<}v\ \mathsf{then}\ s\ \mathsf{else}\ t = t_2 \in T$ ⌊Ax⌋
by lessReduceTrue

$\Gamma \vdash s = t_2 \in T$ ⌊Ax⌋
$\Gamma \vdash u < v$ ⌊Ax⌋

$\Gamma \vdash \mathsf{if}\ u{<}v\ \mathsf{then}\ s\ \mathsf{else}\ t = t_2 \in T$ ⌊Ax⌋
by lessReduceFalse

$\Gamma \vdash t = t_2 \in T$ ⌊Ax⌋
$\Gamma \vdash u \geq v$ ⌊Ax⌋

$\Gamma \vdash C$ $\lfloor$**ext** $t\rfloor$
by `arith` j
$\Gamma \vdash s_i \in \mathbb{Z}$ $\lfloor$Ax$\rfloor$

Decision procedure for elementary arithmetic

– subgoals for all non-arithmetical expressions s_i in C –

Basic Inference Rule	*Corresponding Tactic*	
	with required arguments	*with optional tacticals*
`intFormation`	`---`	
`intEquality`	`EqCD`	
`natural_numberEquality`	`EqCD`	
`natural_numberFormation` n	`With` n `(D 0)`	
`minusEquality`	`EqCD`	
`addEquality`	`EqCD`	
`addFormation`	`---`	
`subtractEquality`	`EqCD`	
`subtractFormation`	`---`	
`multiplyEquality`	`EqCD`	
`multiplyFormation`	`---`	
`divideEquality`	`EqCD`	
`divideFormation`	`---`	
`remainderEquality`	`EqCD`	
`remainderFormation`	`---`	
`remainderBounds1`		
`remainderBounds2`		
`remainderBounds3`		
`remainderBounds4`		
`divideRemainderSum`		
`indEquality` z T x f_x v	`StrongEqCD`	
`intElimination` i x f_x v	`D` i	
`indReduceDown`	`PrimReduceFirstEquand ‘down‘`	`PrimReduceEquands ‘down‘ [1]`
`indReduceUp`	`PrimReduceFirstEquand ‘up‘`	`PrimReduceEquands ‘up‘ [1]`
`indReduceBase`	`PrimReduceFirstEquand ‘base‘`	`PrimReduceEquands ‘base‘ [1]`
`int_eqEquality`	`EqCD`	
`int_eqReduceTrue`	`PrimReduceFirstEquand ‘true‘`	`PrimReduceEquands ‘true‘ [1]`
`int_eqReduceFalse`	`PrimReduceFirstEquand ‘false‘`	`PrimReduceEquands ‘false‘ [1]`
`lessEquality`	`EqCD`	
`lessReduceTrue`	`PrimReduceFirstEquand ‘true‘`	`PrimReduceEquands ‘true‘ [1]`
`lessReduceFalse`	`PrimReduceFirstEquand ‘false‘`	`PrimReduceEquands ‘false‘ [1]`
`arith` j	`Arith`	

A.3.9 Less_Than Proposition

$\Gamma \vdash s_1 < t_1 = s_2 < t_2 \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
by less_thanEquality
  $\Gamma \vdash s_1 = s_2 \in \mathbb{Z}$ $\lfloor$Ax$\rfloor$
  $\Gamma \vdash t_1 = t_2 \in \mathbb{Z}$ $\lfloor$Ax$\rfloor$

$\Gamma \vdash \mathbb{U}_j$ $\lfloor$**ext** $s<t\rfloor$
by less_thanFormation
  $\Gamma \vdash \mathbb{Z}$ $\lfloor$**ext** $s\rfloor$
  $\Gamma \vdash \mathbb{Z}$ $\lfloor$**ext** $t\rfloor$

$\Gamma \vdash$ Ax $\in s<t$ $\lfloor$Ax$\rfloor$
by less_thanMember
  $\Gamma \vdash s < t$ $\lfloor$Ax$\rfloor$

Basic Inference Rule	*Corresponding Tactic* *with required arguments with optional tacticals*
`less_thanEquality`	`EqCD`
`less_thanFormation`	`---`
`less_thanMember`	`EqCD`

A.3.10 Lists

$\Gamma \vdash \mathbb{U}_j \quad \lfloor \textbf{ext } T \text{ list} \rfloor$
by `listFormation`
$\quad \Gamma \vdash \mathbb{U}_j \quad \lfloor \textbf{ext } T \rfloor$

$\Gamma \vdash T_1 \text{ list} = T_2 \text{ list} \in \mathbb{U}_j \quad \lfloor \text{Ax} \rfloor$
by `listEquality`
$\quad \Gamma \vdash T_1 = T_2 \in \mathbb{U}_j \quad \lfloor \text{Ax} \rfloor$

$\Gamma \vdash [] = [] \in T \text{ list} \quad \lfloor \text{Ax} \rfloor$
by `nilEquality` j
$\quad \Gamma \vdash T \in \mathbb{U}_j \quad \lfloor \text{Ax} \rfloor$

$\Gamma \vdash T \text{ list} \quad \lfloor \textbf{ext } [] \rfloor$
by `nilFormation` j
$\quad \Gamma \vdash T \in \mathbb{U}_j \quad \lfloor \text{Ax} \rfloor$

$\Gamma \vdash t_1 :: l_1 = t_2 :: l_2 \in T \text{ list} \quad \lfloor \text{Ax} \rfloor$
by `consEquality`
$\quad \Gamma \vdash t_1 = t_2 \in T \quad \lfloor \text{Ax} \rfloor$
$\quad \Gamma \vdash l_1 = l_2 \in T \text{ list} \quad \lfloor \text{Ax} \rfloor$

$\Gamma \vdash T \text{ list} \quad \lfloor \textbf{ext } t :: l \rfloor$
by `consFormation`
$\quad \Gamma \vdash T \quad \lfloor \textbf{ext } t \rfloor$
$\quad \Gamma \vdash T \text{ list} \quad \lfloor \textbf{ext } l \rfloor$

$\Gamma \vdash \text{list\_ind}(s_1; \mathit{base}_1; x_1, l_1, f_{xl1}.t_1) = \text{list\_ind}(s_2; \mathit{base}_2; x_2, l_2, f_{xl2}.t_2) \in T[s_1/z] \quad \lfloor \text{Ax} \rfloor$
by `list_indEquality` z T $S \text{ list}$ x l f_{xl}
$\quad \Gamma \vdash s_1 = s_2 \in S \text{ list} \quad \lfloor \text{Ax} \rfloor$
$\quad \Gamma \vdash \mathit{base}_1 = \mathit{base}_2 \in T[[]/z] \quad \lfloor \text{Ax} \rfloor$
$\quad \Gamma,\ x{:}S,\ l{:}S \text{ list},\ f_{xl}{:}T[l/z] \vdash t_1[x, l, f_{xl}/x_1, l_1, f_{xl1}] = t_2[x, l, f_{xl}/x_2, l_2, f_{xl2}] \in T[x::l/z] \quad \lfloor \text{Ax} \rfloor$

$\Gamma,\ z{:}T \text{ list},\ \Delta \vdash C \quad \lfloor \textbf{ext } \text{list\_ind}(z; \mathit{base}; x, l, f_{xl}.t) \rfloor$
by `listElimination` i f_{xl} x l
$\quad \Gamma,\ z{:}T \text{ list},\ \Delta \vdash C[[]/z] \quad \lfloor \textbf{ext } \mathit{base} \rfloor$
$\quad \Gamma,\ z{:}T \text{ list},\ \Delta,\ x{:}T,\ l{:}T \text{ list},\ f_{xl}{:}C[l/z] \vdash C[x::l/z] \quad \lfloor \textbf{ext } t \rfloor$

$\Gamma \vdash \text{list\_ind}([]; \mathit{base}; x, l, f_{xl}.t) = t_2 \in T \quad \lfloor \text{Ax} \rfloor$
by `list_indReduceBase`
$\quad \Gamma \vdash \mathit{base} = t_2 \in T \quad \lfloor \text{Ax} \rfloor$

$\Gamma \vdash \text{list\_ind}(s::u; \mathit{base}; x, l, f_{xl}.t) = t_2 \in T \quad \lfloor \text{Ax} \rfloor$
by `list_indReduceUp`
$\quad \Gamma \vdash t[s, u, \text{list\_ind}(u; \mathit{base}; x, l, f_{xl}.t)/x, l, f_{xl}] = t_2 \in T \quad \lfloor \text{Ax} \rfloor$

Basic Inference Rule	*Corresponding Tactic* *with required arguments*	*with optional tacticals*
`listFormation`	`---`	
`listEquality`	`EqCD`	
`nilEquality` j	`EqCD`	
`nilFormation` j	`D 0`	
`consEquality`	`EqCD`	
`consFormation`	`---`	
`list_indEquality` z T $S \text{ list}$ x l f_{xl}	`EqCD`	
`listElimination` i f_{xl} x l	`D` i	
`list_indReduceBase`	`ReduceEquands 0`	`ReduceAtAddr [2] 0`
`list_indReduceUp`	`ReduceEquands 0`	`ReduceAtAddr [2] 0`

A.3.11 Inductive Types

$\Gamma \vdash \text{rectype } X_1 = T_{X1} \;=\; \text{rectype } X_2 = T_{X2} \;\in\; \mathbb{U}_j$ ⌊Ax⌋
by recEquality X
$\Gamma,\; X{:}\mathbb{U}_j \vdash T_{X1}[X/X_1] \;=\; T_{X2}[X/X_2] \;\in\; \mathbb{U}_j$ ⌊Ax⌋

$\Gamma \vdash s \;=\; t \;\in\; \text{rectype } X = T_X$ ⌊Ax⌋
by rec_memberEquality j
$\Gamma \vdash s \;=\; t \;\in\; T_X[\text{rectype } X = T_X / X]$ ⌊Ax⌋
$\Gamma \vdash \text{rectype } X = T_X \;\in\; \mathbb{U}_j$ ⌊Ax⌋

$\Gamma \vdash \text{rectype } X = T_X$ ⌊**ext** t⌋
by rec_memberFormation j
$\Gamma \vdash T_X[\text{rectype } X = T_X / X]$ ⌊**ext** t⌋
$\Gamma \vdash \text{rectype } X = T_X \;\in\; \mathbb{U}_j$ ⌊Ax⌋

$\Gamma \vdash \text{let}^* \; f_1(x_1) = t_1 \text{ in } f_1(e_1) \;=\; \text{let}^* \; f_2(x_2) = t_2 \text{ in } f_2(e_2) \;\in\; T[e_1/z]$ ⌊Ax⌋
by rec_indEquality z T rectype $X = T_X$ j P f x
$\Gamma \vdash e_1 \;=\; e_2 \;\in\; \text{rectype } X = T_X$ ⌊Ax⌋
$\Gamma \vdash \text{rectype } X = T_X \;\in\; \mathbb{U}_j$ ⌊Ax⌋
$\Gamma,\; P{:}\, (\text{rectype } X = T_X) \to \mathbb{P}_j,\; f{:}\, (x{:}\{x{:}\text{rectype } X = T_X \mid P(x)\} \to T[y/z]),$
$x{:}\, T_X[\{x{:}\text{rectype } X = T_X \mid P(x)\}/X] \;\vdash\; t_1[f,x/f_1,x_1] \;=\; t_2[f,x/f_2,x_2] \in T[x/z]$ ⌊Ax⌋

$\Gamma,\; z{:}\,\text{rectype } X = T_X,\; \Delta \vdash C$ ⌊**ext** $\text{let}^* \; f(x) = t[\lambda y.\Lambda/P] \text{ in } f(z)$⌋
by recElimination i j P y f x
$\Gamma,\; z{:}\,\text{rectype } X = T_X,\; \Delta \vdash \text{rectype } X = T_X \;\in\; \mathbb{U}_j$ ⌊Ax⌋
$\Gamma,\; z{:}\,\text{rectype } X = T_X,\; \Delta,\; P{:}\, (\text{rectype } X = T_X) \to \mathbb{P}_j,\; f{:}\, (y{:}\{x{:}\text{rectype } X = T_X \mid P(x)\} \to C[y/z]),$
$x{:}\, T_X[\{x{:}\text{rectype } X = T_X \mid P(x)\}/X] \;\vdash\; C[x/z]$ ⌊**ext** t⌋

$\Gamma,\; z{:}\,\text{rectype } X = T_X,\; \Delta \vdash C$ ⌊**ext** $t[z/x]$⌋
by recUnrollElimination i x v
$\Gamma,\; z{:}\,\text{rectype } X = T_X,\; \Delta,\; x{:}\, T_X[\text{rectype } X = T_X/X],\; v{:}\, z{=}x \in T_X[\text{rectype } X = T_X/X] \vdash C[x/z]$ ⌊**ext** t⌋

Basic Inference Rule	*Corresponding Tactic* *with required arguments with optional tacticals*
recEquality X	---
rec_memberEquality j	EqTypeCD
rec_memberFormation j	---
rec_indEquality z T rectype $X = T_X$ j P f x	---
recElimination i j P y f x	RecTypeInduction i
recUnrollElimination i x v	D i

A.3.12 Subset

$\Gamma \vdash \mathbb{U}_j \quad \lfloor \textbf{ext } \{x{:}S \mid T\} \rfloor$
by dependent_setFormation S x
$\quad \Gamma \vdash S \in \mathbb{U}_j \quad \lfloor \mathsf{Ax} \rfloor$
$\quad \Gamma,\ x{:}S \vdash \mathbb{U}_j \quad \lfloor \textbf{ext } T \rfloor$

$\Gamma \vdash \mathbb{U}_j \quad \lfloor \textbf{ext } \{S \mid T\} \rfloor$
by independent_setFormation
$\quad \Gamma \vdash \mathbb{U}_j \quad \lfloor \textbf{ext } S \rfloor$
$\quad \Gamma \vdash \mathbb{U}_j \quad \lfloor \textbf{ext } T \rfloor$

$\Gamma \vdash \{x_1{:}S_1 \mid T_1\} = \{x_2{:}S_2 \mid T_2\} \in \mathbb{U}_j \quad \lfloor \mathsf{Ax} \rfloor$
by setEquality x
$\quad \Gamma \vdash S_1 = S_2 \in \mathbb{U}_j \quad \lfloor \mathsf{Ax} \rfloor$
$\quad \Gamma,\ x{:}S_1 \vdash T_1[x/x_1] = T_2[x/x_2] \in \mathbb{U}_j \quad \lfloor \mathsf{Ax} \rfloor$

$\Gamma \vdash s = t \in \{x{:}S \mid T\} \quad \lfloor \mathsf{Ax} \rfloor$
by dependent_set_memberEquality j x'
$\quad \Gamma \vdash s = t \in S \quad \lfloor \mathsf{Ax} \rfloor$
$\quad \Gamma \vdash T[s/x] \quad \lfloor \mathsf{Ax} \rfloor$
$\quad \Gamma,\ x'{:}S \vdash T[x'/x] \in \mathbb{U}_j \quad \lfloor \mathsf{Ax} \rfloor$

$\Gamma \vdash \{x{:}S \mid T\} \quad \lfloor \textbf{ext } s \rfloor$
by dependent_set_memberFormation j s x'
$\quad \Gamma \vdash s \in S \quad \lfloor \mathsf{Ax} \rfloor$
$\quad \Gamma \vdash T[s/x] \quad \lfloor \mathsf{Ax} \rfloor$
$\quad \Gamma,\ x'{:}S \vdash T[x'/x] \in \mathbb{U}_j \quad \lfloor \mathsf{Ax} \rfloor$

$\Gamma \vdash s = t \in \{S \mid T\} \quad \lfloor \mathsf{Ax} \rfloor$
by independent_set_memberEquality
$\quad \Gamma \vdash s = t \in S \quad \lfloor \mathsf{Ax} \rfloor$
$\quad \Gamma \vdash T \quad \lfloor \mathsf{Ax} \rfloor$

$\Gamma \vdash \{S \mid T\} \quad \lfloor \textbf{ext } s \rfloor$
by independent_set_memberFormation
$\quad \Gamma \vdash S \quad \lfloor \textbf{ext } s \rfloor$
$\quad \Gamma \vdash T \quad \lfloor \mathsf{Ax} \rfloor$

$\Gamma,\ z{:}\{x{:}S \mid T\},\ \Delta \vdash C \quad \lfloor \textbf{ext } (\lambda y.t)\ z \rfloor$
by setElimination i y v
$\quad \Gamma,\ z{:}\{x{:}S \mid T\},\ y{:}S,\ [\![v]\!]{:}T[y/x],\ \Delta[y/z] \vdash C[y/z] \quad \lfloor \textbf{ext } t \rfloor$

Basic Inference Rule	*Corresponding Tactic* *with required arguments with optional tacticals*
dependent_setFormation S x	---
independent_setFormation	---
setEquality x	EqCD
dependent_set_memberEquality j x'	EqTypeCD
dependent_set_memberFormation j s x'	D 0
independent_set_memberEquality	EqTypeCD
independent_set_memberFormation	D 0
setElimination i y v	D i

A.3.13 Intersection

$\Gamma \vdash \mathbb{U}_j \;\lfloor \mathbf{ext}\ \cap x{:}S.T \rfloor$
by `isectFormation` x S
$\Gamma \vdash S \in \mathbb{U}_j \;\lfloor \mathsf{Ax} \rfloor$
$\Gamma,\ x{:}S \vdash \mathbb{U}_j \;\lfloor \mathbf{ext}\ T \rfloor$

$\Gamma \vdash \cap x_1{:}S_1.T_1 = \cap x_2{:}S_2.T_2 \in \mathbb{U}_j \;\lfloor \mathsf{Ax} \rfloor$
by `isectEquality` x
$\Gamma \vdash S_1 = S_2 \in \mathbb{U}_j \;\lfloor \mathsf{Ax} \rfloor$
$\Gamma,\ x{:}S_1 \vdash T_1[x/x_1] = T_2[x/x_2] \in \mathbb{U}_j \;\lfloor \mathsf{Ax} \rfloor$

$\Gamma \vdash t_1 = t_2 \in \cap x{:}S.T \;\lfloor \mathsf{Ax} \rfloor$
by `isect_memberEquality` j x'
$\Gamma,\ x'{:}S \vdash t_1 = t_2 \in T[x'/x] \;\lfloor \mathsf{Ax} \rfloor$
$\Gamma \vdash S \in \mathbb{U}_j \;\lfloor \mathsf{Ax} \rfloor$

$\Gamma \vdash \cap x{:}S.T \;\lfloor \mathbf{ext}\ t \rfloor$
by `isect_memberFormation` j x'
$\Gamma,\ x'{:}S \vdash T[x'/x] \;\lfloor \mathbf{ext}\ t \rfloor$
$\Gamma \vdash S \in \mathbb{U}_j \;\lfloor \mathsf{Ax} \rfloor$

$\Gamma \vdash f_1 = f_2 \in T[t/x] \;\lfloor \mathsf{Ax} \rfloor$
by `isect_member_caseEquality` $\cap x{:}S.T$ t
$\Gamma \vdash f_1 = f_2 \in \cap x{:}S.T \;\lfloor \mathsf{Ax} \rfloor$
$\Gamma \vdash t \in S \;\lfloor \mathsf{Ax} \rfloor$

$\Gamma,\ f{:}\cap x{:}S.T,\ \Delta \vdash C \;\lfloor \mathbf{ext}\ t[f, \mathsf{Ax}/y, z] \rfloor$
by `isectElimination` i s y z
$\Gamma,\ f{:}\cap x{:}S.T,\ \Delta \vdash s \in S \;\lfloor \mathsf{Ax} \rfloor$
$\Gamma,\ f{:}\cap x{:}S.T,\ y{:}T[s/x],\ z{:}\ y = f \in T[s/x],\ \Delta \vdash C \;\lfloor \mathbf{ext}\ t \rfloor$

Basic Inference Rule	*Corresponding Tactic* *with required arguments*	*with optional tacticals*
`isectFormation` x S	`---`	
`isectEquality` x	`EqCD`	
`isect_memberEquality` j x'	`EqTypeCD`	
`isect_memberFormation` j x'	`D 0`	
`isect_member_caseEquality` $\cap x{:}S.T$ t	`GenTypeCD` $t = x \in S$	
`isectElimination` i s y z	`With` s (`D` i)	

A.3.14 Quotient Type

$\Gamma \vdash \mathbb{U}_j$ $\lfloor$**ext** $x,y : T//E\rfloor$
by quotientFormation T E x y z v v'
- $\Gamma \vdash T \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
- $\Gamma,\ x{:}T,\ y{:}T \vdash E \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
- $\Gamma,\ x{:}T,\ \vdash E[x,x/x,y]$ $\lfloor$Ax$\rfloor$
- $\Gamma,\ x{:}T,\ y{:}T,\ v{:}\ E[x,y/x,y] \vdash E[y,x/x,y]$ $\lfloor$Ax$\rfloor$
- $\Gamma,\ x{:}T,\ y{:}T,\ z{:}T,\ v{:}\ E[x,y/x,y],\ v'{:}\ E[y,z/x,y] \vdash E[x,z/x,y]$ $\lfloor$Ax$\rfloor$

$\Gamma \vdash x_1,y_1 : T_1//E_1 = x_2,y_2 : T_2//E_2 \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
by quotientWeakEquality x y z v v'
- $\Gamma \vdash T_1 = T_2 \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
- $\Gamma,\ x{:}T_1,\ y{:}T_1 \vdash E_1[x,y/x_1,y_1] = E_2[x,y/x_2,y_2] \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
- $\Gamma,\ x{:}T_1 \vdash E_1[x,x/x_1,y_1]$ $\lfloor$Ax$\rfloor$
- $\Gamma,\ x{:}T_1,\ y{:}T_1,\ v{:}\ E_1[x,y/x_1,y_1] \vdash E_1[y,x/x_1,y_1]$ $\lfloor$Ax$\rfloor$
- $\Gamma,\ x{:}T_1,\ y{:}T_1,\ z{:}T_1,\ v{:}\ E_1[x,y/x_1,y_1],\ v'{:}\ E_1[y,z/x_1,y_1] \vdash E_1[x,z/x_1,y_1]$ $\lfloor$Ax$\rfloor$

$\Gamma \vdash x_1,y_1 : T_1//E_1 = x_2,y_2 : T_2//E_2 \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
by quotientEquality
- $\Gamma \vdash x_1,y_1 : T_1//E_1 \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
- $\Gamma \vdash x_2,y_2 : T_2//E_2 \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
- $\Gamma \vdash T_1 = T_2 \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
- $\Gamma,\ v{:}\ T_1{=}T_2{\in}\mathbb{U}_j,\ x{:}T_1,\ y{:}T_1 \vdash E_1[x,y/x_1,y_1] \Rightarrow E_2[x,y/x_2,y_2]$ $\lfloor$Ax$\rfloor$
- $\Gamma,\ v{:}\ T_1{=}T_2{\in}\mathbb{U}_j,\ x{:}T_1,\ y{:}T_1 \vdash E_2[x,y/x_2,y_2] \Rightarrow E_1[x,y/x_1,y_1]$ $\lfloor$Ax$\rfloor$

$\Gamma \vdash s = t \in x,y : T//E$ $\lfloor$Ax$\rfloor$
by quotient_memberWeakEquality j
- $\Gamma \vdash x,y : T//E \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
- $\Gamma \vdash s = t \in T$ $\lfloor$Ax$\rfloor$

$\Gamma \vdash x,y : T//E$ $\lfloor$**ext** $t\rfloor$
by quotient_memberFormation j
- $\Gamma \vdash x,y : T//E \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
- $\Gamma \vdash T$ $\lfloor$**ext** $t\rfloor$

$\Gamma \vdash s = t \in x,y : T//E$ $\lfloor$Ax$\rfloor$
by quotient_memberEquality j
- $\Gamma \vdash x,y : T//E \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
- $\Gamma \vdash s \in T$ $\lfloor$Ax$\rfloor$
- $\Gamma \vdash t \in T$ $\lfloor$Ax$\rfloor$
- $\Gamma \vdash E[s,t/x,y]$ $\lfloor$Ax$\rfloor$

$\Gamma,\ v{:}\ s{=}t{\in}x,y : T//E,\ \Delta \vdash C$ $\lfloor$**ext** $u\rfloor$
by quotient_equalityElimination i j v'
- $\Gamma,\ v{:}\ s{=}t{\in}x,y : T//E,\ [\![v']\!]{:}\,E[s,t/x,y],\ \Delta \vdash C$ $\lfloor$**ext** $u\rfloor$
- $\Gamma,\ v{:}\ s = t{\in}x,y : T//E,\ \Delta \vdash E[s,t/x,y] \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$

$\Gamma,\ z{:}\ x,y : T//E,\ \Delta \vdash s = t \in S$ $\lfloor$Ax$\rfloor$
by quotientElimination i j x' y' v
- $\Gamma,\ z{:}\ x,y : T//E,\ \Delta,\ x'{:}T,\ y'{:}T \vdash E[x',y'/x,y] \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
- $\Gamma,\ z{:}\ x,y : T//E,\ \Delta \vdash S \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
- $\Gamma,\ z{:}\ x,y : T//E,\ \Delta,\ x'{:}T,\ y'{:}T,\ v{:}\ E[x',y'/x,y] \vdash s[x'/z] = t[y'/z] \in S[x'/z]$ $\lfloor$Ax$\rfloor$

Γ, $z{:}\,x,y : T/\!/E$, $\Delta \vdash s = t \in S$ ⌊Ax⌋
  by quotientElimination_2 i j x' y' v
  Γ, $z{:}\,x,y : T/\!/E$, Δ, $x'{:}T$, $y'{:}T \vdash E[x',y'/x,y] \in \mathbb{U}_j$ ⌊Ax⌋
  Γ, $z{:}\,x,y : T/\!/E$, $\Delta \vdash S \in \mathbb{U}_j$ ⌊Ax⌋
  Γ, $z{:}\,x,y : T/\!/E$, $x'{:}T$, $y'{:}T$, $v{:}\,E[x',y'/x,y]$, $\Delta[x'/z] \vdash s[x'/z] = t[y'/z] \in S[x'/z]$ ⌊Ax⌋

Basic Inference Rule	*Corresponding Tactic* *with required arguments*	*with optional tacticals*
`quotientFormation` T E x y z v v'	`---`	
`quotientWeakEquality` x y z v v'	`EqCD`	
`quotientEquality`	`QuotEqCD`	
`quotient_memberWeakEquality` j	`WeakEqTypeCD`	
`quotient_memberFormation` j	`D 0`	
`quotient_memberEquality` j	`EqTypeCD`	
`quotient_equalityElimination` i j v'	`EqTypeD` i	
`quotientElimination` i j x' y' v	`D` i	
`quotientElimination_2` i j x' y' v	`QuotD` i	`QuotientHD'` $[x';y']$ i

A.3.15 Direct Computation

$\Gamma \vdash C \quad \lfloor \textbf{ext } t \rfloor$
  by direct_computation *tagC*
  $\Gamma \vdash C{\downarrow}_{tagC} \quad \lfloor \textbf{ext } t \rfloor$

$\Gamma \vdash C \quad \lfloor \textbf{ext } t \rfloor$
  by reverse_direct_computation *tagC*
  $\Gamma, \vdash C{\uparrow}_{tagC} \quad \lfloor \textbf{ext } t \rfloor$

$\Gamma,\ z\!:\!T,\ \Delta \vdash C \quad \lfloor \textbf{ext } t \rfloor$
  by direct_computation_hypothesis i *tagT*
  $\Gamma,\ z\!:\!T{\downarrow}_{tagT},\ \Delta \vdash C \quad \lfloor \textbf{ext } t \rfloor$

$\Gamma,\ z\!:\!T,\ \Delta \vdash C \quad \lfloor \textbf{ext } t \rfloor$
  by reverse_direct_computation_hypothesis i *tagT*
  $\Gamma,\ z\!:\!T{\uparrow}_{tagT},\ \Delta \vdash C \quad \lfloor \textbf{ext } t \rfloor$

Basic Inference Rule	*Corresponding Tactic*	
	with required arguments	*with optional tacticals*
direct_computation *tagC*	ComputeWithTaggedTerm *tagC* 0	
reverse_direct_computation *tagC*	RevComputeWithTaggedTerm *tagC* 0	
direct_computation_hypothesis i *tagT*	ComputeWithTaggedTerm *tagT* i	
reverse_direct_computation_hypothesis i *tagT*	RevComputeWithTaggedTerm *tagT* i	

A.3.16 Miscellaneous

Γ, x:T, Δ $\vdash$ T $\lfloor$**ext** $x\rfloor$
  by `hypothesis` i

Γ, x:T, Δ $\vdash$ C $\lfloor$**ext** $t\rfloor$
  by `thin` i
  Γ, Δ $\vdash$ C $\lfloor$**ext** $t\rfloor$

Γ, Δ $\vdash$ C $\lfloor$**ext** $(\lambda x.t)$ $s\rfloor$
  by `cut` i T x
  Γ, Δ $\vdash$ T $\lfloor$**ext** $s\rfloor$
  Γ, x:T, Δ $\vdash$ C $\lfloor$**ext** $t\rfloor$

Γ $\vdash$ T $\lfloor$**ext** $t\rfloor$
  by `introduction` t
  Γ $\vdash$ t $\in$ T $\lfloor$Ax$\rfloor$

Γ, z:T, Δ $\vdash$ C $\lfloor$**ext** $t\rfloor$
  by `hyp_replacement` i S j
  Γ, z:S, Δ $\vdash$ C $\lfloor$**ext** $t\rfloor$
  Γ, z:T, Δ $\vdash$ T = S $\in$ $\mathbb{U}_j$ $\lfloor$Ax$\rfloor$

Γ $\vdash$ C $\lfloor$**ext** $t\rfloor$
  by `lemma` "*theorem-name*"

Γ $\vdash$ t $\in$ T $\lfloor$Ax$\rfloor$
  by `extract` "*theorem-name*"

Γ $\vdash$ C $\lfloor$**ext** $t[\sigma]\rfloor$
  by `instantiate` Γ' C' σ
  Γ' $\vdash$ C' $\lfloor$**ext** $t\rfloor$

Γ $\vdash$ C $\lfloor$**ext** $t[y/x]\rfloor$
  by `rename` y x
  $\Gamma[x/y]$ $\vdash$ $C[x/y]$ $\lfloor$**ext** $t\rfloor$

Γ $\vdash$ C $\lfloor$Ax$\rfloor$
  by `because`

Basic Inference Rule	*Corresponding Tactic*	
	with required arguments	*with optional tacticals*
`hypothesis` i	`Hypothesis`	`NthHyp` i
`thin` i	`Thin` i	
`cut` i T x	`Assert` T	`AssertDeclAtHyp` i T x
`introduction` t	`UseWitness` t	`UseWitness` t
`hyp_replacement` i S j	`SubstClause` S i	
`lemma` "*theorem-name*"	`Lemma` "*theorem-name*"	`Lemma` "*theorem-name*"
`extract` "*theorem-name*"		
`instantiate` Γ' C' σ		
`rename` y x	`RenameVar` x i *	
`because`	`Fiat`	`Fiat`

*: y is the variable declared in hypothesis i.

Appendix B

Introduction to Nuprl ML

Whenever Nuprl is fired up, several ML *top loops* are created. Running in these windows is an ML interpreter that is embedded into the library, editor, or refiner process. Whenever one has the `ML>` prompt one can type an ML expression, terminate it with `;;` and press ↵. ML will evaluate the expression, and print its value and type.[1]

One's primary interaction with Nuprl 5 is through the navigator and the windows opened by it. However, advanced users will find it necessary to interact with the Nuprl 5 processes for examining parts of Nuprl objects or customizing the behavior of Nuprl 5. In particular, all Nuprl tactics are written in ML, as are a variety of utility functions. The tactics are documented in Chapter 8. The utility functions are described throughout this Chapter.

B.1 The History of ML

Several versions of the programming language ML have appeared over the years, between the time it was first designed and implemented by Milner, Morris and Wadsworth at the University of Edinburgh in the early 1970's, and the time it was settled and standardized in the mid-1980's. The original ML, the *meta-language* of the Edinburgh LCF system, is defined in [GMW79].

The ML used in the Nuprl system is fairly close to the original. It is derived from a early version that Huet at INRIA and Paulson at the University of Cambridge were working on in 1981. Todd Knoblock at Cornell made most of the Nuprl specific modifications in the mid-1980's. Nuprl's ML hasn't changed since then and is not compatible with the ML versions that are widely used today.

The ML of Huet and Paulson is described in the preface to '*The* ML *Handbook*' [CHP84]. Huet used this version in the Formel project; and it subsequently evolved into a version of ML called CAML. Paulson also used it, as part of the first version of Cambridge LCF, but switched to Standard ML in the later versions of Cambridge LCF [Pau87].

The CAML language [CH90, WAL+90] is now rarely used. But there is a scaled down version called CAML-Light which is actively used in teaching programming to over 10,000 engineers a year in France. Its object-oriented version OCaml [Ler00] have become quite popular in recent years and has been used in the implementation of the group communication toolkit Ensemble [Hay98, BCH+00]. The Standard ML language has also become increasingly popular for implementing theorem provers such as HOL [GM93] or Isabelle [Pau90].

[1] Note that the ML prompt is different in each window. It is `ML[(ORB)]>` in the Nuprl process windows for the library, editor, or refiner and may later change into `ML[(lib)]>`, `ML[(edd)]>`, and `ML[(ref)]>`. In the Nuprl 5 top loop it is `ML[EDD]>`, `ML[LIB]>`, or `ML[REF]>`. The latter already provide the double semicolon for terminating an ML expression, so the user does not have to enter `;;` in these windows.

The description of ML that appears in Sections B.3 to B.6 is based very closely on 'The ML Handbook' [CHP84]. It was adapted for NUPRL purposes from LaTeX sources provided by the HOL theorem proving group in Cambridge. For completeness (and historical interest), the preface to '*The* ML *Handbook*' and the preface to '*Edinburgh LCF: a Mechanised Logic of Computation*' are reproduced below.

B.1.1 Preface to 'The ML Handbook'

This handbook is a revised edition of Section 2 of 'Edinburgh LCF', by M. Gordon, R. Milner, and C. Wadsworth, published in 1979 as Springer Verlag Lecture Notes in Computer Science n$^{\underline{o}}$ 78. ML was originally the meta-language of the LCF system. The ML system was adapted to Maclisp on Multics by Gérard Huet at INRIA in 1981, and a compiler was added. Larry Paulson from the University of Cambridge completely redesigned the LCF proving system, which stabilized in 1984 as Cambridge LCF. Guy Cousineau from the University Paris VII added concrete types in the summer of 1984. Philippe Le Chenadec from INRIA implemented an interface with the Yacc parser generator system, for the versions of ML running under Unix. This permits the user to associate a concrete syntax with a concrete type.

The ML language is still under design. An extended language was implemented on the VAX by Luca Cardelli in 1981. It was then decided to completely re-design the language, in order to accommodate in particular the call by pattern feature of the language HOPE designed by Rod Burstall and David MacQueen. A committee of researchers from the Universities of Edinburgh and Cambridge, the Bell Laboratories and INRIA, headed by Robin Milner, is currently working on the new extended language, called Standard ML. Progress reports appear in the Polymorphism Newsletter, edited by Luca Cardelli and David MacQueen from Bell Laboratories. The design of a core language is now frozen, and its description will appear in a forthcoming report of the University of Edinburgh, as 'The Standard ML Core Language' by Robin Milner.

This handbook is a manual for ML version 6.1, released in December 1984. The language is somewhere in between the original ML from LCF and standard ML, since Guy Cousineau added the constructors and call by patterns. This is a LISP based implementation, compatible for Maclisp on Multics, Franzlisp on VAX under Unix, Zetalisp on Symbolics 3600, and Le_Lisp on 68000, VAX, Multics, Perkin-Elmer, etc... Video interfaces have been implemented by Philippe Le Chenadec on Multics, and by Maurice Migeon on Symbolics 3600. The ML system is maintained and distributed jointly by INRIA and the University of Cambridge.

B.1.2 Preface to 'Edinburgh LCF'

ML is a general purpose programming language. It is derived in different aspects from ISWIM, POP2 and GEDANKEN, and contains perhaps two new features. First, it has an escape and escape trapping mechanism, well-adapted to programming strategies which may be (in fact usually are) inapplicable to certain goals. Second, it has a polymorphic type discipline which combines the flexibility of programming in a typeless language with the security of compile-time type checking (as in other languages, you may also define your own types, which may be abstract and/or recursive).

For those primarily interested in the design of programming languages, a few remarks here may be helpful both about ML as a candidate for comparison with other recently designed languages, and about the description of ML which we provide. On the first point, although we did not set out with programming language design as a primary aim, we believe that ML does contain features worthy of serious consideration; these are the escape mechanism and the polymorphic type discipline mentioned above, and also the attempt to make programming with functions—including those of higher type—as easy and natural as possible. We are less happy about the imperative aspects of the language, and would wish to give them further thought if we were mainly concerned with language design. In particular, the constructs for controlling iteration both by boolean conditions and by escape-trapping (which we included partly for experiment) are perhaps too complex taken together, and we are sensitive to the criticism that escape (or failure, as we call it) reports information only in the form of a string. This latter constraint results mainly from our type discipline; we do not know how best to relax the constraint while maintaining the discipline.

Concerning the description of ML, we have tried both to initiate users by examples of programming and to give a precise definition.

B.2 Introduction and Examples

ML is an interactive language. At top-level one can:

- evaluate expressions
- perform declarations

To give a first impression of the system, we reproduce below a session at a terminal in which simple uses of various ML constructs are illustrated. To make the session easier to follow, it is split into a sequence of sub-sessions. A complete description of the syntax and semantics of ML is given in Section B.3 and Section B.4 respectively.

B.2.1 Expressions

In this tutorial, the ML prompt is `#` so lines beginning with this contain the user's contribution; all other lines are output by the system. The NUPRL ML prompt is different; usually `ML>` is used for the first line of user input, and `>` is used for continuation lines.

```
# 2+3;;                                   1
5 : int

# it;;
5 : int
```

ML prompted with `#`, the user then typed `2+3;;` followed by a carriage return ↵; ML then responded with `5 : int`, a new line, and then prompted again. The user then typed `it;;` ↵ and the system responded by typing `5 : int` again. In general to evaluate an expression e one types e followed by a carriage return; the system then prints e's value and type (the type prefaced by a colon). The *value of the last expression* evaluated at top level is remembered in the identifier `it`.

B.2.2 Declarations

The declaration `let` x `=` e evaluates e and binds the resulting value to x.

```
# let x=2*3;;                             2
x = 6 : int

# it=x;;
false : bool
```

Notice that declarations do not affect the identifier `it`. To bind the variables $x_1, \ldots, x_n$ simultaneously to the values of the expressions $e_1, \ldots, e_n$ one can perform either the declaration `let` x_1`=`e_1 `and` x_2`=`e_2 `...and` x_n`=`e_n or `let` x_1`,`x_2`,...,`x_n `=` e_1`,`e_2`...,`e_n. These two declarations are equivalent.

```
# let y=10 and z=x;;
y = 10 : int
z = 6 : int

# let x,y = y,x;;
x = 10 : int
y = 6 : int
```
3

A declaration d can be made *local* to the evaluation of an expression e by evaluating the expression d `in` e. The expression e `where` b (where b is a *binding* such as `x=2`) is equivalent to `let` b `in` e.

```
# let x=2 in x*y;;
12 : int

# x;;
10 : int

# x*y where x=2;;
12 : int
```
4

B.2.3 Assignment

Identifiers can be declared *assignable* using `letref` instead of `let`. Values bound to such identifiers can be changed with the assignment expression x`:=`e, which changes the value bound to x to be the value of e. Attempts to assign to non-assignable variables are detected by the type checker.

```
# x:=1;;

unbound or non-assignable variable x
1 error in typing
typecheck failed

# letref x=1 and y=2;;
x = 1 : int
y = 2 : int

# x:=6;;
6 : int

# x;;
6 : int
```
5

The value of an assignment x`:=`e is the value of e (hence the value of `y:=6` is `6`). Simultaneous assignments can also be done:

```
# x,y := y,x;;
(2,6) : (int # int)

# x,y;;
(2,6) : (int # int)
```
6

The type `(int # int)` is the type of pairs of integers.

B.2.4 Functions

To define a function f with formal parameter x and body e one performs the declaration `let` f x `=` e. To apply the function f to an actual parameter e one evaluates the expression f e.

7
```
# let f x = 2*x;;
f = - : (int -> int)

# f 4;;
8 : int
```

Functions are printed as a dash, `-`, followed by their type, since a function as such is not printable. Application binds more tightly than anything else in the language; thus, for example, f `3 + 4` means `(`f `3) + 4` not f `(3 + 4)`. Functions of several arguments can be defined:

8
```
# let add x y = x+y;;
add = - : (int -> int -> int)

# add 3 4;;
7 : int

# let f = add 3;;
f = - : (int -> int)

# f 4;;
7 : int
```

Application associates to the left so `add 3 4` means `(add 3) 4`. In the expression `add 3`, the function `add` is partially applied to `3`; the resulting value is the function of type `int -> int` which adds `3` to its argument. Thus `add` takes its arguments one at a time. We could have made `add` take a single argument of the cartesian product type `(int # int)`:

9
```
# let add(x,y) = x+y;;
add = - : ((int #  int) -> int)

# add(3,4);;
7 : int

# let z = (3,4) in add z;;
7 : int

# add 3;;

ill-typed phrase: 3
has an instance of type  int
which should match type  (int #  int)
1 error in typing
typecheck failed
```

As well as taking structured arguments (e.g. `(3,4)`) functions may also return structured results.

10
```
# let sumdiff(x,y) = (x+y,x-y);;
sumdiff = - : ((int #  int) -> (int #  int))

# sumdiff(3,4);;
(7, -1) : (int #  int)
```

B.2.5 Recursion

The following is an attempt to define the factorial function:

```
# let fact n = if n=0 then 1 else n*fact(n-1);;                          11

unbound or non-assignable variable fact
1 error in typing
typecheck failed
```

The problem is that any free variables in the body of a function have the bindings they had just before the function was declared; `fact` is such a free variable in the body of the declaration above, and since it is not defined before its own declaration, an error results. To make things clear consider:

```
# let f n = n+1;;                                                        12
f = - : (int -> int)

# let f n = if n=0 then 1 else n*f(n-1);;
f = - : (int -> int)

# f 3;;
9 : int
```

Here `f 3` results in the evaluation of `3*f(2)`, but now the first `f` is used so `f(2)` evaluates to `2+1=3`, hence the expression `f 3` results in `3*3=9`. To make a function declaration hold within its own body, `letrec` instead of `let` must be used. The correct recursive definition of the factorial function is thus:

```
# letrec fact n = if n=0 then 1 else n*fact(n-1);;                       13
fact = - : (int -> int)

# fact 3;;
6 : int
```

B.2.6 Iteration

The construct `if` e_1 `then` e_2 `loop` e_3 is the same as `if` e_1 `then` e_2 `else` e_3 in the true case; when e_1 evaluates to false, e_3 is evaluated and control loops back to the front of the construct again. As an illustration, here is an iterative definition of `fact` using two local assignable variables: `count` and `result`.

```
# let fact n =                                                           14
#     letref count=n and result=1
#     in     if count=0
#            then result
#            loop count,result := count-1,count*result;;
fact = - : (int -> int)

# fact 4;;
24 : int
```

Replacing the `then` in `if` e_1 `then` e_2 `else` e_3 by `loop` causes iteration when e_1 evaluates to true: e.g., `if` e_1 `loop` e_2 `else` e_3 is equivalent to `if not(`e_1`)` `then` e_3 `loop` e_2. The conditional/loop construct can have a number of conditions, each preceded by `if`. The expression guarded by each condition may be preceded by `then`, or by `loop` when the whole construct is to be re-evaluated after evaluating the guarded expression:

```
# let gcd(x,y) =                                                    15
#     letref x,y = x,y
#     in     if x>y loop x:=x-y
#            if x<y loop y:=y-x
#            else x;;
gcd = - : ((int #  int) -> int)

# gcd(12,20);;
4 : int
```

B.2.7 Lists

If $e_1, \ldots, e_n$ all have type *ty* then the ML expression `[`e_1`;...;`e_n`]` has type `(`*ty* `list)`. The standard functions on lists are `hd` (head),]`tl` (tail), `null` (which tests whether a list is empty – i.e. is equal to `[]`), and the infixed operators `.` (cons) and `@` (append, or concatenation).

```
# let m = [1;2;(2+1);4];;                                           16
m = [1; 2; 3; 4] : int list

# hd m , tl m;;
(1, [2; 3; 4]) : (int #  int list)

# null m , null [];;
(false, true) : (bool #  bool)

# 0.m;;
[0; 1; 2; 3; 4] : int list

# [1; 2] @ [3; 4; 5; 6];;
[1; 2; 3; 4; 5; 6] : int list

# [1;true;2];;

ill-typed phrase: true
has an instance of type   bool
which should match type   int
1 error in typing
typecheck failed
```

All the members of a list must have the same type (although this type could be a sum, or disjoint union type—see Section B.5).

B.2.8 Tokens

A sequence of characters enclosed between token quotes (\` – i.e. ascii 96) is a *token*.

```
# `this is a token`;;                                               17
`this is a token` : tok

# ``this is a token list``;;
[`this`; `is`; `a`; `token`; `list`] : tok list

# it = ``this is a`` @ [`token`;`list`];;
true : bool
```

The expression ``` ``tok ```$_1$```tok```$_2$```...tok```$_n$``` `` ``` is an alternative syntax for ``[`tok``$_1$`` `;`tok``$_2$`` `;...;`tok``$_n$`` `]``.

B.2.9 Strings

A sequence of characters enclosed between string quotes (" – i.e. ascii 34) is a *string*.

```
# "this is a string";;                                           18
"this is a string" : string

# "";;
"" : string
```

Although similar, strings and tokens are implemented differently in Lisp; strings are implemented as character arrays, and tokens as symbols. The implementation affects the efficiency of such operations as comparison and concatenation; Tokens are much slower to concatenate, but faster to compare.

B.2.10 Polymorphism

The list processing functions `hd`, `tl` etc. can be used on all types of lists.

```
# hd [1;2;3];;                                                   19
1 : int

# hd [true;false;true];;
true : bool

# hd [1,2;3,4];;
(1, 2) : (int #  int)
```

Thus `hd` has several types; for example, it is used above with types `(int list) -> int`, `(bool list) -> bool`, and `(int # int) list -> (int # int)`. In fact if *ty* is *any* type then `hd` has the type `(`*ty* `list) ->` *ty*. Functions, like `hd`, with many types are called *polymorphic*, and ML uses type variables `*`, `**`, `***` etc. to represent their types.

```
# hd;;                                                           20
- : (* list -> *)

# letrec map f l = if null l then []
#                             else f(hd l).map f (tl l);;
map = - : ((* -> **) -> * list -> ** list)

# map fact [1;2;3;4];;
[1; 2; 6; 24] : int list
```

The ML function `map` takes a function f (with argument type `*` and result type `**`), and a list l (of elements of type `*`), and returns the list obtained by applying f to each element of l (which is a list of elements of type `**`). `map` can be used at any instance of its type: above, both `*` and `**` were instantiated to `int`; below, `*` is instantiated to `(int list)` and `**` to `bool`. Notice that the instance need not be specified; it is determined by the type checker.

```
# map null [[1;2]; []; [3]; []];;                                21
[false; true; false; true] : bool list
```

B.2.11 Lambda-expressions

The expression `\`x`.`e evaluates to a function with formal parameter x and body e. Thus the declaration `let` f x `=` e is equivalent to `let` f `=` `\`x`.`e. Similarly `let` f`(`x`,`y`)` z `=` e is equivalent to `let` f `=` `\(`x`,`y`).\`z`.`e. Repeated \'s, as in `\(`x`,`y`).\`z`.`e may be abbreviated by `\(`x`,`y`)` z`.`e. The character \ is our `\`x`.`e and `\(`x`,`y`)` z`.`e are called lambda-expressions.

22

```
# \x.x+1;;
- : (int -> int)

# it 3;;
4 : int

# map (\x.x*x) [1;2;3;4];;
[1; 4; 9; 16] : int list

# let doubleup = map (\x.x@x);;
doubleup = - : (* list list -> * list list)

# doubleup [ [1;2]; [3;4;5] ];;
[[1; 2; 1; 2]; [3; 4; 5; 3; 4; 5]] : int list list

# doubleup [];;
[] : * list list
```

B.2.12 Failure

Some standard functions *fail* at run-time on certain arguments, yielding a string (which is usually the function name) to identify the sort of failure. A failure with token `‘`t`‘` may also be generated explicitly by evaluating the expression `failwith ‘`t`‘` (or more generally `failwith` e where e has type `tok`).

23

```
# hd(tl[2]);;
evaluation failed     hd

# 1/0;;
evaluation failed     div

# (1/0)+1000;;
evaluation failed     div

# failwith (hd [‘a‘;‘b‘]);;
evaluation failed     a
```

A failure can be *trapped* by `?`; the value of the expression e_1 `?` e_2 is that of e_1, unless e_1 causes a failure, in which case it is the value of e_2.

24

```
# hd(tl[2]) ? 0;;
0 : int

# (1/0)?1000;;
1000 : int

# let half n =
#     if n=0 then failwith ‘zero‘
#            else let m=n/2
#                 in  if n=2*m then m else failwith‘odd‘;;
half = - : (int -> int)
```

The function `half` only succeeds on non-zero even numbers; on `0` it fails with `‘zero‘`, and on odd numbers it fails with `‘odd‘`.

```
# half 4;;                                                          25
2 : int

# half 0;;
evaluation failed zero

# half 3;;
evaluation failed odd

# half 3 ? 1000;;
1000 : int
```

Failures may be *trapped selectively* (on string) by `??`; if e_1 fails with token t, then the value of e_1 `??` `[`t_1`;...;`t_n`]` e_2 is the value of e_2 if t is one of t_1,...,t_n; otherwise the expression fails with the value of t.

```
# half(0) ?? [‘zero‘;‘plonk‘] 1000;;                                26
1000 : int

# half(1) ?? [‘zero‘;‘plonk‘] 1000;;
evaluation failed     odd
```

One may add several `??` traps to an expression, and one may add a `?` trap at the end as a catch-all.

```
# half(1)                                                           27
#   ??[‘zero‘] 1000
#   ??[‘odd‘]  2000;;
2000 : int

# hd(tl[half(4)])
#   ??[‘zero‘] 1000
#   ??[‘odd‘]  2000
#   ? 3000;;
3000 : int
```

One may use `!` or `!!` in place of `?` or `??` to cause re-iteration of the whole construct, analogously to using `loop` in place of `then`.

```
# let same(x,y) =                                                   28
#     if x>y then failwith ‘greater‘
#     if x<y then failwith ‘less‘
#            else x;;
same = - : ((int #  int) -> int)

# let gcd(x,y) =
#     letref x,y = x,y
#     in  same(x,y)
#           !![‘greater‘] x:=x-y
#           !![‘less‘]    y:=y-x;;
gcd = - : ((int #  int) -> int)

# gcd(12,20);;
4 : int
```

B.2.13 Type abbreviations

Types can be given names:

```
# lettype intpair = int #  int;;                                    29
type intpair defined

# let p = 12,20;;
p = (12, 20) : intpair
```

The new name is simply an abbreviation; for example, `intpair` and `int # int` are completely equivalent. The system always uses the most recently defined name when printing types.

```
# gcd;;                                                             30
- : (intpair -> int)

# gcd p;;
4 : int
```

B.2.14 Abstract types

New types can also be defined by abstraction. For example, to define a type `time` we could use the construct `abstype`:

```
# abstype time = int #  int                                         31
#  with maketime(hrs,mins) = if  hrs<0 or 23<hrs or
#                                   mins<0 or 59<mins
#                                then fail
#                                else abs_time(hrs,mins)
#  and hours t = fst(rep_time t)
#  and minutes t = snd(rep_time t);;
maketime = - : (intpair -> time)
hours = - : (time -> int)
minutes = - : (time -> int)
```

This defines an abstract type `time` and three primitive functions: `maketime`, `hours` and `minutes`. In general, an abstract type declaration has the form `abstype` *ty* `=` *ty′* `with` *b* where *b* is a binding, i.e. the kind of phrase that can follow `let` or `letrec`. Such a declaration introduces a new type *ty* which is represented by *ty′*. Only within *b* can one use the (automatically declared) functions `abs_`*ty* (of type *ty′* `->` *ty*) and `rep_`*ty* (of type *ty* `->` *ty′*), which map between a type and its representation. In the example above `abs_time` and `rep_time` are only available in the definitions of `maketime`, `hours` and `minutes`; these latter three functions, on the other hand, are defined throughout the scope of the declaration. Thus an abstract type declaration simultaneously declares a new type together with primitive functions for the type. The representation of the type (i.e. *ty′*), and of the primitives (i.e. the right hand sides of the definitions in *b*), is not accessible outside the `with`-part of the declaration.

```
# let t = maketime(8,30);;                                          32
t = - : time

# hours t , minutes t;;
(8, 30) : intpair
```

Notice that values of an abstract type are printed as `-`, like functions.

B.2.15 Type constructors

Both `list` and `#` are examples of type constructors; `list` has one argument (hence `* list`) whereas `#` has two (hence `* # **`). Each type constructor has various primitive operations associated with it, for example `list` has `null`, `hd`, `tl`, ... etc, and `#` has `fst`, `snd` and the infix `,`.

```
# let z = it;;                                                          33
z = (8, 30) : intpair

# fst z;;
8 : int

# snd z;;
30 : int
```

Another standard constructor of two arguments is `+`; `* + **` is the disjoint union of types `*` and `**`, and associated with it are the following primitives:

`isl  : (* + **) -> bool`	tests membership of left summand
`inl  : * -> (* + **)`	injects into left summand
`inr  : * -> (** + *)`	injects into right summand
`outl : (* + **) -> *`	projects out of left summand
`outr : (* + **) -> **`	projects out of right summand

These are illustrated by:

```
# let x = inl 1                                                         34
# and y = inr 2;;
x = inl 1 : (int + *)
y = inr 2 : (* + int)

# isl x;;
true : bool

# isl y;;
false : bool

# outl x;;
1 : int

# outl y;;
evaluation failed     outl

# outr x;;
evaluation failed     outr

# outr y;;
2 : int
```

Abstract types such as `time` defined above can be thought of as type constructors with no arguments (i.e. nullary constructors). The `abstype...with...` construct may also be used to define non-nullary type constructors (with `absrectype` in place of `abstype` if these are recursive). For example, trees analogous to LISP S-expressions could be defined by:

```
#  absrectype * sexp = * + (* sexp) #  (* sexp)                  35
#    with cons(s1,s2) = abs_sexp (inr (s1,s2))
#    and car s = fst (outr(rep_sexp s))
#    and cdr s = snd (outr(rep_sexp s))
#    and atom s = isl(rep_sexp s)
#    and makeatom a = abs_sexp(inl a);;
cons = - : ((* sexp #  * sexp) -> * sexp)
car = - : (* sexp -> * sexp)
cdr = - : (* sexp -> * sexp)
atom = - : (* sexp -> bool)
makeatom = - : (* -> * sexp)
```

B.3 Syntax of ML

We shall use variables to range over the various constructs of ML as follows:

Variable	Ranges over
var	variables
con	constructors
ce	constant expressions
ty	types
tab	type abbreviation bindings (see B.5.4)
ab	abstract type bindings (see B.5.5)
d	declarations
b	bindings
p	patterns
e	expressions

Variables and constructors are both represented by identifiers but they are different syntax classes. Identifiers and constant expressions are described in Section B.3.2 below. Types and type-bindings are explained in Section B.5. Declarations, bindings, patterns and expressions are defined by the following BNF-like syntax equations in which:

1. Each variable ranges over constructs as above.
2. The numbers following the various variables are there merely to distinguish between different occurrences—this will be convenient when we describe the semantics in Section B.4.
3. $\{C\}$ denotes an optional occurrence of C, and for $n{>}1$ $\{C_1 \mid C_2 \ldots \mid C_n\}$ denotes a choice of exactly one of C_1,C_2,...,C_n.
4. The constructs are listed in order of decreasing binding power.
5. 'L' or 'R' following a construct means that it associates to the left (L) or right (R) when juxtaposed with itself (where this is syntactically admissible).
6. Certain constructs are equivalent to others and this is indicated by 'equiv.' followed by the equivalent construct.

B.3.1 Syntax equations for ML

Table B.1 describes ML declarations, Table B.2 bindings, Table B.3 patterns, and Table B.4 on page 187 describes expressions.

d	::=	`let` b	ordinary variables
	|	`letref` b	assignable variables
	|	`letrec` b	recursive functions
	|	`lettype` tab	concrete types
	|	`rectype` cb	recursive concrete types
	|	`abstype` ab	abstract types
	|	`absrectype` ab	recursive abstract types

Table B.1: Declarations

b	::=	p`=`e	simple binding
	|	$id\ p_1\ p_2\ \ldots\ p_n\ \{$`:`$ty\}$ `=` e	function definition
	|	b_1 `and` b_2 $\ldots$ `and` b_n	multiple binding

Table B.2: Bindings

p	::=	`()`		empty pattern
	|	id		variable
	|	p`:`ty		type constraint
	|	p_1`.`p_2	R	list cons
	|	p_1`,`p_2	R	pairing
	|	`[]`		empty list
	|	`[`p_1`;`p_2 $\ldots$ `;`p_n`]`		list of n elements
	|	`(`p`)`		equivalent to p

Table B.3: Patterns

In the syntax equations constructs are listed in order of decreasing binding power. For example, since e_1e_2 is listed before e_1`;`e_2 function application binds more tightly than sequencing and thus e_1e_2`;` e_3 parses as `(`e_1e_2`);` e_3. This convention determines only the relative binding power of different constructs. The left or right association of a construct is indicated explicitly by 'L' for left and 'R' for right. For example, as application associates to the left, the expression $e_1e_2e_3$ parses as `(`e_1e_2`)` e_3, and since e_1 `=>` e_2 `|` e_3 associates to the right, the expression e_1 `=>` e_2 `|` e_3 `=>` e_4 `|` e_5 parses as e_1 `=>` e_2 `|` `(`e_3 `=>` e_4 `|` e_5`)`.

Only functions can be defined with `letrec`. For example, `letrec x = 2-x` would cause a syntax error.

All the variables occurring in a pattern must be distinct. On the other hand, a pattern can contain multiple occurrences of the wildcard `()`.

Spaces (ASCII 32), carriage returns (ASCII 13), line feeds (ASCII 10) form feeds (`^L`, ASCII 12) and tabs (`^I`, ASCII 9) can be inserted and deleted arbitrarily without affecting the meaning (as long as obvious ambiguities are not introduced). For example, the space in `-` x but not in `not` x can be omitted. *Comments*, which are arbitrary sequences of characters surrounded by `%`'s, can be inserted anywhere a space is allowed.

e	::=	ce		constant
	$\mid$	var		variable
	$\mid$	$e_1\ e_2$	L	function application
	$\mid$	e:ty		type constraint
	$\mid$	`-`e		unary minus
	$\mid$	e_1`*`e_2	L	multiplication
	$\mid$	e_1`/`e_2	L	division
	$\mid$	e_1`+`e_2	L	addition
	$\mid$	e_1`-`e_2	L	subtraction
	$\mid$	e_1`<`e_2		less than
	$\mid$	e_1`>`e_2		greater than
	$\mid$	e_1`.`e_2	R	list cons
	$\mid$	e_1`@`e_2	R	list append
	$\mid$	e_1`=`e_2	L	equality
	$\mid$	`not` e		negation
	$\mid$	e_1 `&` e_2	R	conjunction
	$\mid$	e_1 `or` e_2	R	disjunction
	$\mid$	e_1 *user-infix* e_2	L	user declared infix identifier
	$\mid$	e_1`=>`$e_2 \mid e_3$	R	equivalent to `if` e_1 `then` e_2 `else` e_3
	$\mid$	`do` e		evaluate e for side effects
	$\mid$	e_1`,`e_2	R	pairing
	$\mid$	p`:=`e		assignment
	$\mid$	`fail`		equivalent to `failwith 'fail'`
	$\mid$	`failwith e`		failure with explicit token
	$\mid$	`if` e_1 $\{\texttt{then}\mid\texttt{loop}\}$ e'_1 $\{$ `if` e_2 $\{\texttt{then}\mid\texttt{loop}\}$ e'_2 $\vdots$ `if` e_n $\{\texttt{then}\mid\texttt{loop}\}$ e'_n $\}$ $\{$ $\{\texttt{else}\mid\texttt{loop}\}$ e''_n $\}$		conditional and loop
	$\mid$	e $\{\texttt{??}\mid\texttt{!!}\}$ e_1 e'_1 $\{$ $\{\texttt{??}\mid\texttt{!!}\}$ e_2 e'_2 $\vdots$ $\{\texttt{??}\mid\texttt{!!}\}$ e_n e'_n $\}$ $\{$ $\{\texttt{?}\mid\texttt{!}\mid\texttt{?}\backslash id\mid\texttt{!}\backslash id\}$ e''_n $\}$		failure trap and loop
	$\mid$	e_1`;`e_2 ... `;`e_n		sequencing
	$\mid$	`[]`		empty list
	$\mid$	`[`e_1`;`e_2 ... `;`e_n`]`		list of n elements
	$\mid$	e `where` b	R	equivalent to `let` b `in` e
	$\mid$	e `whereref` b	R	equivalent to `letref` b `in` e
	$\mid$	e `whererec` b	R	equivalent to `letrec` b `in` e
	$\mid$	e `wheretype` db		equivalent to `lettype` db `in` e
	$\mid$	e `whereabstype` ab		equivalent to `abstype` ab `in` e
	$\mid$	e `whereabsrectype` ab		equivalent to `absrectype` ab `in` e
	$\mid$	d `in` e		local declaration
	$\mid$	*backslash* $p_1\ p_2\ \ldots\ p_n$`.`e		abstraction
	$\mid$	`(`e`)`		equivalent to e

Table B.4: Expressions

B.3.2 Identifiers and other lexical matters

In this section the lexical structure of ML is defined.

B.3.2.1 Identifiers

A variable (*var*) or *identifier* is a sequence of alphanumerics starting with a letter, where an alphanumeric is either a letter, a digit, a prime (') or an underbar (_). ML is case-sensitive: upper and lower case letters are considered to be different.

B.3.2.2 Constant expressions

The ML constant expressions (*ce*'s) used in Table B.4 are:

1. Integers, i.e. sequences of digits `0,1...,9`.
2. Truth values `true` and `false`.
3. Tokens and token-lists:
   (a) Tokens consist of any sequence of characters surrounded by token quotes (\`), e.g. `` `This is a single token` ``.
   (b) Token-lists consist of any sequence of tokens (separated by spaces, returns, line-feed or tabs) surrounded by token-list quotes (\`\`). e.g. ``` ``this is a token-list containing 7 members`` ```. ``` `` tok1 tok2 ... tokn`` ``` is equivalent to ``[`tok1`; `tok2`; ...; `tokn`]``.

   In any token or token-list, the occurrence of \\x has the following meanings for different x's:

   `\0` = ten spaces
   `\n` = `n` spaces (0<`n`<10)
   `\S` = one space
   `\R` = return
   `\L` = line-feed
   `\T` = tab
   `\x` = `x` taken literally otherwise (e.g. \\\` to include token quotes in a token or token-list)
4. Strings, consisting of any sequence of characters surrounded by string quotes ("), e.g. `"This is a single string"`. Any " characters within a string must be preceded by \\. The escape sequence \\x for any other character means always to insert the character x.
5. The expression `()`, called *thing*, which evaluates to the unique object of ML type `unit`.

B.3.2.3 Prefixes and infixes

The ML *prefixes px* and *infixes ix* are given by:

```
px ::= not | - | do
ix ::= * | / | + | - | . | @ | = | < | > | & | or | ,
```

In addition, any identifier (and certain single characters) can be made into an infix. Such user-defined infixes bind more tightly than `...=>...|...` but more weakly than `or`. All of them have the same power binding and associate to the left.

Except for `&` and `or`, each infix ix (or prefix px) has correlated with it a special identifier $\$ix$ (or $\$px$) which is bound to the associated function. For example, the identifier `$+` is bound to the addition function, and `$@` to the list-append function (see Section B.6 for the meaning of dollared infixes). This is useful for passing functions as arguments; for example, f`$@` applies f to the append function.

See the descriptions of the functions `ml_paired_infix` and `ml_curried_infix` in Section B.6.1 for details of how to give an identifier infix status.

B.4 Semantics of ML

The evaluation of all ML constructs takes place in the context of an *environment* and a *store*. The environment specifies what the variables and constructors in use denote. Variables may be bound either to *values* or to *locations*. The contents of locations—which must be values—are specified in the *store*. If a variable is bound to a location then (and only then) is it *assignable*. Thus bindings are held in the environment, whereas location contents are held in the store. Constructors may only be bound to values (constructor constants or constructor functions) and this binding occurs when they are declared in a concrete type definition.

The evaluation of ML constructs may either *succeed* or *fail*. In the case of success:

1. The evaluation of a declaration, d say, changes the bindings in the environment of the identifiers declared in d. If d is at top-level, then the scope of the binding is everything following d. In d `in` e the scope of d is the evaluation of e, and so when this is finished the environment reverts to its original state (see Section B.4.1).
2. The evaluation of an expression yields a value: the value of the expression (see Section B.4.2).

If an assignment is done during an evaluation, then the store will be changed — we shall refer to these changes as *side effects* of the evaluation.

If the evaluation of a construct fails, then failure is signalled, and a string is passed to the context which invoked the evaluation. This string is called the *failure string*, and it normally indicates the cause of the failure. During evaluation, failures may be generated either *implicitly* by certain error conditions, or *explicitly* by the construct `failwith` e (which fails with e's value as failure string). For example, the evaluation of the expression `1/0` fails implicitly with failure string `` `div` ``, while that of `failwith` `` ` ``str`` ` `` fails explicitly with failure string `` ` ``str`` ` ``. We shall say two evaluations fail *similarly* if they both fail with the same failure string. For example, the evaluation of `1/0` and `` failwith `div` `` fail similarly. Side effects are not undone by failures.

If during the evaluation of a construct a failure is generated, then unless the construct is a failure trap (i.e. an expression built from `?` and/or `!`) the evaluation of the construct itself fails similarly. Thus failures propagate up until trapped, or reaching top level. For example, when evaluating `(1/0)+1000`, the expression `1/0` is first evaluated, and the failure which this evaluation generates causes the evaluation of the whole expression (viz. `(1/0)+1000`) to fail with `` `div` ``. On the other hand, the evaluation of `(1/0)?1000` traps the failure generated by the evaluation of `1/0`, and succeeds with value `1000`. (In general, the evaluation of e_1`?`e_2 proceeds by first evaluating e_1, and if this succeeds with value E, then E is returned as the value of e_1`?`e_2; however, if e_1 fails, then the result of evaluating e_1`?`e_2 is determined by evaluating e_2).

In describing evaluations, when we say that we *pass control* to a construct, we mean that the outcome of the evaluation is to be the outcome of evaluating the construct. For example, if when evaluating e_1?e_2 the evaluation of e_1 fails, then we pass control to e_2.

Expressions and patterns can be optionally decorated with types by writing $:ty$ after them (e.g. `[]:int list`). The effect of this is to force the type checker to assign an instance of the asserted type to the construct; this is useful as a way of constraining types more than the type checker would otherwise (i.e. more than context demands), and it can also serve as helpful documentation. Details of types and type checking are given in Section B.5, and will be ignored in describing the evaluation of ML constructs in the rest of this section.

If we omit types, precedence information and those constructs which are equivalent to others, then the syntax of ML can be summarized by:

$$
\begin{array}{lll}
d & \texttt{::=} & \texttt{let}\ b \mid \texttt{letref}\ b \mid \texttt{letrec}\ b \\
b & \texttt{::=} & p\texttt{=}e \mid var\ p_l\ p_2 \ldots p_n\ \texttt{=}\ e \mid b_l\ \texttt{and}\ b_2 \ldots\ \texttt{and}\ b_n \\
p & \texttt{::=} & \texttt{()}\ \mid var \mid p_l\texttt{.}p_2 \mid p_l\texttt{,}p_2 \mid \texttt{[]} \mid \texttt{[}p_l\texttt{;}p_2 \ldots\ \texttt{;}p_n\texttt{]} \\
e & \texttt{::=} & ce \mid var \mid e_1\ e_2 \\
 & \mid & px\ e \mid e_1\ ix\ e_2 \mid v\texttt{:=}e \mid\ \texttt{failwith}\ e \\
 & \mid & \quad \texttt{if}\ e_1\ \{\texttt{then}|\texttt{loop}\}\ e_1' \\
 & & \{\ \texttt{if}\ e_2\ \{\texttt{then}|\texttt{loop}\}\ e_2' \\
 & & \qquad\qquad \vdots \\
 & & \quad \texttt{if}\ e_n\ \{\texttt{then}|\texttt{loop}\}\ e_n'\ \} \\
 & & \{ \qquad\qquad \{\texttt{else}|\texttt{loop}\}\ e_n''\ \} \\
 & \mid & \quad e\ \ \{\texttt{??}|\texttt{!!}\}\ e_1\ e_1' \\
 & & \{\ \ \{\texttt{??}|\texttt{!!}\}\ e_2\ e_2' \\
 & & \qquad\qquad \vdots \\
 & & \quad \{\texttt{??}|\texttt{!!}\}\ e_n\ e_n'\ \} \\
 & & \{\ \ \{\texttt{?}|\texttt{!}|\texttt{?}\backslash id|\texttt{!}\backslash id\}\ e_n''\ \} \\
 & \mid & e_l\texttt{;}e_2 \ldots\texttt{;}e_n \mid \texttt{[]}\ \mid \texttt{[}e_l\texttt{;}e_2 \ldots\texttt{;}e_n\texttt{]} \mid d\ \texttt{in}\ e \\
 & \mid & \backslash p_1 p_2 \ldots p_n\texttt{.}e
\end{array}
$$

B.4.1 Declarations

Any declaration must be one of the three kinds: `let` b, `letref` b or `letrec` b, where b is a binding. Each such declaration is evaluated by first evaluating the binding b to produce a (possibly empty) set of variable-value pairs, and then extending the environment (in a manner determined by the kind of declaration) so that each variable in this set of pairs denotes its corresponding value. The evaluation of bindings is described below in Section B.4.1.1.

1. Evaluating `let` b declares the variables specified in b to be an ordinary (i.e. non assignable) variable, and binds (in the environment) each one to the corresponding value produced by evaluating b. To understand what are the variables defined in a declaration may require some knowledge about the environment. For example, a declaration `let` $f\ x$ `=` e declares x if f is a constructor and declares f as the function $\backslash x\texttt{.}e$ otherwise.
2. Evaluating `letref` b declares the variables specified in b to be assignable and thus binds (in the environment) each one to a new location, whose contents (in the store) is set to the corresponding value. The effect of subsequent assignments to the variables will be to change the contents of the locations they are bound to. Bindings (in the environment) of variables to locations can only be changed by evaluating another declaration to supersede the original one.
3. Evaluating `letrec` b is similar to evaluating `let` b except that:
   (a) The binding b in `letrec` b must consist only of function definitions.

(b) These functions are made mutually recursive.

For example, consider:

(a) `let f n = if n=0 then 1 else n*f(n-1)`

(b) `letrec f n = if n=0 then 1 else n*f(n-1)`

The meaning of `f` defined by the first case depends on whatever `f` is bound before the declaration is evaluated, while the meaning of `f` defined by the second case is independent of this (and is the factorial function).

B.4.1.1 The evaluation of bindings

There are three kinds of variable binding each of which, when evaluated, produces a set of variable-value pairs (or fails):

1. *Simple bindings*, which have the form p=e where p is a pattern and e an expression.
2. *Function definitions*, which have the form $id\ p_1 \ldots p_n$ `=` e. This is just an abbreviation for the simple binding id `=` \\$p_1 \ldots p_n$`.`$e$.
3. *Multiple bindings*, which have the form b_1 `and` $b_2 \ldots$ `and` b_n where $b_1, b_2 \ldots, b_n$ are simple bindings or function definitions. As a function definition is just an abbreviation for a certain simple binding, each b_i $(0<i<n+1)$ either is, or is an abbreviation for, some simple binding p_i=e_i. The multiple binding b_1`and` $b_2 \ldots$ `and` b_n then abbreviates p_1`,`$p_2 \ldots$`,`p_n `=` e_1`,` $e_2 \ldots e_n$, which is a simple binding.

As function definitions and multiple bindings are abbreviations for simple bindings we need only describe the evaluation of the latter.

A simple binding p=e is evaluated by first evaluating e to obtain a value E (if the evaluation fails then the evaluation of p=e fails similarly). Next the pattern p is *matched* with E to see if they have the same form (precise details are given in Section B.4.1.2). If so, then to each identifier in p there is a corresponding component of E. The evaluation of p=e then returns the set of each identifier paired with its corresponding component. If p and E do not match then the evaluation of p=e fails with failure token `‘MATCH‘`.

B.4.1.2 Matching patterns and expression values

When a pattern p is matched with a value E, either the match succeeds and a set of identifier-value pairs is returned (each identifier in p being paired with the corresponding component of E), or the match fails. We describe, by cases on p, the conditions for p to match E and the sets of pairs returned:

`()`**:** Always matches E. The empty set of pairs is returned.

*var***:** Always matches E. The set consisting of *var* paired with E is returned.

p_1`.`p_2**:** E must be a non-empty list E_1`.`E_2 such that p_1 matches E_1 and p_2 matches E_2. The union of the sets of pairs returned from matching p_1 with E_1 and p_2 with E_2 is returned.

p_1`,`p_2**:** E must be a pair E_1`,`E_2 such that p_1 matches E_1 and p_2 matches E_2. The union of the sets of pairs returned from matching p_1 with E_1 and p_2 with E_2 is returned.

`[`p_1`;`$p_2 \ldots$`;`p_n`]`**:** E must be a list `[`E_1`;`$E_2 \ldots$`;`E_n`]` of length n such that for each i p_i matches E_i. The union of the sets of pairs returned by matching p_i with E_i is produced.

Thus if p matches E, then p and E have a similar 'shape', and each identifier in p corresponds to some component of E (namely that component paired with the identifier in the set returned by the match). Here are some examples:

1. `[`x`;`y`;`z`]` matches `[1;2;3]` with x, y and z corresponding to `1`, `2`, and `3` respectively.
2. `[`x`;`y`;`z`]` does not match `[1;2]` or `[1;2;3;4]`.
3. x`.`y matches `[1;2;3]` with x and y corresponding to `1` and `[2;3]` respectively, because E_1`.[`E_2`;`E_3`...;`E_n`]` = `[`E_1`;`E_2`;`E_3`...;`E_n`]`.
4. x`.`y does not match `[]` or `1,2`.
5. x`,`y matches `1,2` with x and y corresponding to `1` and `2` respectively.
6. x`,`y does not match `[1;2]` .
7. `(`x`,`y`),[(`z`.`w`);()]` matches `(1,2),[[3;4;5];[6;7]]` with x, y, z and w corresponding to `1`, `2`, `3`, and `[4;5]` respectively.

B.4.2 Expressions

If the evaluation of an expression terminates, then either it succeeds with some value, or it fails; in either case assignments performed during the evaluation may cause side effects. If the evaluation succeeds with some value we shall say that value is *returned*.

We shall describe the evaluation of expressions by considering the various cases, in the order in which they are listed in the syntax equations.

ce: The appropriate constant value is returned.

var: The value associated with *var* is returned. If *var* is ordinary, then the value returned is the value bound to *var* in the environment. If *var* is assignable, then the value returned is the contents of the location to which *var* is bound.

e_1 e_2: e_1 and e_2 are evaluated and the result of applying the value of e_1 (which must be a function) to that of e_2 is returned. Due to optimizations in the ML compiler, the order of evaluation may vary.

px e: e is evaluated and then the result of applying *px* to the value of e is returned. `-`e and `not` e have the obvious meanings; `do` e evaluates e for its side effects and then returns `()`.

e_1 *ix* e_2: e_1 `&` e_2 is equivalent to `if` e_1 `then` e_2 `else false`, so sometimes only e_1 needs be evaluated to evaluate e_1 `&` e_2. e_1 `or` e_2 is equivalent to `if` e_1 `then true else` e_2, so sometimes only e_1 needs to be evaluated to evaluate e_1 `or` e_2.

In all other cases e_1 and e_2 are evaluated (in that order) and the result of applying *ix* to their two values is returned. e_1`,`e_2 returns a pair whose first component is the value of e_1, and whose second component is the value of e_2. The meaning of the other infixes are given in Section B.6.

p`:=`e: Every variable in p must be assignable and bound to some location in the environment. The effect of the assignment is to update the contents of these locations (in the store) with the values corresponding to the variables produced by evaluating the binding p=e (see Section B.4.1.1). If the evaluation of e fails, then no updating of locations occurs, and the assignment fails similarly. If the matching to p fails, then the assignment fails with `'MATCH'`. The value of p`:=`e is the value of e.

`failwith` e: e is evaluated and then a failure with e's value (which must be a token) is generated.

$$\begin{array}{l} \ \texttt{if}\ e_1\ \{\texttt{then|loop}\}\ e'_1 \\ \{\texttt{if}\ e_2\ \{\texttt{then|loop}\}\ e'_2 \\ \quad\vdots \\ \ \texttt{if}\ e_n\ \{\texttt{then|loop}\}\ e'_n\ \} \\ \qquad\{\ \{\texttt{else|loop}\}\ e'\ \} \end{array}$$ **:** e_1, e_2,...,e_n are evaluated in turn until one of them, say e_m, returns `true` (each e_i must be a boolean expression). When the phrase following e_m is `then` e'_m control is passed to e'_m. However, when the phrase is `loop` e'_m then e'_m is evaluated for its side effects, and then control is passed back to the beginning of the whole expression again (i.e. to the beginning of `if` e_1 ...).

In the case that all of e_1,e_2 ...,e_n return `false` and there is a phrase following e'_n, then if this is `else` e' control is passed to e', while if it is `loop` e' then e' is evaluated for its side effects and control is then passed back to the beginning of the whole expression again.

In the case that all of e_1,e_2 ...,e_n return `false`, but no phrase follows e'_n then `()`, the unique value of type `void` is returned.

$$\begin{array}{l} e\ \{\texttt{??|!!}\}\ e_1 e'_1 \\ \{\ \{\texttt{??|!!}\}\ e_2 e'_2 \\ \quad\vdots \\ \quad\{\texttt{??|!!}\}\ e_n e'_n\ \} \\ \{\ \{\texttt{?|!|?}\backslash id|\backslash id\}\ e'\ \} \end{array}$$ **:** e is evaluated and if this succeeds its value is returned. If e fails with failure token *tok*, then each of e_1,e_2 ...,e_n are evaluated in turn until one of them, say e_m, returns a token list containing *tok* (each e_i must be a token). If `??` immediately precedes e_m, then control is passed to e'_m. If `!!` precedes it, then e'_m is evaluated and control is passed back to the beginning of the whole expression e `{??|!!} ...`.

If none of e_1,e_2 ..., e_n produces a token list containing *tok*, and `?\`e' follows e'_n, then control is passed to e'. But if `!\`e' follows e'_n, then e' is evaluated, and control is passed back to the beginning of the whole expression.

If `?\`*id* e' or `?\`*id* e' follows e'_n, then e' is evaluated in an environment in which *id* is bound to the failure string *tok* (i.e. an evaluation equivalent to `let` *id*`=`*tok* `in` e' is done), and then depending on whether a `?` of a `!` occurred, the value of e' is returned or control is passed back to the beginning of the whole expression respectively.

If none of e_1,e_2 ...,e_n returns a token list containing *tok* and nothing follows e'_n, then the whole expressions fails with *tok*.

e_1`;`e_2 ...`;`e_n**:** e_1,e_2 ...,e_n are evaluated in that order, and the value of e_n is returned.

`[`e_1`;`e_2 ...`;`e_n`]`**:** $e_1, e_2, \ldots, e_n$ are evaluated in that order and the list of their values returned. `[]` evaluates to the empty list.

d `in` e**:** d is evaluated and then e is evaluated in the extended environment and its value returned. The declaration d is local to e, so that after the evaluation of e, the former environment is restored.

`\`$p_1 p_2 \ldots p_n$`.`e**:** The evaluation of lambda-expressions always succeeds and yields a function value. The environment in which the evaluation occurs (i.e. in which the function value is created) is called the *definition environment*.

1. *Simple lambda-expressions*: `\`p`.`e evaluates to that function which, when applied to some argument yields the result of evaluating e in the current (i.e. application time) store, and in the environment obtained from the definition environment by binding any variables in p to the corresponding components of the argument (see Section B.4.1.1).
2. *Compound lambda-expressions*: A lambda-expression with more than one parameter is curried, i.e. `\`$p_1 p_2 \ldots p_n$`.`e is equivalent to `\`p_1`. (\`p_2`. ... \`p_n`.`e`) ...`.

Thus the free variables in a function keep the same binding they had in the definition environment. So if a free variable is non-assignable in that environment, then its value is fixed to the value it has there. On the other hand, if a free variable is assignable in the definition environment, then it will be bound to a location. Although that binding is fixed, the contents of the location in the store is not, and can be subsequently changed with assignments.

B.5 ML Types

So far, little mention has been made of types. For ML in its original role as the meta-language for proof in LCF, the importance of strict type checking was principally to ensure that every computed value of the type representing theorems was indeed a theorem.[2]

The same effect could probably have been achieved by run-time type checking, but compile-time type checking was adopted instead, in the design of ML. This was partly for the considerable debugging aid that it provides; partly for efficient execution; and partly to explore the possibility of combining polymorphism with type checking. This last reason is of general interest in programming languages and has nothing to do specifically with proof; the problem is that there are many operations (list mapping functions, functional composition, etc.) which work at an infinity of types, and therefore their types should somehow be parameterized – but it is rather inconvenient to have to mention the particular type intended at each of their uses.

The ML type checking system is implemented in such a way that, although the user may occasionally (either for documentation or as a constraint) ascribe a type to an ML expression or pattern, it is hardly ever necessary to do so. The user of ML will almost always be content with the types ascribed and presented by the type checker, which checks every top-level phrase before it is evaluated. (The type checker may sometimes find a more general type assignment than expected.)

B.5.1 Types and objects

Every data object in ML possesses a type. Such an object may possess many types, in which case it is said to be *polymorphic* and possesses a *polytype* – i.e. a type containing type variables (for which we use a sequence of asterisks possibly followed by an identifier or integer) – and moreover it possesses all types which are *instances* of its polytype, formed by substituting types for zero or more type variables in the polytype. A type containing no type variables is a *monotype*.

We saw several examples of types in Section B.2. To understand the following syntax, note that `list` is a postfixed unary (one-argument) type constructor (thereafter abbreviated to *tycon*). The user may introduce new n-argument type constructors. A binary type operator `directory`, for example, can be introduced. The following type expressions will then be types of different kinds of `directory`:

- `(tok, int) directory`
- `(int, int -> int) directory`

The user may even deal with lists of directories, with the type `(int, bool) directory list`

B.5.1.1 The syntax of types

The syntax of ML types is summarized in Table B.5. Type abbreviations are introduced by a `lettype` declaration (see Section B.5.4 below) which allows an identifier to abbreviate an arbitrary monotype. An abstract type likewise consists of an identifier (introduced by an `abstype` or

[2]The NUPRL system also relies on strict type checking to ensure that objects of type `proof` can only be constructed by reference to a fixed set of inference rules.

Types	*ty*	::=	*sty*		Standard (non-infix) type
		|	*ty* # *ty*	R	Cartesian product
		|	*ty* + *ty*	R	Disjoint sum
		|	*ty* -> *ty*	R	Function type
Standard Types	*sty*	::=	unit | int		
		|	bool		
		|	tok | string		Basic types
		|	*vty*		Type variable
		|	*tycon*		Type abbreviation (see Section B.5.4)
		|	*tycon*		Nullary abstract type
		|	*tyarg tycon*	L	Abstract type (see Section B.5.5)
		|	(*ty*)		
Type arguments	*tyarg*	::=	*sty*		Single type argument
		|	(*ty*,...,*ty*)		One or more type arguments
Type variables	*vty*	::=	* | ** | ...		
		|	**id* | ***id* | ...		
		|	*0 | **0 | ...		
		|	*1 | **1 | ...		
		|	⋮ | ⋮ | ...		

Table B.5: ML Type Syntax

`absrectype` declaration; see Section B.5.5) postfixed to zero or more type arguments. Two or more arguments must be separated by commas and enclosed by parentheses. The type operator `list` is a predeclared unary type operator; and `#`, `+` and `->` may be regarded as infix forms of three predeclared binary type operators.

For an object to possess[3] a type means the following: For basic types, all integers possess `int`, both booleans possess `bool` , all strings possess `string`, etc. The only object possessing `void` (or `unit`) is that denoted by `()` in ML. For a type abbreviation *tycon*, an object possesses *tycon* (during execution of phrases in the scope of the declaration of *tycon*) if and only if it possesses the type which *tycon* abbreviates. For compound monotypes ,

1. The type *ty* `list` is possessed by any list of objects, all of which possess type *ty* (so that the empty list possesses type *ty* `list` for every *ty*).
2. The type ty_1 `#` ty_2 is possessed by any pair of objects possessing the types ty_1 and ty_2, respectively.
3. The type ty_1 `+` ty_2 is possessed by the left-injection of any object possessing ty_1, and by the right-injection of any object possessing ty_2. These injections are denoted by the ML function identifiers `inl : * -> * + **` and `inr : ** -> * + **` (see Section B.6).
4. A function possesses type ty_1 `->` ty_2 if, whenever its argument possesses type ty_1, its result (if defined) possesses type ty_2. (This is not an exact description; for example, a function defined in ML with non-local variables may possess this type even though some assumption about the types of the values of these non-local variables is necessary for the above condition to hold. The constraints on programs listed below ensure that the non-locals will always have the right types).

[3]We shall talk of objects *possessing* types and phrases *having* types, to emphasize the distinction.

5. An object possesses the abstract type *tyarg id* if and only if it is represented (via the abstract type representation) by an object possessing the *tyarg* instance of the right-hand side of the declaration of *id*.

Finally, an object possesses a polytype *ty* if and only if it possesses all monotypes which are substitution instances of *ty*.

B.5.2 Typing of ML phrases

We now explain the constraints used by the type checker in ascribing types to ML expressions, patterns and declarations.

The significance of expression e having type *ty* is that the value of e (if evaluation terminates successfully) possesses type *ty*. As consequences of the well-typing constraints listed below, it is impossible for example to apply a non-function to an argument, or to form a list of objects of different types, or (as mentioned earlier) to compute an object of the type corresponding to theorems which is not a theorem.

The type ascribed to a phrase depends in general on the entire surrounding ML program. In the case of top-level expressions and declarations, however, the type ascribed depends only on preceding top-level phrases. Thus you know that types ascribed at top-level are not subject to further constraint.

Before each top-level phrase is executed, types are ascribed to all its sub-expressions, sub-declarations and sub-patterns according to the following rules. Most of the rules are fairly natural; those which are less so are discussed later. You are only presented with the types of top-level phrases; the types of sub-phrases will hardly ever concern you.

Before giving the list of constraints, let us discuss an example which illustrates some important points. To map a function over a list we may define the polymorphic function `map` recursively as follows (where we have used an explicit abstraction, rather than `letrec map f l = ...`, to make the typing clearer):

```
letrec map = \f.\l. null l => [] | f(hd l).map f(tl l) ;;
```

From this declaration the type checker will infer a *generic* type for `map`. By 'generic' we mean that each later occurrence of `map` will be ascribed a type which is a substitution instance of the generic type.

Now the free identifiers in this declaration are `null`, `hd` and `$.`, which are ML primitives whose *generic* (poly)types are `* list -> bool`, `* list -> *`, and `* # * list -> * list` respectively. The first constraint used by the type checker is that the occurrences of these identifiers in the declaration are ascribed instances of their generic types. Other constraints which the type checker will use to determine the type of `map` are:

- All occurrences of a lambda-bound variable receive the same type.
- Each arm of a conditional receives the same type, and the condition receives type `bool`.
- In each application e `=` `(`$e_1 e_2$`)`, if e_2 receives *ty* and e receives *ty*$'$ then e_1 receives *ty* `->` *ty*$'$.
- In each abstraction e `=` `\`v`.`e_1, if v receives *ty* and e_1 receives *ty*$'$ then e receives *ty* `->` *ty*$'$.
- In a `letrec` declaration, all free occurrences of the declared variable receive the same type.

Now the type checker will ascribe the type `(*->**)->* list->** list` to `map`. This is in fact the most general type consistent with the constraints mentioned. Moreover, it can be shown that any instance of this type also allows the constraints to be satisfied; this is what allows us to claim that the declaration is indeed polymorphic.

In the following constraint list, we say p has ty to indicate that the phrase p is ascribed a type ty which satisfies the stated conditions. We use x, p, e, d to stand for variables, patterns, expressions and declarations respectively.

Constants:

1. `()` has type `unit`
2. `0` has type `int` , `1` has type `int`, ...
3. `true` has type `bool`, `false` has type `bool`
4. `‘...‘` has type `tok`
5. `"..."` has type `string`

Variables and constructors: The constraints described here are discussed in Section B.5.3 below.

1. If x is a variable bound by `\`, `fun` or `letref`, then x is ascribed the same type as its binding occurrence. In the case of `letref`, this must be monotype if the `letref` is top-level or an assignment to x occurs within a lambda-expression within its scope.
2. If x is bound by `let` or `letrec`, then x has ty, where ty is an instance of the type of the binding occurrence of x (i.e. the *generic* type of x), in which type variables occurring in the types of current lambda-bound or `letref`-bound identifiers are not instantiated.
3. If x is not bound in the program (in which case it must be an ML primitive), then x has ty, where ty is an instance of the type of x given in Section B.6.

Patterns: Cases for a pattern p:

- `()`: p has ty, where ty is any type.
- p_1`:`ty:
  p_1 and p have an instance of ty.
- p_1`,`p_2: If p_1 has ty_1 and p_2 has ty_2, then p has ty_1 `#` ty_2.
- p_1`.`p_2: If p_1 has ty then p_2 and p have ty `list`.
- `[`p_1`;...;`p_n`]`: For some ty, each p_i has ty and p has ty `list`.

Expressions: Cases for an expression e (not a constant or identifier):

- $e_1 e_2$: If e_2 has ty and e has ty' then e_1 has ty `->` ty'.
- e_1`:`ty: e_1 and e have an instance of ty.
- px e_1: Treated as `$`px`(`e_1`)` when px is a prefix. If e is `-`e_1, then e and e_1 have `int`.
- e_1 ix e_2: Treated as `$`ix`(`e_1`,`e_2`)` if ix is introduced with `ml_paired_infix`, and as `($`ix e_1 e_2`)` if ix is introduced by `ml_curried_infix`. If e is `(`e_1 `&` e_2`)` or `(`e_1 `or` e_2`)` then e, e_1 and e_2 have `bool`.
- e_1`,`e_2: If e_1 has ty_1 and e_2 has ty_2 then e has ty_1`#`ty_2.
- p`:=`e_1: For some ty, p, e_1 and e all have ty.
- `failwith` e_1: e_1 has `tok`, and e has any type.

- `if` e_1 `then` $e'_1 \ldots$ `if` e_n `then` e'_n `else` e': Each e_i has `bool`, and e, each e'_i, and e' all have ty for some ty. However, this constraint does not apply to an e'_i preceded by `loop` in place of `then`, nor to e' preceded by `loop` in place of `else`. If e' is absent, then $ty =$ `void`.
- e'_0 `??` e_1 $e'_1 \ldots$ `??` e_n e'_n `?{`$\backslash x$`}`e': Each e_i has `tok list`, and e, e'_0, each e'_i and e' all have ty for some ty. However, this constraint does not apply to an e'_i preceded by `!!` in place of `??` nor to e' preceded by `!` in place of `?`. If $\backslash x$ is present, x has `tok`.
- e_1`;...;`e_n: If e_n has ty then e has ty.
- `[`e_1`;...;`e_n`]`: For some ty, each e_i has ty and e has ty `list`.
- d `in` e_1: If e_1 has ty then e has ty. If d is a type definition (see Sections B.5.4 and B.5.5) then ty must contain no type defined in d.
- $\backslash p$`.`e_1: If p has ty and e_1 has ty' then e has ty `->`ty'.

Declarations:

1. Each binding$x$ $p_1 \ldots p_n$ `=` e is treated as x `=` $\backslash p_1$`. ...`$\backslash p_n$`.`e.
2. `let` p_1 `=` e_1 `and ...and` p_n `=` e_n is treated as `let` p_1`,...,`p_n `=` e_1`,...,`e_n (similarly for `letrec` and `letref`).
3. If d is `let` p`=`e, then d, p and e all have ty for some ty (similarly for `letref`). Note that e is not in the scope of the declaration.
4. If d is `letrec` x_1`,...,`x_n `=` e_1`,...,`e_n, then x_i and e_i have ty_i, and d has ty_1`#...#`ty_n for some ty_i. In addition, each free occurrence of x_i in $e_1,\ldots,e_n$ has ty_i, so that the type of recursive calls of x_i is the same as the declaring type.

B.5.3 Discussion of type constraints

We give here reasons for our constraints on the types ascribed to occurrences of identifiers. The reader may like to skip this section at first reading.

1. Consider constraint (1) for lambda-bound identifiers. This constraint implies that the expression `let` x `=` e `in` e' may be well-typed even if the semantically-equivalent expression `let` f x `=` e' `in` f e is not, since in the former expression x may occur in e' with two incompatible types which are both instances of the declaring type. The greater constraint on f is associated with the fact that f may be applied to many different arguments during evaluation. To show the need for the constraint, suppose that it is replaced by the weaker constraint for `let`-bound identifiers, so that for example `let f x = if x then 1+x else x(1)` is a well-typed declaration of type `*->int`, in which the occurrences of `x` receive types `*`, `bool`, `int`, `int->int` respectively. In the absence of an explicit argument for the abstraction, no constraint exists for the type of the binding occurrence of `x`. But, because `f` is `let`-bound, expressions such as `f true` and `` f `dog` `` are admissible in the scope of `f`, although their evaluation should result in either nonsense or *run-time* type-errors; one of our purposes is to preclude these.

   The only exception to this rule is for expressions of the form `(`$\backslash x$`.`e'`)`e, which is treated exactly as `let` x`=`e `in` e'. Here we know the unique instance of type of the argument x, namely the type of e.

2. The analogous restriction for `letref`-bound identifiers is also due to the possibility that the identifier-value binding may change during evaluation (this time because of assignments). Consider the following:

   ```
   letref x = [] in
         (if e then do(x := 1.x) else do(x := [true]) ; hd x) ;;
   ```

   If `letref` were treated like `let`, this phrase would be well-typed and indeed have type `*`, despite the fact that the value returned is either `1` or `true`. So, calling the whole expression e, all manner of larger expressions involving e would be well-typed, even including e`(`e`)`!

3. Top level `letref`s must be monomorphic to avoid retrospective type constraints at top-level. If this restriction were removed the following would be allowed:

   ```
   letref x = [] ;;
       ⋮
     2.x ;;
   ```

   But on type checking the last phrase, it would appear that the type of `x` at declaration should have been `int list`, not `* list`, and the types of intervening phrases may likewise need constraining.

4. To see the need for the exclusion of polymorphic non-local assignments, consider this example in the HOL system (this example is originally due to Lockwood Morris). (The type `thm` is the type of theorems.)

   ```
   let store,fetch =
       letref x = [] in (\y. x:=[y]) , (\(). hd x ) ;;
   store "T = F" ;;
   let eureka :thm = fetch() ;;
   ```

   Now suppose we lift our constraint. Then in the declaration, `x` has type `* list` throughout its (textual) scope, and `store`, `fetch` receive types `*->* list`, `**->*` respectively. In the two ensuing phrases they get respective types `term->term list`, `void->thm` (instances of their declaring types), and the value of `eureka` is a contradictory formula masquerading as a theorem!

   The problem is that the type checker has no simple way of discovering the types of all values assigned to the polymorphic `x`, since these assignments may be invoked by calls of the function `store` outside the (textual) scope of `x`. This is not possible under our constraint.
   However, polymorphic assignable identifiers are still useful: consider

   ```
   let rev l =
        letref l,l' = l,[] in
          if null l then l' loop (l,l':= tl l, hd l.l') ;;
   ```

   Such uses of assignable identifiers for iteration may be avoided given a suitable syntax for iteration, but assignable identifiers are useful for a totally different purpose, namely as 'own variables' shared between one or more functions (as in the store-fetch example). Our constraint of course requires them to be monomorphic; this is one of the few cases where the user occasionally needs to add an explicit type to a program.

B.5.4 Type abbreviations

The syntax of type abbreviation bindings *tab* is

tab `::=` id_1 `=` ty_1 `and` ... `and` id_n `=` ty_n

Then the declaration

`lettype` tab

in which each ty_i must be a monotype, built from basic types and previously defined types, allows you to introduce new names id_i for the types ty_i. Within the scope of the declaration the expression e`:`id_i behaves exactly like e`:`ty_i, and the type ty_i will always be printed as id_i.

One aspect of such type abbreviations should be emphasized. Suppose for the rational numbers you declare `lettype rat = int # int;;` and set up the standard operations on rationals. Within the scope of this declaration *any* expression of type `int # int` will be treated as though it had type `rat`, and this could be not only confusing but also incorrect (in which case it ought to cause a type failure). If you wish to introduce the type `rat`, isomorphic to `int # int` but not matching it for type checking purposes, then you should use abstract types.

B.5.5 Abstract types

As with concrete types, abstract type constructors may be introduced by a declaration in which type variables are used as dummy arguments (or formal parameters) of the operators. The syntax of abstract type bindings *ab* is

ab `::=` $vtyarg_1$ $tycon_1$ `=` ty_1 `and` ...`and` $vtyarg_n$ $tycon_n$ `=` ty_n `with` b

where each $vtyarg_i$ must contain no type variable more than once, and all the type variables in ty_i must occur in $vtyarg_i$. An abstract type declaration takes the form

{`abstype`|`absrectype`} ab

The declaration introduces a set of type operators, and also incorporates a normal binding b (treated like `let`) of ML identifiers. Throughout the scope of the abstract type declaration the type operators and ML identifiers are both available, but it is only within b that the *representation* of the type operators (as declared in terms of other operators) is available. In an abstract type declaration

`abs`{`rec`}`type` $vtyarg_1$ id_1 `=` ty_1 `and` ...`and` $vtyarg_n$ id_n `=` ty_n `with` b

the sense in which the representation of each id_i is available only within b is as follows: the isomorphism between objects of types ty_i and $vtyarg_i$ id_i is available (only in b) via a pair of implicitly declared polymorphic functions

`abs_`id_i `:` ty_i `->` $vtyarg_i$ id_i
`rep_`id_i `:` $vtyarg_i$ id_i `->` ty_i

which are to be used as coercions between the abstract types and their representations. Thus in the simple case `abstype` a`=` ty `with` x`=`e' `in` e the scope of a is e' and e, the scope of `abs_`a and `rep_`a is e', and the scope of x is e.

As an illustration, consider the definition of the type `rat` of rational numbers, represented by pairs of integers, together with operations `plus` and `times` and the conversion functions

```
inttorat : int->rat
rattoint : rat->int
```

Since `rat` is a nullary type operation, no type variables are involved, and `rat` can be defined by:

```
abstype rat = int# int
   with plus(x,y) = (abs_rat(x1*y2+x2*y1, x2*y2)
                     where x1,x2 = rep_rat x
                       and y1,y2 = rep_rat y    )
   and times(x,y) = (abs_rat(x1*y1, x2*y2)
                     where x1,x2 = rep_rat x
                       and y1,y2 = rep_rat y    )
   and inttorat n = abs_rat(n,1)
   and rattoint x = ((x1/x2)*x2=x1 => x1/x2 | failwith `rattoint`
                     where x1,x2 = rep_rat x    ) ;;
```

Most abstract type declarations are probably used at top-level, so that their scope is the remainder of the top-level program. But for non-top-level declarations, a simple constraint ensures that a value of abstract type cannot exist except during the execution of phrases within the scope of the type declaration. In the expression

abs{rec}type $vtyarg_1$ id_1 = ty_1 and ...and $vtyarg_n$ id_n = ty_n with b in e

the type of e, and the types of any non-local assignments within b and e, must not involve any of the id_i.

Finally, in keeping with the abstract nature of objects of abstract type, the value of a top-level expression of abstract type is printed as a dash, `-` , as functional values are. Users who wish to 'see' such an object should declare a coercion function in the 'with' part of the type declaration, to yield a suitable concrete representation of the abstract objects.

B.6 Primitive ML Identifier Bindings

The primitive ML identifier bindings are described in this Section. Some useful derived functions are in Section B.7. The primitive bindings are of two kinds:

- ordinary bindings;
- dollared bindings (which are preceded by $) having prefix or infix status.

The description of the ML value to which an identifier is bound is omitted if the semantics is clear from the identifier name and type given. For those functions whose application may fail, the failure string is the function identifier.

Predeclared identifiers are not regarded as constants of the language. As with all other ML identifiers, the user is free to rebind them, by `let`, `letref`, etc., but note that in the case of infix or prefix operators rebinding the dollared operator will affect even its non-dollared uses. Predeclared bindings are to be understood as if they had been bound by `let`, rather than by `letref`. In particular, therefore, none of them can be changed by assignment (except, of course, within the scope of a rebinding of the identifier by a `letref`-declaration).

B.6.1 Predeclared ordinary identifiers

```
fst  : * #  ** -> *
snd  : * #  ** -> **                       inl  : * -> * + **
                                           inr  : ** -> * + **
null : * list -> bool                      outl : * + ** -> *
hd   : * list -> *                         outr : * + ** -> **
tl   : * list -> * list                    isl  : * + ** -> bool
```

The functions `hd` and `tl` fail if their argument is an empty list. The functions `outl` and `outr` fail if their arguments are not in the left or right summand, respectively. A function `isr` is not provided because it is just the complement of `isl`.

```
explode : tok -> tok list
implode : tok list -> tok
```

The function `explode` maps a token into the list of its single character tokens in order. The function `implode` maps a list of single character tokens (fails if any token is not of length one) into the token obtained by concatenating these characters. For example:

```
# explode `whosit`;;                                                    1
[`w`; `h`; `o`; `s`; `i`; `t`] : tok list

# implode [`c`;`a`;`t`];;
`cat` : tok

# implode [`cd`;`ab`;`tu`];;
evaluation failed     implode
```

```
int_to_char : int -> char
char_to_int : char -> int
```

The function `int_to_char` on argument i returns the ith character in NUPRL's font. The integer i must be non-negative and less than `256`. For arguments less than `128` the integer-character correspondence is the same as in ASCII. The function `char_to_int` returns integer code of its argument, which must be a one-character token.

```
string_to_toks : string -> tok list
toks_to_string : tok list -> string
```

These functions are similar to `explode` and `implode` except that they work on strings rather than tokens.

```
int_to_tok : int -> tok
tok_to_int : tok -> int
```

These are bound to the obvious type coercion functions, with `tok_to_int` failing if its argument is not a non-negative integer token.

```
ml_curried_infix : tok -> unit
ml_paired_infix  : tok -> unit
```

The functions `ml_curried_infix` and `ml_paired_infix` declare their argument tokens to the ML parser as having *infix status*. Infixed functions can either be curried or take a pair as an argument. For example, after executing

```
ml_paired_infix `plus`;;  let x plus y = x+y;;
```

`1 plus 2` is synonymous with `$plus(1,2)` and after executing

```
ml_curried_infix `plus` ;;  let x plus y = x+y ;;
```

`1 plus 2` is synonymous with `$plus 1 2`.[4]

B.6.2 Predeclared dollared identifiers

The following prefix and infix operators are provided as primitives (where the dollar symbol is omitted from the table; the constants are `$do`, and so on):

```
do          : * -> void
not         : bool -> bool
*, /, +, -  : int #  int -> int
>, <        : int #  int -> bool
 =          : * #  * -> bool
 @          : * list #  * list -> * list
 .          : * #  * list -> * list
```

Clarifying remarks:

- `$do` is equivalent to `\x.()`. `do` e evaluates e for its side effects.
- `/` returns the integer part of the result of a division, for example

  ```
  $/(7,3) = 7/3 = 2
  $/(-7,3) = -7/3 = -2
  ```

  The failure token for division by zero is `` `div` ``.
- `-` is the binary subtraction function. Negation (unary minus) is not available as a predeclared function of ML, only as a prefix operator. Of course, the user can define negation if he or she wishes, e.g. by

  ```
  let minus x = -x
  ```

- Not all dollared infix operators are included above: `$,` is not provided since it would be equivalent (as a function) to the identity on pairs, nor is `&` as it has no corresponding call-by-value function (because e `&` e' evaluates to `false` when e does even if evaluation of e' would fail to terminate), nor is `or` analogously.
- The period symbol `.` is an infixed Lisp cons:

  x`.[`x_1`;...;`x_n`]` = `[`x`;`x_1`;...;`x_n`]`
- `=` is bound to the expected predicate for an equality test at non-function types, but is necessarily rather weak, and may give surprising results, at function types. You can be sure that semantically (i.e. extensionally) different functions are not equal, and that semantically equivalent functions are equal when they originate from the same evaluation of the same textual occurrence of a function-denoting expression; for other cases the equality of functions is unreliable (i.e. implementation dependent). For example, after the top-level declarations

[4] Only ordinary identifiers should be used as infixes; infixing other tokens may have unpredictable effects on the parser.

```
let f x = x+1 and g x = x+2;;
let f' = f and h x = f x and h' x = x+1;;
```

`f=f'` evaluates to `true` and `f=g` evaluates to `false`, but the truth values of `f=h`, `f=h'`, and `h=h'` are unreliable. Furthermore, after declaring

```
let plus = \x.\y.x+y;;
let f = plus 1 and g = plus 1;;
```

the truth value of `f=g` is also unreliable.

- `@` is a predeclared list concatenation operator; the symbol `@` has a special parser status and cannot be redeclared as a curried infix.

B.7 General Purpose and List Processing Functions

This Section describes a selection of commonly useful ML functions applicable to pairs, lists and other ML values. All the functions are definable in ML. Each function is documented by:

1. Its name and type.
2. A brief description.
3. An ML declaration defining the function (note that this is not necessarily the definition used: some of the functions are coded directly in Lisp).

Functions preceded by `$` may be used as infix operators (without the `$`), or in normal prefix form or as arguments to other functions (with the `$`).

The functions usually fail with failure string equal to their name; sometimes, however, the failure string is the one generated by the subfunction that caused the failure.

B.7.1 General purpose functions and combinators

The standard primitive combinators are: `I`, `K`, and `S`.

```
I : * -> *
K : * -> ** -> *
S : (* -> ** -> ***) -> (* -> **) -> * -> ***
```

Description: $\mathtt{I}\ x = x \qquad \mathtt{K}\ x\ y = x \qquad \mathtt{S}\ f\ g\ x = f\ x\ (g\ x)$

Definition:

```
let I x = x

let K x y = x

let S f g x = f x (g x)
```

The derived combinators `KI` (the dual of `K`), `C` (the permutator), `W` (the duplicator), `B` (the compositor) and `CB` (which is declared to be infix) have types:

```
KI : * -> ** -> **
C  : (* -> ** -> ***) -> ** -> * -> ***
W  : (* -> * -> **) -> * -> **
B  : (* -> **) -> (*** -> *) -> *** -> **
CB : (* -> **) -> (** -> ***) -> * -> ***
```

Description:

$\texttt{KI}\ x\ y = y$ $\qquad \texttt{C}\ f\ x\ y = f\ y\ x$ $\qquad \texttt{B}\ f\ g\ x = f(g\ x)$ $\qquad \texttt{W}\ f\ x = f\ x\ x$ $\qquad \texttt{CB}\ f\ g\ x = g(f\ x)$

Definition:

```
let KI = K I

let C f x y = f y x

let W f x = f x x

let B f g x = f(g x)

let (f CB g) x = g(f x)
```

The next group of functions are various useful infixed function-composition operators:

```
$o  : ((* -> **) #  (*** -> *)) -> *** -> **
$#   : ((* -> **) #  (*** -> ****)) -> (* #  ***) -> (** #  ****)
$Co : ((* -> ** -> ***) #  (**** -> *)) -> ** -> **** -> ***
```

Description:

$$(f\ \texttt{o}\ g)\ x = f(g\ x)$$
$$(f\ \texttt{\#}\ \ g)(x,y) = (f\ x,\ g\ y)$$
$$(f\ \texttt{Co}\ g)\ x\ y = \texttt{C}(f\ \texttt{o}\ g)\ x\ y = f\ (g\ y)\ x$$

Definition:

```
ml_paired_infix `o`
let (f o g) x = f(g x)

ml_paired_infix `#`
let (f # g)(x,y) = (f x, g y)

ml_paired_infix `Co`
let (f Co g) x y = f (g y) x
```

The following two functions convert between curried and uncurried versions of a binary function.

```
curry   : ((* #  **) -> ***) -> (* -> ** -> ***)
uncurry : (* -> ** -> ***) -> ((* #  **) -> ***)
```

Description: $\texttt{curry}\ f\ x\ y = f(x,y)$ $\qquad \texttt{uncurry}\ f\ (x,y) = f\ x\ y$

Definition:

```
let curry f x y = f (x,y)

let uncurry f (x,y) = f x y
```

The next function tests for failure.

```
can : (* -> **) -> * -> bool
```

Description: `can` f x evaluates to `true` if the application of f to x succeeds; it evaluates to `false` if the evaluation fails.

Definition:

```
let can f x = (f x; true) ? false
```

The next function iterates a function a fixed number of times.

```
funpow : int -> (* -> *) -> * -> *
```

Description: `funpow` n f x applies f to x n-times: `funpow` n f $= f^n$

Definition:

```
letrec funpow n f x = if n=0 then x else funpow (n-1) f (f x)
```

B.7.2 Miscellaneous list processing functions

The function `length` computes the length of a list.

```
length : * list -> int
```

Description: `length [`x_1`;...;`x_n`]` $= n$

Definition:

```
letrec length = fun [] . 0 | (_.l) . 1+(length l)
```

The function `append` concatenates lists; `@` is an uncurried and infixed version of `append`.

```
append : * list -> * list -> * list
```

Description: `append [`x_1`;...;`x_n`] [`y_1`;...;`y_m`]` $=$ x_1`;...;`x_n`;`y_1`;...;`y_m`]`

Definition:

```
letrec append l1 l2 = if null l1 then l2 else hd l1.append (tl l1) l2
```

The function `el` extracts a specified element from a list. It fails if the integer argument is less than 1 or greater than the length of the list.

```
el : int -> * list -> *
```

Description: `el` i `[`x_1`;...;`x_n`]` $= x_i$

Definition:

```
letrec el i l =
  if null l or i < 1 then failwith `el`
     else if i = 1 then hd l
     else el (i-1) (tl l)
```

The functions `last` and `butlast` compute the last element of a list and all but the last element of a list. Both fail if the argument list is empty.

```
last    : * list -> *
butlast : * list -> * list
```

Description: `last` [x_1;...;x_n] $= x_n$ `butlast` [x_1;...;x_n] $=$ [x_1;...;x_{n-1}]

Definition:

```
letrec last l = last (tl l) ? hd l ? failwith `last`

letrec butlast l =
  if null (tl l) then [] else (hd l).(butlast(tl l)) ? failwith `butlast`
```

The next function makes a list consisting of a value replicated a specified number of times. It fails if the specified number is less than zero.

```
replicate : * -> int -> * list
```

Description: `replicate` x n evaluates to [x;...;x], a list of length n.

Definition:

```
letrec replicate x n =
  if n < 0 then failwith `replicate`
     else if n = 0 then []
     else x . (replicate x (n-1))
```

B.7.3 List mapping and iterating functions

```
map : (* -> **) -> * list -> ** list
```

Description: `map` f l returns the list obtained by applying f to the elements of l in turn.

Definition:

```
letrec map f l = if null l then [] else f(hd l). map f (tl l)
```

The following three functions are versions of 'reduce'.

```
itlist     : (* -> ** -> **) -> * list -> ** -> **
rev_itlist : (* -> ** -> **) -> * list -> ** -> **
end_itlist : (* -> * -> *) -> * list -> *
```

Description: `itlist` f [x_1;x_2;...;x_n] x $= f\ x_1\ (f\ x_2\ (\ \ldots\ (f\ x_n\ x)\ \ldots\ ))$
$= ((f\ x_1) \circ (f\ x_2) \circ\ \ldots\ \circ (f\ x_n))\ x$

`rev_itlist` f [x_1;...;x_{n-1};x_n] x $= f\ x_n\ (f\ x_{n-1}\ (\ \ldots\ (f\ x_1\ x)\ \ldots\ ))$
$= ((f\ x_n) \circ (f\ x_{n-1}) \circ\ \ldots\ \circ (f\ x_1))\ x$

`end_itlist` f [x_1;x_2;...;x_{n-1};x_n] $= f\ x_1\ (f\ x_2\ (\ \ldots\ (f\ x_{n-1}\ x_n)\ \ldots\ ))$
$= ((f\ x_1) \circ (f\ x_2) \circ\ \ldots\ \circ (f\ x_{n-1}))\ x_n$

Definition:

```
letrec itlist f l x =
   if null l then x else f (hd l) (itlist f (tl l) x)

letrec rev_itlist f l x =
   if null l then x else rev_itlist f (tl l) (f (hd l) x)

let end_itlist ff l =
   if null l then failwith `end_itlist`
             else (let last.rest = rev l in  rev_itlist ff rest last)
```

or, equivalently:

```
letrec itlist f     = fun [] . I | (y.l) . \x. f y (itlist f l x)

letrec rev_itlist f = fun [] . I | (y.l) . \x. rev_itlist f l (f y x)
```

B.7.4 List searching functions

The functions described in this section search lists for elements with various properties. Those functions that return elements fail if no such element is found; those that return booleans never fail (`false` is returned if the element is not found).

```
find    : (* -> bool) -> * list -> *
tryfind : (* -> **) -> * list -> **
```

Description: `find` p l returns the first element of l that satisfies the predicate p. `tryfind` f l returns the result of applying f to the first member of l for which the application of f succeeds.

Definition:

```
letrec find p    = fun []    . failwith `find`
                     | (x.l) . if p x then x else find p l

letrec tryfind f = fun []    . failwith `tryfind`
                     | (x.l) . (f x ? tryfind f l)
```

The next two functions are analogous to the quantifiers $\exists$ and $\forall$.

```
exists : (* -> bool) -> * list -> bool
forall : (* -> bool) -> * list -> bool
```

Description: `exists` p l applies p to the elements of l in order until one is found which satisfies p, or until the list is exhausted, returning `true` or `false` accordingly; `forall` is the dual.

Definition:

```
let exists p l = can (find p) l

let forall p l = not(exists ($not o p) l)
```

The next function tests for membership of a list.

```
mem : * -> * list -> bool
```

Description: `mem` x l returns `true` if some element of l is equal to x, otherwise it returns `false`.

Definition:

```
let mem = exists o (curry $=)
```

The following two functions are ML versions of Lisp's `assoc`.

```
assoc     : * -> (* #  **) list -> (* #  **)
rev_assoc : * -> (** #  *) list -> (** #  *)
```

Description: `assoc` x l searches a list l of pairs for one whose first component is equal to x, returning the first pair found as result; similarly, `rev_assoc` y l searches for a pair whose second component is equal to y. For example:

```
# assoc 2 [(1,4);(3,2);(2,5);(2,6)];;                    1
(2, 5) : (int #  int)

# rev_assoc 2 [(1,4);(3,2);(2,5);(2,6)];;
(3, 2) : (int #  int)
```

Definition:

```
let assoc x     = find (\(x',y). x=x')

let rev_assoc y = find (\(x,y'). y=y')
```

B.7.5 List transforming functions

The next function reverses a list:

```
rev : * list -> * list
```

Description: `rev` $[x_1;\ldots;x_n]$ = $[x_n;\ldots;x_1]$

Definition:

```
let rev = rev1 []
   whererec rev1 l = fun [] . l | (x.l') . rev1 (x.l) l'
```

The following two functions filter a list to the sublist of elements satisfying a predicate.

```
filter    : (* -> bool) -> * list -> * list
mapfilter : (* -> **) -> * list -> * list
```

Description: `filter` p l applies p to every element of l, returning a list of those that satisfy p; evaluating `mapfilter` f l applies f to every element of l, returning a list of results for those elements for which application of f succeeds.

Definition:

```
letrec filter p    = fun []    . []
                       | (x.l) . if p x then (x.filter p l) else filter p l

letrec mapfilter f = fun []    . []
                       | (x.l) . let l' = mapfilter f l in (f x).l' ? l'
```

The following three functions break-up lists.

```
remove    : (* -> bool) -> * list -> (* #  * list)
partition : (* -> bool) -> * list -> (* list #  * list)
chop_list : int -> * list -> (* list #  * list)
```

Description: `remove` p l separates from the rest of l the first element that satisfies the predicate p; it fails if no element satisfies the predicate. `partition` p l returns a pair of lists. The first list contains the elements of l which satisfy p. The second list contains all the other elements of l.

$$\texttt{chop\_list}\ i\ \texttt{[}x_1\texttt{;}\ldots\texttt{;}x_n\texttt{]} = \texttt{[}x_1\texttt{;}\ldots\texttt{;}x_i\texttt{],[}x_{i+1}\texttt{;}\ldots\texttt{;}x_n\texttt{]}$$

`chop_list` fails if i is negative or greater than n.

Definition:

```
letrec remove p l =
  if p (hd l) then (hd l, tl l)
  else (I # (\r. ((hd l) . r))) (remove p (tl l))

let partition p l =
  itlist (\a (yes,no). if p a then ((a.yes),no) else (yes,(a.no))) l ([],[])

letrec chop_list i l =
  if i = 0 then ([],l)
  else (let l1,l2 = chop_list (i-1) (tl l) in  hd l . l1 , l2)
  ? failwith `chop_list`
```

The next function flattens a list of lists:

```
flat : * list list -> * list
```

Description: $\texttt{flat [[}l_{11}\texttt{;}\ldots\texttt{;}l_{1m_1}\texttt{]; [}l_{21}\texttt{;}\ldots\texttt{;}l_{2m_2}\texttt{]; }\ldots\texttt{[}l_{n1}\texttt{;}\ldots\texttt{;}l_{nm_n}\texttt{]]}$
$= \texttt{[}l_{11}\texttt{;}\ldots\texttt{;}l_{1m_1}\texttt{; }l_{21}\texttt{;}\ldots\texttt{;}l_{2m_2}\texttt{; }\ldots l_{n1}\texttt{;}\ldots\texttt{;}l_{nm_n}\texttt{]}$

Definition:

```
letrec flat = fun [] . [] | (x.l) . x@(flat l)
```

The next two functions 'zip' and 'unzip' between lists of pairs and pairs of lists.

```
combine : (* list #  ** list) -> (* #  **) list
split   : (* #  **) list -> (* list #  ** list)
```

Description:

$$\texttt{combine [}x_1\texttt{;}\ldots\texttt{;}x_n\texttt{] [}y_1\texttt{;}\ldots\texttt{;}y_n\texttt{]} = \texttt{[(}x_1\texttt{,}y_1\texttt{);}\ldots\texttt{;(}x_n\texttt{,}y_n\texttt{)]}$$
$$\texttt{split [(}x_1\texttt{,}y_1\texttt{);}\ldots\texttt{;(}x_n\texttt{,}y_n\texttt{)]} = \texttt{[}x_1\texttt{;}\ldots\texttt{;}x_n\texttt{],[}y_1\texttt{;}\ldots\texttt{;}y_n\texttt{]}$$

Definition:

```
letrec combine = fun ([],[])          . []
                  |  ((x.lx),(y.ly)) . ((x,y).combine(lx,ly))
                  |  _                . failwith `combine`

letrec split   = fun []        . ([],[])
                  |  ((x,y).l) . let lx,ly = split l in (x.lx,y.ly)
```

B.7.6 Functions for lists representing sets

The following functions behave like the corresponding set-theoretic operations on sets (represented as lists without repetitions).

```
intersect : * list -> * list -> * list
subtract  : * list -> * list -> * list
union     : * list -> * list -> * list
```

Description: `intersect` l_1 $l_2 = l_1 \cap l_2$ `subtract` l_1 $l_2 = l_1 - l_2$ `union` l_1 $l_2 = l_1 \cup l_2$

Definition:

```
let intersect l1 l2 = filter (\x. mem x l2) l1

let subtract l1 l2  = filter (\x. not(mem x l2)) l1

let union l1 l2     = l1 @ subtract l2 l1
```

There are also functions to test if a list is a set, remove duplicates from a list and test two lists for set equality.

```
distinct  : * list -> bool
setify    : * list -> * list
set_equal : * list -> * list -> bool
```

Description: `distinct` l returns `true` if all the elements of l are distinct; otherwise it returns `false`. `setify` l removes repeated elements from l, leaving the last occurrence of each duplicate in the list. `set_equal` l_1 l_2 returns `true` if every element of l_1 appears in l_2 and every element of l_2 appears in l_1; otherwise it returns `false`.

Definition:

```
letrec distinct l =
 (null l) or (not (mem (hd l) (tl l)) & distinct (tl l))

let setify l = itlist (\a s. if mem a s then s else a.s) l []

let set_equal l1 l2 = (subtract l1 l2 = []) & (subtract l2 l1 = [])
```

B.7.7 Miscellaneous string processing functions

The following functions split strings into 'words'; `words2` uses a user supplied separator, while `words` uses space and carriage-return as separators.

```
words2 : string -> string -> string list
words  : string -> string list
```

Description: `words2` `` `$c$` `` `` `$s_1 c s_2 c \ldots c s_n$` `` $=$ `` [`$s_1$`;`$s_2$`; ...;`$s_n$`] ``

`words` `` `$s_1$ $s_2$ ... $s_n$` `` $=$ `` [`$s_1$`;`$s_2$`; ...;`$s_n$`] ``

Definition:

```
let words2 sep string =
 snd (itlist (\ch (chs,tokl).
                if ch = sep then
                   if null chs then [],tokl
                   else [], (implode chs . tokl)
                else (ch.chs), tokl)
            (sep . explode string)
            ([],[]))

let word_separators = [` `;`\L`]
```

```
let words string =
 snd (itlist (\ch (chs,tokl).
                if mem ch word_separators then
                   if null chs then [],tokl
                   else [], (implode chs . tokl)
                else (ch.chs), tokl)
            (` ` . explode string)
            ([],[]))
```

The next three functions (the second of which is an infixed version of the first) are string concatenation operators.

```
concat  : string -> string -> string
$^      : string -> string -> string
concatl : string list -> string
```

Description: `concat` concatenates two strings, `$^` is an infixed version of `concat`, and `concatl` concatenates all the strings in a list of strings.

Definition:

```
let concat s1 s2 = implode(explode s1 @ explode s2)

ml_curried_infix `^`
let s1 ^ s2 = concat s1 s2

let concatl sl = implode(itlist append (map explode sl) [])
```

B.7.8 Failure handling functions

The failure handling functions described here are useful for writing code that fails with a backtrace.

```
set_fail_prefix : string -> (* -> **) -> * -> **
set_fail        : string -> (* -> **) -> * -> **
```

Description: `set_fail_prefix` s f x applies f to x and returns the result of the application if it is successful; if the application fails then the string s is concatenated to the failure string and the resulting string propagated as the new failure string.
`set_fail` s f x applies f to x and returns the result of the application if it is successful; if the application fails then the string s is propagated as the new failure string.

Definition:

```
let set_fail_prefix s f x = f x ?\s' failwith(concatl[s;`--`;s'])

let set_fail s f x        = f x ? failwith s
```

Bibliography

[ACE+00] Stuart Allen, Robert Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. The Nuprl open logical environment. In D. McAllester, editor, *17th Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 170–176. Springer Verlag, 2000.

[BBS+98] Holger Benl, Ulrich Berger, Helmut Schwichtenberg, Monika Seisenberger, and Wolfgang Zuber. Proof theory at work: Program development in the minlog system. In W. Bibel and P. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume II, chapter II.1.2, pages 41–71. Kluwer, 1998.

[BCH+00] Ken Birman, Robert Constable, Mark Hayden, Jason Hickey, Christoph Kreitz, Robbert van Renesse, Ohad Rodeh, and Werner Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 149–160. IEEE Computer Society Press, 2000.

[Ble75] W. W. Bledsoe. A new method for proving certain Presburger formulas. In Patrick H. Winston and Carl Hewitt, editors, *IJCAI-75 – 4th International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, September 1975*. Morgan Kaufmann, 1975.

[CAB+86] Robert L. Constable, Stuart F. Allen, H. Mark Bromley, W. Rance Cleaveland, J. F. Cremer, Robert W. Harper, Douglas J. Howe, Todd B. Knoblock, Nax Paul Mendler, Prakash Panangaden, Jim T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl proof development system*. Prentice Hall, 1986.

[CDE+99a] Manuel Clavel, Francisco Duran, Steven Eker, P. Lincoln, N. Marti-Oliet, Jose Meseguer, and J.F.Quesada. The Maude system. In P.Narendran and M. Rusinowitch, editors, *10th International Conference on Rewriting Techniques and Applications (RTA'99)*, number 1631 in Lecture Notes in Computer Science, pages 240–243. Springer Verlag, 1999.

[CDE+99b] Manuel Clavel, Francisco Duran, Steven Eker, Jose Meseguer, and Mark-Oliver Stehr. Maude as a formal meta-tool. In J. Wing, J. Woodcook, and J. Davies, editors, *FM'99, The World Congress On Formal Methods In The Development Of Computing Systems*, number 1709 in Lecture Notes in Computer Science, pages 1684–1703. Springer Verlag, 1999.

[CH90] Guy Cousineau and Gerard Huet. The CAML primer. Rapports Techniques 122, Institut National de Recherche en Informatique et en Automatique, September 1990.

[Cha82] T. Chan. A decision procedure for checking PL/CV arithmetic inferences. In *Introduction to the PL/CV2 Programming Logic*, volume 135 of *Lecture Notes in Computer Science*, pages 227–264. Springer Verlag, 1982.

[CHP84] Guy Cousineau, Gérard Huet, and Larry Paulson. The ML handbook, 1984.

[Dil96] David Dill. The Murphi verification system. In R. Alur and T. Henzinger, editors, *Computer Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393. Springer Verlag, 1996.

[GM93] Michael Gordon and T. Melham. *Introduction to HOL: a theorem proving environment for higher-order logic.* Cambridge University Press, 1993.

[GMW79] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A mechanized Logic of Computation.* Number 78 in Lecture Notes in Computer Science. Springer Verlag, 1979.

[Hay98] Mark Hayden. *The Ensemble System.* PhD thesis, Cornell University. Department of Computer Science, 1998. Technical Report TR98-1662.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[Jac93a] Paul Jackson. *Nuprl ML Manual.* Cornell University. Department of Computer Science, 1993.

[Jac93b] Paul Jackson. *The Nuprl Proof Development System, Version 4.1: Reference Manual and User's Guide.* Cornell University. Department of Computer Science, 1993.

[Jac94] Paul Jackson. *The Nuprl Proof Development System, Version 4.2: Reference Manual and User's Guide.* Cornell University. Department of Computer Science, 1994.

[KHH98] Christoph Kreitz, Mark Hayden, and Jason Hickey. A proof environment for the development of group communication systems. In C. Kirchner and H. Kirchner, editors, *15^{th} Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 317–331. Springer Verlag, 1998.

[KO99] Christoph Kreitz and Jens Otten. Connection-based theorem proving in classical and non-classical logics. *Journal of Universal Computer Science*, 5(3):88–112, 1999.

[KOSP00] Christoph Kreitz, Jens Otten, Stephan Schmitt, and Brigitte Pientka. Matrix-based constructive theorem proving. In Steffen Hölldobler, editor, *Intellectics and Computational Logic. Papers in honor of Wolfgang Bibel*, number 19 in Applied Logic Series, pages 289–205. Kluwer, 2000.

[Kre97] Christoph Kreitz. Formal reasoning about communication systems I: Embedding ML into type theory. Technical Report TR97-1637, Cornell University. Department of Computer Science, June 1997.

[Kre03] Christoph Kreitz. Building reliable, high-performance networks with the Nuprl proof development system. *Journal of Functional Programming*, 2003.

[KS00] Christoph Kreitz and Stephan Schmitt. A uniform procedure for converting matrix proofs into sequent-style systems. *Journal of Information and Computation*, 162(1–2):226–254, 2000.

[Ler00] Xavier Leroy. *The Objective Caml system release 3.00.* Institut National de Recherche en Informatique et en Automatique, 2000.

[LSBB92] Reinhold Letz, Johann Schumann, Stephan Bayerl, and Wolfgang Bibel. SETHEO: A high-performance theorem prover. *Journal of Automated Reasoning*, 8:183–212, 1992.

[Map] Maple home page. `http://www.maplesoft.com/`.

[Mat] The MathBus Term Structure. `www.nuprl.org/mathbus/mathbusTOC.htm`.

[McC97] W. McCune. Solution of the Robbins problem. *Journal of Automated Reasoning*, 19:263–276, 1997.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[Met] Metaprl home page. `http://metaprl.org`.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory Lecture Notes.* Bibliopolis, Napoli, 1984.

[Nau98] Pavel Naumov. Publishing formal mathematics on the web. Technical Report TR98-1689, Cornell University. Department of Computer Science, 1998.

[Nau99] Pavel Naumov. Importing Isabelle formal mathematics into Nuprl. Technical Report TR99-1734, Cornell University. Department of Computer Science, 1999.

[NO79] Greg Nelson and Derek. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[ORR+96] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer Verlag, 1996.

[Pau87] Lawrence C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF.* Cambridge University Press, 1987.

[Pau90] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[Sho84] R. E. Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, 1984.

[SLKN01] Stephan Schmitt, Lori Lorigo, Christoph Kreitz, and Alexey Nogin. JProver: Integrating connection-based theorem proving into interactive proof assistants. In R. Gore, A. Leitsch, and T. Nipkow, editors, *International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 421–426. Springer Verlag, 2001.

[SVC] The Stanford Validity Checker home page. `http://verify.stanford.edu/SVC/`.

[WAL+90] Pierre Weis, Maria-Virginia Aponte, Alain Laville, Michel Mauny, and Acsander Suarez. The CAML reference manual. Rapports Techniques 121, Institut National de Recherche en Informatique et en Automatique, September 1990.

[Wol88] S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison Wesley, 1988.

[WWM+90] L. Wos, S. Winker, W. McCune, R. Overbeek, E. Lusk, R. Stevens, and R. Butler. Automated reasoning contributes to mathematics and logic. In M. E. Stickel, editor, *10th Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 485–499. Springer Verlag, 1990.