Matita V0.99.3 User Manual (rev. 0.99.3)

COLLABORATORS

	TITLE : Matita V0.99.3 User Manual (rev. 0.99.3)	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Andrea Asperti, Claudio Sacerdoti Coen, Ferruccio Guidi, Enrico Tassi, and Stefano Zacchiroli	May 23, 2016	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.99.3	12/07/2006		

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Features

Matita is an interactive theorem prover (or proof assistant) with the following characteristics:

- It is based on a variant of the Calculus of (Co)Inductive Constructions (CIC). CIC is also the logic of the Coq proof assistant.
- It adopts a procedural proof language, but it has a new set of small step tacticals that improve proof structuring and debugging.
- It has a stand-alone graphical user interface (GUI) inspired by CtCoq/Proof General. The GUI is implemented according to the state of the art. In particular:
  - It is based and fully integrated with Gtk/Gnome.
  - An on-line help can be browsed via the Gnome documentation browser.
  - Mathematical formulae are rendered via Unicode.
- It allows the use of the typical ambiguous mathematical notation by means of a disambiguating parser.

1.2 Matita vs Coq

The system shares a common look&feel with the Coq proof assistant and its graphical user interface. The two systems have variants of the same logic, close proof languages and similar sets of tactics. From the user point of view the main lacking features with respect to Coq are:

- proof extraction;
- an extensible language of tactics;
- automatic implicit arguments;
- several ad-hoc decision procedures;
- several rarely used variants for most of the tactics;
- sections and local variables.

Still from the user point of view, the main differences with respect to Coq are:

- the language of tacticals that allows execution of partial tactical application;
- the unification of the concept of metavariable and existential variable;

- terms with subterms that cannot be inferred are always allowed as arguments of tactics or other commands;
- ambiguous terms are disambiguated by direct interaction with the user;
- theorems and definitions in the library are always accessible without needing to require/include them; right now, only notation needs to be included to become active, but we plan to remove this limitation.

Chapter 2

Installation

Matita is a quite complex piece of software, we thus recommend you to either install al precompiled version or use the LiveCD. If you are running Debian GNU/Linux (or one of its derivatives like Ubuntu), you can install matita typing

```
aptitude install matita matita-standard-library
```

If you are running MacOSX or Windows, give the LiveCD a try before trying to compile Matita from its sources.

2.1 Using the LiveCD

In the following, we will assume you have installed virtualbox for your platform and downloaded the .iso image of the LiveCD

2.1.1 Creating the virtual machine

Click on the New button, a wizard will popup, you should ask to its questions as follows

1. The name should be something like Matita, but can be any meaningful string.
2. The OS type should be Debian
3. The base memory size can be 256 mega bytes, but you may want to increase it if you are going to work with huge formalizations.
4. The boot hard disk should be no hard disk. It may complain that this choice is not common, but it is right, since you will run a LiveCD you do not need to emulate an hard drive.

Now that you are done with the creation of the virtual machine, you need to insert the LiveCD in the virtual cd reader unit.

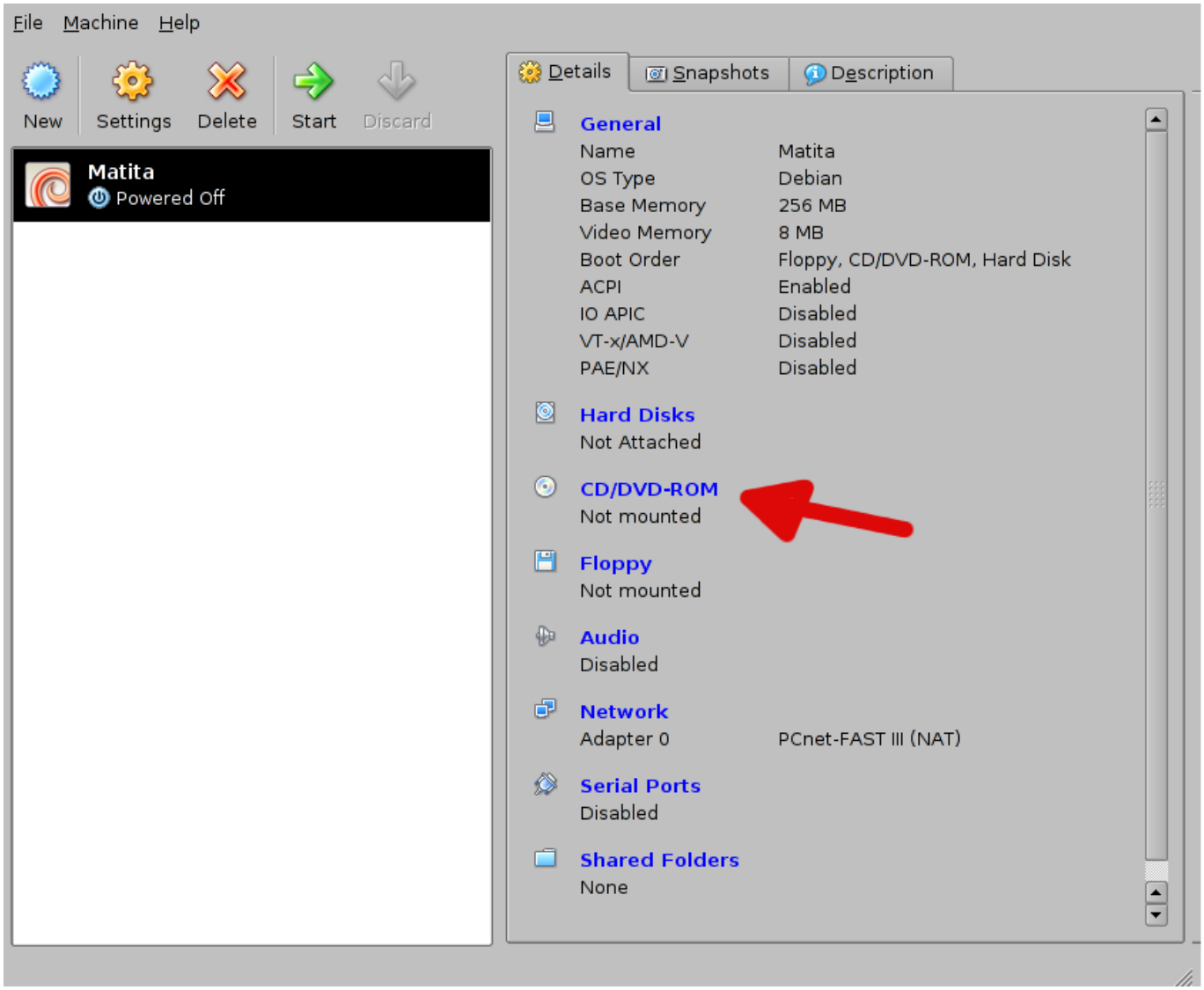

Figure 2.1: The brand new virtual machine

Click on CD/DVD-ROM (that should display something like: Not mouted). Then click on mount CD/DVD drive and select the ISO image option. The combo-box should display no available image, you need to add the ISO image you downloaded from the Matita website clicking on the button near the combo-box. to start the virtual machine.

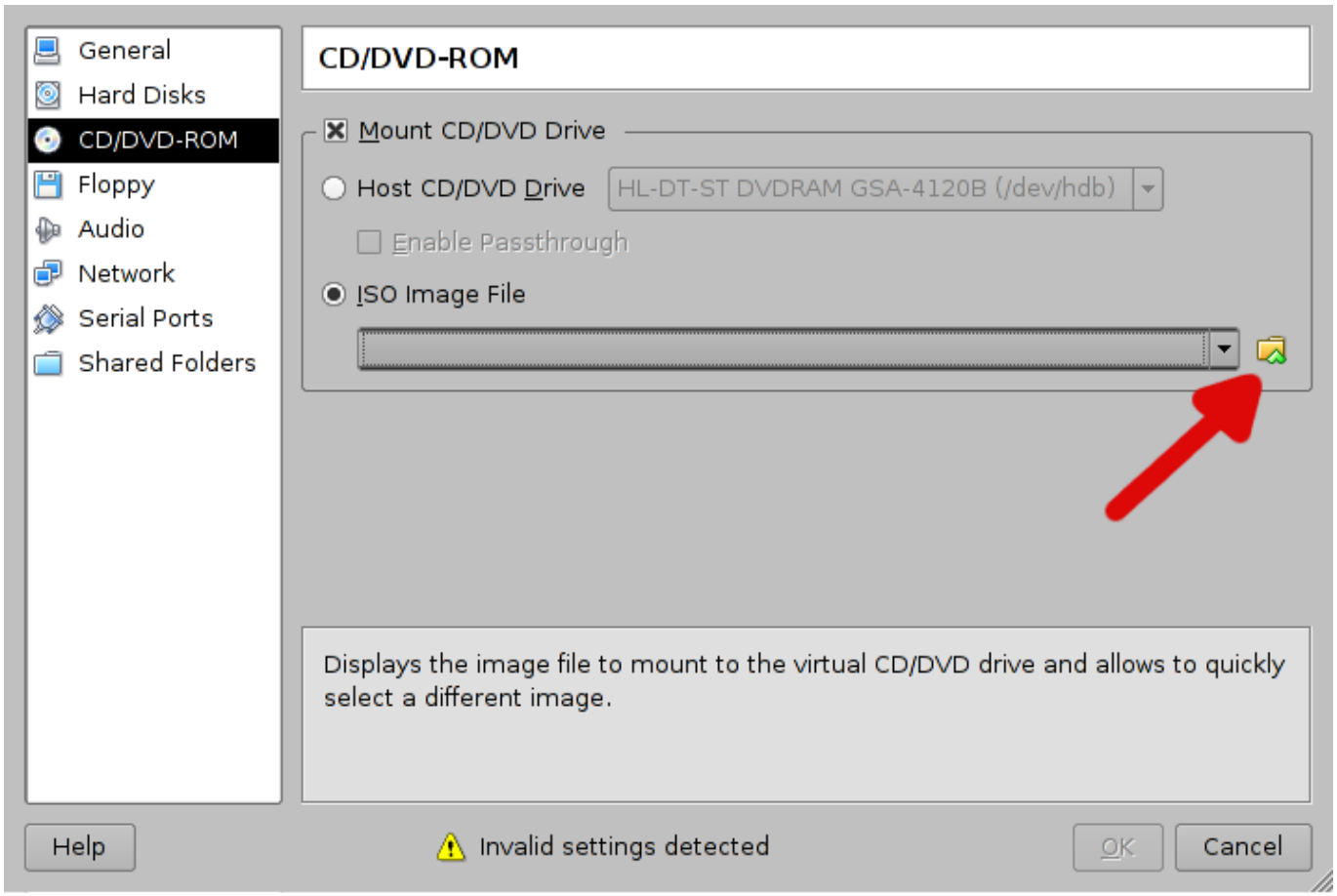

Figure 2.2: Mounting an ISO image

In the newely opened window click the Add button

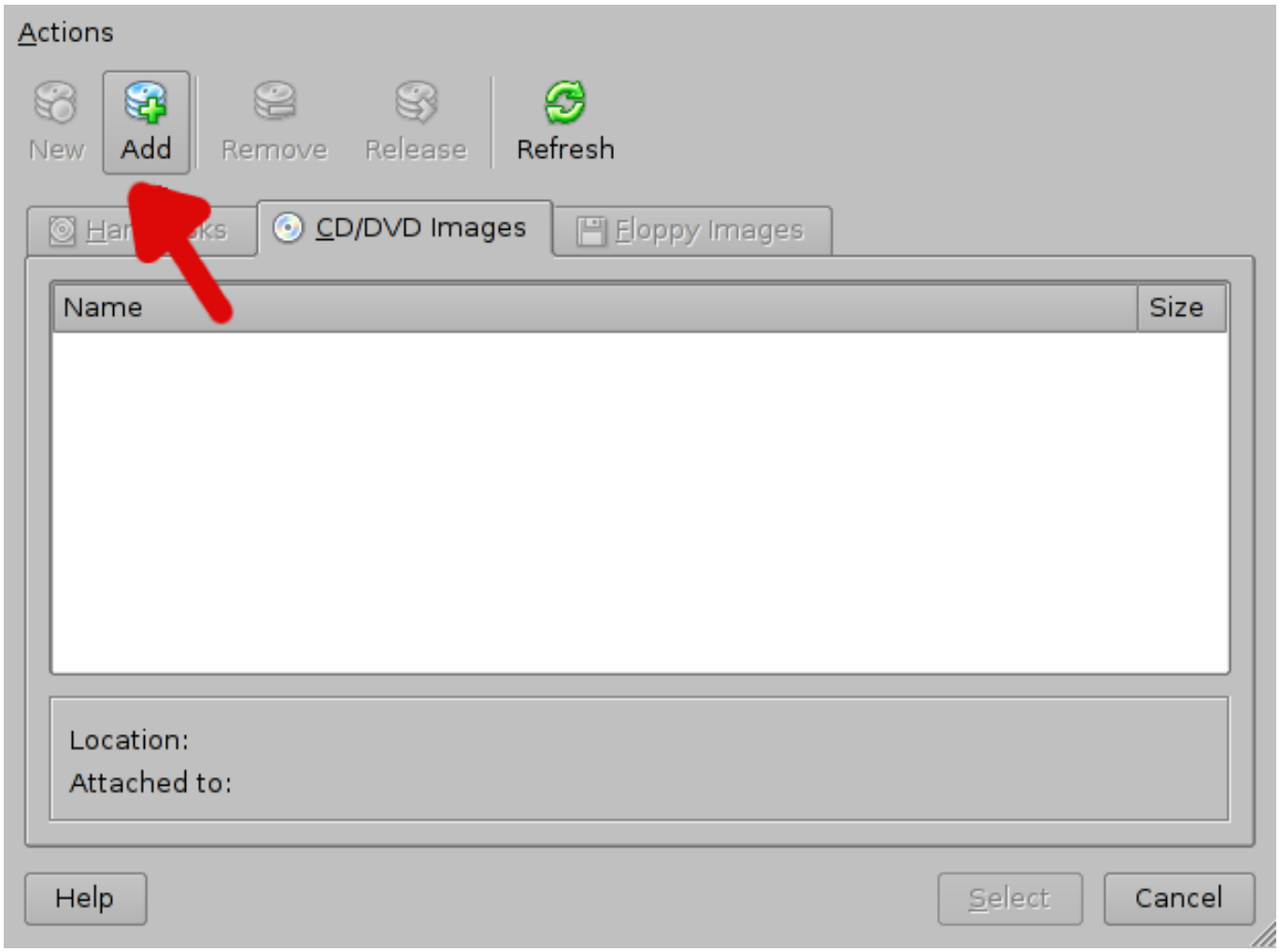

Figure 2.3: Choosing the ISO image

A new windows will pop-up: choose the file you downloaded (usually matita-version.iso) and click open.

Figure 2.4: Choosing the ISO image

Now select the new entry you just added as the CD image you want to insert in the virtual CD drive. You are now ready to start the virtual machine.

2.1.2 Sharing files with the real PC

The virtual machine Matita will run on, has its own file system, that is completely separated from the one of your real PC (thus your files are not available in the emulated environment) and moreover it is a non-presistent file system (thus you data is lost every time you turn off the virtual machine).

Virtualbox allows you to share a real folder (beloging to your real PC) with the emulated computer. Since this folder is persistent, you are encouraged to put your work there, so that it is not lost when the virtual machine is powered off.

The first step to set up a shared folder is to click on the shared folder configuration entry of the virtual machine.

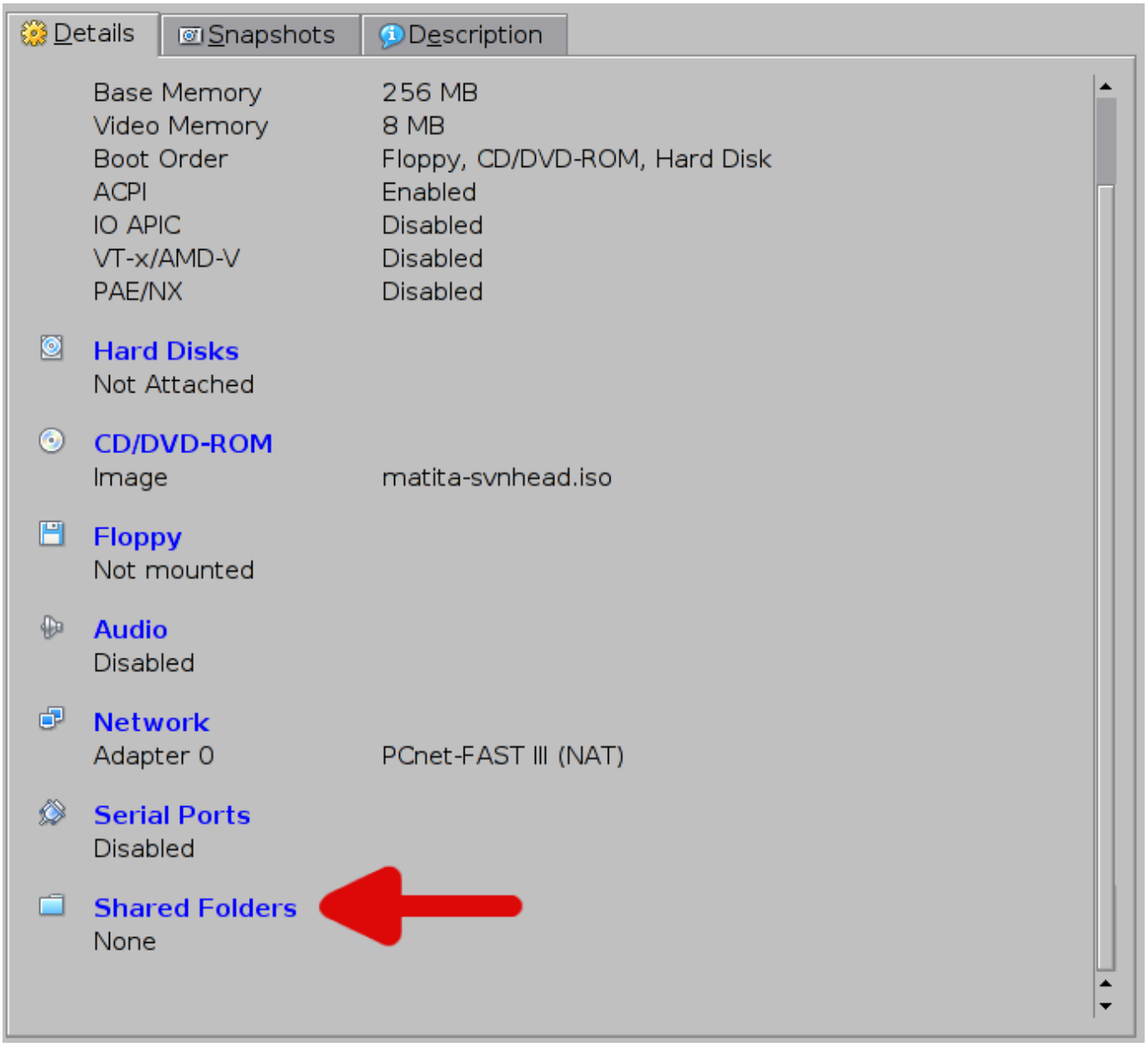

Figure 2.5: Set up a shared folder

Then you shuld add a shared folder clicking on the plus icon on the right

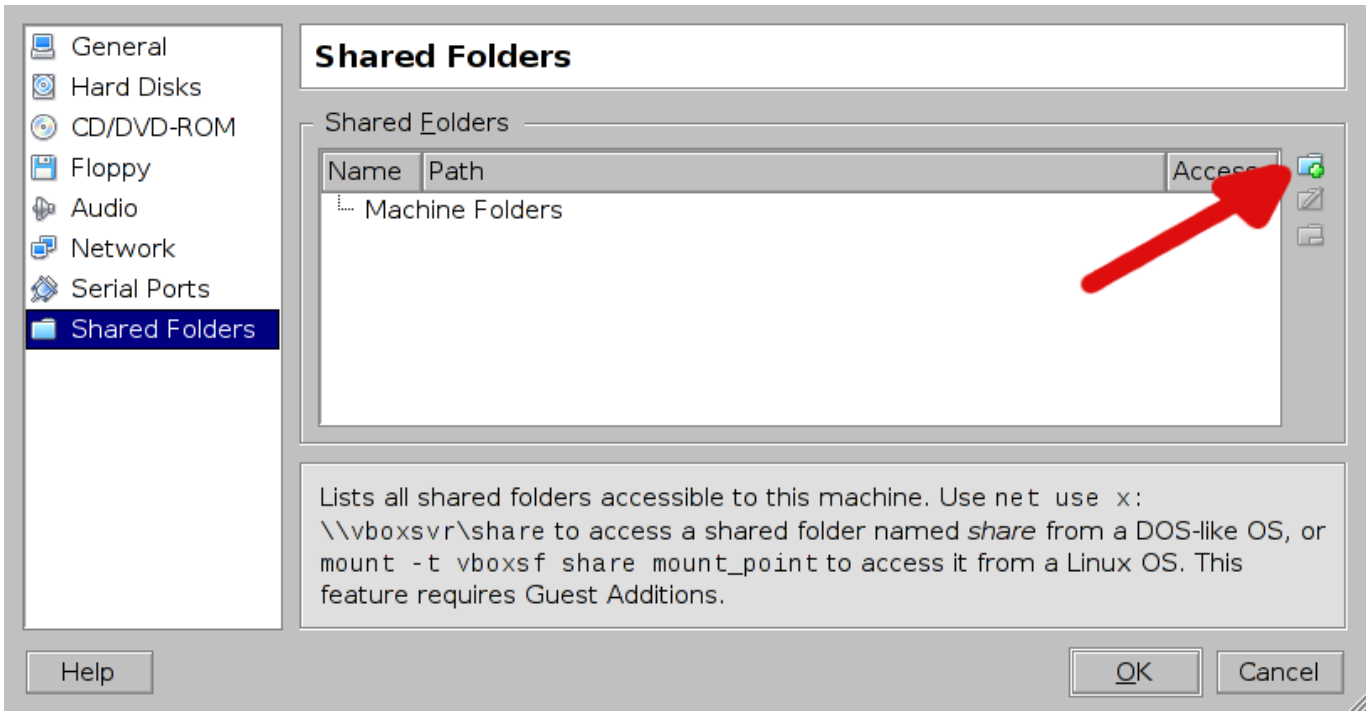

Figure 2.6: Choosing the folder to share

Then you have to specify the real PC folder you want to share and name it. A reasonable folder to share is /home on a standard Unix system, while /Users on MaxOSX. The name you give to the share is important, you should remember it.

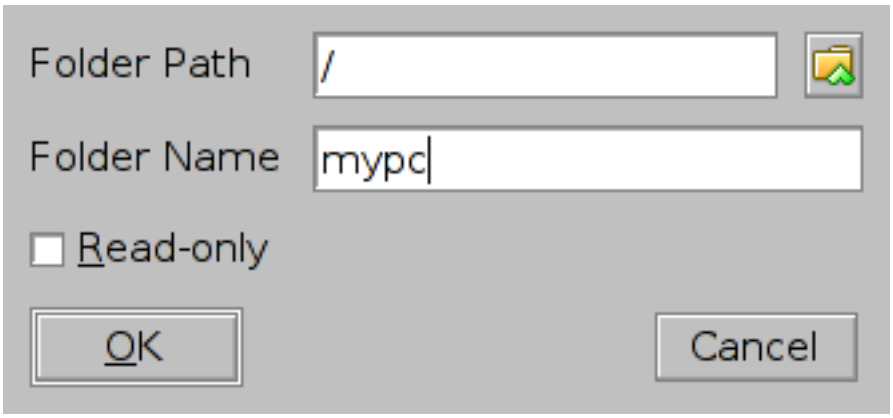

Figure 2.7: Naming the shared folder

Once your virtual machine is up and running, you can mount (that meand have access to) the shared folder by clicking on the Mount VirtualBox share icon, and typing the name of the share.

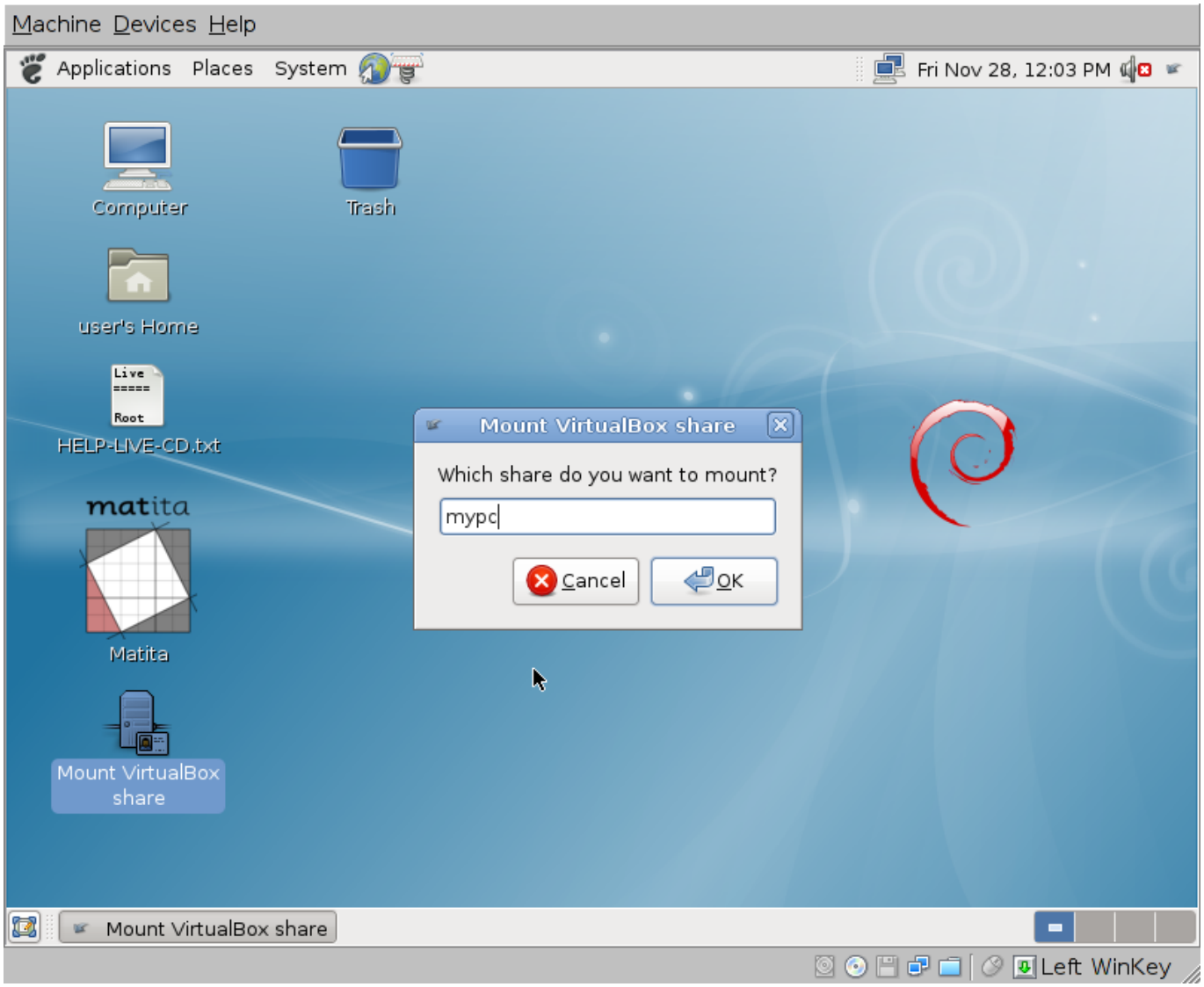

Figure 2.8: Using it from the virtual machine

A window will then pop-up, and its content will be the the content of the real PC folder.

2.2 Installing from sources

Install Matita from the sources is hard, you have been warned!

2.2.1 Getting the source code

You can get the Matita source code in two ways:

1. go to the download page and get the latest released source tarball;
2. get the development sources from our SVN repository. You will need the components/ and matita/ directories from the `trunk/helm/software/` directory, plus the `configure` and `Makefile*` stuff from the same directory.

   In this case you will need to run **autoconf** before proceding with the building instructions below.

2.2.2 Requirements

In order to build Matita from sources you will need some tools and libraries. They are listed below.

Note for Debian (and derivatives) users
If you are running a Debian GNU/Linux distribution, or any of its derivative like Ubuntu, you can use APT to install all the required tools and libraries since they are all part of the Debian archive.
apt-get install ocaml ocaml-findlib libgdome2-ocaml-dev liblablgtk2-ocaml-dev liblablgtkmathview-ocaml-dev liblablgtksourceview-ocaml-dev libsqlite3-ocaml-dev libocamlnet-ocaml-dev libzip-ocaml-dev libhttp-ocaml-dev ocaml-ulex08 libexpat-ocaml-dev libmysql-ocaml-dev camlp5
An official debian package is going to be added to the archive too.

REQUIRED TOOLS AND LIBRARIES

OCaml the Objective Caml compiler, version 3.09 or above

Findlib OCaml package manager, version 1.1.1 or above

OCaml Expat OCaml bindings for the expat library

LablGTK OCaml bindings for the GTK+ library , version 2.6.0 or above

GtkSourceView , LablGtkSourceView extension for the GTK+ text widget (adding the typical features of source code editors) and its OCaml bindings

Ocamlnet collection of OCaml libraries to deal with application-level Internet protocols and conventions

ulex Unicode lexer generator for OCaml

CamlZip OCaml library to access `.gz` files

2.2.3 Compiling and installing

Once you get the source code the installations steps should be quite familiar.

First of all you need to configure the build process executing **`./configure`**. This will check that all needed tools and library are installed and prepare the sources for compilation and installation.

Quite a few (optional) arguments may be passed to the configure command line to change build time parameters. They are listed below, together with their default values:

CONFIGURE COMMAND LINE ARGUMENTS

`--with-runtime-dir=`*`dir`* (*Default:* `/usr/local/matita`) Runtime base directory where all Matita stuff (executables, configuration files, standard library, ...) will be installed

`--enable-debug` (*Default:* disabled) Enable debugging code. Not for the casual user.

Then you will manage the build and install process using make as usual. Below are reported the targets you have to invoke in sequence to build and install:

MAKE TARGETS

`world` builds components needed by Matita and Matita itself (in bytecode or native code depending on the availability of the OCaml native code compiler)

`install` installs Matita related tools, standard library and the needed runtime stuff in the proper places on the filesystem.

Chapter 3

Getting started

If you are already familiar with the Calculus of (Co)Inductive Constructions (CIC) and with interactive theorem provers with procedural proof languages (expecially Coq), getting started with Matita is relatively easy. You just need to learn how to type Unicode symbols, how to browse and search the library and how to author a proof script.

3.1 How to type Unicode symbols

Unicode characters can be typed in several ways:

- Using the "Ctrl+Shift+Unicode code" standard Gnome shortcut. E.g. Ctrl+Shift+3a9 generates "Ω".
- Typing the ligature "\name" where "name" is a standard Unicode or LaTeX name for the character or an ASCII art resembling the shape of the character. Pressing "Alt+L" or Space or Enter just after the last character of the name converts the ligature to the Unicode symbol. E.g. "\Delta" followed by Alt+L generates "Δ", while pressing Alt-L after "=>" generates "⇒"
- Typing a symbol and rotating trough its equivalence class with Alt-L. E.g. pressing Alt-L after an "l" generates a "λ", while pressing Alt-L after an "→" once generates "⇉" and pressing Alt-L again generates "⇒".

The comprehensive list of symbols names or shortcuts and their equivalence classes is available clicking on the "TeX/UTF-8 table" item of the "View" menu.

There is a memory mechanism related to equivalence classes that remembers your last choice, making it the default one. For example, if you use "_" to generate "⎻" (that is the third choice, after "⎽" and "⎼"), the next time you type Alt-L after "_" you immediately get "⎻".

3.2 Browsing and searching

The CIC browser is used to browse and search the library. You can open a new CIC browser selecting "New Cic Browser" from the "View" menu of Matita, or by pressing "F3". The CIC browser is similar to a traditional Web browser.

3.2.1 Browsing the library

To browse the library, type in the location bar the absolute URI of the theorem, definition or library fragment you are interested in. "cic:/" is the root of the library. Contributions developed in Matita are under "cic:/matita"; all the others are part of the library of Coq.

Following the hyperlinks it is possible to navigate in the Web of mathematical notions. Proof are rendered in pseudo-natural language and mathematical notation is used for formulae. For now, mathematical notation must be included in the current script to be activated, but we plan to remove this limitation.

3.2.2 Looking at a proof under development

A proof under development is not yet part of the library. The special URI "about:proof" can be used to browse the proof currently under development, if there is one. The "home" button of the CIC browser sets the location bar to "about:proof".

3.3 Authoring

3.3.1 How to compile a script

Scripts are compiled to base URIs. Base URIs are of the form "cic:/matita/path" and are given once for all for a set of scripts using the "root" file.

A "root" file has to be placed in the root of a script set, for example, consider the following files and directories, and assume you keep files in "list" separated from files in "sort" (for example the former directory may contain functions and proofs about lists, while latter sorting algorithms for lists):

```
list/
  list.ma (* depending just on the standard library *)
  utils/
    swap.ma (* including list.ma *)
sort/
  qsort.ma (* including utils/swap.ma *)
```

To be able to compile properly the contents of "list" a file called root has to be placed in it. The file should be like the following snippet.

```
baseuri=cic:/matita/mydatastructures
```

This file tells Matita that objects generated by "list.ma" have to be placed in "cic:/matita/mydatastructures/list" while objects generated by "swap.ma" have to be placed in "cic:/matita/mydatastructures/utils/swap".

Once you created the root file, you must generate a depend file. Enter the "list" directory (the root of yuor file set) and type "matitadep". Remember to regenerate the depend file every time you alter the dependencies of your files (for example including other scripts). You can now compile you files typing "matitac".

To compile the "sort" directory, create a root file in "sort/" like the following one and then run "matitadep".

```
baseuri=cic:/matita/myalgorithms
include_paths=../list
```

The include_paths field can declare a list of paths separated by space. Please omit any "/" from the end of base URIs or paths.

3.3.2 The authoring interface

TODO

Chapter 4

Syntax

To describe syntax in this manual we use the following conventions:

1. Non terminal symbols are emphasized and have a link to their definition. E.g.: *term*
2. Terminal symbols are in bold. E.g.: **theorem**
3. Optional sequences of elements are put in square brackets. E.g.: [**in** *term*]
4. Alternatives are put in square brakets and they are separated by vertical bars. E.g.: [**<|>**]
5. Repetitions of a sequence of elements are given by putting the sequence in square brackets, that are followed by three dots. The empty sequence is a valid repetition. E.g.: [**and** *term*]...
6. Characters belonging to a set of characters are given by listing the set elements in square brackets. Hyphens are used to specify ranges of characters in the set. E.g.: [**a-zA-Z0-9_-**]

4.1 Terms & co.

4.1.1 Lexical conventions

qstring	::=	**"**⟨⟨*any sequence of characters excluded "*⟩⟩**"**

Table 4.1: qstring

id	::=	⟨⟨*any sequence of letters, underscores or valid XML digits prefixed by a latin letter ([a-zA-Z]) and post-fixed by a possible empty sequence of decorators ([?``])*⟩⟩

Table 4.2: id

nat	::=	⟨⟨*any sequence of valid XML digits*⟩⟩

Table 4.3: nat

char	::=	[**a-zA-Z0-9_-**]

Table 4.4: char

uri-step	::=	*char*[*char*]...

Table 4.5: uri-step

uri	::=	[**cic:/**|**theory:/**]*uri-step*[**/***uri-step*]....*id*[**.***id*]... [**#xpointer(***nat***/***nat*[**/***nat*]...)]

Table 4.6: uri

csymbol	::=	**'***id*

Table 4.7: csymbol

symbol	::=	⟨⟨**None of the above**⟩⟩

Table 4.8: symbol

term	::=	*sterm*	simple or delimited term
	|	*term term*	application
	|	λ*args***.***term*	λ-abstraction
	|	Π*args***.***term*	dependent product meant to define a datatype
	|	$\forall$*args***.***term*	dependent product meant to define a proposition
	|	*term* $\rightarrow$ *term*	non-dependent product (logical implication or function space)
	|	**let** [*id*|(*id***:** *term*)] $\stackrel{def}{=}$ *term* **in** *term*	local definition
	|	**let** [**co**]**rec** *rec_def* [**and** *rec_def*]... **in** *term*	(co)recursive definitions
	|	...	user provided notation
rec_def	::=	*id* [*id*|_|(*id*[**,***id*]... **:***term*)]... [**on** *id*] [**:** *term*] $\stackrel{def}{=}$ *term*]	

Table 4.9: Terms

sterm	::=	(*term*)	
	|	*id*[**\subst[** *id***:=***term* [**;***id***:=***term*]...]]	identifier with optional explicit named substitution
	|	*uri*	a qualified reference
	|	**Prop**	the impredicative sort of propositions
	|	**Set**	the impredicate sort of datatypes
	|	**CProp**	one fixed predicative sort of constructive propositions
	|	**Type**	one predicative sort of datatypes
	|	**?**	implicit argument
	|	**?n** [[[_|*term*]...]]	metavariable
	|	**match** *term* [**in** *id*] [**return** *term*] **with** [*match_branch*[|*match_branch*]...]	case analysis
	|	(*term***:***term*)	cast
	|	...	user provided notation at precedence 90

Table 4.10: Simple terms

args	::=	_[**:** *term*]	ignored argument
	|	(_[**:** *term*])	ignored argument
	|	*id*[**,***id*]... [**:** *term*]	
	|	(*id*[**,***id*]... [**:** *term*])	
args2	::=	*id*	
	|	(*id*[**,***id*]... **:** *term*)	

Table 4.11: Arguments

match_branch	::=	*match_pattern* ⇒ *term*	
match_pattern	::=	*id*	0-ary constructor
	|	(*id id* [*id*]...)	n-ary constructor (binds the n arguments)
	|	*id id* [*id*]...	n-ary constructor (binds the n arguments)
	|	_	any remaining constructor (ignoring its arguments)

Table 4.12: Pattern matching

4.1.2 Terms

4.2 Definitions and declarations

4.2.1 axiom *id*: *term*

```
axiom H: P
```

H is declared as an axiom that states **P**

4.2.2 definition *id*[: *term*] [$\stackrel{def}{=}$ *term*]

```
definition f:  T ≝ t
```

f is defined as **t**; **T** is its type. An error is raised if the type of **t** is not convertible to **T**.

T is inferred from **t** if omitted.

t can be omitted only if **T** is given. In this case Matita enters in interactive mode and **f** must be defined by means of tactics.

Notice that the command is equivalent to **theorem f: T** $\stackrel{def}{=}$ **t**.

4.2.3 discriminator *id*

```
discriminator i
```

Defines a new discrimination (injectivity+conflict) principle à la McBride for the inductive type **i**.

The principle will use John Major's equality if such equality is defined, otherwise it will use Leibniz equality; in the former case, it will be called **i_jmdiscr**, in the latter, **i_discr**. The command will fail if neither equality is available.

Discrimination principles are used by the destruct tactic and are usually automatically generated by Matita during the definition of the corresponding inductive type. This command is thus especially useful when the correct equality was not loaded at the time of that definition.

4.2.4 inverter *id* for *id* (*path*) [*term*]

```
inverter n for i (path) :  s
```

Defines a new induction/inversion principle for the inductive type **i**, called **n**.

(path) must be in the form **(# # # ... #)**, where each **#** can be either **?** or **%**, and the number of symbols is equal to the number of right parameters (indices) of **i**. Parentheses are mandatory. If the j-th symbol is **%**, Matita will generate a principle providing equations for reasoning on the j-th index of **i**. If the symbol is a **?**, no corresponding equation will be provided.

s, which must be a sort, is the target sort of the induction/inversion principle and defaults to **Prop**.

4.2.5 letrec *TODO*

TODO

4.2.6 [inductive|coinductive] *id* [*args2*]... : *term* $\stackrel{def}{=}$ [|] [*id*:*term*] [| *id*:*term*]... [with *id* : *term* $\stackrel{def}{=}$ [|] [*id*:*term*] [| *id*:*term*]...]...

```
inductive i x y z:  S def= k1:T1 | ...  | kn:Tn with i' :  S' def= k1':T1' | ...  | km':Tm'
```

Declares a family of two mutually inductive types **i** and **i'** whose types are **S** and **S'**, which must be convertible to sorts.

The constructors **ki** of type **Ti** and **ki'** of type **Ti'** are also simultaneously declared. The declared types **i** and **i'** may occur in the types of the constructors, but only in strongly positive positions according to the rules of the calculus.

The whole family is parameterized over the arguments **x,y,z**.

If the keyword **coinductive** is used, the declared types are considered mutually coinductive.

Elimination principles for the record are automatically generated by Matita, if allowed by the typing rules of the calculus according to the sort **S**. If generated, they are named **i_ind**, **i_rec** and **i_rect** according to the sort of their induction predicate.

4.2.7 record *id* [*args2*]... : *term* $\stackrel{def}{=}$ {[*id* [:|:>] *term*] [;*id* [:|:>] *term*]...}

```
record id x y z:  S def= { f1:  T1; ...; fn:Tn }
```

Declares a new record family **id** parameterized over **x,y,z**.

S is the type of the record and it must be convertible to a sort.

Each field **fi** is declared by giving its type **Ti**. A record without any field is admitted.

Elimination principles for the record are automatically generated by Matita, if allowed by the typing rules of the calculus according to the sort **S**. If generated, they are named **i_ind**, **i_rec** and **i_rect** according to the sort of their induction predicate.

For each field **fi** a record projection **fi** is also automatically generated if projection is allowed by the typing rules of the calculus according to the sort **S**, the type **T1** and the definability of depending record projections.

If the type of a field is declared with **:>**, the corresponding record projection becomes an implicit coercion. This is just syntactic sugar and it has the same effect of declaring the record projection as a coercion later on.

4.3 Proofs

4.3.1 theorem *id*[: *term*] [$\stackrel{def}{=}$ *term*]

```
theorem f:  P def= p
```

Proves a new theorem **f** whose thesis is **P**.

If **p** is provided, it must be a proof term for **P**. Otherwise an interactive proof is started.

P can be omitted only if the proof is not interactive.

A warning is raised if the name of the theorem cannot be obtained by mangling the name of the constants in its thesis.

Notice that the command is equivalent to **definition f: T** $\stackrel{def}{=}$ **t**.

4.3.2 corollary *id*[: *term*] [$\stackrel{def}{=}$ *term*]

```
corollary f:  T def= t
```

Same as **theorem f: T** $\stackrel{def}{=}$ **t**

4.3.3 lemma *id*[: *term*] [$\stackrel{def}{=}$ *term*]

`lemma f: T` $\stackrel{def}{=}$ **`t`**

Same as **theorem f: T** $\stackrel{def}{=}$ **t**

4.3.4 fact *id*[: *term*] [$\stackrel{def}{=}$ *term*]

`fact f: T` $\stackrel{def}{=}$ **`t`**

Same as **theorem f: T** $\stackrel{def}{=}$ **t**

4.3.5 example *id*[: *term*] [$\stackrel{def}{=}$ *term*]

`example f: T` $\stackrel{def}{=}$ **`t`**

Same as **theorem f: T** $\stackrel{def}{=}$ **t**, but the example is not indexed nor used by automation.

4.4 Tactic arguments

This section documents the syntax of some recurring arguments for tactics.

4.4.1 pattern

pattern	::=	**in** [*id*[**:** *path*]]... [⊢ *path*]]**;**	simple pattern
	|	**in match** *path* [**in** [*id*[**:** *path*]]... [⊢ *path*]]**;**	full pattern

Table 4.13: pattern

path	::=	⟨⟨*any* sterm *without occurrences of* ***Set***, ***Prop***, ***CProp***, ***Type***, id, uri *and user provided notation; however,* **%** *is now an additional production for* sterm⟩⟩

Table 4.14: path

A *path* locates zero or more subterms of a given term by mimicking the term structure up to:

1. Occurrences of the subterms to locate that are represented by **%**.
2. Subterms without any occurrence of subterms to locate that can be represented by **?**.

Warning: the format for a path for a **match** ... **with** expression is restricted to: **match** *path* **with [** _ ⇒ *path* | ... | _ ⇒ *path* **]** Its semantics is the following: the n-th "_ ⇒ *path*" branch is matched against the n-th constructor of the inductive data type. The head λ-abstractions of *path* are matched against the corresponding constructor arguments.

For instance, the path ∀**`_,_:?.(? ? % ?)`**→**`(? ? ? %)`** locates at once the subterms **`x+y`** and **`x*y`** in the term ∀**`x,y:nat.x+y=1`**→**`0=x*y`** (where the notation **`A=B`** hides the term **`(eq T A B)`** for some type **`T`**).

A *simple pattern* extends paths to locate subterms in a whole sequent. In particular, the pattern **{ H: p K: q ⊢ r }** locates at once all the subterms located by the pattern **r** in the conclusion of the sequent and by the patterns **p** and **q** in the hypotheses **H** and **K** of the sequent.

If no list of hypotheses is provided in a simple pattern, no subterm is selected in the hypothesis. If the **⊢ p** part of the pattern is not provided, no subterm will be matched in the conclusion if at least one hypothesis is provided; otherwise the whole conclusion is selected.

Finally, a *full pattern* is interpreted in three steps. In the first step the **match T in** part is ignored and a set S of subterms is located as for the case of simple patterns. In the second step the term **T** is parsed and interpreted in the context of each subterm $s \in S$. In the last term for each $s \in S$ the interpreted term **T** computed in the previous step is looked for. The final set of subterms located by the full pattern is the set of occurrences of the interpreted **T** in the subterms s.

A full pattern can always be replaced by a simple pattern, often at the cost of increased verbosity or decreased readability.

Example: the pattern **{ match x+y in ⊢ ∀_,_:?.(? ? % ?) }** locates only the first occurrence of **x+y** in the sequent **x,y: nat ⊢ ∀z,w:nat. (x+y) * (z+w) = z * (x+y) + w * (x+y)**. The corresponding simple pattern is **{ ⊢ ∀_,_:?.(? ? (? % ?) ?) }**.

Every tactic that acts on subterms of the selected sequents have a pattern argument for uniformity. To automatically generate a simple pattern:

1. Select in the current goal the subterms to pass to the tactic by using the mouse. In order to perform a multiple selection of subterms, hold the Ctrl key while selecting every subterm after the first one.
2. From the contextual menu select "Copy".
3. From the "Edit" or the contextual menu select "Paste as pattern"

4.4.2 reduction-kind

Reduction kinds are normalization functions that transform a term to a convertible but simpler one. Each reduction kind can be used both as a tactic argument and as a stand-alone tactic.

reduction-kind	::=	**normalize** [**nodelta**]	Computes the $\beta\delta\iota\zeta$-normal form. If **nodelta** is specified, δ-expansions are not performed.
	|	**whd** [**nodelta**]	Computes the $\beta\delta\iota\zeta$-weak-head normal form. If **nodelta** is specified, δ-expansions are not performed.

Table 4.15: reduction-kind

4.4.3 auto-params

auto_params	::=	[*nat*] [*simple_auto_param*]... [**by** [*sterm*... | _]]	The natural number, which defaults to 1, gives a bound to the depth of the search tree. The terms listed is the only knowledge base used by automation together with all indexed factual and equational theorems in the included library. If the list of terms is empty, only equational theorems and facts in the library are used. If the list is omitted, it defaults to all indexed theorems in the library. Finally, if the list is _, the automation command becomes a macro that is expanded in a new automation command where _ is replaced with the list of theorems required to prove the sequent.

Table 4.16: auto-params

simple_auto_param	::=	**width=*nat***	The maximal width of the search tree
	|	**size=*nat***	The maximal number of nodes in the proof
	|	**demod**	Simplifies the current sequent using the current set of equations known to automation
	|	**paramod**	Try to close the goal performing unit-equality paramodulation
	|	**fast_paramod**	A bounded version of **paramod** that is granted to terminate quickly

Table 4.17: simple-auto-param

Chapter 5

Extending the syntax

5.1 notation

```
notation usage "presentation" associativity with precedence p for content
```

Synopsis: **notation** [*usage*] "*notation_lhs*" [*associativity*] **with precedence** *nat* **for** *notation_rhs*

Action: Declares a mapping between the presentation AST **presentation** and the content AST **content**. The declared presentation AST fragment **presentation** is at precedence level **p**. The precedence level is used to determine where parentheses must be inserted. In particular, the content AST fragment **content** is actually a pattern, since it contains placeholders (variables) for sub-ASTs. Every placeholder for a term is given an expected precedence level. Parentheses must be inserted around sub-ASTs having a precedence level strictly smaller than the expected one.

If **presentation** describes a binary infix operator and if no precedence level is explicitly given for the operator arguments, an **associativity** declaration can be given to automatically choose the right level for the operands. Otherwise, no **associativity** can be given.

If **direction** is omitted, the mapping is bi-directional and is used both during parsing and pretty-printing of terms. If **direction** is **>**, the mapping is used only during parsing; if it is **<**, it is used only during pretty-printing. Thus it is possible to use simple notations to type for writing the term, and nicer ones for rendering it.

Notation arguments:

usage	::=	**<**	Only for pretty-printing
	|	**>**	Only for parsing

Table 5.1: usage

associativity	::=	**left associative**	Left associative
	|	**right associative**	Right associative
	|	**non associative**	Non associative (default)

Table 5.2: associativity

notation_rhs	::=	*unparsed_ast*	*TODO*
	|	*unparsed_meta*	*TODO*

Table 5.3: notation_rhs

unparsed_ast	::=	@{*enriched_term*}	A content level AST (a term which is parsed, but not disambiguated).
	|	@*id*	**@id** is just an abbreviation for **@{id}**
	|	@*csymbol*	**@'symbol** is just an abbreviation for **@{'symbol}**

Table 5.4: unparsed_ast

enriched_term	::=	⟨⟨A term that may contain occurrences of *unparsed_meta*, even as variable names in binders, and occurrences of *csymbol*⟩⟩	*TODO*

Table 5.5: enriched_term

unparsed_meta	::=	**${***level2_meta***}**	*TODO*
	|	**$***id*	**$id** is just an abbreviation for **${id}**
	|	**$_**	**$_** is just an abbreviation for **${_}**

Table 5.6: unparsed_meta

level2_meta	::=	*unparsed_ast*	*TODO*
	|	**term** *nat id*	*TODO*
	|	**number** *id*	*TODO*
	|	**ident** *id*	*TODO*
	|	**fresh** *id*	*TODO*
	|	**anonymous**	*TODO*
	|	*id*	*TODO*
	|	**fold** [**left**|**right**] *level2_meta* **rec** *id level2_meta*	*TODO*
	|	**default** *level2_meta level2_meta*	*TODO*
	|	**if** *level2_meta* **then** *level2_meta* **else** *level2_meta*	*TODO*
	|	**fail**	*TODO*

Table 5.7: level2_meta

notation_lhs	::=	*layout* [*layout*]...

Table 5.8: notation_lhs

layout	::=	*layout* **\sub** *layout*	Subscript
	|	*layout* **\sup** *layout*	Superscript
	|	*layout* **\below** *layout*	
	|	*layout* **\above** *layout*	
	|	*layout* **\over** *layout*	
	|	*layout* **\atop** *layout*	
	|	*layout* **\frac** *layout*	Fraction
	|	**\infrule** *layout layout layout*	Inference rule (premises, conclusion, rule name)
	|	**\sqrt** *layout*	Square root
	|	**\root** *layout* **\of** *layout*	Generalized root
	|	**hbox** (*layout* [*layout*]...)	Horizontal box
	|	**vbox** (*layout* [*layout*]...)	Vertical box
	|	**hvbox** (*layout* [*layout*]...)	Horizontal and vertical box
	|	**hovbox** (*layout* [*layout*]...)	Horizontal or vertical box
	|	**break**	Breakable space
	|	(*layout* [*layout*]...)	Group
	|	*id*	Placeholder for a term with no explicit precedence
	|	**term** *nat id*	Placeholder for a term with explicit expected precedence
	|	**number** *id*	Placeholder for a natural number
	|	**ident** *id*	Placeholder for an identifier
	|	*literal*	Literal
	|	**opt** *layout*	Optional layout (it can be omitted for parsing)
	|	**list0** *layout* [**sep** *literal*]	List of layouts separated by **sep** (default: any blank)
	|	**list1** *layout* [**sep** *literal*]	Non empty list of layouts separated by **sep** (default: any blank)
	|	**mstyle** *id* value (*layout*)	Style attributes like color #ff0000
	|	**mpadded** *id* value (*layout*)	padding attributes like width -150%
	|	**maction** (*layout*) [(*layout*) ...]	Alternative notations (output only)

Table 5.9: layout

literal	::=	*symbol*	Unicode symbol
	|	*nat*	Natural number (a constant)
	|	**'***id***'**	New keyword for the lexer

Table 5.10: literal

5.2 interpretation

`interpretation "description" 'symbol p1 ... pn = rhs`

Synopsis: **interpretation** *qstring csymbol* [*interpretation_argument*]... **=** *interpretation_rhs*

Action: It declares a bi-directional mapping **{...}** between the content-level AST **'symbol t_1 ... t_n** and the semantic term **rhs[{t_1}/p_1;...;{t_n}/p_n]** (the simultaneous substitution in **rhs** of the interpretation **{...}** of every content-level actual argument t_i for its corresponding formal parameter p_i). The **description** must be a textual description of the meaning associated to **'symbol** by this interpretation, and is used by the user interface of Matita to provide feedback on the interpretation of ambiguous terms.

Interpretation arguments:

interpretation_argument	::=	[η.]... *id*	A formal parameter. If the name of the formal parameter is prefixed by n symbols "η", then the mapping performs (multiple) η-expansions to grant that the semantic actual parameter begins with at least n λ-abstractions.

Table 5.11: interpretation_argument

interpretation_rhs	::=	*uri*	A constant, specified by its URI
	|	*id*	A constant, specified by its name, or a bound variable. If the constant name is ambiguous, the one corresponding to the last implicitly or explicitly specified alias is used.
	|	**?**	An implicit parameter
	|	(*interpretation_rhs* [*interpretation_rhs*]...)	An application

Table 5.12: interpretation_rhs

Chapter 6

Tacticals

6.1 Interactive proofs and definitions

An interactive definition is started by giving a definition command omitting the definiens. An interactive proof is started by using one of the proof commands omitting an explicit proof term.

An interactive proof or definition can and must be terminated by a qed command when no more sequents are left to prove. Between the command that starts the interactive session and the qed command the user must provide a procedural proof script made of tactics structured by means of tacticals.

In the tradition of the LCF system, tacticals can be considered higher order tactics. Their syntax is structured and they are executed atomically. On the contrary, in Matita the syntax of several tacticals is destructured into a sequence of tokens and tactics in such a way that is is possible to stop execution after every single token or tactic. The original semantics is preserved: the execution of the whole sequence yields the result expected by the original LCF-like tactical.

6.2 The proof status

During an interactive proof, the proof status is made of the set of sequents to prove and the partial proof built so far.

The partial proof can be inspected on demand in the CIC browser. It will be shown in pseudo-natural language produced on the fly from the proof term.

The set of sequents to prove is shown in the notebook of the authoring interface, in the top-right corner of the main window of Matita. Each tab shows a different sequent, named with a question mark followed by a number. The current role of the sequent, according to the following description, is also shown in the tab tag.

1. **Selected sequents** (name in boldface, e.g. **?3**). The next tactic will be applied to every selected sequent, producing new selected sequents. Tacticals such as branching ("**[**") or "**focus**" can be used to change the set of selected sequents.
2. **Sibling sequents** (name prefixed by a vertical bar and their position, e.g. $|_3$?2). When the set of selected sequents has more than one element, the user can decide to focus in turn on each of them. The branching tactical ("**[**") selects the first sequent only, marking every previously selected sequent as a sibling sequent. Each sibling sequent is given a different position. The tactical "**2,3:**" can be used to select one or more sibling sequents, different from the one proposed, according to their position. Once the user starts to work on the selected sibling sequents it becomes impossible to select a new set of siblings until the ("|") tactical is used to end work on the current one.
3. **Automatically solved sibling sequents** (name strokethrough, e.g. $|_3$*?2*). Sometimes a tactic can close by side effects a sibling sequent the user has not selected yet. The sequent is left in the automatically solved status in order for the user to explicitly accept (using the "**skip**" tactical) the automatic instantiation in the proof script. This way the correspondence between the number of branches in the proof script and the number of sequents generated in the proof is preserved.

6.3 Tacticals

proof-script	::=	*proof-step* [*proof-step*]...

Table 6.1: proof script

Every proof step can be immediately executed.

proof-step	::=	*LCF-tactical*	The tactical is applied to each selected sequent. Each new sequent becomes a selected sequent.
	|	**.**	The first selected sequent becomes the only one selected. All the remaining previously selected sequents are proposed to the user one at a time when the next "**.**" is used.
	|	**;**	Nothing changes. Use this proof step as a separator in concrete syntax.
	|	**[**	Every selected sequent becomes a sibling sequent that constitute a branch in the proof. Moreover, the first sequent is also selected.
	|	**|**	Stop working on the current branch of the innermost branching proof. The sibling branches become the sibling sequents and the first one is also selected.
	|	*nat*[**,***nat*]. . . **:**	The sibling sequents specified by the user become the next selected sequents.
	|	***:**	Every sibling branch not considered yet in the innermost branching proof becomes a selected sequent.
	|	**skip**	Accept the automatically provided instantiation (not shown to the user) for the currently selected automatically closed sibling sequent.
	|	**]**	Stop analyzing branches for the innermost branching proof. Every sequent opened during the branching proof and not closed yet becomes a selected sequent.
	|	**focus** *nat* [*nat*]. . .	Selects the sequents specified by the user. The selected sequents must be completely closed (no new sequents left open) before doing an "**unfocus** that restores the current set of sibling branches.
	|	**unfocus**	Used to match the innermost "**focus**" tactical when all the sequents selected by it have been closed. Until "**unfocus**" is performed, it is not possible to progress in the rest of the proof.

LCF-tactical	::=	*tactic*	Applies the specified tactic.
	|	*LCF-tactical* **;** *LCF-tactical*	Applies the first tactical first and the second tactical to each sequent opened by the first one.
	|	*LCF-tactical* **[** [*LCF-tactical*] [| *LCF-tactical*]. . . **]**	Applies the first tactical first and each tactical in the list of tacticals to the corresponding sequent opened by the first one. The number of tacticals provided in the list must be equal to the number of sequents opened by the first tactical.
	|	**do** *nat LCF-tactical*	*TODO*
	|	**repeat** *LCF-tactical*	*TODO*
	|	**first [** [*LCF-tactical*] [| *LCF-tactical*]. . . **]**	*TODO*
	|	**try** *LCF-tactical*	*TODO*
	|	**solve [** [*LCF-tactical*] [| *LCF-tactical*]. . . **]**	*TODO*
	|	**(***LCF-tactical***)**	Used for grouping during parsing.

Table 6.3: tactics and LCF tacticals

Chapter 7

Tactics

7.1 Quick reference card

tactic	::=	@ *sterm*
	|	applyS *sterm auto_params*
	|	assumption
	|	**/***auto_params***/.**
	|	cases *term pattern*
	|	change *pattern* **with** *sterm*
	|	**-***id*
	|	**%** [*nat*] [{*sterm*...}]
	|	cut *sterm*
	|	***** [**as** *id*]
	|	destruct [(*id*...)] [**skip** (*id*...)]
	|	elim *sterm pattern*
	|	generalize *pattern*
	|	**#** *id*
	|	**#_**
	|	inversion *sterm*
	|	lapply *sterm*
	|	letin *id* $\stackrel{def}{=}$ *sterm*
	|	**##**
	|	normalize *pattern* [**nodelta**]
	|	[**<|>**] *sterm pattern*
	|	whd *pattern* [**nodelta**]

Table 7.1: tactics

7.2 apply

`@t`

Synopsis: @ *sterm*

Pre-conditions: **t** must have type $\mathbf{T_1} \to \ldots \to \mathbf{T_n} \to \mathbf{G}$ where **G** can be unified with the conclusion of the current sequent.

Action: It closes the current sequent by applying **t** to **n** implicit arguments (that become new sequents).

New sequents to prove: It opens a new sequent for each premise $\mathbf{T_i}$ that is not instantiated by unification. $\mathbf{T_i}$ is the conclusion of the **i**-th new sequent to prove.

7.3 auto

/params/

Synopsis: /*auto_params*/.

Pre-conditions: None, but the tactic may fail finding a proof if every proof is in the search space that is pruned away. Pruning is controlled by the optional **params**. Moreover, only lemmas whose type signature is a subset of the signature of the current sequent are considered. The signature of a sequent is essentially the set of constats appearing in it.

Action: It closes the current sequent by repeated application of rewriting steps (unless **paramodulation** is omitted), hypothesis and lemmas in the library.

New sequents to prove: None

7.4 intro

#H

Synopsis: #*id*

Pre-conditions: The conclusion of the sequent to prove must be an implication or a universal quantification.

Action: It applies the right introduction rule for implication, closing the current sequent.

New sequents to prove: It opens a new sequent to prove adding to the hypothesis the antecedent of the implication and setting the conclusion to the consequent of the implicaiton. The name of the new hypothesis is **H**.

7.5 intro_clear

#_

Synopsis: #_

Pre-conditions: The conclusion of the sequent to prove must be an implication.

Action: It applies the ``a fortiori'' rule for implication, closing the current sequent.

New sequents to prove: It opens a new sequent whose conclusion is the conclusion of the implication of the original sequent.

7.6 macro_input

##

Synopsis: ##

Pre-conditions: None.

Action: This macro expands to the longest possible list of $\#\mathbf{H_i}$ tactics. The names of the introduced hypotheses are automatically generated.

7.7 clear

```
-H
```

Synopsis: *-id*

Pre-conditions: **H** must be an hypothesis of the current sequent to prove.

Action: It hides the hypothesis **H** from the current sequent.

New sequents to prove: None

7.8 constructor

```
%n {args}
```

Synopsis: *%* [*nat*] [*{sterm...}*]

Pre-conditions: The conclusion of the current sequent must be an inductive type or the application of an inductive type with at least **n** constructors.

Action: It applies the **n**-th constructor of the inductive type of the conclusion of the current sequent to the arguments **args**. If **n** is omitted, it defaults to 1.

New sequents to prove: It opens a new sequent for each premise of the constructor that can not be inferred by unification. For more details, see the **apply** tactic.

7.9 decompose

```
* as H
```

Synopsis: * [**as** *id*]

Pre-conditions: The current conclusion must be of the form $\mathbf{T} \rightarrow \mathbf{G}$ where **I** is an inductive type applied to its arguments, if any.

Action: It introduces a new hypothesis **H** of type **T**. Then it proceeds by cases over **H**. Finally, if the name **H** is not specified, it clears the new hypothesis from all contexts.

New sequents to prove: The ones generated by case analysis.

7.10 rewrite

```
> p patt
```

Synopsis: [**<**|**>**] *sterm pattern*

Pre-conditions: **p** must be the proof of an equality, possibly under some hypotheses.

Action: It looks in every term matched by **patt** for all the occurrences of the left hand side of the equality that **p** proves (resp. the right hand side if **<** is used). Every occurence found is replaced with the opposite side of the equality.

New sequents to prove: It opens one new sequent for each hypothesis of the equality proved by **p** that is not closed by unification.

7.11 applyS

`applyS t auto_params`

Synopsis: **applyS** *sterm auto_params*

Pre-conditions: **t** must have type $\mathbf{T_1} \rightarrow \mathbf{...} \rightarrow \mathbf{T_n} \rightarrow \mathbf{G}$.

Action: **applyS** is useful when **apply** fails because the current goal and the conclusion of the applied theorems are extensionally equivalent up to instantiation of metavariables, but cannot be unified. E.g. the goal is **P(n*O+m)** and the theorem to be applied proves ∀**m.P(m+O)**.

It tries to automatically rewrite the current goal using auto paramodulation to make it unifiable with **G**. Then it closes the current sequent by applying **t** to **n** implicit arguments (that become new sequents). The **auto_params** parameters are passed directly to **auto paramodulation**.

New sequents to prove: It opens a new sequent for each premise $\mathbf{T_i}$ that is not instantiated by unification. $\mathbf{T_i}$ is the conclusion of the **i**-th new sequent to prove.

7.12 assumption

`assumption`

Synopsis: **assumption**

Pre-conditions: There must exist an hypothesis whose type can be unified with the conclusion of the current sequent.

Action: It closes the current sequent exploiting an hypothesis.

New sequents to prove: None

7.13 cases

`cases t pattern`

Synopsis: **cases** *term pattern*

Pre-conditions: **t** must inhabit an inductive type

Action: It proceed by cases on **t**. The new generated hypothesis in each branch are named according to **hyps**. The elimintation predicate is restricted by **pattern**. In particular, if some hypothesis is listed in **pattern**, the hypothesis is generalized and cleared before proceeding by cases on **t**. Currently, we only support patterns of the form $\mathbf{H_1}$ **...** $\mathbf{H_n} \vdash$ **%**. This limitation will be lifted in the future.

New sequents to prove: One new sequent for each constructor of the type of **t**. Each sequent has a new hypothesis for each argument of the constructor.

7.14 change

`change patt with t`

Synopsis: **change** *pattern* **with** *sterm*

Pre-conditions: Each subterm matched by the pattern must be convertible with the term **t** disambiguated in the context of the matched subterm.

Action: It replaces the subterms of the current sequent matched by **patt** with the new term **t**. For each subterm matched by the pattern, **t** is disambiguated in the context of the subterm.

New sequents to prove: None.

7.15 cut

```
cut P
```

Synopsis: **cut** *sterm*

Pre-conditions: **P** must be a type.

Action: It closes the current sequent.

New sequents to prove: It opens two new sequents. The first one has conclusion $\mathbf{P} \to \mathbf{G}$ where **G** is the old conclusion. The second sequent has conclusion **P** and hypotheses the hypotheses of the current sequent to prove.

7.16 destruct

```
destruct (H0 ... Hn) skip (K0 ... Km)
```

Synopsis: **destruct** [(*id*...)] [**skip** (*id*...)]

Pre-conditions: Each hypothesis $\mathbf{H_i}$ must be either a Leibniz or a John Major equality where the two sides of the equality are possibly applied constructors of an inductive type.

Action: The tactic recursively compare the two sides of each equality looking for different constructors in corresponding position. If two of them are found, the tactic closes the current sequent by proving the absurdity of **p**. Otherwise it adds a new hypothesis for each leaf of the formula that states the equality of the subformulae in the corresponding positions on the two sides of the equality. If the newly added hypothesis is an equality between a variable and a term, the variable is substituted for the term everywhere in the sequent, except for the hypotheses $\mathbf{K_j}$, and it is then cleared from the list of hypotheses.

New sequents to prove: None.

7.17 elim

```
elim t pattern
```

Synopsis: **elim** *sterm pattern*

Pre-conditions: **t** must inhabit an inductive type.

Action: It proceeds by cases on the values of **t**, according to the most appropriate elimination principle for the current goal. The induction predicate is restricted by **pattern**. In particular, if some hypothesis is listed in **pattern**, the hypothesis is generalized and cleared before eliminating **t**

New sequents to prove: It opens one new sequent for each case.

7.18 generalize

```
generalize patt
```

Synopsis: **generalize** *pattern*

Pre-conditions: All the terms matched by **patt** must be convertible and close in the context of the current sequent.

Action: It closes the current sequent by applying a stronger lemma that is proved using the new generated sequent.

New sequents to prove: It opens a new sequent where the current sequent conclusion **G** is generalized to $\forall$**x.G{x/t}** where **{x/t}** is a notation for the replacement with **x** of all the occurrences of the term **t** matched by **patt**. If **patt** matches no subterm then **t** is defined as the **wanted** part of the pattern.

7.19 inversion

`inversion t`

Synopsis: **inversion** *sterm*

Pre-conditions: The type of the term **t** must be an inductive type or the application of an inductive type.

Action: It proceeds by cases on **t** paying attention to the constraints imposed by the actual "right arguments" of the inductive type.

New sequents to prove: It opens one new sequent to prove for each case in the definition of the type of **t**. With respect to a simple elimination, each new sequent has additional hypotheses that states the equalities of the "right parameters" of the inductive type with terms originally present in the sequent to prove. It uses either Leibniz or John Major equality for the new hypotheses, according to the included files.

7.20 lapply

`lapply t`

Synopsis: **lapply** *sterm*

Pre-conditions: None.

Action: It generalizes the conclusion of the current goal adding as a premise the type of **t**, closing the current goal.

New sequents to prove: The new sequent has conclusion $\mathbf{T} \to \mathbf{G}$ where **T** is the type of **t** and **G** the old conclusion.

7.21 letin

`letin x` $\stackrel{def}{=}$ `t`

Synopsis: **letin** *id* $\stackrel{def}{=}$ *sterm*

Pre-conditions: None.

Action: It adds to the context of the current sequent to prove a new definition $\mathbf{x} \stackrel{def}{=} \mathbf{t}$.

New sequents to prove: None.

7.22 normalize

`normalize patt nodelta`

Synopsis: **normalize** *pattern* [**nodelta**]

Pre-conditions: None.

Action: It replaces all the terms matched by **patt** with their $\beta\delta\iota\zeta$-normal form. If **nodelta** is specified, δ-expansions are not performed.

New sequents to prove: None.

7.23 whd

`whd patt nodelta`

Synopsis: **whd** *pattern* [**nodelta**]

Pre-conditions: None.

Action: It replaces all the terms matched by **patt** with their $\beta\delta\iota\zeta$-weak-head normal form. If **nodelta** is specified, δ-expansions are not performed.

New sequents to prove: None.

Chapter 8

Other commands

8.1 alias

```
alias id "s" = "def"
alias symbol "s" (instance n) = "def"
alias num (instance n) = "def"
```

Synopsis: **alias** [**id** *qstring* **=** *qstring* | **symbol** *qstring* [(**instance** *nat*)] **=** *qstring* | **num** [(**instance** *nat*)] **=** *qstring*]

Action: Used to give an hint to the disambiguating parser. When the parser is faced to the identifier (or symbol) **s** or to any number, it will prefer interpretations that "map **s** (or the number) to **def**". For identifiers, "def" is the URI of the interpretation. E.g.: **cic:/matita/nat/nat.ind#xpointer(1/1/1)** for the first constructor of the first inductive type defined in the block of inductive type(s) **cic:/matita/nat/nat.ind**. For symbols and numbers, "def" is the label used to mark the wanted interpretation.

When a symbol or a number occurs several times in the term to be parsed, it is possible to give an hint only for the instance **n**. When the instance is omitted, the hint is valid for every occurrence.

Hints are automatically inserted in the script by Matita every time the user is interactively asked a question to disambiguate a term. This way the user won't be posed the same question twice when the script will be executed again.

8.2 check

```
check t
```

Synopsis: **check** *sterm*

Action: Opens a CIC browser window that shows **t** together with its type. The command is immediately removed from the script.

8.3 coercion

```
coercion nocomposites c :  ty ≝ u on s :  S to T
```

Synopsis: **coercion** [**nocomposites**] *id* [: *term* $\stackrel{def}{=}$ *term* **on** *id* : *term* **to** *term*]

Action: Declares **c** as an implicit coercion. If only **c** is given, **u** is the constant named by **c**, **ty** its declared type, **s** the name of the last variable abstracted in in **ty**, **S** the type of this last variable and **T** the target of **ty**. The user can specify all these component to have full control on how the coercion is indexed. The type of the body of the coercion **u** must be convertible to the declared one **ty**. Let it be ∀**x1:T1. ...** ∀**x(n-1):T(n-1).Tn**. Then **s** must be one of **x1** ... **xn** (possibly prefixed by _ if the product is non dependent). Let **s** be **xi** in the following. Then **S** must be **Ti** where all bound variables are replaced by **?**, and **T** must be **Tn** where all bound variable are replaced by **?**. For example the following command declares a coercions from vectors of any length to lists of natural numbers.

```
coercion nocomposites v2l : ∀n:nat.∀v:Vect nat n. List nat ≝ l_of_v on _v : Vect nat ? to List nat
```

Every time a term **x** of a type that matches **S** (**Vect nat ?** here) is used with an expected type that matches **T** (**List nat** here), Matita automatically replaces **x** with **(u ? ... ? x ? ... ?)** to avoid a typing error. Note that the position of **x** is determined by **s**.

Implicit coercions are not displayed to the user: **(u ? ... ? x)** is rendered simply as **x**.

When a coercion **u** is declared from source **s** to target **t** and there is already a coercion **u'** of target **s** or source **t**, a composite implicit coercion is automatically computed by Matita unless **nocomposites** is specified.

Note that **Vect nat ?** can be replaced with **Vect ? ?** to index the coercion is a loose way.

8.4 include

```
include "s"
```

Synopsis: **include** *qstring*

Action: Every coercion, notation and interpretation that was active when the file **s** was compiled last time is made active. The same happens for declarations of disambiguation hints (aliases). On the contrary, theorem and definitions declared in a file can be immediately used without including it.

The file **s** is automatically compiled if it is not compiled yet.

If the file **s** was already included, either directly or recursively, the commands does nothing.

8.5 include alias

```
include alias "s"
```

Synopsis: **include alias** *qstring*

Action: Every interpretation declared in the file **s** is re-declared so to make it the preferred choice for disambiguation.

8.6 qed

```
qed
```

Synopsis: **qed**

Action: Saves and indexes the current interactive theorem or definition. In order to do this, the set of sequents still to be proved must be empty.

8.7 qed-

qed-

Synopsis: qed-

Action: Saves the current interactive theorem or definition without indexing. Therefore automation will ignore it. In order to do this, the set of sequents still to be proved must be empty.

8.8 unification hint

unification hint n := v1 : T1,... vi : Ti; h1 ≟ t1, ... hn ≟ tn ⊢ tl ≡ tr.

Synopsis: unification hint *nat* **:=** [*id* [**:** *term*] ,..] **;** [*id* **≟** *term* ,..] ⊢ *term* ≡ *term*

Action: Declares the hint at precedence **n**

The file **hints_declaration.ma** must be included to declare hints with that syntax.

Unification hints are described in the paper "Hints in unification" by Asperti, Ricciotti, Sacerdoti and Tassi.

8.9 universe constraint

TODO

Chapter 9

License