Lambdapi User Manual Documentation

Deducteam

Jul 31, 2023

Contents

Lambdapi is a proof assistant for the $\lambda\Pi$-calculus modulo rewriting. See *What is Lambdapi?* for more details.

Lambdapi files must end with `.lp`. But Lambdapi can also read Dedukti files ending with `.dk` and convert them to Lambdapi files (see *Compatibility with Dedukti*).

Installation instructions - Frequently Asked Questions - Issue tracker

Learn Lambdapi in 15 minutes

Examples of developments made with Lambdapi:

- Some logic definitions
- Library on natural numbers, integers and polymorphic lists
- Example of inductive-recursive type definition
- Example of inductive-inductive type definition
- Test files

Opam repository of Lambdapi libraries

CHAPTER 1

What is Lambdapi?

Lambdapi is an interactive proof system featuring dependent types like in Martin-Lőf's type theory, but allowing to define objects and types using oriented equations, aka rewriting rules, and reason modulo those equations.

This allows to simplify some proofs, and formalize complex mathematical objects that are otherwise impossible or difficult to formalize in more traditional proof systems.

Lambdapi can also take as input Dedukti files, and can thus be used as an interoperability tool.

Lambdapi is a logical framework and does not come with a pre-defined logic. However, it is easy to define a logic by a few symbols and rules. See for instance, the file FOL.lp which defines (polymorphic) first-order logic. There also exist definitions for the logics of HOL, Coq or Agda.

Here are some of the features of Lambdapi:

- Emacs and VSCode plugins (based on LSP)
- support for unicode (UTF-8) and user-defined infix operators
- symbols can be declared commutative, or associative and commutative
- some arguments can be declared as implicit: the system will try to find out their value automatically
- symbol and rule declarations are separated so that one can easily define inductive-recursive types or turn a proved equation into a rewriting rule
- support for interactive resolution of typing goals, and unification goals as well, using tactics
- a rewrite tactic similar to the one of SSReflect in Coq
- the possibility of calling external automated provers
- a command is provided for automatically generating an induction principle for (mutually defined) strictly-positive inductive types
- Lambdapi can call external provers for checking the confluence and termination of user-defined rewriting rules by translating them to the XTC and HRS formats used in the termination and confluence competitions

1.1 Some bibliographic references

- Dedukti: a Logical Framework based on the $\lambda\Pi$-Calculus Modulo Theory, Ali Assaf, Guillaume Burel, Raphaël Cauderlier , David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Draft, 2016.
- Typechecking in the $\lambda\Pi$-Calculus Modulo: Theory and Practice, Ronan Saillard. PhD thesis, 2015.
- Definitions by rewriting in the Calculus of Constructions, Frédéric Blanqui, in Mathematical Structures in Computer Science 15(1):37-92.
- The New Rewriting Engine of Dedukti, Gabriel Hondet and Frédéric Blanqui, 2020.
- Hints in Unification, Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Cohen and Enrico Tassi.
- More papers can be found here.

CHAPTER 2

Getting started

This guide is intended for anyone wishing to give Lambdapi a try, but also for everyone who wants to get started with the development of a new library in the system. The first thing to know is that Lambdapi deliberately enforces a strict discipline on file and module management. As a consequence, it is not in general possible to run `lambdapi` on a source file directly. In particular, actions must be taken so that the system knows where, in the global module hierarchy, a given module (or source file) should be placed. This may seem like a strong restriction but this will allow us to develop simple, well-integrated and powerful tooling in the future (including a proper build system and a package manager). Note however that we provide tools to hide as much of the restriction to the user as possible.

2.1 Creating a new package

The very first thing to do to start using Lambdapi, assuming it has already been installed on your system, is to create a new **Lambdapi package**. Every source file must be part of some package in Lambdapi, and this package will determine the name space under which the objects of the package will be made accessible to other objects of the package, but also to other libraries if the package is ever installed. It is hence important to choose an appropriate name for the packages you create. In particular, this name should uniquely identify the package as it will never be possible to install two distinct packages with the same name.

To create your first package, simply run `lambdapi init my_package`. The effect of this command is to create a new directory `my_package` containing a `Makefile` and a Lambdapi package file `lambdapi.pkg`. After entering the new directory, you can run the `make` command to build the object files of the package. And you can then start working on your project by modifying and creating new source files and running `make` again to type-check and run the commands they contain (don't forget to add in the Makefile the files you want to check). Note that dependencies are handled automatically.

When creating a package, it is also possible to specify a module path under which the package should be placed: `lambdapi init contrib.libs.my_package`. In that case, the name of your package (and of the created folder) is still `my_package`, but every object defined in the package will be accessed with a module path prefixed with `contrib.libs.my_package` instead of `my_package` only. This allows for a more fine-grained management of name spaces.

TL;DR

```
lambdapi init my_package # Create a new package "my_package".
cd my_package            # Move to the created directory.
make                     # Build the example project.
```

2.2 Subdirectories and module hierarchy

Every file with the `.lp` extension under a package's directory can be part of the package. In particular, files can be classified into several directories and subdirectories. However, it is important to note that the directory structure is reflected into the way objects are qualified. As an example, the objects defined in file `dir1/dir2/file.lp` will be accessible in any other module with the qualification `my_packake.dir1.dir2.file` (or `contrib.libs.my_package.dir1.dir2.file` if the package is placed under the more complex module path discussed in the previous section). Note that this implies that the same qualification will be used in other modules of the same package, but also (after installation) in the modules of another packages that depend on your package.

2.3 Experimenting with the system

While you keep reading the following sections of this manual, we encourage you to experiment with the system. You can do that by creating new files in your test package, and using `make` to type-check them after having added those files in the Makefile. Note that by default the system will not give you much feedback. However, once you are editing files in a package (i.e., when there is a `lambdapi.pkg` file available in a parent directory of the files you consider), you can very well call Lambdapi directly (i.e., without relying on the `Makefile`).

Note: do not hesitate to report any problem you may have using our issue tracker.

2.4 Installing a package

Packages must be installed following specific conventions, and hence this should not be attempted directly. Instead, we provide a `lambdapi install` command that takes care of installing the source and object files it is given as argument in the right place, according to the package file. The `Makefile` that is generated at package creation time comes with an `install` target, so installing your package is as simple as running `make install`. Of course, you can also uninstall your package using the command `make uninstall`.

CHAPTER 3

Command line interface

The main Lambdapi executable is called `lambdapi`, and it can be invoked using `lambdapi COMMAND ...`. To see the list of the supported commands, simply run `lambdapi help` or `lambdapi --help`. To get the documentation of a specific command run `lambdapi COMMAND --help`. It will contain the list of options that are supported by the command.

The available commands are:

- `check`: check the correctness of input files.
- `decision-tree`: output the decision tree of a symbol as a Dot graph (see *Decision trees*)
- `export`: translate the input file to other formats.
- `help`: display the main help message.
- `index`: create an index of symbols and rules of input files.
- `init`: create a new Lambdapi package (see *Getting started*).
- `install`: install the specified files according to package configuration.
- `lsp`: run the Lambdapi LSP server.
- `parse`: parse the input files.
- `search`: runs a search query against the index.
- `uninstall`: uninstalls the specified package.
- `version`: give the current version of Lambdapi.
- `websearch`: starts a webserver to search the library.

The commands `parse`, `export` and `index` can trigger the compilation of dependencies if the required object files (`.lpo` extension) are not present.

Input files:

The commands `check`, `parse`, `export` and `index` expect input files with either the `.lp` extension or the `.dk` extension. The appropriate parser is selected automatically. The `export` command accept only one file as argument.

If a command takes several files as argument, the files are handled independently in the order they are given. The program immediately stops on the first failure, without going to the next file (if any).

index:

The `index` command generates the file `~/.LPSearch.db`. This file contains an indexation of all the symbols and rules occurring in the dk/lp files given in argument. By default, the file `~/.LPSearch.db` is erased first. To append new symbols and rules, use the option `--add`. It is also possile to normalize terms wrt some rules before indexation by using `--rules` options.

search:

The command `search` takes as argument a query and runs it against the index file `~/.LPSearch.db`. See *Query Language* for the specification of the query language.

Common flags:

The commands `check`, `decision-tree`, `export`, `parse`, `lsp` all support the following command line arguments and flags.

- `--debug=<FLAGS>` enables the debugging modes specified by every character of `FLAGS`. Details on available character flags are obtained using `--help`.
- `--lib-root=<DIR>` sets the library root, that is, the folder corresponding to the entry point of the Lambdapi package system. This is the folder under which every package is installed, and a default value is only known if the program has been installed. In development mode, `--lib-root lib` must be given (assuming Lambdapi is run at the root of the repository).
- `--map-dir=<MOD>:<DIR>` maps an arbitrary directory `DIR` under a module path `MOD` (relative to the root directory). This option is mainly useful during the development of a package (before it has been installed). However it can also be accessed using a package configuration file (`lambdapi.pkg`) at the root of the library's source tree. More information on that is given in the section about the module system.
- `--no-sr-check` disables subject reduction checking.
- `--timeout=<NUM>` gives up type-checking after the given number of seconds. Note that the timeout is reset between each file, and that the parameter of the command is expected to be a natural number.
- `-v <NUM>`, `--verbose=<NUM>` sets the verbosity level to the given natural number (the default value is 1). A value of 0 should not print anything, and the higher values print more and more information.

3.1 check

- `-c`, `--gen-obj` instructs Lambdapi to generate object files for every checked module (including dependencies). Object files have the extension `.lpo` and they are automatically read back when necessary if they exist and are up to date (they are regenerated otherwise).
- `--too-long=<FLOAT>` gives a warning for each interpreted source file command taking more than the given number of seconds to be checked. The parameter `FLOAT` is expected to be a floating point number.

3.2 export

- `-o <FMT>`, `--output=<FMT>` instructs Lambdapi to translate the files given in argument according to `<FMT>`:
  - `lp`: Lambdapi format
  - `dk`: Dedukti format

- `hrs`: HRS format of the confluence competition
- `xtc`: XTC format of the termination competition
- `raw_coq`: Coq format
- `stt_coq`: Coq format assuming that the input file is in an encoding of simple type theory

WARNING: The options `raw_coq` and `stt_coq` are still experimental.

With the options `raw_coq` and `stt_coq`, rules are ignored. The encoding of simple type theory can however be defined in Coq using STTfa.v.

For the format `stt_coq`, several other options are available:

- `--encoding <LP_FILE>` (mandatory option) where `<LP_FILE>` contains the following sequence of builtin declarations:

```
builtin "Set"  ...; // : TYPE
builtin "prop"  ...; // : Set
builtin "arr"  ...; // : Set → Set → Set
builtin "El"  ...; // : Set → TYPE
builtin "Prf"  ...; // : El prop → TYPE
builtin "eq"  ...; // : Π [a : Set], El a → El a → El prop
builtin "not"  ...; // : El prop → El prop
builtin "imp"  ...; // : El prop → El prop → El prop
builtin "and"  ...; // : El prop → El prop → El prop
builtin "or"  ...; // : El prop → El prop → El prop
builtin "all"  ...; // : Π [a : Set], (El a → El prop) → El prop
builtin "ex"  ...; // : Π [a : Set], (El a → El prop) → El prop
```

It tells Lambdapi which symbols of the input files are used for the encoding. Example: encoding.lp. The first argument `a` of the symbols corresponding to the builtins `"eq"`, `"all"` and `"ex"` need not be declared as implicit.

- `--no-implicits` instructs Lambdapi that the symbols of the encoding have no implicit arguments.
- `--renaming <LP_FILE>` where `<LP_FILE>` contains a sequence of builtin declarations of the form

```
builtin "coq_expr"  lp_id;
```

It instructs Lambdapi to replace any occurrence of the unqualified identifier `lp_id` by `coq_expr`, which can be any Coq expression. Example: renaming.lp.

- `--requiring <COQ_FILE>` to add `Require Import <COQ_FILE>` at the beginning of the output. `<COQ_FILE>` usually needs to contain at least the following definitions:

```
Definition arr (A:Type) (B:Type) := A -> B.
Definition imp (P Q: Prop) := P -> Q.
Definition all (A:Type) (P:A->Prop) := forall x:A, P x.
```

if the symbols corresponding to the builtins `"arr"`, `"imp"` and `"all"` occurs partially applied in the input file. Example: coq.v.

- `--erasing <LP_FILE>` where `<LP_FILE>` contains a sequence of builtin declarations like for the option `--renaming` except that, this time, `lp_id` can be a qualified identifier. It has the same effect as the option `--renaming` plus it removes any declaration of the renamed symbols. `coq_expr` therefore needs to be defined in Coq standard library or in the Coq file specified with the option `-requiring`. It is not necessary to have entries for the symbols corresponding to the builtins `"El"` and `"Prf"` declared with the option `--encoding` since they are erased automatically. Example: erasing.lp.
- `--use-notations` instructs Lambdapi to use the usual Coq notations for the symbols corresponding to the builtins `"eq"`, `"not"`, `"and"` and `"or"`.

Examples of libraries exported to Coq:

- In the Lambdapi sources, see how to export the Holide Dedukti library obtained from OpenTheory in README.md.
- See in hol2dk how to export the Lambdapi library obtained from HOL-Light.

3.3 index

- `--add` tells lambdapi to not erase `~/.LPSearch.db` before adding new symbols and rules.
- `--rules <LPSearch.lp>` tells lambdapi to normalize terms using the rules given in the file `<LPSearch.lp>` before indexing. Several files can be specified by using several `--rules` options. In these files, symbols must be fully qualified but no `require` command is needed. Moreover, the rules do not need to preserve typing. On the other hand, right hand-side of rules must contain implicit arguments.

  For instance, to index the Matita library, you can use the following rules:

```
rule cic.Term _ $x  $x;
rule cic.lift _ _ $x  $x;
```

3.4 websearch

- `--port=<N>` specifies the port number to use (default is 8080).

3.5 lsp

- `--standard-lsp` restricts to standard LSP protocol (no extension).
- `--log-file=<FILE>` sets the log file for the LSP server. If not given, the file `/tmp/lambdapi_lsp_log.txt` is used.

3.6 (un)install

- `--dry-run` prints the system commands that should be called instead of running them.

3.7 decision-tree

- `--ghost` print the decision tree of a ghost symbol. Ghost symbols are symbols used internally that cannot be used in the concrete syntax.

3.8 confluence

- `--confluence=<CMD>` checks the confluence of the rewriting system by calling an external prover with the command `CMD`. The given command receives HRS formatted text on its standard input, and is expected to output on the first line of its standard output either `YES`, `NO` or `MAYBE`. As an example, `echo MAYBE` is the simplest possible (valid) confluence checker that can be used.

For now, only the CSI^ho confluence checker has been tested with Lambdapi. It can be called using the flag `--confluence "path/to/csiho.sh --ext trs --stdin"`.

To inspect the `.trs` file generated by Lambdapi, one may use the following dummy command: `--confluence "cat > output.trs; echo MAYBE"`.

3.9 termination

- `--termination=<CMD>` checks the termination of the rewriting system by calling an external prover with the command `CMD`. The given command receives XTC formatted text on its standard input, and is expected to output on the first line of its standard output either `YES`, `NO` or `MAYBE`. `echo MAYBE` is the simplest (valid) command for checking termination.

To the best of our knowledge, the only termination checker that is compatible with all the features of Lambdapi is SizeChangeTool. It can be called using the flag `--termination "path/to/sct.native --no-color --stdin=xml"`

If no type-level rewriting is used Wanda can also be used. However, it does not directly accept input on its standard input, so it is tricky to have Lambdapi call it directly. Alternatively, one can first generate a `.xml` file as described below.

To inspect the `.xml` file generated by Lambdapi, one may use the following dummy command:`--termination "cat > output.xml; echo MAYBE"`.

CHAPTER 4

User interfaces

Lambdapi provides a language server for (an extension of) the LSP protocol, which is supported by most editors. See below for setting up common editors.

The server is run using the command `lambdapi lsp`. The flag `--standard-lsp` can be used to enforce strict LSP protocol and, thus, currently, deactivate goal display.

4.1 Emacs

MELPA package

Lambdapi source code can be edited with the emacs editor with the major mode `lambdapi-mode`. It requires Emacs 26.1 or higher and provides:

- Syntax highlighting for Lambdapi (`*.lp` files) and Dedukti (`*.dk` files)
- Auto indentation for Lambdapi
- Easier unicode input
- Completion for Lambdapi
- Typing of symbol at point (in minibuffer)
- Type checking declarations

4.1.1 Short-cuts

- `C-c C-c`: jump to cursor position
- `C-c C-n`: next tactic or command
- `C-c C-p`: previous tactic or command
- `C-c C-f`: next proof
- `C-c C-b`: previous proof

- `C-c C-e`: toggle electric mode
- `M-;` : (un)comment region
- `C-x RET C-\`: enter unicode characters in minibuffer using LaTeX
- `M-.` : goto definition
- `M-x customize-group lambdapi`: customize window layout
- `C-c C-r`: refresh window layout
- `C-c C-k`: shutdown LSP server
- `C-c C-r`: reconnect LSP server
- Click on a symbol to discover its type in the bottom line

As always with emacs, if you were to be dissatisfied with these keybindings, you can change them easily!

4.1.2 Installation

The `lambdapi-mode` can be installed from MELPA using any package manager (`package.el`, straight, ...). Provided that Emacs is properly configured (see https://melpa.org/#/getting-started to configure Emacs to use MELPA), the mode can be installed with `M-x package-install RET lambdapi-mode`.

4.1.3 Usage

Make sure you have a lambdapi.pkg file in your folder when editing a Lambdapi file in it.

Commenting regions

Lambdapi handles single-line and multi-line comments with `//` and `/* ... */` respectively. To comment a region in Emacs, select it and use `M-;`.

Entering unicode

Company: if company and company-math are installed, LaTeX commands are autocompleted with the latter tool. Any (or at least a lot of) LaTeX symbol can be entered via its LaTeX command: start typing it, and an autocompletion tooltip should suggest the unicode symbol.

This method is the more complete and easier to use, but depends on `company`.

LambdaPi input method: if company or company-math is not installed, LaTeX characters can be entered via the LambdaPi input method. Greek characters can be accessed using backquoted letters (as done in cdlatex), or with the LaTeX command: α can be accessed with `` `a `` or `\alpha`, β with `` `b `` or `\beta`, and similarly for other Greek letters.

NOTE on the interaction between the input method and company: the dropdown window of company-math will not appear as long as the current word is a candidate for a completion of the input method. To favour company over the input method, the input method can be disabled setting the variable `lambdapi-unicode-prefer-company` to a non-nil value in `~/.emacs` or `~/.emacs.d/init.el`:

```
(setq lambdapi-unicode-prefer-company 1)
```

abbrev mode: the `abbrev` mode is an emacs minor mode allowing the user to define abbreviations. For instance, one may define "btw" to be an abbreviation of "by the way" with, `add-global-abbrev`. Doing so will cause the sequence "btw" to be automatically expanded when the user hits `SPC` or `TAB`. The expansion can be inhibited by hitting `C-q` before `SPC`.

The function `lambdapi-local-abbrev` can be called when the cursor is at the end of a word to define the word as an abbreviation. When called, the user can input the expanded form in the minibuffer. Additionnally, the abbreviation is added as a directory local variable, so it will be available the next time a file of the project is opened. The function `lambdapi-local-abbrev` is bound to `C-c a`.

To enter unicode characters in the minibuffer using LaTeX, the TeX input method can be used, for this, once in the minibuffer, enter `C-x RET C-\` and select `TeX` in the list.

LSP server

Navigating goals

On lambdapi-mode startup, the window is split into three buffers:

- the top buffer contains the Lambdapi file,
- the middle `*Goals*` buffer is where goals are displayed,
- the bottom `*lp-logs*` buffer is where Lambdapi messages are displayed.

It is possible to print the goals to solve at some point in the file by using the following short-cuts or the navigation buttons "Prev" and "Next":

- `C-c C-c`: jump to cursor position
- `C-c C-n` or button "Next": next tactic or command
- `C-c C-p` or button "Prev": previous tactic or command
- `C-c C-f`: next proof
- `C-c C-b`: previous proof

The part of the file up to the current goal is displayed with a green background. In case of error, the background gets red. If an edition occurs in the green zone, the green zone is automatically shrinked and the goals buffer updated.

It is possible to make the green zone expand automatically each time a new command is typed by toggling the electric mode with `C-c C-e`.

Clicking on the `i`-th goal of the `*Goals*` buffer puts the focus on it by inserting a `focus i` tactic in the proof script.

Electric Terminator mode

You can toggle electric terminators either from the toolbar or using `C-c C-e`. This will evaluate the region till the cursor whenever you type the `;` terminator or `begin`.

Customize window layout

The window layout can be customized in the LambdaPi customization group (Do `M-x customize-group lambdapi`). The layout can be refreshed with `C-c C-r`.

CPU usage and deactivation

If for any reason the LSP server consumes too much power (e.g. if a non-terminating rewrite system is edited), it can be disabled with `M-x eglot-shutdown`.

4.1.4 Other relevant packages

- company: auto-completion
- company-math: unicode symbols auto completion
- unicode-fonts: to configure correctly Emacs' unicode fonts
- rainbow-delimiters: to appreciate having a lot of parentheses
- paredit: to help keeping the parentheses balanced
- quickrun: for code evaluation

To have everything configured using use-package, use

```
(use-package lambdapi-mode
    :hook (paredit-mode rainbow-delimiters-mode-enable))
```

4.2 VSCode

You can get the VSCode extension for Lambdapi from the Marketplace. To install it from the sources, see INSTALL.md.

The extension provides syntax highlighting, go-to-definition, key-bindings for proof navigation, and snippets for inputing common mathematical symbols.

Logs (command `debug`) are displayed in a terminal which opens automatically when needed.

Usage

Make sure you have a lambdapi.pkg file in your folder when editing a Lambdapi file in it.

Proof navigation

Goals are visualised in a panel on the right side of the editor. You can navigate in proof with the following keybindings:

- `Ctrl+Right`: go one step forward
- `Ctrl+Left`: go one step backward
- `Ctrl+Up`: go to the previous proof (or the beginning)
- `Ctrl+Down`: go to the next proof (or the end)
- `Ctrl+Enter`: go to the position of the cursor
- `Ctrl+Alt+c`: toggle cursor mode (proof highlight follows the cursor or not)
- `Ctrl+Alt+w`: toggle follow mode (proof highlight is always centered in the window when keybindings are pressed)
- `Shift+Alt+w`: center proof highlight in the current window

Hover and go-to-definition

Hovering a token will display its type if available. For the go-to-definition, you can either:

- press `F12` when the cursor is within the range of a certain symbol
- or `right-click` on the symbol -> "Go to definition". It is advised to set up a key-binding for "Go back" in File -> Preferences -> Keyboard shortcuts.

Snippets

Type one of the suggested snippets described below, then press `Enter` or `Tab` to confirm adding the chosen Unicode character. If a snippet completion does not seem to work, try pressing `Ctrl+Space` to see completion suggestions.

Common symbols: `` `ra `` : →, `` `is `` : , `` `re `` : , `` `all `` : , `` `ex `` : , `` `imp `` : , `` `or `` : , `` `and `` : , `` `not `` : ¬, `` `th `` : , `` `eq `` : , `` `box `` : , `` `cons `` :

Greek letters: For every letter `l`, typing `` `l `` will suggest a corresponding unicode greek letter (for instance `` `b `` will suggest β). Some greek letters are present in a variant form as in LaTeX, accessible with `` `vl `` (for instance, `` `f `` will suggest and `` `vf `` will suggest ϕ).

Fonts: For every letter `l`, the following prefixes change the font of `l`: `` `dl `` for double-struck (), `` `il ``: italic (), `` `Il ``: bold italic (), `` `sl ``: script ( ), `` `Sl ``: bold script (), `` `fl ``: Fraktur ().

Recommended additional extension

- unicode-math replaces `->` by →, `_1` by $_1$, and many other unicode characters by simply pressing `Tab`.

4.3 Vim

A minimal Vim mode is provided to edit Lambdapi files. It provides syntax highlighting and abbreviations to enter unicode characters. It does not provide support for the LSP server yet.

4.3.1 Installation

Installing from sources

The Vim mode can be installed using the command `make install_vim` in the `lambdapi` repository.

Installing with Opam

If Lambdapi is installed with Opam or using `dune build` from the sources, then the line

```
set rtp+=~/.opam/$OPAM_SWITCH_PREFIX/share/vim
```

must be added to the Vim configuration file (`~/.vimrc` for Vim, `~/.config/nvim/init.vim` for NeoVim).

CHAPTER 5

Module system

Lambdapi has a very light module system based on file paths. It allows you to split your developments across files and folders. The main thing to know is that a file holds a single module, and accessing this module in other files requires using its *full* module path.

5.1 Module path

The module path that corresponds to a source file is defined using the name of the file, but also the position of the file under the *library root* (a folder under which all libraries are installed).

By default, the library root is `/usr/local/lib/lambdapi/lib_root` or `$OPAM_SWITCH_PREFIX/lib/lambdapi/lib_root` if `OPAM_SWITCH_PREFIX` is defined. An alternative library root can be specified using the environment variable `LAMBDAPI_LIB_ROOT` or the *command line flag* `--lib-root`.

The typical case is when we want to access a module of some installed library. In that case, the module path is built using the file path as follows: if our source file is at `<LIB_ROOT>/std/bool.lp` it is accessed with module path `std.bool`. And if there are nested folders then the module path gets more members. File `<LIB_ROOT>/std/a/b/c/d.lp` has module path `std.a.b.c.d`.

By default, all modules are looked up under the library root. However, there are cases where the files we want to work with are not yet placed under the library root. The typical case is when a library is under development. In that case, the development folder can be mapped under the library root, similarly to what would happen when mounting a volume in a file system. There are two ways of doing that, the first one is to use the `--map-dir MOD:DIR` *command line option*. However, the best way is to use a package configuration file.

5.2 Package configuration file

A package configuration file `lambdapi.pkg` can be placed at the root of the source tree of a library under development. It must contain the following fields (an example is given below for the syntax):

- `package_name` giving a globally unique name for the package being defined. This is not used yet, but will be necessary when packages will eventually be published online so that they can be automatically downloaded when users (the idea is to come up with a system similar to Cargo in Rust).
- `root_path` gives the module path under which the library is to be placed. Assuming that our configuration file is at `<REPO_ROOT>/lambdapi.pkg`, this means that `<REPO_ROOT>/a/b/c.lp` will get module path `<ROOT_PATH>.a.b.c`. In other words, this is equivalent to `--map-dir <ROOT_PATH>:<REPO_ROOT>` on the command line.

Remark: *.lpo* files needs to be removed and regenerated if the root_path is changed.

In the future, more useful meta data may be added to the configuration file, for example the name of the author, the version number, the dependencies, ...

Example of configuration file (syntax reference):

```
# Lines whose first non-whitespace charater is # are comments
# The end of a non-comment line cannot be commented.
# The config file MUST be called "lambdapi.pkg".
# The syntax for entries is:
key = value
# If the key is not known then it is ignored.
# There are two used keys for now:
package_name = my_package_name
root_path    = a.b.c
# We will use more entries later (e.g., authors, version, ...)
```

5.3 Installation procedure for third-party packages

A package should be installed under the library directory, under the specified root path. In other words a package whose root path is `x.y.z` should have its files installed under the directory `<LIB_ROOT>/x/y/z`. Moreover the directory structure relative to the package configuration file should be preserved.

Of course, we should enforce that a root path is uniquely used by one package. This can be enforced using a central authority, but this can only be done when we actually decide to implement automatic package distribution. For now, it is assumed that every user uses reasonable root paths for their packages.

Note that if a package uses root path `a.b.c` then no other package should use root path `a` or `a.b` (and obviously not `a.b.c` either). However there is no problem in using `a.d` or `a.b.d`. The rules are as follows:

- the root path of a package cannot be a prefix of the root path of another,
- the root path of a package cannot extend the root path of another.

We will use the following conventions:

- root path `std` is reserved for the standard library,
- extracted libraries are installed under `libraries.<NAME>` (for example, we would use root path `libraries.matita` for `matita`).

CHAPTER 6

Syntax of terms

The BNF grammar of Lambdapi is in lambdapi.bnf.

6.1 Identifiers

An identifier can be:

- a *regular* identifier: `/` or an arbitrary non-empty sequence of UTF-8 codepoints not among `\t\r\n :,; ` ( ) { } [ ] " . @ $ | ? /`
- an *escaped* identifier: an arbitrary sequence of characters enclosed between `{|` and `|}`

Remark: for any regular identifier `i`, `{|i|}` and `i` are identified.

Remark: Escaped identifiers or regular identifiers ending with a non-negative integer with leading zeros cannot be used for bound variable names.

Convention: identifiers starting with an uppercase letter denote types (e.g. `Nat`, `List`), and identifiers starting with a lowercase letter denote constructors, functions and proofs (e.g. `zero`, `add`, `refl`).

6.2 Qualified identifiers

A qualified identifier is an identifier of the form `dir1. ... dirn.file.id` denoting the function symbol `id` defined in the file `dir1/ ... /dirn/file.lp`. To be used, `dir1. ... dirn.file` must be required first.

Remark: `dir1`, ..., `dirn` cannot be natural numbers.

6.3 Terms

A user-defined term can be either:

- `TYPE`, the sort for types

- an unqualified identifier denoting a bound variable
- a qualified or a non-qualified symbol previously declared in the current file or in some previously open module, possibly prefixed by `@` to disallow implicit arguments
- an anonymous function λ `(x:A) y z,t` mapping `x`, `y` and `z` (of type `A` for `x`) to `t`
- a dependent product Π `(x:A) y z,T`
- a non-dependent product `A` $\rightarrow$ `T` (syntactic sugar for Π `x:A,T` when `x` does not occur in `T`)
- a `let f (x:A) y z : T t in u` construction
- an application written by space-separated juxtaposition, except for symbols having an infix *notation* (e.g. `x + y`)
- an infix symbol application `x + y`
- an identifier wrapped in parentheses to access its notationless value (e.g. `(+)`)
- a metavariable application `?0.[x;y]` that has been generated previously. `?0` alone can be used as a short-hand for `?0.[]`.
- a pattern-variable application `$P.[x;y]` (in rules only). `$P` alone can be used as a shorthand for `$P.[]`, except under binders (to avoid mistakes).
- _ for an unknown term or a term we don't care about. It will be replaced by a fresh metavariable in terms, or a fresh pattern variable in a rule left-hand side, applied to all the variables of the context.
- an integer between 0 and 2^30-1 if the *builtins* `"0"` and `"+1"` are defined
- a term enclosed between square brackets `[ ... ]` for explicitly giving the value of an argument declared as implicit

Subterms can be parenthesized to avoid ambiguities.

CHAPTER 7

Commands

The BNF grammar of Lambdapi is in lambdapi.bnf.

Lambdapi files are formed of a list of commands. A command starts with a particular reserved keyword and ends with a semi-colon.

One-line comments are introduced by `//`:

```
// These words are ignored
```

Multi-line comments are opened with `/*` and closed with `*/`. They can be nested.

```
/* These
   words are
   ignored /* these ones too */ */
```

7.1 require

Informs the type-checker that the current module depends on some other module, which must hence be compiled.

A required module can optionally be aliased, in which case it can be referred to with the provided name.

```
require std.bool;
require church.list as list;
```

Note that `require` always take as argument a qualified identifier. See *Module system* for more details.

7.2 open

Puts into scope the symbols of the previously required module given in argument. It can also be combined with the `require` command.

```
require std.bool;
open std.bool;
require open church.sums;
```

Note that `open` always take as argument a qualified identifier. See *Module system* for more details.

7.3 `symbol`

Allows to declare or define a symbol as follows:

modifiers `symbol` *identifier parameters* [`:` *type*] [*term*] [`begin` *proof* `end`] `;`

The identifier should not have already been used in the current module. It must be followed by a type or a definition (or both).

The following proof (if any) allows the user to solve typing and unification goals the system could not solve automatically. It can also be used to give a definition interactively (if no defining term is provided).

Without , this is just a symbol declaration. Note that, in this case, the following proof script does *not* provide a proof of *type* but help the system solve unification constraints it couldn't solve automatically for checking the well-sortedness of *type*.

For defining a symbol or proving a theorem, which is the same thing, is mandatory. If no defining *term* is provided, then the following proof script must indeed include a proof of *type*. Note that `symbol f:A  t` is equivalent to `symbol f:A  begin refine t end`.

Examples:

```
symbol N:TYPE;

// with no proof script
symbol add : N → N → N; // a type but no definition (axiom)
symbol double n  add n n; // no type but a definition
symbol triple n : N  add n (double n); // a type and a definition

// with a proof script (theorem or interactive definition)
symbol F : N → TYPE;
symbol idF n : F n → F n
begin
  assume n x; apply x;
end;
```

Modifiers:

Modifiers are keywords that precede a symbol declaration to provide the system with additional information on its properties and behavior.

- **Opacity modifier**:
  - `opaque`: The symbol will never be reduced to its definition. This modifier is generally used for actual theorems.
- **Property modifiers** (used by the unification engine or the conversion):
  - `constant`: No rule or definition can be given to the symbol
  - `injective`: The symbol can be considered as injective, that is, if `f t1 .. tn f u1 .. un`, then `t1u1`, ..., `tnun`. For the moment, the verification is left to the user.
  - `commutative`: Adds in the conversion the equation `f t u  f u t`.

  - `associative`: Adds in the conversion the equation `f (f t u) v  f t (f u v)` (in conjonction with `commutative` only).

    For handling commutative and associative-commutative symbols, terms are systemically put in some canonical form following a technique described here.

    If a symbol `f` is `commutative` and not `associative` then, for every canonical term of the form `f t u`, we have `t  u`, where is a total ordering on terms left unspecified.

    If a symbol `f` is `associative left` then there is no canonical term of the form `f t (f u v)` and thus every canonical term headed by `f` is of the form `f ...  (f (f` t_1 t_2`) t3) ...  t`. If a symbol `f` is `associative` or `associative right` then there is no canonical term of the form `f (f t u) v` and thus every canonical term headed by `f` is of the form `f` t_1 `(f` t_2 `(f` t_3 `...  t) ...  )`. Moreover, in both cases, if `f` is also `commutative` then we have t_1 t_2 `...  t`.

- **Exposition modifiers** define how a symbol can be used outside the module where it is defined. By default, the symbol can be used without restriction.
  - `private`: The symbol cannot be used.
  - `protected`: The symbol can only be used in left-hand side of rewrite rules.

  Exposition modifiers obey the following rules: inside a module,
  - Private symbols cannot appear in the type of public symbols.
  - Private symbols cannot appear in the right-hand side of a rewriting rule defining a public symbol.
  - Externally defined protected symbols cannot appear at the head of a left-hand side.
  - Externally defined protected symbols cannot appear in the right hand side of a rewriting rule.
- **Matching strategy modifier:**
  - `sequential`: modifies the pattern matching algorithm. By default, the order of rule declarations is not taken into account. This modifier tells Lambdapi to apply rules defining a sequential symbol in the order they have been declared (note that the order of the rules may depend on the order of the `require` commands). An example can be seen in `tests/OK/rule_order.lp`. *WARNING:* using this modifier can break important properties.

Examples:

```
constant symbol Nat : TYPE;
constant symbol zero : Nat;
constant symbol succ (x:Nat) : Nat;
symbol add : Nat → Nat → Nat;
opaque symbol add0 n : add n 0 = n  begin ... end; // theorem
injective symbol double n  add n n;
constant symbol list : Nat → TYPE;
constant symbol nil : List zero;
constant symbol cons : Nat → Π n, List n → List(succ n);
private symbol aux : Π n, List n → Nat;
```

Implicit arguments: Some arguments can be declared as implicit by encloding them into square brackets `[ ... ]`. Then, they must not be given by the user later. Implicit arguments are replaced by _ at parsing time, generating fresh metavariables. An argument declared as implicit can be explicitly given by enclosing it between square brackets `[ ... ]` though. If a function symbol is prefixed by `@` then the implicit arguments mechanism is disabled and all the arguments must be explicitly given.

```
symbol eq [a:U] : T a → T a → Prop;
// The first argument of "eq" is declared as implicit and must not be given
```

(continues on next page)

(continued from previous page)

```
// unless "eq" is prefixed by "@".
// Hence, "eq t u", "eq [_] t u" and "@eq _ t u" are all valid and equivalent.
```

Notations: Some notation can be declared for a symbol using the commands *notation* and *builtin*.

7.4 notation

The `notation` command allows to change the behaviour of the parser.

When declared as notations, identifiers then must be used at correct places and as such cannot make valid terms on their own anymore. To reaccess the value of the identifier without the notation properties, wrap it in parentheses.

infix The following code defines infix symbols for addition and multiplication. Both are associative to the left, and they have priority levels `6` and `7` respectively.

```
notation + infix left 6;
notation × infix left 7;
```

The modifier `infix`, `infix right` and `infix left` can be used to specify whether the defined symbol is non-associative, associative to the right, or associative to the left. Priority levels are non-negative floating point numbers, hence a priority can (almost) always be inserted between two different levels.

As explained above, at this point, + is not a valid term anymore, as it was declared infix. The system now expects + to only appear in expressions of the form `x + y` To get around this, you can use `(+)` instead.

prefix The following code defines a prefix symbol for negation with some priority level.

```
notation ¬ prefix 5;
```

Remarks:

- Prefix and infix operators share the same levels of priority, hence depending on the binding power, `-x + z` may be parsed `(-x) + z` or `-(x + z)`.
- Non-operator application (such as `f x` where `f` and `x` are not operators) has a higher binding power than operator application: let − be a prefix operator, then `- f x` is always parsed `- (f x)`, no matter what the binding power of − is.
- The functional arrow has a lower binding power than any operator, therefore for any prefix operator −, `- A → A` is always parsed `(- A) → A`
- Parsing of operators is performed with the pratter library.

quantifier Allows to write `` `f x, t `` instead of `f (λ x, t)`:

```
symbol  {a} : (T a → Prop) → Prop;
notation  quantifier;
compute λ p,  (λ x:T a, p); // prints ` x, p
type λ p, ` x, p; // quantifiers can be written as such
type λ p, `f x, p; // works as well if f is any symbol
```

7.5 builtin

The command `builtin` allows to map a "builtin" string to a user-defined symbol identifier. Those mappings are necessary for other commands or tactics. For instance, to use decimal numbers, one needs to map the builtins "0" and "+1" to some symbol identifiers for zero and the successor function (see hereafter); to use tactics on equality, one needs to define some specific builtins; etc.

notation for natural numbers It is possible to use the standard decimal notation for natural numbers by defining the builtins `"0"` and `"+1"` as follows:

```
builtin "0"   zero; // : N
builtin "+1"  succ; // : N → N
type 42;
```

7.6 rule

Rewriting rules for definable symbols are declared using the `rule` command.

```
rule add zero      $n  $n;
rule add (succ $n) $m  succ (add $n $m);
rule mul zero      _   zero;
```

Identifiers prefixed by `$` are pattern variables.

User-defined rules are assumed to form a confluent (the order of rule applications is not important) and terminating (there is no infinite rewrite sequences) rewriting system when combined with β-reduction.

The verification is left to the user, who can call external provers for trying to check those properties automatically using the *command line options* `--confluence` and `--termination`.

Lambdapi will however try to check at each `rule` command that the added rules preserve local confluence, by checking the joinability of critical pairs between the added rules and the rules already added in the signature (critical pairs involving AC symbols or non-nullary pattern variables are currently not checked). A warning is output if Lambdapi finds a non-joinable critical pair. To avoid such a warning, it may be useful to declare several rules in the same `rule` command by using the keyword `with`:

```
rule add zero      $n  $n
with add (succ $n) $m  succ (add $n $m);
```

Rules must also preserve typing (subject-reduction property), that is, if an instance of a left-hand side has some type, then the corresponding instance of the right-hand side should have the same type. Lambdapi implements an algorithm trying to check this property automatically, and will not accept a rule if it does not pass this test.

Higher-order pattern-matching. Lambdapi allows higher-order pattern-matching on patterns à la Miller but modulo β-equivalence only (and not $\beta\eta$).

```
rule diff (λx, sin $F.[x])  λx, diff (λx, $F.[x]) x × cos $F.[x];
```

Patterns can contain abstractions `λx, _` and the user may attach an environment made of *distinct* bound variables to a pattern variable to indicate which bound variable can occur in the matched term. The environment is a semicolon-separated list of variables enclosed in square brackets preceded by a dot: `.[x;y;...]`. For instance, a term of the form `λx y,t` matches the pattern `λx y,$F.[x]` only if `y` does not freely occur in `t`.

```
rule lam (λx, app $F.[] x)  $F; // η-reduction
```

Hence, the rule `lam (λx, app $F.[] x)  $F` implements η-reduction since no valid instance of `$F` can contain `x`.

Pattern variables cannot appear at the head of an application: `$F.[] x` is not allowed. The converse `x $F.[]` is allowed.

A pattern variable `$P.[]` can be shortened to `$P` when there is no ambiguity, i.e. when the variable is not under a binder (unlike in the rule η above).

It is possible to define an unnamed pattern variable with the syntax `$_.[x;y]`.

The unnamed pattern variable _ is always the most general: if `x` and `y` are the only variables in scope, then _ is equivalent to `$_.[x;y]`.

In rule left-hand sides, λ-expressions cannot have type annotations.

Important. In contrast to languages like OCaml, Coq, Agda, etc. rule left-hand sides can contain defined symbols:

```
rule add (add x y) z  add x (add y z);
```

They can overlap:

```
rule add zero x  x
with add x zero  x;
```

And they can be non-linear:

```
rule minus x x  zero;
```

Other examples of patterns are available in patterns.lp.

7.7 unif_rule

The unification engine can be guided using *unification rules*. Given a unification problem `t  u`, if the engine cannot find a solution, it will try to match the pattern `t  u` against the defined rules (modulo commutativity of) and rewrite the problem to the right-hand side of the matched rule. Variables of the RHS that do not appear in the LHS are replaced by fresh metavariables on rule application.

Examples:

```
unif_rule Bool  T $t  [ $t  bool ];
unif_rule $x + $y  0  [ $x  0; $y  0 ];
unif_rule $a → $b  T $c  [ $a  T $a'; $b  T $b'; $c  arrow $a' $b' ];
```

Thanks to the first unification rule, a problem `T ?x  Bool` is transformed into `?x  bool`.

WARNING This feature is experimental and there is no sanity check performed on the rules.

7.8 coerce_rule

Lambdapi can be instructed to insert function applications into terms whenever needed for typability. These functions are called *coercions*. For instance, assuming we have a type `Float`, a type `Int` and a function `FloatOfInt : Int → Float`, the latter function can be declared as a coercion from integers to floats with the declaration

```
coerce_rule coerce Int Float $x  FloatOfInt $x;
```

Symbol `coerce` is a built-in function symbol that computes the coercion. Whenever a term `t` of type `Int` is found when Lambadpi expected a `Float`, `t` will be replaced by `coerce Int Float t` and reduced. The declared coercion will allow the latter term to be reduced to `FloatOfInt t`.

Coercions can call the function `coerce` recursively, which allows to write, e.g.

```
coerce_rule coerce (List $a) (List $b) $l  map (λ e: El $a, coerce $a $b e) $l;
```

where `Set : TYPE;`, `List : Set → TYPE`, `El : Set → TYPE` and `map` is the usual map operator on lists such that `map f (cons x l)  cons (f x) (map l)`.

WARNING Coercions are still experimental and may not mix well with metavariables. Indeed, the term `coerce ?1 Float t` will not reduce to `FloatOfInt t` even if the equation `?1  Int` has been registered during typing. Furthermore, for the moment, it is unsafe to have symbols that can be reduced to protected symbols in the right-hand side of coercions: reduction may occur during coercion elaboration, which may generate unsound protected symbols.

7.9 `inductive`

The commands `symbol` and `rules` above are enough to define inductive types, their constructors, their induction principles/recursors and their defining rules.

We however provide a command `inductive` for automatically generating the induction principles and their rules from an inductive type definition, assuming that the following builtins are defined:

```
builtin "Prop"  ...; // : TYPE, for the type of propositions
builtin "P"     ...; // : Prop → TYPE, interpretation of propositions as types
```

An inductive type can have 0 or more constructors.

The name of the induction principle is `ind_` followed by the name of the type.

The command currently supports parametrized mutually defined dependent strictly-positive data types only. As usual, polymorphic types can be encoded by defining a type `Set` and a function τ`:Set → TYPE`.

Example:

```
inductive  : TYPE
| zero:
| succ:  → ;
```

is equivalent to:

```
constant symbol  : TYPE;
constant symbol zero : ;
constant symbol succ :  → ;
symbol ind_ p : π(p zero) → (Π x, π(p x) → π(p(succ x))) → Π x, π(p x);
rule ind_ _ $pz _ zero  $pz
with ind_ $p $pz $ps (succ $n)  $ps $n (ind_ $p $pz $ps $n);
```

For mutually defined inductive types, one needs to use the `with` keyword to link all inductive types together.

Inductive definitions can also be parametrized as follows:

```
(a:Set) inductive T: TYPE
| node: τ a → F a → T a
with F: TYPE
| nilF: F a
| consF: T a → F a → F a;
```

Note that parameters are set as implicit in the types of constructors. So, one has to write `consF t l` or `@consF a t l`.

For mutually defined inductive types, an induction principle is generated for each inductive type:

```
assert  ind_F: Π a, Π p:T a → Prop, Π q:F a → Prop,
  (Π x l, π(q l) → π(p (node x l))) →
  π(q nilF) →
  (Π t, π(p t) → Π l, π(q l) → π(q (consF t l))) →
  Π l, π(q l);
assert  ind_T: Π a, Π p:T a → Prop, Π q:F a → Prop,
  (Π x, Π l, π(q l) → π(p (node x l))) →
  π(q nilF) →
  (Π t, π(p t) → Π l, π(q l) → π(q (consF t l))) →
  Π t, π(p t);
```

Finaly, here is an example of strictly-positive inductive type:

```
inductive :TYPE  z: | s: →  | l:( → ) → ;

assert  ind_: Π p, π (p z) → (Π x, π (p x) → π (p (s x)))
  → (Π x, (Π y, π (p (x y))) → π (p (l x))) → Π x, π (p x);

assert p a b c  ind_ p a b c z  a;
assert p a b c x  ind_ p a b c (s x)  b x (ind_ p a b c x);
assert p a b c x y  ind_ p a b c (l x)  c x (λ y, ind_ p a b c (x y));
```

CHAPTER 8

Proof tactics

The `symbol` command allows the user to enter an interactive mode to solve typing goals of the form x_1`:`A_1`, ..., x:A ?M : U` (find a term of a given type materialized by a metavariable `?M`) and unification goals of the form x_1`:`A_1`, ..., x:A U V` (prove that two terms are equivalent modulo the user-defined rewriting rules).

The following tactics help users to refine typing goals and transform unification goals step by step. A tactic application may generate new goals/metavariables.

Except for the `solve` tactic which applies to all the unification goals at once, all the other tactics applies to the first goal only, which is called the *focused* goal, and this focused goal must be a typing goal.

The proof is complete only when all generated goals have been solved.

Proof scripts must be structured. The general rule is: when a tactic generates several subgoals, the proof of each subgoal must be enclosed between curly brackets:

```
opaque symbol 0 [x] : π(x  0  x = 0)
begin
  have l : Π x y, π(x  y  y = 0  x = 0)
  { // subproof of l
    refine ind_ _ _ _
    { /* case 0 */ reflexivity }
    { /* case s */ assume x y xy h a; apply ; apply s0 _ a }
  };
  assume x h; apply l _ _ h _; reflexivity
end;
```

Reminder: the BNF grammar of tactics is in lambdapi.bnf.

8.1 `begin`

Start a proof.

8.2 end

Exit the proof mode when all goals have been solved. It then adds in the environment the symbol declaration or definition the proof is about.

8.3 abort

Exit the proof mode without changing the environment.

8.4 admitted

Add axioms in the environment to solve the remaining goals and exit of the proof mode.

8.5 apply

If `t` is a term of type Π `x1:T1, ..., Π x:T,U`, then `apply t` refines the focused typing goal with `t _ ... _` (with n underscores).

8.6 assume

If the focused typing goal is of the form Π `x1 ... x,T`, then `assume h1 ... h` replaces it by `T` with `x` replaced by `h`.

8.7 remove

`remove h1 ... h` erases the hypotheses `h1 ... h` from the context of the current goal. The remaining hypotheses and the goal must not depend directly or indirectly on the erased hypotheses. The order of removed hypotheses is not relevant.

8.8 generalize

If the focused goal is of the form `x1:A1, ..., x:A, y1:B1, ..., y:B ?1 : U`, then `generalize y1` transforms it into the new goal `x1:A1, ..., x:A ?2 : Π y1:B1, ..., Π y:B, U`.

8.9 have

`have x: t` generates a new goal for `t` and then let the user prove the focused typing goal assuming `x: t`.

8.10 induction

If the focused goal is of the form Π `x:I ...,  ...` with `I` an inductive type, then `induction` refines it by applying the induction principle of `I`.

8.11 refine

The tactic `refine t` tries to instantiate the focused goal by the term `t`. `t` can contain references to other goals by using `?n` where `n` is a goal name. `t` can contain underscores _ or new metavariable names `?n` as well. The type-checking and unification algorithms will then try to instantiate some of the metavariables. The new metavariables that cannot be solved are added as new goals.

8.12 simplify

With no argument, `simpl` normalizes the focused goal with respect to β-reduction and the user-defined rewriting rules.

If `f` is a non-opaque symbol having a definition (introduced with), then `simpl f` replaces in the focused goal every occurrence of `f` by its definition.

If `f` is a symbol identifier having rewriting rules, then `simpl f` applies these rules bottom-up on every occurrence of `f` in the focused goal.

8.13 solve

Simplify unification goals as much as possible.

8.14 why3

The tactic `why3` calls a prover (using the why3 platform) to solve the current goal. The user can specify the prover in two ways :

- globally by using the command `prover` described in *Queries*
- locally by the tactic `why3 "<prover_name>"` if the user wants to change the prover inside an interactive mode.

If no prover name is given, then the globally set prover is used (`Alt-Ergo` by default).

A set of symbols should be defined in order to use the `why3` tactic. The user should define those symbols using builtins as follows :

```
builtin "T"    ... // : U → TYPE
builtin "P"    ... // : Prop → TYPE
builtin "bot"  ... // : Prop
builtin "top"  ... // : Prop
builtin "imp"  ... // : Prop → Prop → Prop
builtin "and"  ... // : Prop → Prop → Prop
builtin "or"   ... // : Prop → Prop → Prop
builtin "not"  ... // : Prop → Prop
```

(continues on next page)

(continued from previous page)

```
builtin "all"  ... // : Π x: U, (T x → Prop) → Prop
builtin "ex"   ... // : Π x: U, (T x → Prop) → Prop
```

Important note: you must run `why3 config detect` to make Why3 know about the available provers.

8.15 admit

Adds in the environment new symbols (axioms) proving the focused goal.

8.16 fail

Always fails. It is useful when developing a proof to stop at some particular point.

8.17 Tactics on equality

The tactics `reflexivity`, `symmetry` and `rewrite` assume the existence of terms with approriate types mapped to the builtins `T`, `P`, `eq`, `eqind` and `refl` thanks to the following builtin declarations:

```
builtin "T"      ... // : U → TYPE
builtin "P"      ... // : Prop → TYPE
builtin "eq"     ... // : Π [a], T a → T a → Prop
builtin "refl"   ... // : Π [a] (x:T a), P(x = x)
builtin "eqind"  ... // : Π [a] x y, P(x = y) → Π p:T a → Prop, P(p y) → P(p x)
```

8.18 reflexivity

Solves a goal of the form `Π x1, ..., Π x, P (t = u)` when `t ≡ u`.

8.19 symmetry

Replaces a goal of the form `P (t = u)` by the goal `P (u = t)`.

8.20 rewrite

The `rewrite` tactic takes as argument a term `t` of type `Π x1 ... x,P(l = r)` prefixed by an optional `left` (to indicate that the equation should be used from right to left) and an optional rewrite pattern in square brackets prefixed by a dot, following the syntax and semantics of SSReflect rewrite patterns:

```
<rw_patt> ::=
  | <term>
  | "in" <term>
  | "in" <ident> "in" <term>
  | <ident> "in" <term>
```

(continues on next page)

(continued from previous page)

```
  | <term> "in" <ident> "in" <term>
  | <term> "as" <ident> "in" <term>
```

See examples in rewrite1.lp and A Small Scale Reflection Extension for the Coq system, by Georges Gonthier, Assia Mahboubi and Enrico Tassi, INRIA Research Report 6455, 2016, section 8, p. 48, for more details.

In particular, if `u` is a subterm of the focused goal matching `l`, that is, of the form `l` with x_1 replaced by u_1, ..., `x` replaced by `u`, then the tactic `rewrite t` replaces in the focused goal all occurrences of `u` by the term `r` with x_1 replaced by u_1, ..., `x` replaced by `u`.

CHAPTER 9

Queries

9.1 assert, assertnot

The `assert` and `assertnot` are convenient for checking that the validity, or the invalidity, of typing judgments or equivalences. This can be used for unit testing of Lambdapi, with both positive and negative tests.

```
assert  zero : Nat;
assert  add (succ zero) zero  succ zero;
assertnot  zero  succ zero;
assertnot  succ : Nat;
```

9.2 compute

Computes the normal form of a term.

9.3 debug

The user can activate (with +) or deactivate (with −) the debug mode for some functionalities as follows:

```
debug +ts;
debug -s;
```

Each functionality is represented by a single character. For instance, `i` stands for type inference. To get the list of all debuggable functionalities, run the command `lambdapi check --help`.

9.4 flag

Sets some flags `on` or `off`, mainly for modifying the printing behavior. Only the flag `"eta_equalify"` changes the behavior of the rewrite engine by reducing η-redexes.

```
flag "eta_equality" on; // default is off
flag "print_implicits" on; // default is off
flag "print_contexts" on; // default is off
flag "print_domains" on; // default is off
flag "print_meta_types" on; // default is off
```

9.5 print

When called with a symbol identifier as argument, displays information (type, notation, rules, etc.) about that symbol. Without argument, displays the list of current goals (in proof mode only).

9.6 proofterm

Outputs the current proof term (in proof mode only).

9.7 prover

Changes the prover used by the `why3` tactic. By default, it uses *Alt-Ergo*.

```
prover "Eprover";
```

9.8 prover_timeout

Changes the timeout (in seconds) for the `why3` tactic. At the beginning, the timeout is set to 2s.

```
prover_timeout 60;
```

9.9 search

Runs a query between double quotes against the index file `~/.LPSearch.db`. See *Query Language* for the query language specification.

```
search "spine: (nat → nat) , hyp: bool";
```

9.10 type

Returns the type of a term.

9.11 verbose

Takes as argument a non-negative integer. Higher is the verbose level, more details are printed. At the beginning, the verbose is set to 1.

```
verbose 3;
```

CHAPTER 10

Query Language

Queries can be expressed according to the following syntax:

```
Q ::= B | Q,Q | Q;Q | Q|PATH
B ::= WHERE HOW GENERALIZE? PATTERN
PATH ::= << string >>
WHERE ::= name | anywhere | rule | lhs | rhs | type | concl | hyp | spine
HOW ::= > | = | >= |
GENERALIZE ::= generalize
PATTERN ::= << term possibly containing placeholders _ (for terms) and V# (for␣
↪variable occurrences >>
```

where

- the precedence order is `, > ; > |`
- parentheses can be used as usual to force a different precedence order
- `anywhere` can be paired only with >= and `name` can be paired only with >= and no `generalize`
- a pattern should be wrapped in parentheses, unless it is atomic (e.g. an identifier or a placeholder)

The semantics of the query language is the following:

- a query `Q` is either a base query `B`, the conjunction `Q1,Q2` of two queries `Q1` and `Q2`, their disjunction `Q1;Q2` or the query `Q|PATH` that behaves as `Q`, but only keeps the results whose path is a suffix of `PATH` (that must be a valid path prefix)
- a base query `name = ID` looks for symbols with name `ID` in the library. The identifier `ID` must not be qualified.
- a base query `WHERE HOW GENERALIZE? PATTERN` looks in the library for occurrences of the pattern `PATTERN` **up to normalization rules** and, if `generalize` is specified, also **up to generalization** of the pattern. The normalization rules are library specific and are employed during indexing. They can be used, for example, to remove the clutter associated to encodings, to align concepts by mapping symbols to cross-library standard ones, or to standardize the shape of statements to improve recall (e.g. replacing occurrence of `x > y` with `y < x`).
- `WHERE` restricts the set of occurrences we are interested in as follow:

  - `anywhere` matches without restrictions
  - `rule` matches only in rewriting rules
  - `lhs`/`rhs` matches only in the left-hand-side/right-hand-side of rewriting rules
  - `typ` matches only in the type of symbols
  - `spine` matches only in the spine of the type of symbols, i.e. what is left of the type skipping zero or more (but not all) universal quantifications/implications
  - `concl` matches only in the conclusion of the type of symbols, i.e. what is left skipping all universal quantifications/implications
  - `hyp` matches only in the hypotheses of the type of symbols, i.e. in the type of an universal quantification/in the right left of an implication that occur in the spine
- `HOW` further restricts the set of occurrences we are interested in as follows, where positions have already been restricted by `WHERE`:
  - >= and matches without restrictions
  - = the pattern must match the whole position
  - > the pattern must match a strict subterm of the position

Examples:

- `hyp = (nat → bool) , hyp >= (list nat)` searches for theorem that have an hypothesis `nat → bool` and such that `list nat` occurs in some (other) hypothesis. The query can return `filter_nat_list: list nat → (nat → bool) → list nat`
- `concl > plus | math.arithmetics` searches for theorems having an hypothesis containing `plus` and located in a module whose path is a suffix of `math.arithmetics`. The query can return `plus_O : x: nat. plus x O = x` where `plus_O` has fully qualified name `math.arithmetics.addition.plus`
- `name = nat ; name = NAT` searches for symbols named either `nat` or `NAT`

CHAPTER 11

Compatibility with Dedukti

Lambdapi can read Dedukti files with the extension `.dk`, and translate Lambdapi files to Dedukti files, and vice versa, by using the `export` *command*.

Moreover, a Lambdapi file can refer to a symbol declared in a Dedukti file.

In case there are two files `file.dk` and `file.lp`, `file.lp` is used.

Remarks on the export to Dedukti:

When a `lp` identifier or module/file name is not a valid `dk` identifier or module/file name (the `lp` and `dk` formats do not accept the same class of identifiers and module/file names), we try to rename them instead of failing:

- `lp` identifiers that are not valid `dk` identifiers or that are `dk` keywords are enclosed between `{|` and `|}` and, in escaped identifiers, spaces and newlines are replaced by underscores.
- In module names, dots are replaced by underscores and, if a `lp` file requires the module `mylib.logic.untyped.fol`, its translation will require the file `mylib_logic_untyped_fol.dk`. Therefore, in a package whose `root_path` is `mylib.logic`, the file `untyped/fol.lp` should be translated into `mylib_logic_untyped_fol.dk`.

CHAPTER 12

Include Lambdapi code in a Latex document

With the listings package:

You need to include lambdapi.tex.

See an example here.

CHAPTER 13

For developers

Guidelines for contributing

13.1 Overview of directories and files

- `doc/`: documentation in ReStructured Text format
  - `doc/README.md`: introduction to the user manual and guidelines
- `editors/`: editor plugins for handling Lambdapi files
  - `emacs/`: code for Emacs
  - `vim/`: code for Vim
  - `vscode/`: code for VSCode
    - `.vscode/*.json`: config for launching and debugging the extension
    - `lp.configuration.json`: specific characters
    - `media/styles.css`: styles
    - `package.json`: manifest of the plugin (activation events, scripts, dependencies, ...)
    - `snippets/unicode.json`: short-cuts for entering unicode characters
    - `src/*.ts`: source code of the extension
    - `syntaxes/lp.tmLanguage.json`: grammar of Lambdapi
    - `tsconfig.json`: TypeScript configuration (directories, ...)
- `libraries/`: libraries of Dedukti files (see `Makefile`)
- `src/cli/`: command line interface
  - `config.ml`: main program configuration
  - `init.ml`: lambdapi init command

  - `install.ml`: lambdapi install command
  - `lambdapi.ml`: main program
- `src/common/`: miscellaneous modules and libraries
  - `console.ml`: flag management
  - `debug.ml`: debugging tools
  - `error.ml`: warning and error management
  - `escape.ml`: basic functions on escaped identifiers
  - `library.ml`: Lambdapi library management
  - `logger.ml`: logging tools
  - `path.ml`: module paths in the Lambdapi library
  - `pos.ml`: source file position management
- `src/core/`: core of Lambdapi
  - terms:
    - `term.ml`: internal representation of terms
    - `libTerm.ml`: basic operations on terms
    - `libMeta.ml`: basic operations on metavariables
    - `print.ml`: pretty printing of terms
    - `env.ml`: maps identifier -> variable and type
  - signatures:
    - `builtin.ml`: managing builtins
    - `sign.ml`: compiled module signature (symbols and rules in a module)
    - `sig_state.ml`: signature under construction
  - rewriting:
    - `tree_type.ml`: types and basic functions for decision trees
    - `tree.ml`: compilation of rewriting rules to decision trees
    - `eval.ml`: rewriting engine
  - type inference/checking:
    - `ctxt.ml`: typing contexts (maps variable -> type)
    - `infer.ml`: constraints generating type inference and checking
  - unification:
    - `unif_rule.ml`: ghost signature for unification rules
    - `unif.ml`: unification algorithm
    - `inverse.ml`: inverse of injective functions
- `src/handle`: signature building
  - `command.ml`: command handling
  - `compile.ml`: file parsing and compiling (.lpo files)

  - `inductive.ml`: generation of induction principles
  - `query.ml`: handling of queries (commands that do not change the signature or the proof state)
  - tactics:
    * `proof.ml`: proof state
    * `rewrite.ml`: rewrite tactic (similar to Ssreflect)
    * `tactic.ml`: tactic handling
    * `why3_tactic.ml`: why3 tactic
- `src/lplib/`: extension of the Ocaml standard library
  - `range_intf.ml`: module type of abstract intervals
  - `rangeMap_intf.ml`: module type of abstract maps on intervals
  - `rangeMap.ml`: instance of `rangeMap_intf` using `range`
  - `range.ml`: instance of `range_intf` with integer intervals
  - `realpath.c`: C implementation of `Filename.realpath`
- `src/lsp/`: LSP server
  - `lp_doc.ml`: document type
  - `lp_lsp.ml`: LSP server
  - `lsp_base.ml`: basic functions for building messages
  - `lsp_io.ml`: basic functions for reading and sending messages
- `src/parsing/`: parsing Dedukti and Lambdapi files
  - pkg file parsing;
    * `package.ml`: parsing of package files `lambdapi.pkg`
  - abstract syntax:
    * `syntax.ml`: abstract syntax
  - dk file parsing:
    * `dkBasic.ml`: basic definitions for dk parsing
    * `dkTokens.ml`: lexing tokens for dk syntax
    * `dkLexer.mll`: lexer for dk syntax
    * `dkRule.ml`: convert dk rules into lp rules
    * `dkParser.mly`: parser for dk syntax
  - lp file parsing:
    * `lpLexer.ml`: lexer for Lambdapi syntax
    * `lpParser.mly`: parser for Lambdapi syntax
    * `parser.ml`: interfaces for parsers
    * `pretty.ml`: pretty print the abstract syntax (used to convert Dedukti files into Lambdapi files)
  - scoping:
    * `pratt.ml`: parsing of applications wrt symbol notations

    * `scope.ml`: convert the abstract syntax into terms

- `src/pure/`: pure interface (mainly used by the LSP server)
  - `pure.ml`: provide utilities to roll back the state
- `src/tool/`: tools
  - `external.ml`: call of external tools
  - `hrs.ml`: export to the .hrs format of the confluence competition
  - `sr.ml`: algorithm for checking subject reduction
  - `tree_graphviz.ml`: representation of trees as graphviz files
  - `xtc.ml`: export to the .xtc format of the termination competition
- `tests/`: unit tests
  - `OK/`: tests that should succeed
  - `KO/`: tests that should fail
- `misc/`:
  - `gen_version.ml`: script used by dune to generate `_build/default/src/core/version.ml` used in `lambdapi.ml`
  - `sanity_check.sh`: script checking some style guidelines below (called by `make sanity_check`)
  - `generate_tests.ml`: creates test files in `tests/OK` that can be parametrised
  - `listings.tex`: setup of the LaTeX package listings for including Lambdapi code into a LaTeX document
  - `deps.ml`: gives the `#REQUIRE` commands that should be added at the beginning of a Dedukti file

13.2 Implementation choices

- Back-tracking in interactive proofs is implemented using the "timed" references of the Timed library.
- Bindings in terms are implemented using the Bindlib library.
- Parsing uses the Menhir library.

13.3 Decision trees

The pattern matching algorithm uses decision trees. These decision trees are attached to symbols and can be inspected for debugging purposes. To print the decision tree of a symbol `s` of a module whose *module path* is `M` (see *Module path*), its decision tree may be printed with

```
lambdapi decision-tree M.s
```

The package configuration file of module `M` must be above the current working directory (closer to the root of the file system), or in the same directory.

The decision trees are printed to the standard output in the dot language. A dot file `tree.gv` can be converted to a png image using `dot -Tpng tree.gv > tree.png`. The one-liner

```
lambdapi decision-tree M.s | dot -Tpng | display
```

displays the decision tree of symbol `M.s` (`display` is part of imagemagick). For other output formats, see graphviz documentation.

13.3.1 Description of the generated graphs

Decision trees are interpreted during evaluation of terms to get the correct rule to apply. A node is thus an instruction for the evaluation algorithm. There are labeled nodes, labeled edges and leaves.

- Circle represent *regular* nodes. Let `n` be the label of the node, the next node is reached by performing an atomic match between the `n` th term of the stack and the labels of the edges between the node and its children. Let `t` be the term taken from the stack and matched against the labels. The labels of the edges can be
  - `s_n`, the atomic match succeeds if `t` is the symbol `s` applied to `n` arguments, the `n` arguments are put back in the stack;
  - λ`vn`, the atomic match succeeds if `t` is an abstraction. the body is substituted with (fresh) variable `vn`. Both the domain of the abstraction and the substituted body are put back into the stack;
  - Π`vn`, the atomic match succeeds if `t` is a product. The body is substituted with a (fresh) variable `vn`. Both the domain of the product and the substituted body are put back into the stack
  - `*`, the atomic match succeeds whatever `t` is.
- Rectangles represent *storage* nodes. They behave like regular nodes, except that the term of the stack is saved for later use.
- Diamonds represent *condition* nodes. The next node is reached by performing a condition check on terms that have been saved. If the condition is validated, the ✓-labeled edge is followed, and the -labeled one is followed otherwise. The label of the nodes indicates the condition, it can be
  - `n  m` which succeeds if the `n`th and `m`th saved terms are convertible,
  - `xs  FV(n)` which succeeds if the free variables of the `n`th saved term is a superset of the free variables `xs`.
- Triangles represent *stack check* nodes. The next node is the left child if the stack of arguments is empty, the right child otherwise. These nodes can only appear in trees built for sequential symbols.

Note for developers: the decision tree of ghost symbols can be printed as well using the `--ghost` flag. For instance,

```
lambdapi decision-tree --ghost M.
```

13.4 Testing

You can run tests using the following commands.

```
make tests        # Unit tests (not stopping on failure).
make real_tests   # Unit tests (stopping on first failure).

make dklib        # Checks files at https://github.com/rafoo/dklib/
make focalide     # Checks files generated from the Focalize library
make holide       # Checks files generated from the OpenTheory library
make matita       # Checks the traduction of Matita's arithmetic library.
make verine       # Checks files generated by the VeriT prover.
```

(continues on next page)

(continued from previous page)

```
make iprover      # Checks files generated by iProverModulo.
make zenon_modulo # Checks files generated by ZenonModulo.
```

13.5 Profiling

This document explains the use of standard profiling tools for the development of `lambdapi`.

13.5.1 Using Linux `perf`

The quickest way to obtain a per-symbol execution time is `perf`. It is simple to use, provided that you have the right privileges on your machine. No change is required in the build procedure, but `lambdapi` must be invoked as follows.

```
dune exec -- perf record lambdapi [LAMBDAPI_OPTIONS]
```

The program behaves as usual, but a trace is recorded in file `perf.data`. The data can then be displayed with the following command.

```
perf report
```

13.5.2 Profiling using Gprof

The `gprof` tool can be used to obtain a more precise (and thorough) execution trace. However, it requires modifying the `src/dune` file by replacing

```
(executable
 (name lambdapi)
```

with the following.

```
(executable
 (name lambdapi)
 (ocamlopt_flags (:standard -p))
```

This effectively adds the `-p` flag to every invocation of `ocamlopt`.

After doing that, `lambdapi` can be launched on the desired example, to record an execution trace. This has the (side-)effect of producing a `gmon.out` file. To retrieve the data, the following command can then be used.

```
gprof _build/install/default/lambdapi gmon.out > profile.txt
```

It takes two arguments: the path to the `lambdapi` binary used to generate the profiling data, and the profiling data itself.

CHAPTER 14

Indices and tables

- genindex
- search