Lem Manual

March 3, 2014

Contents

1 Introduction

Lem is a lightweight tool for writing, managing, and publishing large scale semantic definitions. It is also intended as an intermediate language for generating definitions from domain-specific tools, and for porting definitions between interactive theorem proving systems (such as Coq, HOL4, and Isabelle).

The language combines features familiar from functional programming languages with logical constructs. From functional programming languages we take pure higher-order functions, general recursion, recursive algebraic datatypes, records, lists, pattern matching, parametric polymorphism, a simple type class mechanism for overloading, and a simple module system. To these we add logical constructs familar in provers: universal and existential quantification, sets (including set comprehensions), relations, finite maps, inductive relation definitions, and lemma statements. Then there are facilities to let the user tune how Lem definitions are mapped into the various targets (by declaring target representations and controlling notation, renaming, inlining, and type classes), to generate witness types and executable functions from inductive relations, and for assertions.

Lem typechecks its input and can generate executable OCaml, theorem prover definitions in Coq, HOL4 and Isabelle/HOL, typeset definitions in LaTeX, and simple HTML.

1.1 Supported software

Lem is tested against the following versions of the backend software:

- OCaml: 3.12.1. and 4.00.0
- Coq: 8.4pl3 and 8.4pl2
- Isabelle: Isabelle-2013-2
- HOL: HOL4 Kananaskis 9

Older or newer versions of this software may work correctly with Lem, but are unsupported.

2 Installation

2.1 Lem binary

To build Lem run `make` in the top-level directory. This builds the executable `lem`, and places a symbolic link to it in that directory. Lem depends on OCaml. Lem is tested against OCaml 3.12.1. and 4.00.0. Other versions might or might not work.

Lem needs to access its library, which is by default stored in `PATH_TO_LEM/library`. If you want to use a different library directory, please either set the environment variable `LEMLIB` or pass the command-line argument `-lib YOUR_LIB_DIR` to Lem when running it.

2.2 Backend libraries

Running `make` only generates Lem itself. It does not generate the libraries needed to use Lem's output for certain backends. To generate the libraries for a specific backend, please run

```
for OCaml   : make ocaml-libs
for HOL 4   : make hol-libs
for Isabelle: make isa-libs
for Coq     : make coq-libs
```

These targets depend on the corresponding tool being installed, because they might run some automated tests or compile the Lem generated files. If you just want to generate the input which Lem gives to these tools, please run `make libs`.

2.3 Documentation

2.3.1 Papers

In subdirectory `doc`, a draft version of a conference submission describing Lem can be found, `lem-draft.pdf`.

2.3.2 Manual

Lem's manual can be found in subdirectory `doc`. It's written in *Markdown* and tested with Pandoc 1.9.1.1. However, it tries to avoid Pandoc specific extensions of Markdown.

Running `make` in subdirectory `doc` invokes Pandoc to generate html- and pdf-versions of the manual. Since the manual is written in Markdown, you can easily read it with the text-editor of your choice as well.

2.3.3 Library documentation

Similar to generating backend libraries, one can also generate documentation for the libraries by running `make tex-libs`. This generates a file `tex-lib/lem-libs.pdf`. In order to not pretty print the whole library, but just get interface information, one can use Lem's command line argument `print_env`. Running `lem -print_env library/pervasives_extra.lem` loads all of the libraries and afterwards prints the environment in a concise form.

2.3.4 Syntax

The input syntax of Lem is described later in this document. The syntax is defined using the Ott tool, and the language definition can be found in file `language/lem.ott`. You don't need Ott to compile or use Lem. However, if Ott is installed, the makefile in directory `language` can be used to generate a PDF documenting the syntax of Lem. A snapshot of that is in `doc/lem.pdf`.

2.3.5 Source code documentation

The makefile in Lem's root directory contains targets to generate Ocamldoc documentation for Lem's sources. Running `make lem-doc` generates

- directory `html-doc` (the source documentation as HTML)
- file `lem-doc.pdf` (the source documentation as PDF)
- file `lem-doc-dep.pdf` (a dependency graph as PDF)

2.3.6 Old Manual

Lem's old manual can be found in subdirectory `manual`. It is now out of date, though.

3 Invoking Lem

The most basic usage of Lem is running a command like

```
lem input1.lem ... inputn.lem -target
```

This command loads the lem files `input1.lem` through `inputn.lem` and outputs their translation to target `target` in the same directory as the input files. Multiple target arguments are possible. For example

```
lem name1.lem name2.lem -ocaml -hol -isa -coq
```

creates the following files (assuming there are no type errors, and no explicit renaming in the source files):

- `name1.ml` and `name2.ml` for target `ocaml`
- `name1Script.sml`, `name2Script.sml` for target `hol`
- `Name1.thy`, `Name2.thy` for target `isa`
- `name1.v`, `name2.v` for target `coq`

There are auxiliary files generated as well, which are discussed later.

Lem has many command line options to configure its behaviour. Running `lem --help` provides a short documentation of these options. The most common ones are explained below.

3.1 Backends

The following command line options tell Lem to generate output for certain backends. They are discussed in more detail in the corresponding backend sections later. Notice that multiple backend options can be given in order to generate output for more than one backend.

- `-ocaml` generate OCaml output
- `-hol` generate HOL4 output
- `-isa` generate Isabelle/HOL output
- `-coq` generate Coq output
- `-tex` generate LaTeX output for each module separately. This means that for each input file, a separate output `.tex` file is created. These files contain the pretty-printed input.
- `-tex_all output_filename.tex` generate LaTeX output in a single file. All input files are added as separate sections to the file `output_filename.tex` and a table of contents is added before.
- `-html` generate HTML output for each module separately
- `-lem` generate Lem output after simple transformations. This is used for refactoring Lem developments.

3.2 Dependency Resolution / Libraries

Lem by default searches the given input files for explicit *import* statements. It then tries to load the imported modules from either the directory of the files importing them or from one of the library directories. Lem only generates output for the files given explicitly as arguments. No output is generated for automatically imported files.

By default Lem uses the directory `library` as its library directory. This can be changed by either setting the environment-variable `LEMLIB` or by using the command-line argument `-lib`. Multiple usages of `-lib` allow using more than one library directory. Sometimes, users might be interested when a module is imported and from which file. Setting the warning-level of auto-imports to `warn` via the command-line option `-wl_auto_import warn` allows to keep track of auto-imports. Setting it to `err` via `-wl_auto_import err` turns off automatic imports and therefore requires the user to explicitly provide all needed input files on the command line. Notice however, that dependency resolution still happens between these explicitly given files and they might be processed in a different order than specified. To turn off resorting the explicit inputs, one can use the command-line flag `-no_dep_reorder`. When providing all inputs explicitly, it might be useful to turn off output for some of them via the command-line argument `-i`.

3.3 Output Directory

By default, output files are generated in the same directory as the corresponding input file. This remains the case even if input files come from multiple directories. For example

```
lem -tex dir1/file1.lem dir2/file2.lem
```

generates the files `dir1/File1.tex` and `dir2/File2.tex`. The command line option `-outdir` allows one to specify a different output directory. When using `-outdir` all explicitly, the given input files need to live in the same directory.

3.4 Auxiliary Output

Lem generates two kinds of output files, the main output and *auxiliary* outputs. Auxiliary outputs do not contain the main specification, but some related content that might be useful to the user. Examples of such auxiliary output are templates for termination proofs of recursive functions. Other are proof obligations generated by explicit lemmata as well as automatic consistency checks. This kind of auxiliary output should be copied by the user manually to some other files and be used there in whatever way the user thinks best. However, there is also auxiliary output that can be processed completely automatically. Examples are assertions, which for the Ocaml and HOL backends generate executable tests.

By default Lem generates all available auxiliary output. The command-line option `-auxiliary_level` can be used to control this behaviour. By default it is set to `full`. The command line option `-auxiliary_level auto` causes only automatically processable output like testing code of assertions to be generated. `-auxiliary_level none` turns off the generation of auxiliary files. One can also turn off the generation of the main files and only generate the auxiliary ones using `-only_auxiliary`.

3.5 Updating Existing Output

When using multi-file Lem developments, it might be handy to only update the output files that really changed. This allows the build-tools of backends like OCaml, HOL, Isabelle or Coq to only recompile files that really need to. Lem supports this via the command line option `-only_changed_output`.

3.6 Warnings

Lem can print warning messages about various things. Common warnings are about unused variables, name clashes that required automatic renaming of some entities or the need for pattern compilation, but there are many more. Warning options start with the prefix `wl`. They can be set to 4 different values

- `ign` ignore this warning and do nothing
- `warn` print a normal warning message and continue. This is the default for most options.
- `verb` print a verbose warning message and continue
- `err` stop with a verbose error message

The option `-wl` controls all warning messages at once. This is useful to turn off all warning messages (`-wl ign`). It can also be used to first turn all messages off and then activate selected ones again to concentrate on certain problems with the input. `-wl ign -wl_rename warn` causes - for example - Lem to print only warnings about renamings of constants. So, the user can look the renamings up and provide manual renamings, which generally look better than the auto-generated ones. Later, when there should be no auto-renamings any more, one could enforce this property by using `-wl_rename err`.

There are currently the following warnings. Since this list changes frequently, it is recommended to check the warning-options for your version of Lem via `lem --help`. This also prints the default setting for these warning options.

- `-wl` warning level of all warnings
- `-wl_gen` warning level of miscellaneous warnings
- `-wl_amb_code` warning level of ambiguous code. This means code that is can easily confuse users and should perhaps be written more clearly.
- `-wl_auto_import` warning level of automatically imported modules. Setting this option to `err` is used to turn off automatic imports. Together with '`-no_dep_reorder` it effectively turns off dependency resolution.

- `-wl_comp_message` warning level of compile messages. Compile messages are messages associated with certain functions. They contain information for the user how to use these functions. Such messages might point out that a function is not supported by certain backends or that its semantics might be underspecified of deliberately different for different backends.
- `-wl_inst_over` warning level of overridden instance declarations
- `-wl_no_dec_eq` warning level of equality of type is undecidable
- `-wl_pat_comp` warning level of pattern compilation. This causes a warning message if certain patterns are not natively supported by a backend and therefore need pattern compilation. Since pattern compilation changes the input significantly, sometimes users might prefer to write the pattern match in a style supported by the backend.
- `-wl_pat_exh` warning level of non-exhaustive pattern matches
- `-wl_pat_fail` warning level of failed pattern compilation
- `-wl_pat_red` warning level of redundant patterns
- `-wl_rename` warning level of automatic renamings
- `-wl_resort` warning level of re-sorted record fields and function clauses. Some backends require the fields of a record to be given in the same order as in the definition of the record type. Lem is more relaxed but warns if it needs to re-sort. Similarly, some backends require the clauses of mutually recursive function definitions to be grouped together, which might require resorting.
- `-wl_unused_vars` warning level of unused variables. To turn off this warning of a certain variable, one can change the variable name to start with an underscore. For example, for a variable `_x` no warning is issued.

3.7 Miscellaneous Command-Line Options

- `-print_env` print the environment signature on stdout. This feature gives a very brief overview of the current state and allows for example to get a short list of all the types and functions defined in a certain module.
- `-add_loc_annots` add location annotations to the output.
- `-v` print Lem's version

3.8 Command-line Options for Debugging

- `-ident` generate the input on stdout. This is used for debugging Lem.
- `-debug` print a backtrace for all errors. This is used for debugging Lem. In order for it to work, Lem needs to be compiled in debug mode (which is done by default).
- `-ident_pat_compile` activates pattern compilation for the identity backend. This is used for debugging.
- `-ident_dict_passing` activates dictionary passing transformations for the identity backend. This is used for debugging.

4 Backends

4.1 OCaml

The command line option `-ocaml` instructs Lem to generate OCaml output. A module with name `Mymodule` generates a file `mymodule.ml` and possibly `mymoduleAuxiliary.lem`.

4.1.1 Compilation

Lem-generated OCaml relies on some Lem-specific OCaml code as well as OCaml versions of the Lem library. Calling `make ocaml-libs` in Lem's main directory generates these files in subdirectory `ocaml_lib` and compiles them.

When compiling Lem-generated OCaml-code, it needs to be linked with the files in directory `ocaml_lib`. To make this simpler, an OCaml-package `Lem` (using `extract.cma`) is defined in this directory. One can for example compile a file `name1.ml` by

```
ocamlc -I path_to_lem/ocaml-lib/_build -o name extract.cma name1.ml
```

or, using *ocamlbuild* and *findlib*, by

```
export OCAMLPATH=/absolute/path/to/lem/ocaml-lib:$OCAMLPATH
ocamlbuild -libs nums -use-ocamlfind -pkg lem name.native
```

4.1.2 Auxiliary Files

OCaml auxiliary files do not need modifications by the user. They contain tests generated from *assertions* in the input files. When compiled as described above and run as standalone programs, OCaml auxiliary files execute the tests and print the results.

4.2 HOL4

The command line option `-hol` instructs Lem to generate HOL4 output. A module with name `Mymodule` generates a file `mymoduleScript.sml` and possibly `mymoduleAuxiliaryScript.sml`.

4.2.1 Compilation

Lem-generated HOL theories depend on some Lem-specific HOL4 code as well as HOL4 versions of the Lem library. Calling `make hol-libs` in Lem's main directory generates these files in subdirectory `hol_lib` and compiles them using `Holmake`. During this compilation process a heap with name `lemheap` is generated. It is recommended to use this heap for your own HOL4 development based on Lem-generated files. Using the generated files in `hol-libs` directly is possible as well, though. In order to use the heap, add the following line to the `Holmakefile` of the directory, where your HOL4-files are stored:

```
HOLHEAP = path_to_lem/hol-lib/lemheap
```

A template `Holmakefile` file using other useful options as well can be found in directory `library`.

4.2.2 Auxiliary Files

HOL4 auxiliary files contain both executable tests generated from assertions as well as templates for termination proofs and lemmata that need manual labour by the user. The command line option `-auxiliary_level auto` allows to generate only the executable tests.

4.3 Isabelle/HOL

The command line option `-isa` instructs Lem to generate Isabelle/HOL output. A module with name `Mymodule` generates a files `Mymodule.thy` and possibly `MymoduleAuxiliary.thy`.

4.3.1 Generating Isabelle Library

Lem-generated Isabelle theories depend on some Lem-specific Isabelle theories as well as Isabelle versions of the Lem library. Calling `make isa-libs` in Lem's main directory generates these files in subdirectory `isa_lib`. In contrast to the HOL and OCaml libraries the generation of these libraries does not trigger automatic tests. If you want to check the sanity of the library, please use `make isa-lib-tests` in subdirectory library. This creates a directory `library/isa-build-dir` and the library auxiliary files within this directory. Moreover, there is a file `LemTests.thy`, which imports all other files and is therefore useful for testing all these files in Isabelle.

4.3.2 Adapting Isabelle Imports

The theory import-statement in the header of generated Isabelle files contains the absolute path to Lem's library directory. If you move the library directory, this path needs adapting. If you want to use a backend specific `import` statement in your own Lem development, that imports some theory in the library directory, you can use the variable `$LIB_DIR` as in the following example

```
open import {isabelle} `$LIB_DIR/Lem`
```

4.3.3 Auxiliary Files

Isabelle auxiliary contain both executable tests generated from assertions as well as templates for termination proofs and lemmata that need manual labour by the user. In contrast to the auxiliary output of HOL, the templates for lemmata and termination proofs make use of Isabelle's automation and therefore often succeed without user intervention. Therefore, using the command line option `-auxiliary_level auto` in order to generate only code for assertions is possible but not imperative.

4.3.4 Automatic Proof Tools / Counter Example Generation

The auxiliary files contain templates for lemmata that use Isabelle's automation. Therefore these templates might be useful even for users not familiar with Isabelle, who want to use tools like automatic counter example generation.

The Lem-lemma

```
lemma unzip_zip:
   forall l1 l2. unzip (zip l1 l2) = (l1, l2)
```

is for example translated to the following Isabelle code:

```
lemma unzip_zip:
   "! l1 l2. list_unzip (zip l1 l2) = (l1, l2)"
   (* try *) by auto
```

The automated proof attempt by the `auto` method fails. If the user removes the comment around `try`, various automated methods are run to either prove the lemma or find a counterexample. These methods include running external SMT and first order provers, internal natural deduction tools as well as a sophisticated counter example generator. In this example, Isabelle quickly finds a counterexample:

```
Nitpick found a counterexample for card 'a = 2 and card 'b = 2:
  Skolem constants: l1 = [a1], l2 = []
```

While this is a trivial example, counterexamples and proofs are also found for more interesting cases. So, writing lemmata in Lem and translating them to Isabelle might be useful, even if you are not familiar with Isabelle.

4.4 Coq

4.5 LaTeX

The command line option `-tex` instructs Lem to generate LaTeX output. A module with name `Mymodule` generates a files `Mymodule.tex`, `Mymodule-inc.tex` and `Mymodule-use_inc.tex`. No auxiliary files are generated. The generated LaTeX output depends on the style-file `lem.sty` in directory `tex-lib`.

The file `Mymodule.tex` contains a pretty-printed version of the original input file. `Mymodule-inc.tex` defines LaTeX macros that can be used to type-set single definitions inside your own developments. `Mymodule-use_inc.tex` uses these macros to mimic the behaviour of `Mymodule.tex`. It is useful, since it essentially is a list of all the defined macros in the order they appear in the input file.

The command-line-option `-tex` generates separate LaTeX files for each input file. If using the option `-tex_all my_output`, Lem generates the files `my_output.tex`, `my_output-inc.tex` and `my_output-use_inc.tex`, which contain representations / macros for all input files.

4.5.1 LaTeX Macro Names

The `...-inc.tex` files contain macros that allow type-setting single definitions from the original input. As far as possible, the names of the macros are derived from the name of the defined entity. We have

- the definition of a function `myfun` generates a macro `\LEMmyfun`
- the definition of a type `mytype` generates a macro `\LEMTypeMytype`
- the definition of a relation `myrel` generates a macro `\LEMmyrel`
- a val-specification of a function `myfun` generates a macro `\LEMValspecMyfun`

Other entities like declarations, class definitions etc. do not currently get predictable names. Please have a look at the content of the `...-use_inc.tex` or `...-inc.tex` file to figure out the generated name for these.

If the names of macros derived by the scheme above clash, a number is added at the end. Because LaTeX does not allow digets in macro names, these numbers are expressed as English words. Name clashes happen if there are several definitions of a function, which sometimes happens since you might prefer a different definition depending on the target. If there is a val-specification for a function `myfun`, as well as an OCaml-specific, a HOL and Isabelle-specific and Coq-specific one, these generates the macros `\LEMValspecMyfun`, `\LEMmyfun`, `\LEMmyfunZero`, `\LEMmyfunOne`, `\LEMmyfunTwo`.

4.5.2 LaTeX Macro Usage

By default, macros print their full definition without any preceding comment, but with a LaTeX `label` that allows referring to that definition. The generated LaTeX macros accept an optional argument that changes this behaviour. So, for example `\LEMmyfun` prints the definition of the function `myfun`, whereas `\LEMmyfun[name]` prints only the type-set name of `myfun`. There are the following arguments available:

- `default` same as not providing an argument, alias for `def`
- `def` print a label followed by the full definition excluding the preceding comment
- `defWithComment` print a label followed by the full definition including the preceding comment
- `name` print the typeset name of the definition. For definitions defining more than one constant of type, as well as for Lem statements not defining anything, this is empty
- `comment` print the preceding comment
- `commentPre` alias for `comment`

- `commentPost` print the comment directly after the definition (usually empty)
- `core` print the *core* of a definition. Usually that's the right hand side of the definition, but might vary depending on the type of Lem-statement that generated the macro
- `label` print the label that is used by `def` and `defWithComment`.

If you want to learn about details or add your own argument values, please have a look at the definition of macro `\lemdefn` in file `tex-lib/lem.sty`.

4.5.3 Libraries

Running `make tex-libs` in Lem's main directory generates LaTeX output for Lem's library. By running Pdflatex on this output a file `tex-lib/lem-libs.pdf` is generated, which can be used as library documentation. Moreover, there are also `lem-libs.tex`, `lem-libs-inc.tex` and `lem-libs-use_inc.tex`, which can be used as described above.

4.6 HTML

The command line option `-html` instructs Lem to generate HTML output. A module with name `Mymodule` generates a file `Mymodule.html`. No auxiliary files are generated.

5 Lem-library

Lem comes with a default library of types and constants. This library can be found in directory `library`. It contains collections such as lists, sets and maps, basic data types such as disjoint sums, optional types, booleans and tuples, useful combinators on functions, and a library for working with relations.

5.1 General Design

The library follows Haskell's library in terms of names of constants, types and modules. The library is separated into two sets of modules: the *main* and *extra* modules. The main hierarchy of files contain total, terminating functions that we believe are well-specified enough to be portable across all backends. All other functions are are placed in the extra modules. For example, the library file `function.lem` includes various useful combinators such as `flip` and `const`. The `function_extra.lem` file, on the other hand, contains the constant `THE` with type `forall 'a.  ('a --> bool) --> maybe 'a`, inexpressible in Coq.

Lem leaves the choice of using the main library or the extended library to the user. The module `Pervasives` contains the main part of the library, `Pervasives_extra` the extra part. The first line of a common Lem file is usually `open import Pervasives` or `open import Pervasives_extra`, which imports and opens either the main or the extra library.

5.2 Library documentation

For an overview of the Lem library, please generate the pdf-file `tex-lib/lem-libs.pdf` by running `make tex-libs`. If you are just interested in the interface, consider running `lem -print_env library/pervasives_extra.lem`.

6 Writing your own Lem files

Lem's syntax broadly follows OCaml syntax, while the libraries follow the Haskell libraries. Here, only a few selected points of Lem's syntax and its features are discussed. To learn more about its syntax, please have a look at the next section and at the file `doc/lem.pdf`. Another possibility is having a look at the Lem-library in the `library`-directory or at the tests in directory `tests`, especially `tests/backends`.

6.1 Header

6.1.1 Importing Library

A Lem file usually starts with importing the appropriate library. Without such an import, even very simple operations like boolean conjunction are not available. The user thus has the choice of either importing the main library or the extended library. The main library contains total, terminating functions that we believe are well-specified enough to be portable across all backends. All other functions are placed in the extended library.

The main library is imported by

```
open import Pervasives
```

and the extended one by

```
open import Pervasives_extra
```

6.1.2 Setting Module Name

Each Lem file defines a top-level module. A file with name `mymodule.lem` creates a Lem module `Mymodule`. By default this is also the name of the module for all backends. It is however possible (and sometimes necessary) to rename modules for backends. For example, Lem's library contains a file `set.lem`, which defines the Lem module `Set`. In order to avoid clashes with the existing HOL and Isabelle theories called set, it is however renamed to `lem_set` for these backends. This is done via the command

```
declare {isabelle;hol} rename module = lem_set
```

Notice that in contrast to renaming functions, no module name is used behind the keyword `module`. This causes the current module to be renamed. It is also possible to rename other modules. However, this should only be used for submodules defined in the same file as the renaming, because otherwise the module might have different names in different files referring to it.

6.1.3 Importing Modules

Lem provides dependency resolution, but only for explicitly imported modules. Using a statement like

```
import Mymodule
```

causes Lem to search for a file `mymodule.lem` in the current directory as well as in a list of given library directories. If such a file is found, it is automatically processed by Lem and it's contents are used to generate a Lem module `Mymodule`. Import statements do not need to, but are usually placed at the top of Lem files.

6.1.4 Opening / Including Modules

A function `myfun` from a module `Mymod` is usually accessible by `Mymod.myfun`. Lem allows explicitly opening modules via `open Mymod`. After such a statement `myfun` can be used instead of `Mymod.myfun`.

When using `open Mymod` inside a module `Mymod2`, it only affects the state inside this current module `Mymod2`. It does not change the outside view of `Mymod2`. If you want to be all functions `Mymod.myfun` also available as `Mymod2.myfun`, one can use `include` instead of `open`. Including is mostly useful for writing libraries.

Often one wants to import and open a module at the same time. Therefore `open import Mymodule` and `include import Mymodule` are hands for first importing and then opening / including a module. Similarly, Lem allows opening/including/importing multiple modules with just one statement.

6.2 Constant definitions

6.2.1 Simple definitions

A simple function definition is Lem is very similar to an OCaml top-level definition. It is of the form

```
let fun_name arg1 ... argn = rhs_exp
```

The arguments are allowed to be arbitrary Lem-patterns. The right-hand side an arbitrary expression that uses the variables bound by the arguments.

6.2.2 Target specific definitions

Sometimes you might want to use different definitions for different targets. In order to do that the functions needs to be introduced via a val-specification first:

```
val fun_name : type-scheme
```

After this specification multiple target-specific definitions of the form

```
let {target1; ...; targetn} fun_name arg1 ... argn  = rhs_exp
```

or let ∼{target1; . . . ; targetn} fun_name arg1 . . . argn = rhs_exp

are allowed. Thereby {`target1; ...; targetn`} represents the set of the given targets, whereas ∼ represents the set of all targets except the given ones. The targets intended to just typeset the Lem input file, i.e. the LaTeX and HTML do not require definitions and providing one does not change their behaviour. All other targets for which the function should be used, require a definition.

6.2.3 Inlining

Lem allows inlined constant definitions. These definitions are essentially macro expansions. For example consider an emptiness check for List.

```
let inline isEmptyList l = (l = [])
```

It is a simple, straightforward definition, that you might not want to generate special target definitions for. An `inline` definition allows using the function `isEmptyList` in Lem. It is also used in the HTML, Latex, Identity and Refactoring backends. All other backends replace it with the right hand side though. So, Lem would not define HOL4 function for `isEmptyList`, but replace every occurrence of it with the definition.

In order to allow this inlining, the definition has to be simple. Arguments are just allowed to be variables and inlined definition may not be recursive. Moreover, they may not have any type-class constraints attached.

If a val-specification is provided first, it is possible to inline constants only for certain targets and generate proper definitions for other targets. For this, syntax similar to the following example is used:

```
let inline {hol} isEmptyList l = (l = [])
```

6.2.4 Recursive Definitions

Lem allows to define recursive and even mutually recursive functions by using the keyword `let rec`. For example to define (stupidly) functions `even` and `odd`, one can use

```
let rec even (0:nat) = true
    and odd  0 = false
    and even (n + 1) = not (odd n)
    and odd (n + 1) = not (even n)
```

6.2.5 Termination Proofs

Recursive definitions require termination (or well-foundedness) proofs in the theorem prover backends. Isabelle and HOL4 are able to delay these proofs. The user has to fill in these proofs then, before using the defined functions. For simple functions like the ones in the example, this can be annoying. A `termination_argument` declaration can therefore be used to tell Isabelle and HOL to try automatic termination proofs. If multiple functions are defined in a single, mutually recursive definition, an automatic termination proof is only attempted, if automatic termination is declared for all defined functions.

```
declare {hol; isabelle} termination_argument even = automatic
declare {hol; isabelle} termination_argument odd = automatic
```

6.3 Type definitions

Note that when defining new types in Lem it may be necessary to instantiate some of the basic type classes, as described in the Type Classes section below.

6.4 Assertions / Lemmata / Theorems

Lem allows the user to write assertions, lemmata and theorems. These are named boolean expressions, which the user desires to be true. For the append function on lists, one could for example write:

```
assert append_test_1: [(2:nat); 3] ++ [4;5] = [2;3;4;5]
lemma append_spec: (forall l. [] ++ l = l) && (forall x xs ys. (x :: xs) ++ ys = x :: (xs ++ ys))
theorem append_empty: forall l. l ++ [] = l
```

Assertions should be executable. They are intended to be used for unit-testing your Lem specifications. For OCaml and HOL4 they generate executable tests.

Lemmata are non-executable properties. They are used to document properties that are non-executable. They can be used for documentation purposes to write down properties the user had in mind, when defining a function. They generate proof obligation in the auxiliary files. Therefore, they can also be used to express important high-level properties about the whole model, which the user wants to proof correct. *Theorems* are lemmata that the user wants to mark as important.

Writing assertions allows an easy way to unit-test specifications. Lemmata and theorems are beneficial for documentation purposes. The automated translation to Isabelle also allows to use Isabelle's sophisticated automation without knowing much about Isabelle. With that mechanism it is for example very easily possible to search for counter-examples.

6.5 Renaming

The naming conventions of our backends differ. Therefore, it might be beneficial to use different names depending on the backend. Renaming can also be used to avoid name clashes with existing backend functions or just to avoid confusion when similar names already are used for the backend. For example, there is already a HOL4 function `symmetric`. To avoid confusion with the Lem function `isSymmetric` the Lem one can easily be renamed:

```
declare {hol} rename function isSymmetric = lem_is_symmetric
```

Besides `functions`, it is also possible to rename `types`, `fields` and `modules`.

7 The Lem Language

7.1 Metavariables and Identifiers

```
indexvar n , i , j , k   {{ Index variables for meta-lists }}
metavar num              {{ Numeric literal }}
metavar string           {{ String literal }}
metavar backtick_string  {{ String literal preceded by ' }}
metavar regexp           {{ Regular expresion, as a string literal }}
metavar l                {{ Source location }}
metavar x                {{ Name }}
metavar ix               {{ Infix name }}

id  ::= {{ Long identifers }}
  | x1 . .. xn . x l

a  ::=  {{ Type variables }}
  | ' x
```

7.2 Literals

```
lit  ::=  {{ Literal constants }}
  | true
  | false
  | num
  | hex
  | bin
  | string
  | ( )
```

7.3 Types

```
typ  ::=  {{ Types }}
  | _                                {{ Unspecified type }}
  | a                                {{ Type variables }}
  | typ1 -> typ2                     {{ Function types }}
  | typ1 * .... * typn               {{ Tuple types }}
  | id typ1 .. typn                  {{ Type applications }}
  | backtick_string typ1 .. typn     {{ Backend-Type applications }}
  | ( typ )
```

7.4 Patterns

```
pat ::=   {{ Patterns }}
  | _                                {{ Wildcards }}
  | ( pat as x )                     {{ Named patterns }}
  | ( pat : typ )                    {{ Typed patterns }}
  | id pat1 .. patn                  {{ Single variable and constructor patterns }}
  | <| fpat1 ; ... ; fpatn semi_opt |> {{ Record patterns }}
  | ( pat1 , .... , patn )           {{ Tuple patterns }}
```

```
  | [ pat1 ; .. ; patn semi_opt ]       {{ List patterns }}
  | ( pat )
  | pat1 :: pat2                      {{ Cons patterns }}
  | x + num                           {{ constant addition patterns }}
  | lit                                 {{ Literal constant patterns }}


fpat ::=  {{ Field patterns }}
  | id = pat l

bar_opt  ::=         {{ Optional bars }}
  |
  | '|'

semi_opt  ::=       {{ Optional semi-colons }}
  |
  | ;
```

7.5 Expressions

```
exp ::=  {{ Expressions }}
  | id                                                {{ Identifiers }}
  | backtick_string                                   {{ identifier that should be literally used i
  | fun psexp                                         {{ Curried functions }}
  | function bar_opt pexp1 '|' ... '|' pexpn end      {{ Functions with pattern matching }}
  | exp1 exp2                                         {{ Function applications }}
  | exp1 ix exp2                                      {{ Infix applications }}
  | <| fexps |>                                       {{ Records }}
  | <| exp with fexps |>                              {{ Functional update for records }}
  | exp . id                                          {{ Field projection for records }}
  | match exp with bar_opt pexp1 '|' ... '|' pexpn l end     {{ Pattern matching expressions }}
  | ( exp : typ )                                     {{ Type-annotated expressions }}
  | let letbind in exp                                {{ Let expressions }}
  | ( exp1 , .... , expn )                            {{ Tuples }}
  | [ exp1 ; .. ; expn semi_opt ]                     {{ Lists }}
  | ( exp )
  | begin exp end                                     {{ Alternate syntax for (exp) }}
  | if exp1 then exp2 else exp3                       {{ Conditionals }}
  | exp1 :: exp2                                    {{ Cons expressions }}
  | lit                                               {{ Literal constants }}
  | { exp1 | exp2 }                                   {{ Set comprehensions }}
  | { exp1 | forall qbind1 .. qbindn | exp2 }         {{ Set comprehensions with explicit binding }
  | { exp1 ; .. ; expn semi_opt }                     {{ Sets }}
  | q qbind1 ... qbindn . exp                         {{ Logical quantifications }}
  | [ exp1 | forall qbind1 .. qbindn | exp2 ]         {{ List comprehensions (all binders must be q
  | do id pat1 <- exp1 ; .. patn <- expn ; in exp end  {{ Do notation for monads }}

q ::=  {{ Quantifiers }}
  | forall
  | exists

qbind  ::= {{ Bindings for quantifiers}}
  | x
  | ( pat IN exp )                                  {{ Restricted quantifications over sets}}
  | ( pat MEM exp )                                   {{ Restricted quantifications over lists }}

fexp  ::=  {{ Field-expressions }}
```

```
  | id = exp l

fexps ::=  {{ Field-expression lists }}
  | fexp1 ; ... ; fexpn semi_opt l

pexp ::=  {{ Pattern matches }}
  | pat -> exp l

psexp ::=  {{ Multi-pattern matches }}
  | pat1 ... patn -> exp l

tannot_opt ::=  {{ Optional type annotations }}
  |
  | : typ

funcl  ::=  {{ Function clauses }}
  | x pat1 ... patn tannot_opt = exp

letbind  ::= {{ Let bindings }}
  | pat tannot_opt = exp        {{ Value bindings }}
  | funcl                       {{ Function bindings }}
```

7.6 Inductive Relation Definitions

```
name_t ::=  {{ Name or name with type for inductively defined relation clauses }}
  | x
  | ( x : typ )

name_ts ::=  {{ Names with optional types for inductively defined relation clauses }}
  | name_t0 .. name_tn

rule ::=  {{ Inductively defined relation clauses }}
  | x : forall name_t1 .. name_ti . exp ==> x1 exp1 .. expn

witness_opt  ::= {{ Optional witness type name declaration. Must be present for a witness type to be
  |
  | witness type x ;

check_opt  ::=  {{ Option check name declaration }}
  |
  | check x ;

functions_opt  ::=  {{ Optional names and types for functions to be generated. Types should use only
  |
  | x : typ
  | x : typ ; functions_opt

indreln_name ::=  {{ Name for inductively defined relation }}
  | [ x : typschm witness_opt check_opt functions_opt ]
```

7.7 Type Definitions

```
typs ::=      {{ Type lists }}
  | typ1 * ... * typn

ctor_def ::= {{ Datatype definition clauses }}
  | x of typs
```

```
  | x                {{ Constant constructors }}

texp ::=  {{ Type definition bodies }}
  | typ                                            {{ Type abbreviations }}
  | <| x1 : typ1 ; ... ; xn : typn semi_opt |>     {{ Record types }}
  | bar_opt ctor_def1 '|' ... '|' ctor_defn        {{ Variant types }}

name_opt ::=  {{ Optional name specification for variables of defined type }}
  |
  | [ name = regexp ]

td ::= {{ Type definitions }}
  | x tnvars name_opt = texp
  | x       tnvars name_opt                        {{ Definitions of opaque types }}
```

7.8 Type Schemes

```
c ::=  {{ Typeclass constraints }}
  | id tnvar

cs ::=  {{ Typeclass constraint lists }}
  |
  | c1 , .. , ci =>                                        {{ Must have >0 constraints }}

c_pre ::=  {{ Type and instance scheme prefixes }}
  |
  | forall tnvar1 .. tnvarn . cs                           {{ Must have >0 type variables }}

typschm ::=    {{ Type schemes }}
  | c_pre typ

instschm ::=  {{ Instance schemes }}
  | c_pre ( id typ )
```

7.9 Target Descriptions

```
target ::=  {{ Backend target names }}
  | hol
  | isabelle
  | ocaml
  | coq
  | tex
  | html
  | lem

targets  ::=  {{ Backend target name lists }}
  | { target1 ; .. ; targetn }
  | ~{ target1 ; .. ; targetn }              {{ all targets except the listed ones }}

targets_opt ::=    {{ Optional targets }}
  |
  | targets
```

7.10 Import, Open, and Include

```
open_import  ::=  {{ Open or import statements }}
```

```
  | open
  | import
  | open import
  | include
  | include import
```

7.11 Lemmas, Assertions, and Theorems

```
lemma_typ ::= {{ Types of Lemmata }}
  | assert
  | lemma
  | theorem

lemma_decl ::=   {{ Lemmata and Tests }}
  | lemma_typ targets_opt x : exp
```

7.12 Unused?

```
dexp ::=  {{ declaration field-expressions }}
  | name_s = string l
  | format = string l
  | arguments = exp1 ... expn l
  | targuments = texp1 ... texpn l

declare_arg ::=   {{ arguments to a declaration }}
  | string
  | <| dexp1 ; ... ; dexpn semi_opt l |>
```

7.13 Target Representation Declarations

```
component  ::= {{ components }}
  | module
  | function
  | type
  | field

termination_setting ::= {{ termination settings }}
  | automatic
  | manual

exhaustivity_setting  ::= {{ exhaustivity settings }}
  | exhaustive
  | inexhaustive

elim_opt ::=  {{ optional terms used as eliminators for pattern matching }}
  |
  | id

fixity_decl ::= {{ fixity declarations for infix identifiers }}
  | right_assoc nat
  | left_assoc  nat
  | non_assoc   nat
  |

target_rep_rhs ::=  {{ right hand side of a target representation declaration }}
  | infix fixity_decl backtick_string
```

```
  | exp
  | typ
  | special string exp1 ... expn
  |

target_rep_lhs  ::= {{ left hand side of a target representation declaration }}
  | target_rep component id x1 .. xn
  | target_rep component id tnvars

declare_def  ::=   {{ declarations }}
  | declare targets_opt compile_message id = string                    {{ compile_message_decl
  | declare targets_opt rename module = x                              {{ rename_current_module_decl
  | declare targets_opt rename component id = x                        {{ rename_decl
  | declare targets_opt ascii_rep component id = backtick_string       {{ ascii_rep_decl
  | declare target target_rep target_rep_lhs = target_rep_rhs          {{ target_rep_decl
  | declare set_flag x1 = x2                                           {{ set_flag_decl
  | declare targets_opt termination_argument id = termination_setting  {{ termination_argument_decl
  | declare targets_opt pattern_match exhaustivity_setting id tnvars = [ id1 ; ... ; idn semi_opt ]
```

7.14 Value Definitions

```
val_def ::=  {{ Value definitions }}
  | let targets_opt letbind                         {{ Non-recursive value definitions }}
  | let rec targets_opt funcl1 and ... and funcln   {{ Recursive function definitions }}
  | let inline targets_opt letbind                  {{ Function definitions to be inlined }}
  | let lem_transform targets_opt letbind           {{ Function definitions to be transformed }}

ascii_opt ::=   {{ an optional ascii representation }}
  |
  | [ backtick_string ]
```

7.15 Class and Instance Declarations

```
instance_decl ::=   {{ is it an instance or the default instance? }}
  | instance
  | default_instance

class_decl ::=   {{ is a class an inlined one? }}
  | class
  | class inline
```

7.16 Value Type Specifications

```
val_spec ::=  {{ Value type specifications }}
  | val x ascii_opt : typschm
```

7.17 Top-level Definitions

```
semisemi_opt ::=  {{ Optional double-semi-colon }}
  |
  | ;;

def  ::=  {{ Top-level definitions }}
  | type td1 and ... and tdn                               {{ Type definitions }}
  | val_spec                                               {{ Top-level type constraints }}
```

```
  | val_def                                                    {{ Value definitions }}
  | lemma_decl                                                 {{ Lemmata }}
  | module x = struct defs end                                 {{ Module definitions }}
  | module x = id                                              {{ Module renamings }}
  | open_import id1 ... idn                                    {{ importing and/or opening modules
  | open_import targets_opt backtick_string1 ... backtick_stringn
      {{ importing and/or opening only for a target / it does not influence the Lem state }}
  | indreln targets_opt indreln_name1 and ... and indreln_namei rule1 and ... and rulen
      {{ Inductively defined relations }}
  | class_decl ( x tnvar ) val targets_opt1 x1 ascii_opt1 : typ1 l1 ... val targets_optn xn ascii_op
      {{ Typeclass definitions }}
  | instance_decl instschm val_def1 l1 ... val_defn ln end
      {{ Typeclass instantiations }}
  | declare_def                                                {{ modify Lem behaviour }}


defs ::=  {{ Definition sequences }}
  | def1 semisemi_opt1 .. defn semisemi_optn
```

8 Linking to existing Backend Libraries

Lem allows one to use existing backend libraries from your Lem-development. This is done by target-specific imports and target-specific representations.

8.1 Target specific imports

Before using an existing target library, it usually needs to be loaded. There are target-specific `open`, `import` and `include` statements that allow instructing Lem to generate output that loads an existing backend library. These statements are very similar to the corresponding statements for Lem modules. However, they allow specifying targets and the modules are quoted. While - generalising the Lem staments - many possible combinations are allowed, in practice only `open import` statements are used.

As an example, consider Lem's relation library. Some of its existing definitions should be mapped to HOL functions defined in the HOL4 theory `set_relation`. To load this theory for HOL, Lem's relation library contains the statement

```
open import {hol} `set_relationTheory`
```

8.2 Simple Target Representations

A `target_rep` declaration allows specifing which *existing* target function should be used for a Lem-specific one. The boolean conjunction operator is for example mapped as follows

```
val not : bool -> bool
let not b = match b with
  | true -> false
  | false -> true
end

declare ocaml    target_rep function not = `not`
declare hol      target_rep function not x = `~` x
declare isabelle target_rep function not x = `\<not>` x
declare coq      target_rep function not = `negb`

declare html     target_rep function not = `&not;`
declare tex      target_rep function not b = `$\neg$` b
```

- definition + target rep useful for documentation
- however, only val-spec + target rep needed
- definition gets turned into lemma when target-rep is present
- rhs of target_reps can be expression containing quotations
- if arguments are given, they have to be variables
- if not all arguments are present, eta-expansion is used
- eta-expansion necessary sometimes, see `not` for Isabelle and HOL

8.3 Target Representations of Types

```
type map 'k 'v
declare ocaml    target_rep type map = `Pmap.map`
declare isabelle target_rep type map = `Map.map`
declare hol      target_rep type map = `fmap`
declare coq      target_rep type map = `fmap`
```

8.4 Infix Operations

```
val (&&) [`and`] : bool -> bool -> bool
let (&&) b1 b2 = match (b1, b2) with
  | (true, true) -> true
  | _ -> false
end

declare hol      target_rep function (&&) = infix `/\`
declare ocaml    target_rep function (&&) = infix `&&`
declare isabelle target_rep function (&&) = infix `\<and>`
declare coq      target_rep function (&&) = infix `&&`
declare html     target_rep function (&&) = infix `&and;`
declare tex      target_rep function (&&) = infix `$\wedge$`
```

8.5 Special Target Representations

```
class ( NumPow 'a )
  val ( ** ) [`numPow`] : 'a -> nat -> 'a
end
declare tex target_rep function numPow n m = special "{%e}^{%e}" n m
```

9 Type classes

9.1 Type class for Equality

The Lem equality is translated using the type class `Eq`, defined in `library/basic_classes.lem`, with operations `=` (`isEqual`) and `<>` (`isInequal`) for equality and inequality.

For HOL and Isabelle, this is always mapped to their standard equalities.

However, for OCaml, the default mapping to OCaml equality is not always appropriate, and if a Lem development uses equality at any types at which it is not, the user must provide a suitable instantiation for type class `Eq`. For example, one needs to instantiate `Eq` for any inductively defined types that make use of `natural` (mapped to OCaml big integers) or sets or maps, or (recursively) any other types that do.

9.2 Type classes for Sets and Maps

Sets and Maps require comparison operations in OCaml and Coq. This is provided via type classes `SetType` and `MapType`, introduced in `library/basic_classes.lem`; the former has a single method `setElemCompare`. The default OCaml instantiation of `SetType` is with OCaml's `compare`, but if the user constructs sets of types containing any tuples, records, or user-defined inductive types, those types must also have an instance declaration for `SetType` with a suitable comparison function. If this is omitted, the default will be used and there may be a run-time error as the equality test will be incorrect. `MapType` uses `SetType` as default implementation.

For example, for a simple inductive type:

```
type memory_order = Atomic | NA
```

one can make it an instance of `SetType` as follows, as here the default OCaml `compare` and the theorem prover equalities will be correct.

```
instance (SetType memory_order)
  let setElemCompare = defaultCompare
end
```

For a more complex inductive type such as the following, with recursion through a set and pair constructor:

```
type tree 'a =
  | Node of set ('a * tree 'a)
```

one can define an equality function making use of the underlying `setCompareBy` comparison on sets:

```
val treeCompare : forall 'a .
  ('a -> 'a -> ordering) -> (tree 'a) -> (tree 'a) -> ordering
let rec treeCompare cmpa (Node xs) (Node ys) =
  setCompareBy (pairCompare cmpa (treeCompare cmpa)) xs ys
```

and make the `tree` type constructor instantiate `SetType` as follows:

```
instance forall 'a. SetType 'a => (SetType (tree 'a))
  let setElemCompare = treeCompare setElemCompare
end
```

Tuple types up to a certain size are made an instance of `SetType` in `basic_classes.lem`; if one uses sets or maps of wider tuples, they must also be made instances following the same pattern, otherwise Lem will generate incorrect code.

9.3 Other Standard Library Type Classes

The standard library defines several other type classes. In `library/basic_classes.lem` we have, in addition to `Eq` and `SetType`:

- `Ord` for total linear orders with comparison operations
- `OrdMaxMin` extending `Ord` with max and min

In `map.lem` we have `MapKeyType`.

In `num.lem` there are various numeric types and type classes for the operations that they each may or may not support:

- NumNegate
- NumAdd
- NumMinus
- NumMult
- NumPow
- NumDivision
- NumIntegerDivision
- NumRemainder
- NumSucc
- NumPred

In `word.lem` there is a type class `Word` of machine words, bitwise logical operations, and conversions to and from lists of booleans.

10 Refactoring

- backend `lem` used for refactoring
- use command-line option `-lem`
- file `myfile.lem` translated to `myfile-processed.lem`
- compare files, modify `myfile-processed.lem`, when ready rename back to `myfile.lem`

10.1 Types

```
declare {lem} rename type nat = NAT
declare lem target_rep type set 'a = `SET` 'a 'a
```

10.2 Functions / Fields

```
declare {lem} rename function my_fun = my_fun'
declare lem target_rep function my_fun x y z = `my_fun'` (x, y) true z
```

Also possible `lem_transform`. However, better use `declare lem target_rep` instead of `lem_transform`. TODO: remove `lem_transform`

```
let lem_transform my_fun x y z = other_existing_function y z
```

10.3 Modules

```
declare {lem} rename module my_mod = my_mod_new_name
```