Lean 2 Quick Reference

Jeremy Avigad, Leonardo de Moura, Soonho Kong

Version d0dd6d0, updated at 2017-01-30 19:53:44 -0500

Quick Reference

Note that this quick reference guide describes Lean 2 only.

Displaying Information

```
check <expr>                : check the type of an expression
eval <expr>                 : evaluate expression
print <id>                  : print information about <id>
print notation              : display all notation
print notation <tokens>     : display notation using any of the tokens
print axioms                : display assumed axioms
print options               : display options set by user or emacs mode
print prefix <namespace>    : display all declarations in the namespace
print coercions             : display all coercions
print coercions <source>    : display only the coercions from <source>
print classes               : display all classes
print instances <class name> : display all instances of the given class
print fields <structure>    : display all "fields" of a structure
print metaclasses           : show kinds of metadata stored in a namespace
help commands               : display all available commands
help options                : display all available options
```

Common Options

You can change an option by typing `set_option <option> <value>`. The `<option>` field supports TAB-completion. You can see an explanation of all options using `help options`.

```
pp.implicit        : display implicit arguments
pp.universes       : display universe variables
pp.coercions       : show coercions
pp.notation        : display output using defined notations
pp.abbreviations   : display output using defined abbreviations
pp.full_names      : use full names for identifiers
pp.all             : disable notations, implicit arguments, full names,
                     universe parameters and coercions
pp.beta            : beta reduce terms before displaying them
```

```
pp.max_depth       : maximum expression depth
pp.max_steps       : maximum steps for printing expression
pp.private_names   : show internal name assigned to private definitions and theorems
pp.metavar_args    : show arguments to metavariables
pp.numerals        : print output as numerals
```

Attributes

These can generally be declared with a `definition` or `theorem`, or using the `attribute` or `local attribute` commands.

Example: `local attribute nat.add nat.mul [reducible]`.

```
reducible       : unfold at any time during elaboration if necessary
quasireducible  : unfold during higher order unification,
                  but not during type class resolution
semireducible   : unfold when performance is not critical
irreducible     : avoid unfolding during elaboration
coercion        : use as a coercion between types
class           : type class declaration
instance        : type class instance
priority <num>  : add a priority to an instance or notation
parsing-only    : use notation only for input
unfold <num>    : if the argument at position <num> is marked with [constructor]
                  unfold this and that argument (for iota reduction)
constructor     : see unfold <num>
unfold-full     : unfold definition when fully applied
recursor        : user-defined recursor/eliminator, used for the induction tactic
recursor <num>  : user-defined non-dependent recursor/eliminator
                  where <num> is the position of the major premise
refl            : reflexivity lemma, used for calc-expressions, tactics and simplifier
symm            : symmetry lemma, used for calc-expressions, tactics and simplifier
trans           : transitivity lemma, used for calc-expressions, tactics and simplifier
subst           : substitution lemma, used for calc-expressions and simplifier
```

Proof Elements

Term Mode

```
take, assume     : syntactic sugar for lambda
let              : introduce local definitions
have             : introduce auxiliary fact (opaque, in the body)
assert           : like "have", but visible to tactics
show             : make result type explicit
suffices         : show that the goal follows from this fact
obtain ..., from : destruct structures such as exists, sigma, ...
match ... with   : introduce proof or definition by cases
proof ... qed    : introduce a proof or definition block, elaborated separately
```

The keywords `have` and `assert` can be anonymous, which is to say, they can be used without giving a label to the hypothesis. The corresponding element of the context can then be referred to using the keyword `this` until another anonymous element is introduced, or by enclosing the assertion in backticks. To avoid a syntactic ambiguity, the keyword `suppose` is used instead of `assume` to introduce an anonymous assumption.

One can also use anonymous binders (like `lambda`, `take`, `obtain`, etc.) by enclosing the type in backticks, as in `λ `nat`, `nat` + 1`. This introduces a variable of the given type in the context with a hidden name.

Tactic Mode

At any point in a proof or definition you can switch to tactic mode and apply tactics to finish that part of the proof or definition.

```
begin ... end   : enter tactic mode, and blocking mechanism within tactic mode
{ ... }         : blocking mechanism within tactic mode
by ...          : enter tactic mode, can only execute a single tactic
begin+; by+     : same as =begin= and =by=, but make local results available
have            : as in term mode (enters term mode), but visible to tactics
show            : as in term mode (enters term mode)
match ... with  : as in term mode (enters term mode)
let             : introduce abbreviation (not visible in the context)
note            : introduce local fact (opaque, in the body)
```

Normally, entering tactic mode will make declarations in the local context given by "have"-expressions unavailable. The annotations `begin+` and `by+` make all these declarations available.

Sectioning Mechanisms

```
namespace <id> ... end <id>  : begin / end namespace
section ... end              : begin / end section
section <id> .... end <id>   : begin / end section

variable (var : type)        : introduce variable where needed
variable {var : type}        : introduce implicit variable where needed
variable {{var : type}}      : introduce implicit variable where needed,
                               which is not maximally inserted
variable [var : type]        : introduce class inference variable where needed
variable {var} (var) [var]   : change the bracket type of an existing variable
parameter                    : introduce variable, fixed within the section
include                      : include variable in subsequent definitions
omit                         : undo "include"
```

Tactics

We say a tactic is more "aggressive" when it uses a more expensive (and complete) unification algorithm, and/or unfolds more aggressively definitions.

General tactics

```
apply <expr>       : apply a theorem to the goal, create subgoals for non-dependent premises
fapply <expr>      : like apply, but create subgoals also for dependent premises that were
                     not assigned by unification procedure
eapply <expr>      : like apply, but used for applying recursor-like definitions
exact <expr>       : apply and close goal, or fail
rexact <expr>      : relaxed (and more expensive) version of exact
                     (this will fully elaborate <expr> before trying to match it to the goal)
refine <expr>      : like exact, but creates subgoals for unresolved subgoals

intro <ids>        : introduce multiple variables or hypotheses
intros <ids>       : same as intro <ids>
intro              : let Lean choose a name
intros             : introduce variables as long as the goal reduces to a function type
                     and let Lean choose the names

rename <id> <id>   : rename a variable or hypothesis
generalize <expr> : generalize an expression
clear <ids>        : remove variables or hypotheses
revert <ids>       : move variables or hypotheses into the goal
assumption         : try to close a goal with something in the context
eassumption        : a more aggressive ("expensive") form of assumption
```

Equational reasoning

```
esimp                  : simplify expressions (by evaluation/normalization) in goal
esimp at <id>          : simplify hypothesis in context
esimp at *             : simplify everything
esimp [<ids>]          : unfold definitions and simplify expressions in goal
esimp [<ids>] at <id> : unfold definitions and simplify hypothesis in context
esimp [<ids>] at *     : unfold definitions and simplify everything
unfold <id>            : similar to (esimp <id>)
fold <expr>            : unfolds <expr>, search for convertible term in the
                         goal, and replace it with <expr>

beta                   : beta reduce goal
whnf                   : put goal in weak head normal form
change <expr>          : change the goal to <expr> if it is convertible to <expr>

rewrite <rule>         : apply a rewrite rule (see below)
rewrite [<rules>]      : apply a sequence of rewrite rules (see below)
krewrite               : using keyed rewriting, matches any subterm
                         with the same head as the rewrite rule
xrewrite               : a more aggressive form of rewrite

subst <id>             : substitute a variable defined in the context, and clear hypothesis and
                         variable
substvars              : substitute all variables in the context
```

Rewrite rules You can combine rewrite rules from different groups in the following order, starting with the innermost:

```
e                : match left-hand-side of equation e to a goal subterm,
                   then replace every occurence with right-hand-side
{p}e             : apply e only where pattern p (which may contain placeholders) matches

n t              : apply t exactly n times
n>t              : apply t at most n times
*t               : apply t zero or more times (up to rewriter.max_iter)
+t               : apply t one or more times

-t               : apply t in reverse direction

↑id              : unfold id
↑[ids]           : unfold ids
↓id              : fold id
►expr            : reduce goal to expression expr
►*               : equivalent to esimp

t at {i, ...}    : apply t only at numbered occurences
t at -{i, ...}   : apply t only at all but the numbered occurences
t at H           : apply t at hypothesis H
t at H {i, ...}  : apply t only at numbered occurences in H
t at H -{i, ...} : apply t only at all but the numbered occurences in H
t at * ⊢         : apply t at all hypotheses
t at *           : apply t at the goal and all hypotheses
```

Induction and cases

```
cases <expr>                   : decompose an element of an inductive type
cases <expr> with <ids>        : name newly introduced variables as specified by <ids>
induction <expr> (with <ids>)  : use induction
induction <expr> using <def>   : use the definition <def> to apply induction
constructor                    : construct an element of an inductive type by applying the
                                 first constructor that succeeds
constructor <i>                : construct an element of an inductive type by applying the
                                 ith-constructor
fconstructor                   : construct an element of an inductive type by (fapply)ing the
                                 first constructor that succeeds
fconstructor <i>               : construct an element of an inductive type by (fapply)ing the
                                 ith-constructor
injection <id> (with <ids>)    : use injectivity of constructors at specified hypothesis
split                          : equivalent to (constructor 1), only applicable to inductive
                                 datatypes with a single constructor (e.g. and introduction)
left                           : equivalent to (constructor 1), only applicable to inductive
                                 datatypes with two constructors (e.g. left or introduction)
right                          : equivalent to (constructor 2), only applicable to inductive
                                 datatypes with two constructors (e.g. right or introduction)
existsi <expr>                 : similar to (constructor 1) but we can provide an argument,
                                 useful for performing exists/sigma introduction
```

Special-purpose tactics

```
contradiction       : close contradictory goal
exfalso             : implements the "ex falso quodlibet" logical principle
congruence          : solve goals of the form (f a_1 ... a_n = f' b_1 ... b_n) by congruence
```

```
reflexivity          : reflexivity of equality (or any relation marked with attribute refl)
symmetry             : symmetry of equality (or any relation marked with attribute symm)
transitivity <expr> : transitivity of equality (or any relation marked with attribute trans)
trivial              : apply true introduction
```

Combinators

```
and_then <tac1> <tac2> (notation: <tac1> ; <tac2>)
                         : execute <tac1> and then execute <tac2>, backtracking when needed
                           (aka sequential composition)
or_else <tac1> <tac2> (notation: (<tac1> | <tac2>))
                         : execute <tac1> if it fails, execute <tac2>
<tac1>: <tac2>           : apply <tac1> and then apply <tac2> to all subgoals generated by <tac1>
par <tac1> <tac2>        : execute <tac1> and <tac2> in parallel
fixpoint (fun t, <tac>)  : fixpoint tactic, <tac> may refer to t
try <tac>                : execute <tac>, if it fails do nothing
repeat <tac>             : repeat <tac> zero or more times (until it fails)
repeat1 <tac>            : like (repeat <tac>), but fails if <tac> does not succeed at least
                           once
at_most <num> <tac>      : like (repeat <tac>), but execute <tac> at most <num> times
do <num> <tac>           : execute <tac> exactly <num> times
determ <tac>             : discard all but the first proof state produced by <tac>
discard <tac> <num>      : discard the first <num> proof-states produced by <tac>
```

Goal management

```
focus_at <tac> <i>   : execute <tac> to the ith-goal, and fail if it is not solved
focus  <tac>         : equivalent to (focus_at <tac> 0)
rotate_left  <num>   : rotate goals to the left <num> times
rorate_right <num>   : rotate goals to the right <num> times
rotate <num>         : equivalent to (rotate_left <num>)
all_goals <tac>      : execute <tac> to all goals in the current proof state
fail                 : tactic that always fails
id                   : tactic that does nothing and always succeeds
now                  : fail if there are unsolved goals
```

Information and debugging

```
state                           : display the current proof state
check_expr <expr>               : display the type of the given expression in the current goal
trace <string>                  : display the current string
with_options [<options>] <tac> : execute a single tactic with different options
                                  (<options> is a comma-separated list)
```

Emacs Lean-mode commands

Flycheck commands

```
C-c ! n    : next error
C-c ! p    : previous error
C-c ! l    : list errors
C-c C-x    : execute Lean (in stand-alone mode)
```

Lean-specific commands

```
C-c C-k    : show how to enter unicode symbol
C-c C-o    : set Lean options
C-c C-e    : execute Lean command
C-c C-r    : restart Lean process
C-c C-p    : print the definition of the identifier under the cursor
             in a new buffer
C-c C-g    : show the current goal at a line of a tactic proof, in a
             new buffer
C-c C-f    : fill a placeholder by the printed term in the minibuffer.
             Note: the elaborator might need more information
             to correctly infer the implicit arguments of this term
```

Unicode Symbols

This section lists some of the Unicode symbols that are used in the Lean library, their ASCII equivalents, and the keystrokes that can be used to enter them in the Emacs Lean mode.

Logical symbols

Unicode	Ascii	Emacs
true		
false		
¬	not	`\not`, `\neg`
∧	/\	`\and`
∨	\/	`\or`
→	->	`\to`, `\r`, `\implies`
↔	<->	`\iff`, `\lr`
∀	forall	`\all`
∃	exists	`\ex`
λ	fun	`\l`, `\fun`
≠	~=	`\ne`

Types

Π	Pi	`\Pi`
→	`->`	`\to`, `\r`, `\implies`
Σ	Sigma	`\S`, `\Sigma`
×	prod	`\times`
	sum	`\union`, `\u+`, `\uplus`
ℕ	nat	`\nat`
ℤ	int	`\int`
ℚ	rat	`\rat`
ℝ	real	`\real`

When you open the namespaces `prod` and `sum`, you can use `*` and `+` for the types `prod` and `sum` respectively. To avoid overwriting notation, these have to have the same precedence as the arithmetic operations. If you don't need to use notation for the arithmetic operations, you can obtain lower-precedence versions by opening the namespaces `low_precedence_times` and `low_precedence_plus` respectively.

Greek letters

Unicode	Emacs
α	`\alpha`
β	`\beta`
γ	`\gamma`
…	…

Equality proofs (`open eq.ops`)

Unicode	Ascii	Emacs
$^{-1}$	eq.symm	`\sy`, `\inv`, `\-1`
$\cdot$	eq.trans	`\tr`
▶	eq.subst	`\t`

Symbols for the rewrite tactic

Unicode	Ascii	Emacs
↑	^	`\u`
↓	<d	`\d`

Brackets

Unicode	Ascii	Emacs
⌞t⌟	?(t)	`\cll t \clr`
⦃ t ⦄	{{t}}	`\{{ t \}}`
⟨ t ⟩		`\< t \>`
t		`\<< t \>>`

Set theory

Unicode	Ascii	Emacs
∈	mem	`\in`
∉		`\nin`
∩	inter	`\i`
∪	union	`\un`
⊆	subseteq	`\subeq`

Binary relations

Unicode	Ascii	Emacs
≤	<=	`\le`
≥	>=	`\ge`
|	dvd	`\\|`
≡		`\equiv`
≈		`\eq`

Binary operations

Unicode	Ascii	Emacs
∘	comp	`\comp`