The Lean Reference Manual

Release 3.3.0

Jeremy Avigad, Gabriel Ebner, and Sebastian Ullrich

Sep 06, 2018

CONTENTS

CHAPTER

ONE

USING LEAN

1.1 Using Lean Online

You can run a Javascript version of Lean online in your browser. It is much slower than running a version of Lean installed on your computer, but it provides a convenient way to experiment with the system.

The online version of Lean checks your input continuously. Error messages, warnings, and output appear in the window on the right-hand side. The editor shares a number of features with Visual Studio Code; for example, you can type unicode characters with a backslash, so `\and` yields the unicode symbol for conjunction, and `\a`, `\b`, and `\g` yield the unicode α, β, and γ respectively.

1.2 Using Lean with VSCode

Assuming you have installed Lean and Visual Studio Code, you can add the Lean extension to VSCode by clicking the extension icon in the view bar at left and searching for `lean`. Once you have installed the extension, clicking on `lean` in the extensions panel provides additional information.

With the extension installed, if you create a file with the extension `.lean` and edit it, Lean will check the file continuously as you type. For example, if you type the words `#check id`, the word `#check` is underlined in green to indicate a response from the Lean server. Hovering over the word `#check` displays the response, in this case, the type of the identity function.

1.2.1 Features

VSCode Intellisense suggests completions as you type. These are context sensitive: if you are typing an identifier, it suggests suitable completions for identifiers, after `import`, it suggests modules to import, after `set_option`, it suggests suitable options, and so on.

You can enter Unicode characters with a backslash. For example, `\a` inserts an α. You can see the commands provided by the Lean extension by typing `ctrl-shift-P` on Windows/Linux or `cmd-shift-P` on a Mac, and then typing `lean` into the search bar to filter the list. Typing `ctrl-shift-enter` opens up a message window which shows you error messages, warnings, output, and goal information when in tactic mode.

Typing an underscore in an expression asks Lean to infer a suitable value for the expression and fill it in automatically. In cases where Lean is unable to determine a value for the argument, the underscore is highlighted, and the error message indicates the type of the "hole" that needs to be filled. This can be extremely useful when constructing proofs incrementally. You can start typing a proof sketch, using either `sorry` or an underscore for details you intend to fill in later. Assuming the proof is correct modulo these missing pieces of information, the error message at an unfilled underscore tells you the type of the term you need to construct, typically an assertion you need to justify.

1.2.2 Multi-file Projects

If you want to work on a project with multiple files, use the *package_manager* to set up a project folder, and then use `Open Folder` in the VSCode `File` menu to open the root directory for the project.

1.3 Using Lean with Emacs

Assuming you have installed Lean, Emacs, and the Lean Emacs mode according to the instructions on the Lean Download page, you simply need to create a file with the extension `.lean` and edit it in Emacs. The file will be checked continuously as you type. For example, if you type the words `#check id`, the word `#check` is underlined in green to indicate a response from the Lean server. Hovering over the word `#check` displays the response, in this case, the type of the identity function.

1.3.1 Features

Lean mode uses an Emacs package named *Flycheck*, as evidenced by the letters `FlyC` that appear in the information line. Flycheck offers a number of commands that begin with `C-c !`. For example, `C-c ! n` moves the cursor to the next error, and `C-c ! p` moves the cursor to the previous error. You can get to a help menu that lists these key bindings by clicking on the `FlyC` tag.

It is often inconvenient to have to put the cursor on a highlighted identifier to see an error message or the outcome of a `#print` or `#check` command. The keystrokes `C-c C-n` toggle `Lean-Next-Error` mode, in which the next message (or all the messages that occur on the line that the cursor is on, if there are any) appears in a buffer named `lean-info`. You can position this window anywhere you want using Emacs commands for splitting windows and loading buffers. Pressing `C-c C-n` again toggles the mode off.

As with VSCode, the Emacs mode provides context-sensitive tab completion. Typing an underscore in an expression asks Lean to infer a suitable value for the expression and fill it in automatically. As described in the previous section, this provides a convenient way to construct terms and proof interactively.

If you put your cursor on an identifier and hit `M-.`, Emacs will take you to the identifier's definition, whether it is in the same file, in another file in your project , or in the library. This works even in an autocompletion popup window: if you start typing an identifier, press the tab key, choose a completion from the list of options, and press `M-.`, you are taken to the symbol's definition. If you have Emacs 25 or later, you can then press `M-,` to go back to the original location.

In tactic mode, if you put your cursor on a tactic (or the keyword `begin` or `end`) and type `C-c C-g`, Emacs will show you the goal in the `lean-info` buffer. Here is another useful trick: if you see some notation in a Lean file and you want to know how to enter it from the keyboard, put the cursor on the symbol and type `C-c C-k`.

If for some reason the Lean background process does not seem to be responding (for example, the information line no longer shows you type information), type `C-c C-r` or `M-x lean-server-restart-process`, or choose "restart lean process" from the Lean menu, and with luck that will set things right again.

In Lean, the `#exit` command halts processing of a file abruptly. Inserting an `#exit` therefore prevents Lean from checking the file beyond that point.

Some of the main key bindings are summarized in the table below.

Key	Function
`M-.`	jump to definition in source file (`lean-find-definition`)
`M-,`	return to original position (requires Emacs 25)
`TAB`	tab complete identifier, option, filename, etc. (`lean-tab-indent-or-complete`)
`C-c C-k`	shows the keystroke needed to input the symbol under the cursor
`C-c C-g`	show goal in tactic proof (`lean-show-goal-at-pos`)
`C-c C-x`	execute lean in stand-alone mode (`lean-std-exe`)
`C-c C-n`	toggle next-error-mode: shows next error in dedicated lean-info buffer
`C-c C-r`	restart the lean server
`C-c ! n`	flycheck: go to next error
`C-c ! p`	flycheck: go to previous error
`C-c ! l`	flycheck: show list of errors

1.3.2 Multi-file Projects

If you want to work on a project with multiple files, use the *package_manager* to set up a project folder, and use `Open Folder` in the VSCode `File` menu to open the root directory for the project.

1.4 Using the Package Manager

`leanpkg` is the package manager for the Lean theorem prover. It downloads dependencies and manages what modules you can import in your Lean files.

This section explains the general concepts of `leanpkg`. For more information on a specific `leanpkg` command, execute `leanpkg help <command>` as a command line.

1.4.1 Directory Layout

A Lean package is a directory containing the following items:

- `src`: a directory in which the package's Lean files are stored. Imports from other packages are resolved relative to this directory.
- `leanpkg.toml`: a manifest describing the package name, version, and dependencies. Dependencies can be either local paths or git URLs. Git dependencies are pinned to a specific commit and can be upgraded with *leanpkg upgrade*.
- `leanpkg.path` and `_target/deps`: these items are created by `leanpkg configure` and should not be added to git. They contain the paths to the dependencies on the current machine, and their git checkouts, respectively.

1.4.2 Using Lean in a Lean package

- Running the `lean` command-line tool from a directory inside your package will automatically use the `leanpkg.path` file for import resolution.
- In Emacs, `lean-mode` will automatically start a new Lean server process for each visited package.
- In VSCode, open the package as a folder.

1.4.3 Creating new packages

The `leanpkg new` command creates a new package. You can use `leanpkg add` to add dependencies (or add them manually if you prefer):

```
leanpkg new my_awesome_pkg
cd my_awesome_pkg
leanpkg add leanprover/mathlib
# shorthand for `leanpkg add https://github.com/leanprover/mathlib`
```

You can now add new `.lean` files inside the `src` directory.

1.4.4 Scratch files

It is reasonably common to have thousands of "scratch" files lying around that are not part of a package. Files that are not inside a package themselves can still use dependencies fetched via `leanpkg`. These dependencies are stored in `~/.lean/leanpkg.toml` and can be modified with `leanpkg install`:

```
leanpkg install https://github.com/leanprover/smt2_interface
```

After this, you can use the `smt2_interface` package in all files that do not belong to a package themselves.

For experimenting inside a Lean package, you can use a directory separate from `src`, say `scratch`. Files in this folder will still be able to import the package's Lean modules, but will not interfere with `leanpkg build` etc.

1.4.5 Import resolution

Lean supports two kinds of imports:

```
import theory.set_theory   -- absolute
import .basic              -- relative
```

Relative imports are always relative to the current file name.

Absolute imports are resolved according to the entries in the `leanpkg.path` file. That is, when executing `import theory.set_theory`, Lean looks for a file called `theory/set_theory.lean` in the *src* directories of all (transitive) dependencies as well as the current package.

1.4.6 Format of leanpkg.toml

```
[package]
name = "my_awesome_pkg"
version = "0.1"         # no semantic significance currently
lean_version = "3.3.0"  # optional, prints a warning on mismatch with Lean executable
path = "src"            # hard-coded, will be removed in the future
timeout = 100           # optional, passed to `lean` as `-T` parameter

[dependencies]
# local dependency
demopkg = { path = "relative/path/to/demopkg" }
# git dependency
mathlib =
```

```
{ git = "https://github.com/leanprover/mathlib",
  rev = "62f7883d937861b618ae8bd645ee16ec137dd0bd" }
```

CHAPTER

TWO

LEXICAL STRUCTURE

This section describes the detailed lexical structure of the Lean language. Many readers will want to skip this section on a first reading.

Lean input is processed into a stream of tokens by its scanner, using the UTF-8 encoding. The next token is the longest matching prefix of the remaining input.

```
token  ::=  symbol | command | ident | string | char | numeral |
            decimal | quoted_symbol | doc_comment | mod_doc_comment |
            field_notation
```

Tokens can be separated by the whitespace characters space, tab, line feed, and carriage return, as well as comments. Single-line comments start with `--`, whereas multi-line comments are enclosed by `/-` and `-/` and can be nested.

2.1 Symbols and Commands

Symbols are static tokens that are used in term notations and commands. They can be both keyword-like (e.g. the *have* keyword) or use arbitrary Unicode characters.

Command tokens are static tokens that prefix any top-level declaration or action. They are usually keyword-like, with transitory commands like *#print* prefixed by an additional `#`. The set of built-in commands is listed in the Chapter 5 section.

Users can dynamically extend the sets of both symbols (via the commands listed in Section 2.6) and command tokens (via the *[user_command]* attribute).

2.2 Identifiers

An *atomic identifier*, or *atomic name*, is (roughly) an alphanumeric string that does not begin with a numeral. A (hierarchical) *identifier*, or *name*, consists of one or more atomic names separated by periods.

Parts of atomic names can be escaped by enclosing them in pairs of French double quotes «».

```
def foo.«bar.baz» := 0  -- name parts ["foo", "bar.baz"]
```

```
ident               ::=  atomic_ident | ident "." atomic_ident
atomic_ident        ::=  atomic_ident_start atomic_ident_rest*
```

```
atomic_ident_start  ::=  letterlike | "_" | escaped_ident_part
letterlike          ::=  [a-zA-Z] | greek | coptic | letterlike_symbols
greek               ::=  <[α-ωA-Ω - ] except for [λΠΣ]>
coptic              ::=  [ - ]
letterlike_symbols  ::=  [ - ]
escaped_ident_part  ::=  "«" [^«»\r\n\t]* "»"
atomic_ident_rest   ::=  atomic_ident_start | [0-9' ] | subscript
subscript           ::=  [₀-₉ - ᵢ- ]
```

2.3 String Literals

String literals are enclosed by double quotes ("). They may contain line breaks, which are conserved in the string value.

```
string         ::=  '"' string_item '"'
string_item    ::=  string_char | string_escape
string_char    ::=  [^\\]
string_escape  ::=  "\" ("\" | '"' | "'" | "n" | "t" | "x" hex_char hex_char)
hex_char       ::=  [0-9a-fA-F]
```

2.4 Char Literals

Char literals are enclosed by single quotes (').

```
char  ::=  "'" string_item "'"
```

2.5 Numeric Literals

Numeric literals can be specified in various bases.

```
numeral    ::=  numeral10 | numeral2 | numeral8 | numeral16
numeral10  ::=  [0-9]+
numeral2   ::=  "0" [bB] [0-1]+
numeral8   ::=  "0" [oO] [0-7]+
numeral16  ::=  "0" [xX] hex_char+
```

Decimal literals are currently only being used for some *set_option* values.

```
decimal  ::=  [0-9]+ "." [0-9]+
```

2.6 Quoted Symbols

In a fixed set of commands (*notation*, *local notation*, and *reserve*), symbols (known or unknown) can be quoted by enclosing them in backticks (`). Quoted symbols are used by these commands for registering new notations and symbols.

```
quoted_symbol        ::=   "`" " "* quoted_symbol_start quoted_symbol_rest* " "* "`"
quoted_symbol_start  ::=   [^0-9"\n\t `]
quoted_symbol_rest   ::=   [^"\n\t `]
```

A quoted symbol may contain surrounding whitespace, which is customarily used for pretty printing the symbol and ignored while scanning.

While backticks are not allowed in a user-defined symbol, they are used in some built-in symbols (see *Quotations*), which are accessible outside of the set of commands noted above.

2.7 Doc Comments

A special form of comments, doc comments are used to document modules and declarations.

```
doc_comment      ::=   "/--" ([^-] | "-" [^/])* "-/"
mod_doc_comment  ::=   "/-!" ([^-] | "-" [^/])* "-/"
```

2.8 Field Notation

Trailing field notation tokens are used in expressions such as `(1+1).to_string`. Note that `a.to_string` is a single *identifier*, but may be interpreted as a field notation expression by the parser.

```
field_notation  ::=   "." ([0-9]+ | atomic_ident)
```

CHAPTER

THREE

EXPRESSIONS

3.1 Universes

Every type in Lean is, by definition, an expression of type `Sort u` for some universe level `u`. A universe level is one of the following:

- a natural number, `n`
- a universe variable, `u` (declared with the command `universe` or `universes`)
- an expression `u + n`, where `u` is a universe level and `n` is a natural number
- an expression `max u v`, where `u` and `v` are universes
- an expression `imax u v`, where `u` and `v` are universe levels

The last one denotes the universe level `0` if `v` is `0`, and `max u v` otherwise.

```
universes u v

#check Sort u
#check Sort 5
#check Sort (u + 1)
#check Sort (u + 3)
#check Sort (max u v)
#check Sort (max (u + 3) v)
#check Sort (imax (u + 3) v)
#check Prop
#check Type
```

3.2 Expression Syntax

The set of expressions in Lean is defined inductively as follows:

- `Sort u` : the universe of types at universe level `u`
- `c` : where `c` is an identifier denoting an axiomatically declared constant or a defined object
- `x` : where `x` is a variable in the local context in which the expression is interpreted
- Π `x : α, β` : the type of functions taking an element `x` of α to an element of β, where β is an expression whose type is a `Sort`
- `s t` : the result of applying `s` to `t`, where `s` and `t` are expressions
- λ `x : α, t` : the function mapping any value `x` of type α to `t`, where `t` is an expression

- `let x := t in s` : a local definition, denotes the value of `s` when `x` is replaced by `t`

Every well formed term in Lean has a *type*, which itself is an expression of type `Sort u` for some `u`. The fact that a term `t` has type α is written `t : α`.

For an expression to be well formed, its components have to satisfy certain typing constraints. These, in turn, determine the type of the resulting term, as follows:

- `Sort u : Sort (u + 1)`
- `c : α`, where α is the type that `c` has been declared or defined to have
- `x : α`, where α is the type that `x` has been assigned in the local context where it is interpreted
- `(Π x : α, β) : Sort (imax u v)` where `α : Sort u`, and `β : Sort v` assuming `x : α`
- `s t : β[t/x]` where `s` has type `Π x : α, β` and `t` has type α
- `(λ x : α, t) : Π x : α, β` if `t` has type β whenever `x` has type α
- `(let x := t in s) : β[t/x]` where `t` has type α and `s` has type β assuming `x : α`

`Prop` abbreviates `Sort 0`, `Type` abbreviates `Sort 1`, and `Type u` abbreviates `Sort (u + 1)` when `u` is a universe variable. We say "α is a type" to express `α : Type u` for some `u`, and we say "`p` is a proposition" to express `p : Prop`. Using the *propositions as types* correspondence, given `p : Prop`, we refer to an expression `t : p` as a *proof* of `p`. In contrast, given `α : Type u` for some `u` and `t : α`, we sometimes refer to `t` as *data*.

When the expression β in `Π x : α, β` does not depend on `x`, it can be written $\alpha \to \beta$. As usual, the variable `x` is bound in `Π x : α, β`, `λ x : α, t`, and `let x := t in s`. The expression `∀ x : α, β` is alternative syntax for `Π x : α, β`, and is intended to be used when β is a proposition. An underscore can be used to generate an internal variable in a binder, as in `λ _ : α, t`.

In addition to the elements above, expressions can also contain *metavariables*, that is, temporary placeholders, that are used in the process of constructing terms. They can also contain *macros*, which are used to annotate or abbreviate terms. Terms that are added to the environment contain neither metavariable nor variables, which is to say, they are fully elaborated and make sense in the empty context.

Constants can be declared in various ways, such as by the `constant(s)` and `axiom(s)` keywords, or as the result of an `inductive` or `structure` declaration. Similarly, objects can be defined in various ways, such as using `def`, `theorem`, or the equation compiler. See Chapter 4 for more information.

Writing an expression `(t : α)` forces Lean to elaborate `t` so that it has type α or report an error if it fails.

Lean supports anonymous constructor notation, anonymous projections, and various forms of match syntax, including destructuring λ and `let`. These, as well as notation for common data types (like pairs, lists, and so on) are discussed in Chapter 4 in connection with inductive types.

```
universes u v w

variables (p q : Prop)
variable  (α : Type u)
variable  (β : Type v)
variable  (γ : α → Type w)
variable  (η : α → β → Type w)

constants δ ϵ : Type u
constants cnst : δ
constant  f : δ → ϵ

variables (a : α) (b : β) (c : γ a) (d : δ)

variable  g  : α → β
```

```
variable  h  : Π x : α, γ x
variable  h' : Π x, γ x → δ

#check Sort (u + 3)
#check Prop
#check Π x : α, γ x
#check f cnst
#check λ x, h x
#check λ x, h' x (h x)
#check (λ x, h x) a
#check λ _ : ℕ, 5
#check let x := a in h x

#check Π x y, η x y
#check Π (x : α) (y : β), η x y
#check λ x y, η x y
#check λ (x : α) (y : β), η x y
#check let x := a, y := b in η x y

#check (5 : ℕ)
#check (5 : (λ x, x) ℕ)
#check (5 : ℤ)
```

3.3 Implicit Arguments

When declaring arguments to defined objects in Lean (for example, with `def`, `theorem`, `constant`, `inductive`, or `structure`; see Chapter 4) or when declaring variables and parameters in sections (see Chapter 5), arguments can be annotated as *explicit* or *implicit*. This determines how expressions containing the object are interpreted.

- `(x : α)` : an explicit argument of type α
- `{x : α}` : an implicit argument, eagerly inserted
- ⦃`x : α`⦄ or `{{x : α}}` : an implicit argument, weakly inserted
- `[x : α]` : an implicit argument that should be inferred by type class resolution
- `(x : α := t)` : an optional argument, with default value `t`
- `(x : α . t)` : an implicit argument, to be synthesized by tactic `t`

The name of the variable can be omitted from a class resolution argument, in which case an internal name is generated.

When a function has an explicit argument, you can nonetheless ask Lean's elaborator to infer the argument automatically, by entering it as an underscore (`_`). Conversely, writing `@foo` indicates that all of the arguments to be `foo` are to be given explicitly, independent of how `foo` was declared.

```
universe u

def ex1 (x y z : ℕ) : ℕ := x + y + z

#check ex1 1 2 3

def id1 (α : Type u) (x : α) : α := x

#check id1 nat 3
```

```
#check id1 _ 3

def id2 {α : Type u} (x : α) : α := x

#check id2 3
#check @id2 ℕ 3
#check (id2 : ℕ → ℕ)

def id3 {{α : Type u}} (x : α) : α := x

#check id3 3
#check @id3 ℕ 3
#check (id3 : Π α : Type, α → α)

class cls := (val : ℕ)
instance cls_five : cls := ⟨5⟩

def ex2 [c : cls] : ℕ := c.val

example : ex2 = 5 := rfl

def ex2a [cls] : ℕ := ex2

example : ex2a = 5 := rfl

def ex3 (x : ℕ := 5) := x

#check ex3 2
#check ex3
example : ex3 = 5 := rfl

meta def ex_tac : tactic unit := tactic.refine ``(5)

def ex4 (x : ℕ . ex_tac) := x

example : ex4 = 5 := rfl
```

3.4 Basic Data Types and Assertions

The core library contains a number of basic data types, such as the natural numbers (ℕ, or `nat`), the integers (ℤ), the booleans (`bool`), and common operations on these, as well as the usual logical quantifiers and connectives. Some example are given below. A list of common notations and their precedences can be found in a file in the core library. The core library also contains a number of basic data type constructors. Definitions can also be found the data directory of the core library. For more information, see also Chapter 9.

```
/- numbers -/
section
variables a b c d : ℕ
variables i j k : ℤ

#check a^2 + b^2 + c^2
#check (a + b)^c ≤ d
#check i ∣ j * k
end
```

```
/- booleans -/
section
variables a b c : bool

#check a && (b || c)
end

/- pairs -/
section
variables (a b c : ℕ) (p : ℕ × bool)

#check (1, 2)
#check p.1 * 2
#check p.2 && tt
#check ((1, 2, 3) : ℕ × ℕ × ℕ)
end

/- lists -/
section
variables x y z : ℕ
variables xs ys zs : list ℕ
open list

#check (1 :: xs) ++ (y :: zs) ++ [1,2,3]
#check append (cons 1 xs) (cons y zs)
#check map (λ x, x^2) [1, 2, 3]
end

/- sets -/
section
variables s t u : set ℕ

#check ({1, 2, 3} ∩ s) ∪ ({x | x < 7} ∩ t)
end

/- strings and characters -/
#check "hello world"
#check 'a'

/- assertions -/
#check ∀ a b c n : ℕ,
  a ≠ 0 ∧ b ≠ 0 ∧ c ≠ 0 ∧ n > 2 → a^n + b^n ≠ c^n

def unbounded (f : ℕ → ℕ) : Prop := ∀ M, ∃ n, f n ≥ M
```

3.5 Constructors, Projections, and Matching

Lean's foundation, the *Calculus of Inductive Constructions*, supports the declaration of *inductive types*. Such types can have any number of *constructors*, and an associated *eliminator* (or *recursor*). Inductive types with one constructor, known as *structures*, have *projections*. The full syntax of inductive types is described in Chapter 4, but here we describe some syntactic elements that facilitate their use in expressions.

When Lean can infer the type of an expression and it is an inductive type with one constructor, then one can write ⟨a1, a2, ..., an⟩ to apply the constructor without naming it. For example, ⟨a, b⟩ denotes

`prod.mk a b` in a context where the expression can be inferred to be a pair, and ⟨`h₁, h₂`⟩ denotes `and.intro h₁ h₂` in a context when the expression can be inferred to be a conjunction. The notation will nest constructions automatically, so ⟨`a1, a2, a3`⟩ is interpreted as `prod.mk a1 (prod.mk a2 a3)` when the expression is expected to have a type of the form `α1 × α2 × α3`. (The latter is interpreted as `α1 × (α2 × α3)`, since the product associates to the right.)

Similarly, one can use "dot notation" for projections: one can write `p.fst` and `p.snd` for `prod.fst p` and `prod.snd p` when Lean can infer that `p` is an element of a product, and `h.left` and `h.right` for `and.left h` and `and.right h` when `h` is a conjunction.

The anonymous projector notation can used more generally for any objects defined in a *namespace* (see Chapter 5). For example, if `l` has type `list α` then `l.map f` abbreviates `list.map f l`, in which `l` has been placed at the first argument position where `list.map` expects a `list`.

Finally, for data types with one constructor, one destruct an element by pattern matching using the `let` and `assume` constructs, as in the examples below. Internally, these are interpreted using the `match` construct, which is in turn compiled down for the eliminator for the inductive type, as described in Chapter 4.

```
universes u v
variables {α : Type u} {β : Type v}

def p : ℕ × ℤ := ⟨1, 2⟩
#check p.fst
#check p.snd

def p' : ℕ × ℤ × bool := ⟨1, 2, tt⟩
#check p'.fst
#check p'.snd.fst
#check p'.snd.snd

def swap_pair (p : α × β) : β × α :=
⟨p.snd, p.fst⟩

theorem swap_conj {a b : Prop} (h : a ∧ b) : b ∧ a :=
⟨h.right, h.left⟩

#check [1, 2, 3].append [2, 3, 4]
#check [1, 2, 3].map (λ x, x^2)

example (p q : Prop) : p ∧ q → q ∧ p :=
λ h, ⟨h.right, h.left⟩

def swap_pair' (p : α × β) : β × α :=
let (x, y) := p in (y, x)

theorem swap_conj' {a b : Prop} (h : a ∧ b) : b ∧ a :=
let ⟨ha, hb⟩ := h in ⟨hb, ha⟩

def swap_pair'' : α × β → β × α :=
λ ⟨x, y⟩, (y, x)

theorem swap_conj'' {a b : Prop} : a ∧ b → b ∧ a :=
assume ⟨ha, hb⟩, ⟨hb, ha⟩
```

3.6 Structured Proofs

Syntactic sugar is provided for writing structured proof terms:

- `assume h : p, t` is sugar for `λ h : p, t`
- `have h : p, from s, t` is sugar for `(λ h : p, t) s`
- `suffices h : p, from s, t` is sugar for `(λ h : p, s) t`
- `show p, t` is sugar for `(t : p)`

As with λ, multiple variables can be bound with `assume`, and types can be omitted when they can be inferred by Lean. Lean also allows the syntax `assume : p, t`, which gives the assumption the name `this` in the local context. Similarly, Lean recognizes the variants `have p, from s, t` and `suffices p, from s, t`, which use the name `this` for the new hypothesis.

The notation `<p>` is notation for `(by assumption : p)`, and can therefore be used to apply hypotheses in the local context.

As noted in Section 3.5, anonymous constructors and projections and match syntax can be used in proofs just as in expressions that denote data.

```
example (p q r : Prop) : p → (q ∧ r) → p ∧ q :=
assume h₁ : p,
assume h₂ : q ∧ r,
have h₃ : q, from and.left h₂,
show p ∧ q, from and.intro h₁ h₃

example (p q r : Prop) : p → (q ∧ r) → p ∧ q :=
assume : p,
assume : q ∧ r,
have q, from and.left this,
show p ∧ q, from and.intro <p> this

example (p q r : Prop) : p → (q ∧ r) → p ∧ q :=
assume h₁ : p,
assume h₂ : q ∧ r,
suffices h₃ : q, from and.intro h₁ h₃,
show q, from and.left h₂
```

Lean also supports a calculational environment, which is introduced with the keyword `calc`. The syntax is as follows:

```
calc
  <expr>_0  'op_1'  <expr>_1  ':'  <proof>_1
    '...'   'op_2'  <expr>_2  ':'  <proof>_2
     ...
    '...'   'op_n'  <expr>_n  ':'  <proof>_n
```

Each `<proof>_i` is a proof for `<expr>_{i-1} op_i <expr>_i`.

Here is an example:

```
variables (a b c d e : ℕ)
variable h1 : a = b
variable h2 : b = c + 1
variable h3 : c = d
variable h4 : e = 1 + d
```

```
theorem T : a = e :=
calc
  a     = b      : h1
    ... = c + 1  : h2
    ... = d + 1  : congr_arg _ h3
    ... = 1 + d  : add_comm d (1 : ℕ)
    ... =  e     : eq.symm h4
```

The style of writing proofs is most effective when it is used in conjunction with the `simp` and `rewrite` tactics.

3.7 Computation

Two expressions that differ up to a renaming of their bound variables are said to be *α-equivalent*, and are treated as syntactically equivalent by Lean.

Every expression in Lean has a natural computational interpretation, unless it involves classical elements that block computation, as described in the next section. The system recognizes the following notions of *reduction*:

- *β-reduction* : An expression `(λ x, t) s` β-reduces to `t[s/x]`, that is, the result of replacing `x` by `s` in `t`.
- *ζ-reduction* : An expression `let x := s in t` ζ-reduces to `t[s/x]`.
- *δ-reduction* : If `c` is a defined constant with definition `t`, then `c` δ-reduces to to `t`.
- *ι-reduction* : When a function defined by recursion on an inductive type is applied to an element given by an explicit constructor, the result ι-reduces to the specified function value, as described in Section 4.4.

The reduction relation is transitive, which is to say, is `s` reduces to `s'` and `t` reduces to `t'`, then `s t` reduces to `s' t'`, `λ x, s` reduces to `λ x, s'`, and so on. If `s` and `t` reduce to a common term, they are said to be *definitionally equal*. Definitional equality is defined to be the smallest equivalence relation that satisfies all these properties and also includes α-equivalence and the following two relations:

- *η-equivalence* : An expression `(λx, t x)` is η-equivalent to `t`, assuming `x` does not occur in `t`.
- *proof irrelevance* : If `p : Prop`, `s : p`, and `t : p`, then `s` and `t` are considered to be equivalent.

This last fact reflects the intuition that once we have proved a proposition `p`, we only care that is has been proved; the proof does nothing more than witness the fact that `p` is true.

Definitional equality is a strong notion of equalty of values. Lean's logical foundations sanction treating definitionally equal terms as being the same when checking that a term is well-typed and/or that it has a given type.

The reduction relation is believed to be strongly normalizing, which is to say, every sequence of reductions applied to a term will eventually terminate. The property guarantees that Lean's type-checking algorithm terminates, at least in principle. The consistency of Lean and its soundness with respect to set-theoretic semantics do not depend on either of these properties.

Lean provides two commands to compute with expressions:

- `#reduce t` : use the kernel type-checking procedures to carry out reductions on `t` until no more reductions are possible, and show the result
- `#eval t` : evaluate `t` using a fast bytecode evaluator, and show the result

Every computable definition in Lean is compiled to bytecode at definition time. Bytecode evaluation is more liberal than kernel evaluation: types and all propositional information are erased, and functions are

evaluated using a stack-based virtual machine. As a result, `#eval` is more efficient than `#reduce`, and can be used to execute complex programs. In contrast, `#reduce` is designed to be small and reliable, and to produce type-correct terms at each step. Bytecode is never used in type checking, so as far as soundness and consistency are concerned, only kernel reduction is part of the trusted computing base.

```
#reduce (λ x, x + 3) 5
#eval   (λ x, x + 3) 5

#reduce let x := 5 in x + 3
#eval   let x := 5 in x + 3

def f x := x + 3

#reduce f 5
#eval   f 5

#reduce @nat.rec (λ n, ℕ) (0 : ℕ)
                 (λ n recval : ℕ, recval + n + 1) (5 : ℕ)
#eval   @nat.rec (λ n, ℕ) (0 : ℕ)
                 (λ n recval : ℕ, recval + n + 1) (5 : ℕ)

def g : ℕ → ℕ
| 0     := 0
| (n+1) := g n + n + 1

#reduce g 5
#eval   g 5

#eval   g 50000

example : (λ x, x + 3) 5 = 8 := rfl
example : (λ x, f x) = f := rfl
example (p : Prop) (h₁ h₂ : p) : h₁ = h₂ := rfl
```

Note: the combination of proof irrelevance and singleton `Prop` elimination in ι-reduction renders the ideal version of definitional equality, as described above, undecidable. Lean's procedure for checking definitional equality is only an approximation to the ideal. It is not transitive, as illustrated by the example below. Once again, this does not compromise the consistency or soundness of Lean; it only means that Lean is more conservative in the terms it recognizes as well typed, and this does not cause problems in practice. Singleton elimination will be discussed in greater detail in Section 4.4.

```
def R (x y : unit) := false
def accrec := @acc.rec unit R (λ_, unit) (λ _ a ih, ()) ()
example (h) : accrec h = accrec (acc.intro _ (λ y, acc.inv h)) :=
              rfl
example (h) : accrec (acc.intro _ (λ y, acc.inv h)) = () := rfl
example (h) : accrec h = () := sorry   -- rfl fails
```

3.8 Axioms

Lean's foundational framework consists of:

- type universes and dependent function types, as described above
- inductive definitions, as described in Section 4.4 and Section 4.5.

In addition, the core library defines (and trusts) the following axiomatic extensions:

- propositional extensionality:

```
axiom propext {a b : Prop} : (a ↔ b) → a = b
```

- quotients:

```
universes u v

constant quot      : Π {α : Sort u}, (α → α → Prop) → Sort u

constant quot.mk   : Π {α : Sort u} (r : α → α → Prop),
                     α → quot r

axiom    quot.ind  : ∀ {α : Sort u} {r : α → α → Prop}
                       {β : quot r → Prop},
                     (∀ a, β (quot.mk r a)) →
                       ∀ (q : quot r), β q

constant quot.lift : Π {α : Sort u} {r : α → α → Prop}
                       {β : Sort u} (f : α → β),
                     (∀ a b, r a b → f a = f b) → quot r → β

axiom quot.sound   : ∀ {α : Type u} {r : α → α → Prop}
                       {a b : α},
                     r a b → quot.mk r a = quot.mk r b
```

`quot r` represents the quotient of α by the smallest equivalence relation containing `r`. `quot.mk` and `quot.lift` satisfy the following computation rule:

```
quot.lift f h (quot.mk r a) = f a
```

- choice:

```
axiom choice {α : Sort u} : nonempty α → α
```

Here `nonempty` α is defined as follows:

```
class inductive nonempty (α : Sort u) : Prop
| intro : α → nonempty
```

It is equivalent to `∃ x : α, true`.

The quotient construction implies function extensionality. The `choice` principle, in conjunction with the others, makes the axiomatic foundation classical; in particular, it implies the law of the excluded middle and propositional decidability. Functions that make use of `choice` to produce data are incompatible with a computational interpretation, and do not produce bytecode. They have to be declared `noncomputable`.

For metaprogramming purposes, Lean also allows the definition of objects which stand outside the object language. These are denoted with the `meta` keyword, as described in Chapter 7.

CHAPTER

FOUR

DECLARATIONS

4.1 Declaration Names

A declaration name is a *hierarchical identifier* that is interpreted relative to the current namespace as well as (during lookup) to the set of open namespaces.

```
namespace a
  constant b.c : ℕ
  #print b.c -- constant a.b.c : ℕ
end a

#print a.b.c -- constant a.b.c : ℕ
open a
#print b.c -- constant a.b.c : ℕ
```

Declaration names starting with an underscore are reserved for internal use. Names starting with the special atomic name `_root_` are interpreted as absolute names.

```
constant a : ℕ
namespace a
  constant a : ℤ
  #print _root_.a -- constant a : ℕ
  #print a.a      -- constant a.a : ℤ
end a
```

4.2 Contexts and Telescopes

When processing user input, Lean first parses text to a raw expression format. It then uses background information and type constants to disambiguate overloaded symbols and infer implicit arguments, resulting in a fully-formed expression. This process is known as *elaboration*.

As hinted in Section 3.2, expressions are parsed and elaborated with respect to an *environment* and a *local context*. Roughly speaking, an environment represents the state of Lean at the point where an expression is parsed, including previously declared axioms, constants, definitions, and theorems. In a given environment, a *local context* consists of a sequence $(a_1 : \alpha_1)\ (a_2 : \alpha_2)\ \ldots\ (a_n : \alpha_n)$ where each a_i is a name denoting a local constant and each α_i is an expression of type `Sort u` for some `u` which can involve elements of the environment and the local constants a_j for `j < i`.

Intuitively, a local context is a list of variables that are held constant while an expression is being elaborated. Consider the following example:

```
def f (a b : ℕ) : ℕ → ℕ := λ c, a + (b + c)
```

Here the expression `λ c, a + (b + c)` is elaborated in the context `(a : ℕ) (b : ℕ)` and the expression `a + (b + c)` is elaborated in the context `(a : ℕ) (b : ℕ) (c : ℕ)`. If you replace the expression `a + (b + c)` with an underscore, the error message from Lean will include the current *goal*:

```
a b c : ℕ
⊢ ℕ
```

Here `a b c : ℕ` indicates the local context, and the second ℕ indicates the expected type of the result.

A *context* is sometimes called a *telescope*, but the latter is used more generally to include a sequence of declarations occuring relative to a given context. For example, relative to the context $(a_1 : \alpha_1)\ (a_2 : \alpha_2)\ \ldots\ (a_n : \alpha_n)$, the types β_i in a telescope $(b_1 : \beta_1)\ (b_2 : \beta_2)\ \ldots\ (b_n : \beta_n)$ can refer to $a_1, \ldots, a_n$. Thus a context can be viewed as a telescope relative to the empty context.

Telescopes are often used to describe a list of arguments, or parameters, to a declaration. In such cases, it is often notationally convenient to let `(a : α)` stand for a telescope rather than just a single argument. In general, the annotations described in *Implicit Arguments* can be used to mark arguments as implicit.

4.3 Basic Declarations

Lean provides ways of adding new objects to the environment. The following provide straightforward ways of declaring new objects:

- `constant c : α` : declares a constant named `c` of type α, where `c` is a *declaration name*.
- `axiom c : α` : alternative syntax for `constant`
- `def c : α := t` : defines `c` to denote `t`, which should have type α.
- `theorem c : p := t` : similar to `def`, but intended to be used when `p` is a proposition.
- `lemma c : p := t` : alternative syntax for `theorem`

It is sometimes useful to be able to simulate a definition or theorem without naming it or adding it to the environment.

- `example : α := t` : elaborates `t` and checks that it has sort α (often a proposition), without adding it to the environment.

`constant` and `axiom` have plural versions, `constants` and `axioms`.

In `def`, the type (α or `p`, respectively) can be omitted when it can be inferred by Lean. Constants declared with `theorem` or `lemma` are marked as `irreducible`.

Any of `def`, `theorem`, `lemma`, or `example` can take a list of arguments (that is, a context) before the colon. If `(a : α)` is a context, the definition `def foo (a : α) : β := t` is interpreted as `def foo : Π a : α, β := λ a : α, t`. Similarly, a theorem `theorem foo (a : α) : p := t` is interpreted as `theorem foo : ∀ a : α, p := assume a : α, t`. (Remember that ∀ is syntactic sugar for Π, and `assume` is syntactic sugar for λ.)

```
constant  c : ℕ
constants (d e : ℕ) (f : ℕ → ℕ)
axiom     cd_eq : c = d

def foo : ℕ := 5
def bar := 6
def baz (x y : ℕ) (s : list ℕ) := [x, y] ++ s
```

```
theorem foo_eq_five : foo = 5 := rfl
theorem baz_theorem (x y : ℕ) : baz x y [] = [x, y] := rfl
lemma baz_lemma (x y : ℕ) : baz x y [] = [x, y] := rfl

example (x y : ℕ) : baz x y [] = [x, y] := rfl
```

4.4 Inductive Types

Lean's axiomatic foundation allows users to declare arbitrary inductive families, following the pattern described by *[Dybjer]*. To make the presentation more manageable, we first describe inductive *types*, and then describe the generalization to inductive *families* in the next section. The declaration of an inductive type has the following form:

```
inductive foo (a : α) : Sort u
| constructor₁ : Π (b : β₁), foo
| constructor₂ : Π (b : β₂), foo
...
| constructorₙ : Π (b : βₙ), foo
```

Here `(a : α)` is a context and each `(b : βᵢ)` is a telescope in the context `(a : α)` together with `(foo : Sort u)`, subject to the following constraints.

Suppose the telescope `(b : βᵢ)` is `(b₁ : βᵢ₁) ... (bᵤ : βᵢᵤ)`. Each argument in the telescope is either *nonrecursive* or *recursive*.

- An argument `(bⱼ : βᵢⱼ)` is *nonrecursive* if β_{ij} does not refer to `foo`, the inductive type being defined. In that case, β_{ij} can be any type, so long as it does not refer to any nonrecursive arguments.
- An argument `(bⱼ : βᵢⱼ)` is *recursive* if it β_{ij} of the form `Π (d : δ), foo` where `(d : δ)` is a telescope which does not refer to `foo` or any nonrecursive arguments.

The inductive type `foo` represents a type that is freely generated by the constructors. Each constructor can take arbitrary data and facts as arguments (the nonrecursive arguments), as well as indexed sequences of elements of `foo` that have been previously constructed (the recursive arguments). In set theoretic models, such sets can be represented by well-founded trees labeled by the constructor data, or they can defined using other transfinite or impredicative means.

The declaration of the type `foo` as above results in the addition of the following constants to the environment:

- the *type former* `foo : Π (a : α), Sort u`
- for each `i`, the *constructor* `foo.constructorᵢ : Π (a : α) (b : βᵢ), foo a`
- the *eliminator* `foo.rec`, which takes arguments
  - `(a : α)` (the parameters)
  - `{C : foo a → Type u}` (the *motive* of the elimination)
  - for each `i`, the *minor premise* corresponding to `constructorᵢ`
  - `(x : foo)` (the *major premise*)

  and returns an element of `C x`. Here, The ith minor premise is a function which takes
  - `(b : βᵢ)` (the arguments to the constructor)
  - an argument of type `Π (d : δ), C (bⱼ d)` corresponding to each recursive argument `(bⱼ : βᵢⱼ)`, where β_{ij} is of the form `Π (d : δ), foo` (the recursive values of the function being defined)

and returns an element of `C (constructor`$_i$ `a b)`, the intended value of the function at `constructor`$_i$ `a b`.

The eliminator represents a principle of recursion: to construct an element of `C x` where `x : foo a`, it suffices to consider each of the cases where `x` is of the form `constructor`$_i$ `a b` and to provide an auxiliary construction in each case. In the case where some of the arguments to `constructor`$_i$ are recursive, we can assume that we have already constructed values of `C y` for each value `y` constructed at an earlier stage.

Under the propositions-as-type correspondence, when `C x` is an element of `Prop`, the eliminator represents a principle of induction. In order to show $\forall$ `x, C x`, it suffices to show that `C` holds for each constructor, under the inductive hypothesis that it holds for all recursive inputs to the constructor.

The eliminator and constructors satisfy the following identities, in which all the arguments are shown explicitly. Suppose we set `F := foo.rec a C f`$_1$ `...` `f`$_n$. Then for each constructor, we have the definitional reduction:

```
F (constructor_i a b) = f_i b ... (λ d : δ_ij, F (b_j d)) ...
```

where the ellipses include one entry for each recursive argument.

Below are some common examples of inductive types, many of which are defined in the core library.

```
inductive empty : Type

inductive unit : Type
| star : unit

inductive bool : Type
| ff : bool
| tt : bool

inductive prod (α : Type u) (β : Type v) : Type (max u v)
| mk : α → β → prod

inductive sum (α : Type u) (β : Type v)
| inl : α → sum
| inr : β → sum

inductive sigma (α : Type u) (β : α → Type v)
| mk : Π a : α, β a → sigma

inductive false : Prop

inductive true : Prop
| trivial : true

inductive and (p q : Prop) : Prop
| intro : p → q → and

inductive or (p q : Prop) : Prop
| inl : p → or
| inr : q → or

inductive Exists (α : Type u) (p : α → Prop) : Prop
| intro : ∀ x : α, p x → Exists

inductive subtype (α : Type u) (p : α → Prop) : Type u
| intro : ∀ x : α, p x → subtype
```

```
inductive nat : Type
| zero : nat
| succ : nat → nat

inductive list (α : Type u)
| nil : list
| cons : α → list → list

-- full binary tree with nodes and leaves labeled from α
inductive bintree (α : Type u)
| leaf : α → bintree
| node : bintree → α → bintree → bintree

-- every internal node has subtrees indexed by ℕ
inductive cbt (α : Type u)
| leaf : α → cbt
| node : (ℕ → cbt) → cbt
```

Note that in the syntax of the inductive definition `foo`, the context `(a : α)` is left implicit. In other words, constructors and recursive arguments are written as though they have return type `foo` rather than `foo a`.

Elements of the context `(a : α)` can be marked implicit as described in Section 3.3. These annotations bear only on the type former, `foo`. Lean uses a heuristic to determine which arguments to the constructors should be marked implicit, namely, an argument is marked implicit if it can be inferred from the type of a subsequent argument. If the annotation `{}` appears after the constructor, a argument is marked implicit if it can be inferred from the type of a subsequent argument *or the return type*. For example, it is useful to let `nil` denote the empty list of any type, since the type can usually be inferred in the context in which it appears. These heuristics are imperfect, and you may sometimes wish to define your own constructors in terms of the default ones. In that case, use the `[pattern]` *attribute* to ensure that these will be used appropriately by the *equation compiler*.

There are restrictions on the universe `u` in the return type `Sort u` of the type former. There are also restrictions on the universe `u` in the return type `Sort u` of the motive of the eliminator. These will be discussed in the next section in the more general setting of inductive families.

Lean allows some additional syntactic conveniences. You can omit the return type of the type former, `Sort u`, in which case Lean will infer the minimal possible nonzero value for `u`. As with function definitions, you can list arguments to the constructors before the colon. In an enumerated type (that is, one where the constructors have no arguments), you can also leave out the return type of the constructors.

```
inductive weekday
| sunday | monday | tuesday | wednesday
| thursday | friday | saturday

inductive nat
| zero
| succ (n : nat) : nat

inductive list (α : Type u)
| nil {} : list
| cons (a : α) (l : list) : list

@[pattern]
def list.nil' (α : Type u) : list α := list.nil

def length {α : Type u} : list α → ℕ
| (list.nil' .(α)) := 0
| (list.cons a l) := 1 + length l
```

The type former, constructors, and eliminator are all part of Lean's axiomatic foundation, which is to say, they are part of the trusted kernel. In addition to these axiomatically declared constants, Lean automatically defines some additional objects in terms of these, and adds them to the environment. These include the following:

- `foo.rec_on` : a variant of the eliminator, in which the major premise comes first
- `foo.cases_on` : a restricted version of the eliminator which omits any recursive calls
- `foo.no_confusion_type`, `foo.no_confusion` : functions which witness the fact that the inductive type is freely generated, i.e. that the constructors are injective and that distinct constructors produce distinct objects
- `foo.below`, `foo.ibelow` : functions used by the equation compiler to implement structural recursion
- `foo.sizeof` : a measure which can be used for well-founded recursion

Note that it is common to put definitions and theorems related to a datatype `foo` in a namespace of the same name. This makes it possible to use projection notation described in Section 4.9 and Section 5.3.

```
inductive nat
| zero
| succ (n : nat) : nat

#check nat
#check nat.rec
#check nat.zero
#check nat.succ

#check nat.rec_on
#check nat.cases_on
#check nat.no_confusion_type
#check @nat.no_confusion
#check nat.brec_on
#check nat.below
#check nat.ibelow
#check nat.sizeof
```

4.5 Inductive Families

In fact, Lean implements a slight generalization of the inductive types described in the previous section, namely, inductive *families*. The declaration of an inductive family in Lean has the following form:

```
inductive foo (a : α) : Π (c : γ), Sort u
| constructor₁ : Π (b : β₁), foo t₁
| constructor₂ : Π (b : β₂), foo t₂
...
| constructorₙ : Π (b : βₙ), foo tₙ
```

Here `(a : α)` is a context, `(c : γ)` is a telescope in context `(a : α)`, each `(b : βᵢ)` is a telescope in the context `(a : α)` together with `(foo : Π (c : γ), Sort u)` subject to the constraints below, and each `tᵢ` is a tuple of terms in the context `(a : α) (b : βᵢ)` having the types `γ`. Instead of defining a single inductive type `foo a`, we are now defining a family of types `foo a c` indexed by elements `c : γ`. Each constructor, `constructorᵢ`, places its result in the type `foo a tᵢ`, the member of the family with index `tᵢ`.

The modifications to the scheme in the previous section are straightforward. Suppose the telescope `(b :` β_i`)` is `(b`$_1$ `:` β_{i1}`) ... (b`$_u$ `:` β_{iu}`)`.

- As before, an argument `(b`$_j$ `:` β_{ij}`)` is *nonrecursive* if β_{ij} does not refer to `foo`, the inductive type being defined. In that case, β_{ij} can be any type, so long as it does not refer to any nonrecursive arguments.
- An argument `(b`$_j$ `:` β_{ij}`)` is *recursive* if β_{ij} is of the form Π `(d :` δ`), foo s` where `(d :` δ`)` is a telescope which does not refer to `foo` or any nonrecursive arguments and `s` is a tuple of terms in context `(a :` α`)` and the previous nonrecursive `b`$_j$'s with types γ.

The declaration of the type `foo` as above results in the addition of the following constants to the environment:

- the *type former* `foo :` Π `(a :` α`) (c :` γ`), Sort u`
- for each `i`, the *constructor* `foo.constructor`$_i$ `:` Π `(a :` α`) (b :` β_i`), foo a t`$_i$
- the *eliminator* `foo.rec`, which takes arguments
  - `(a :` α`)` (the parameters)
  - `{C :` Π `(c :` γ`), foo a c` → `Type u}` (the motive of the elimination)
  - for each `i`, the minor premise corresponding to `constructor`$_i$
  - `(x : foo a)` (the major premise)

  and returns an element of `C x`. Here, The ith minor premise is a function which takes
  - `(b :` β_i`)` (the arguments to the constructor)
  - an argument of type Π `(d :` δ`), C s (b`$_j$ `d)` corresponding to each recursive argument `(b`$_j$ `:` β_{ij}`)`, where β_{ij} is of the form Π `(d :` δ`), foo s`

  and returns an element of `C t`$_i$ `(constructor`$_i$ `a b)`.

Suppose we set `F := foo.rec a C f`$_1$ `...` `f`$_n$. Then for each constructor, we have the definitional reduction, as before:

```
F (constructor_i a b) = f_i b ... (λ d : δ_ij, F (b_j d)) ...
```

where the ellipses include one entry for each recursive argument.

The following are examples of inductive families.

```
inductive vector (α : Type u) : ℕ → Type u
| nil  : vector 0
| succ : Π n, vector n → vector (n + 1)

-- 'is_prod s n' means n is a product of elements of s
inductive is_prod (s : set ℕ) : ℕ → Prop
| base : ∀ n ∈ s, is_prod n
| step : ∀ m n, is_prod m → is_prod n → is_prod (m * n)

inductive eq {α : Sort u} (a : α) : α → Prop
| refl : eq a
```

We can now describe the constraints on the return type of the type former, `Sort u`. We can always take `u` to be `0`, in which case we are defining an inductive family of propositions. If `u` is nonzero, however, it must satisfy the following constraint: for each type β_{ij} `: Sort v` occurring in the constructors, we must have `u` ≥ `v`. In the set-theoretic interpretation, this ensures that the universe in which the resulting type resides is large enough to contain the inductively generated family, given the number of distinctly-labeled constructors. The restriction does not hold for inductively defined propositions, since these contain no data.

Putting an inductive family in `Prop`, however, does impose a restriction on the eliminator. Generally speaking, for an inductive family in `Prop`, the motive in the eliminator is required to be in `Prop`. But there is an exception to this rule: you are allowed to eliminate from an inductively defined `Prop` to an arbitrary `Sort` when there is only one constructor, and each argument to that constructor is either in `Prop` or an index. The intuition is that in this case the elimination does not make use of any information that is not already given by the mere fact that the type of argument is inhabited. This special case is known as *singleton elimination*.

4.6 Mutual and Nested Inductive Definitions

Lean supports two generalizations of the inductive families described above, namely, *mutual* and *nested* inductive definitions. These are *not* implemented natively in the kernel. Rather, the definitions are compiled down to the primitive inductive types and families.

The first generalization allows for multiple inductive types to be defined simultaneously.

```
mutual inductive foo, bar (a : α)
with foo : Π (c : γ), Sort u
| constructor₁₁ : Π (b : β₁₁), foo t₁₁
| constructor₁₂ : Π (b : β₁₂), foo t₁₂
...
| constructor₁ₙ : Π (b : β₁ₙ), foo t₁ₙ
with bar :
| constructor₂₁ : Π (b : β₂₁), bar t₂₁
| constructor₂₂ : Π (b : β₂₂), bar t₂₂
...
| constructor₂ₘ : Π (b : β₂ₘ), bar t₂ₘ
```

Here the syntax is shown for defining two inductive families, `foo` and `bar`, but any number is allowed. The restrictions are almost the same as for ordinary inductive families. For example, each `(b : β_ij)` is a telescope relative to the context `(a : α)`. The difference is that the constructors can now have recursive arguments whose return types are any of the inductive families currently being defined, in this case `foo` and `bar`. Note that all of the inductive definitions share the same parameters `(a : α)`, though they may have different indices.

A mutual inductive definition is compiled down to an ordinary inductive definition using an extra finite-valued index to distinguish the components. The details of the internal construction are meant to be hidden from most users. Lean defines the expected type formers `foo` and `bar` and constructors `constructor_ij` from the internal inductive definition. There is no straightforward elimination principle, however. Instead, Lean defines an appropriate `sizeof` measure, meant for use with well-founded recursion, with the property that the recursive arguments to a constructor are smaller than the constructed value.

The second generalization relaxes the restriction that in the recursive definition of `foo`, `foo` can only occur strictly positively in the type of any of its recursive arguments. Specifically, in a nested inductive definition, `foo` can appear as an argument to another inductive type constructor, so long as the corresponding parameter occurs strictly positively in the constructors for *that* inductive type. This process can be iterated, so that additional type constructors can be applied to those, and so on.

A nested inductive definition is compiled down to an ordinary inductive definition using a mutual inductive definition to define copies of all the nested types simultaneously. Lean then constructs isomorphisms between the mutually defined nested types and their independently defined counterparts. Once again, the internal details are not meant to be manipulated by users. Rather, the type former and constructors are made available and work as expected, while an appropriate `sizeof` measure is generated for use with well-founded recursion.

```
mutual inductive even, odd
with even : ℕ → Prop
```

```
| even_zero : even 0
| even_succ : ∀ n, odd n → even (n + 1)
with odd : ℕ → Prop
| odd_succ : ∀ n, even n → odd (n + 1)

inductive tree (α : Type u)
| mk : α → list tree → tree

inductive double_tree (α : Type u)
| mk : α → list double_tree × list double_tree → double_tree
```

4.7 The Equation Compiler

The equation compiler takes an equational description of a function or proof and tries to define an object meeting that specification. It expects input with the following syntax:

```
def foo (a : α) : Π (b : β), γ
| [patterns₁] := t₁
...
| [patternsₙ] := tₙ
```

Here `(a : α)` is a telescope, `(b : β)` is a telescope in the context `(a : α)`, and γ is an expression in the context `(a : α) (b : β)` denoting a `Type` or a `Prop`.

Each $\texttt{patterns}_i$ is a sequence of patterns of the same length as `(b : β)`. A pattern is either:

- a variable, denoting an arbitrary value of the relevant type,
- an underscore, denoting a *wildcard* or *anonymous variable*,
- an inaccessible term (see below), or
- a constructor for the inductive type of the corresponding argument, applied to a sequence of patterns.

In the last case, the pattern must be enclosed in parentheses.

Each term $\texttt{t}_i$ is an expression in the context `(a : α)` together with the variables introduced on the left-hand side of the token `:=`. The term $\texttt{t}_i$ can also include recursive calls to `foo`, as described below. The equation compiler does case splitting on the variables `(b : β)` as necessary to match the patterns, and defines `foo` so that it has the value $\texttt{t}_i$ in each of the cases. In ideal circumstances (see below), the equations hold definitionally. Whether they hold definitionally or only propositionally, the equation compiler proves the relevant equations and assigns them internal names. They are accessible by the `rewrite` and `simp` tactics under the name `foo` (see Section 6.7 and Section 6.8). If some of the patterns overlap, the equation compiler interprets the definition so that the first matching pattern applies in each case. Thus, if the last pattern is a variable, it covers all the remaining cases. If the patterns that are presented do not cover all possible cases, the equation compiler raises an error.

When identifiers are marked with the `[pattern]` attribute, the equation compiler unfolds them in the hopes of exposing a constructor. For example, this makes it possible to write `n+1` and `0` instead of `nat.succ n` and `nat.zero` in patterns.

For a nonrecursive definition involving case splits, the defining equations will hold definitionally. With inductive types like `char`, `string`, and `fin n`, a case split would produce definitions with an inordinate number of cases. To avoid this, the equation compiler uses `if ... then ... else` instead of `cases_on` when defining the function. In this case, the defining equations hold definitionally as well.

```
open nat

def sub2 : ℕ → ℕ
| zero            := 0
| (succ zero)     := 0
| (succ (succ a)) := a

def bar : ℕ → list ℕ → bool → ℕ
| 0     _         ff := 0
| 0     (b :: _) _   := b
| 0     []        tt := 7
| (a+1) []        ff := a
| (a+1) []        tt := a + 1
| (a+1) (b :: _) _   := a + b

def baz : char → ℕ
| 'A' := 1
| 'B' := 2
| _   := 3
```

If any of the terms $\texttt{t}_i$ in the template above contain a recursive call to `foo`, the equation compiler tries to interpret the definition as a structural recursion. In order for that to succeed, the recursive arguments must be subterms of the corresponding arguments on the left-hand side. The function is then defined using a *course of values* recursion, using automatically generated functions `below` and `brec` in the namespace corresponding to the inductive type of the recursive argument. In this case the defining equations hold definitionally, possibly with additional case splits.

```
def fib : nat → nat
| 0     := 1
| 1     := 1
| (n+2) := fib (n+1) + fib n

def append {α : Type} : list α → list α → list α
| []     l := l
| (h::t) l := h :: append t l

example : append [(1 : ℕ), 2, 3] [4, 5] = [1, 2, 3, 4, 5] := rfl
```

If structural recursion fails, the equation compiler falls back on well-founded recursion. It tries to infer an instance of `has_sizeof` for the type of each argument, and then show that each recursive call is decreasing under the lexicographic order of the arguments with respect to `sizeof` measure. If it fails, the error message provides information as to the goal that Lean tried to prove. Lean uses information in the local context, so you can often provide the relevant proof manually using `have` in the body of the definition. In this case of well-founded recursion, the defining equations hold only propositionally, and can be accessed using `simp` and `rewrite` with the name `foo`.

```
def div : ℕ → ℕ → ℕ
| x y :=
  if h : 0 < y ∧ y ≤ x then
    have x - y < x,
      from sub_lt (lt_of_lt_of_le h.left h.right) h.left,
    div (x - y) y + 1
  else
    0

example (x y : ℕ) :
  div x y = if 0 < y ∧ y ≤ x then div (x - y) y + 1 else 0 :=
```

```
by rw [div]
```

Note that recursive definitions can in general require nested recursions, that is, recursion on different arguments of `foo` in the template above. The equation compiler handles this by abstracting later arguments, and recursively defining higher-order functions to meet the specification.

The equation compiler also allows mutual recursive definitions, with a syntax similar to that of *mutual inductive definitions*. They are compiled using well-founded recursion, and so once again the defining equations hold only propositionally.

```
mutual def even, odd
with even : ℕ → bool
| 0     := tt
| (a+1) := odd a
with odd : ℕ → bool
| 0     := ff
| (a+1) := even a

example (a : ℕ) : even (a + 1) = odd a :=
by simp [even]

example (a : ℕ) : odd (a + 1) = even a :=
by simp [odd]
```

Well-founded recursion is especially useful with *mutual and nested inductive definitions*, since it provides the canonical way of defining functions on these types.

```
mutual inductive even, odd
with even : ℕ → Prop
| even_zero : even 0
| even_succ : ∀ n, odd n → even (n + 1)
with odd : ℕ → Prop
| odd_succ : ∀ n, even n → odd (n + 1)

open even odd

theorem not_odd_zero : ¬ odd 0.

mutual theorem even_of_odd_succ, odd_of_even_succ
with even_of_odd_succ : ∀ n, odd (n + 1) → even n
| _ (odd_succ n h) := h
with odd_of_even_succ : ∀ n, even (n + 1) → odd n
| _ (even_succ n h) := h

inductive term
| const : string → term
| app   : string → list term → term

open term

mutual def num_consts, num_consts_lst
with num_consts : term → nat
| (term.const n)  := 1
| (term.app n ts) := num_consts_lst ts
with num_consts_lst : list term → nat
| []      := 0
| (t::ts) := num_consts t + num_consts_lst ts
```

The case where patterns are matched against an argument whose type is an inductive family is known as *dependent pattern matching*. This is more complicated, because the type of the function being defined can impose constraints on the patterns that are matched. In this case, the equation compiler will detect inconsistent cases and rule them out.

```
universe u

inductive vector (α : Type u) : ℕ → Type u
| nil {}  : vector 0
| cons    : Π {n}, α → vector n → vector (n+1)

namespace vector

def head {α : Type} : Π {n}, vector α (n+1) → α
| n (cons h t) := h

def tail {α : Type} : Π {n}, vector α (n+1) → vector α n
| n (cons h t) := t

def map {α β γ : Type} (f : α → β → γ) :
  Π {n}, vector α n → vector β n → vector γ n
| 0     nil         nil         := nil
| (n+1) (cons a va) (cons b vb) := cons (f a b) (map va vb)

end vector
```

An expression of the form `.(t)` in a pattern is known as an *inaccessible term*. It is not viewed as part of the pattern; rather, it is explicit information that is used by the elaborator and equation compiler when interpreting the definition. Inaccessible terms do not participate in pattern matching. They are sometimes needed for a pattern to make sense, for example, when a constructor depends on a parameter that is not a pattern-matching variable. In other cases, they can be used to inform the equation compiler that certain arguments do not require a case split, and they can be used to make a definition more readable.

```
variable {α : Type u}

def add [has_add α] :
  Π {n : ℕ}, vector α n → vector α n → vector α n
| ._ nil        nil        := nil
| ._ (cons a v) (cons b w) := cons (a + b) (add v w)

def add' [has_add α] :
  Π {n : ℕ}, vector α n → vector α n → vector α n
| .(0)   nil                nil        := nil
| .(n+1) (@cons .(α) n a v) (cons b w) := cons (a + b) (add' v w)
```

4.8 Match Expressions

Lean supports a `match ... with ...` construct similar to ones found in most functional programming languages. The syntax is as follows:

```
match t₁, ..., tₙ with
| p₁₁, ..., p₁ₙ := s₁
...
| pₘ₁, ..., pₘₙ := sₘ
```

Here $t_1, \ldots, t_n$ are any terms in the context in which the expression appears, the expressions p_{ij} are patterns, and the terms s_i are expressions in the local context together with variables introduced by the patterns on the left-hand side. Each s_i should have the expected type of the entire `match` expression.

Any `match` expression is interpreted using the equation compiler, which generalizes $t_1, \ldots, t_n$, defines an internal function meeting the specification, and then applies it to $t_1, \ldots, t_n$. In contrast to the definitions in Section 4.7, the terms t_i are arbitrary terms rather than just variables, and the expression can occur anywhere within a Lean expression, not just at the top level of a definition. Note that the syntax here is somewhat different: both the terms t_i and the patterns p_{ij} are separated by commas.

```
def foo (n : ℕ) (b c : bool) :=
5 + match n - 5, b && c with
    | 0,      tt := 0
    | m+1,    tt := m + 7
    | 0,      ff := 5
    | m+1,    ff := m + 3
    end
```

When a `match` has only one line, the vertical bar may be left out. In that case, Lean provides alternative syntax with a destructuring `let`, as well as a destructuring lambda abstraction. Thus the following definitions all have the same net effect.

```
def bar₁ : ℕ × ℕ → ℕ
| (m, n) := m + n

def bar₂ (p : ℕ × ℕ) : ℕ :=
match p with (m, n) := m + n end

def bar₃ : ℕ × ℕ → ℕ :=
λ ⟨m, n⟩, m + n

def bar₄ (p : ℕ × ℕ) : ℕ :=
let ⟨m, n⟩ := p in m + n
```

4.9 Structures and Records

The `structure` command in Lean is used to define an inductive data type with a single constructor and to define its projections at the same time. The syntax is as follows:

```
structure foo (a : α) extends bar, baz : Sort u :=
constructor :: (field₁ : β₁) ... (fieldₙ : βₙ)
```

Here `(a : α)` is a telescope, that is, the parameters to the inductive definition. The name `constructor` followed by the double colon is optional; if it is not present, the name `mk` is used by default. The keyword `extends` followed by a list of previously defined structures is also optional; if it is present, an instance of each of these structures is included among the fields to `foo`, and the types β_i can refer to their fields as well. The output type, `Sort u`, can be omitted, in which case Lean infers to smallest non-`Prop` sort possible. Finally, `(field₁ : β₁) ... (fieldₙ : βₙ)` is a telescope relative to `(a : α)` and the fields in `bar` and `baz`.

The declaration above is syntactic sugar for an inductive type declaration, and so results in the addition of the following constants to the environment:

- the type former : `foo : Π (a : α), Sort u`
- the single constructor :

```
foo.constructor : Π (a : α) (_to_foo : foo) (_to_bar : bar)
  (field₁ : β₁) ... (fieldₙ : βₙ), foo a
```

- the eliminator `foo.rec` for the inductive type with that constructor

In addition, Lean defines

- the projections : `field`$_i$ `:` Π `(a :` α`) (c : foo) :` β_i for each `i`

where any other fields mentioned in β_i are replaced by the relevant projections from `c`.

Given `c : foo`, Lean offers the following convenient syntax for the projection `foo.field`$_i$ `c`:

- *anonymous projections* : `c.field`$_i$
- *numbered projections* : `c.i`

These can be used in any situation where Lean can infer that the type of `c` is of the form `foo a`. The convention for anonymous projections is extended to any function `f` defined in the namespace `foo`, as described in Section 5.3.

Similarly, Lean offers the following convenient syntax for constructing elements of `foo`. They are equivalent to `foo.constructor b₁ b₂ f₁ f₁ ... fₙ`, where `b₁ : foo`, `b₂ : bar`, and each `fᵢ :` β_i :

- *anonymous constructor*: ⟨ `b₁, b₂, f₁, ..., fₙ` ⟩
- *record notation*:

```
{ foo . to_bar := b₁, to_baz := b₂, field₁ := f₁, ...,
    fieldₙ := fₙ }
```

The anonymous constructor can be used in any context where Lean can infer that the expression should have a type of the form `foo a`. The unicode brackets are entered as `\<` and `\>` respectively. The tokens `(|` and `|)` are ascii equivalents.

When using record notation, you can omit the annotation `foo .` when Lean can infer that the expression should have a type of the form `foo a`. You can replace either `to_bar` or `to_baz` by assignments to *their* fields as well, essentially acting as though the fields of `bar` and `baz` are simply imported into `foo`. Finally, record notation also supports

- *record updates*: `{ t with ... field`$_i$ `:= f`$_i$ `...}`

Here `t` is a term of type `foo a` for some `a`. The notation instructs Lean to take values from `t` for any field assignment that is omitted from the list.

Lean also allows you to specify a default value for any field in a structure by writing `(field`$_i$ `:` β_i `:= t)`. Here `t` specifies the value to use when the field `field`$_i$ is left unspecified in an instance of record notation.

```
universes u v

structure vec (α : Type u) (n : ℕ) :=
(l : list α) (h : l.length = n)

structure foo (α : Type u) (β : ℕ → Type v) : Type (max u v) :=
(a : α) (n : ℕ) (b : β n)

structure bar :=
(c : ℕ := 8) (d : ℕ)

structure baz extends foo ℕ (vec ℕ), bar :=
(v : vec ℕ n)
```

```
#check foo
#check @foo.mk
#check @foo.rec

#check foo.a
#check foo.n
#check foo.b

#check baz
#check @baz.mk
#check @baz.rec

#check baz.to_foo
#check baz.to_bar
#check baz.v

def bzz := vec.mk [1, 2, 3] rfl

#check vec.l bzz
#check vec.h bzz
#check bzz.l
#check bzz.h
#check bzz.1
#check bzz.2

example : vec ℕ 3 := vec.mk [1, 2, 3] rfl
example : vec ℕ 3 := ⟨[1, 2, 3], rfl⟩
example : vec ℕ 3 := (| [1, 2, 3], rfl |)
example : vec ℕ 3 := { vec . l := [1, 2, 3], h := rfl }
example : vec ℕ 3 := { l := [1, 2, 3], h := rfl }

example : foo ℕ (vec ℕ) := ⟨1, 3, bzz⟩

example : baz := ⟨⟨1, 3, bzz⟩, ⟨5, 7⟩, bzz⟩
example : baz := { a := 1, n := 3, b := bzz, c := 5, d := 7, v := bzz}
def fzz : foo ℕ (vec ℕ) := {a := 1, n := 3, b := bzz}

example : foo ℕ (vec ℕ) := { fzz with a := 7 }
example : baz := { fzz with c := 5, d := 7, v := bzz }

example : bar := { c := 8, d := 9 }
example : bar := { d := 9 }  -- uses the default value for c
```

4.10 Type Classes

(Classes and instances. Anonymous instances. Local instances.)

CHAPTER

FIVE

OTHER COMMANDS

5.1 Universes and Variables

The `universe` command introduces a special variable ranging over a type universe level. After the command `universe u`, a definition or theorem that is declared with a variable ranging over `Sort u` is polymorphic over that universe variable. More generally, universe level variables can appear in any *universe level expression*. The `universes` command can be used to introduce a list of universe level variables.

The `variable` command introduces a single variable declaration, and the `variables` command introduces one more more variable declarations. These have no effect until a subsequent definition or theorem declaration, though variables can also be used in a `#check` command. When Lean detects a variable name occuring in a definition or theorem, either in the type or the body, it inserts that variable and all the variables it depends on into the local context, as though they have all been declared before the colon. In other words, the declaration is abstracted over those variables. Only the variables that appear and their dependences are added, and are inserted in the order that they were declared.

Variables may be annotated as implicit as described in Section 3.3. You can change the annotation of a variable that has previously been declared using another `variable` or `variables` command, listing the variables with the desired annotation, but omitting their types.

Variables that are only used within a tactic block are not automatically included, since the meaning of a name in the context of a tactic block cannot be predicted at parse time. You can force the inclusion of a variable or list of variables in every declaration using the `include` command. To undo the effect of an `include` command, use `omit`.

```
universe u
variables {α β : Type u}
variable  y : α
variable  z : α

def ident (x : α) := x

theorem ident_eq : ∀ x : α, ident x = x := λ x, rfl

theorem ident_eq' : ident y = y := rfl

variables {y z}

variable h : y = z

example : ident z = y := eq.symm h

include h
example : ident z = y :=
```

```
begin
symmetry,
exact h
end

omit h

variable (y)

def ident2 := y
```

5.2 Sections

The scope of a `universe` or `variable` declaration can be scoped in a *section*. A section begins with a command `section foo` and ends with a command `end foo`, where `foo` is an arbitrary name. Alternatively, you can begin a section with the command `section` along, and close it with `end`. The name only serves to help match `section`/`end` pairs, and otherwise does not play any role.

Sections also support the commands `parameter` and `parameters`. These are similar to `variable` and `variables` respectively, except that within the section, later invocations of definitions and theorems that depend on the parameters introduced by these commands do not mention those parameters explicitly. In other words, the parameters are thought of as being fixed throughout the section, whereas definitions and theorems defined in terms of them maintain that fixed dependence. Outside the section, the definitions and theorems are generalized over those variables, just as with the `variables` command.

```
section
variables (x y : ℕ)

def foo := x + y

#check (foo : ℕ → ℕ → ℕ)
end

section
parameters (x y : ℕ)

def bar := x + y

#check (bar : ℕ)
#check (bar + 7 : ℕ)
end
```

As with the `variable` and `variables` commands, variables introduced with `parameter` and `parameters` can be annotated as implicit, and the annotations can be changed after the fact with subsequent declarations that omit the type. The `include` and `omit` commands can be used with these variables as well.

Sections also delimit the scope of local *attributes* and *notation declarations*.

5.3 Namespaces

The commands `namespace foo ... end foo`, where `foo` is a *declaration name*, open and close a namespace named `foo`. Within the namespace, `foo` is added as a prefix to all declarations. So, for exampe, `def bar` adds an object named `foo.bar` to the environment, and declares `bar` to be an alias for `foo.bar` while the

namespace is opened. If there is already an object or alias `bar` in the environment, the name is overloaded. Within the namespace, `foo.bar` is preferred when an ambiguity needs to be resolved. The prefix `_root_` can always be used to specify a full name starting at the top level, so that `_root_.bar` refers to the object whose full name is `bar`.

Namespaces can be nested. In terms of scoping, namespaces behave like sections. For example, variables declared in a namespace stay in scope until the `end` command.

The command `open foo` opens the namespace, so that `foo.bar` is aliased to `bar`. Once again, if there is already an object or alias `bar` in the environment, the name is overloaded (with none of them preferred). The `open` command admits these variations:

- `open foo (bar baz)` : create aliases *only* for `bar` and `baz`
- `open foo (renaming bar -> baz)` : renames `bar` to `baz` when opening `foo`
- `open foo (hiding bar)` : omits creating an alias for `bar` when opening `baz`

Multiple instances of `hiding` and `renaming` can be combined in a single `open` command.

The `export` command is similar to `open`, except that it serves to copy aliases from one namespace to another, or to the top level. For example, if a file exports `bar` from namespace `foo` to the top level, then any file that imports it will have the alias for `foo`.

Declarations within a namespace can bear the `protected` modifier. This means that a shortened alias is not generated when the namespace is open. For example, `nat.rec` is protected, meaning that opening `nat` does *not* generate an alias `rec`.

Declarations in a namespace or at the top level can also bear the `private` modifier, which means that they are added to the environment with an internally generated name and hidden from view outside the file. An alias is generated at the point where the declaration is made and it survives until the namespace is closed, or to the end of the file if the declaration is at the top level. Thus if we declare `private def bar := ...` in namespace `foo`, we can only refer to the object `bar` until the namespace is closed.

```
def baz := 7

namespace foo
namespace bar
    def baz := 5
    def fzz := 9
    protected def bloo := 11
    private def floo := 13

    example : foo.bar.baz = 5 := rfl
    example : bar.baz = 5 := rfl
    example : baz = 5 := rfl
    example : _root_.baz = 7 := rfl
end bar

example : bar.baz = 5 := rfl
end foo

section
open foo.bar

example : fzz = 9 := rfl
-- baz is overloaded and hence ambiguous
example : foo.bar.baz = 5 := rfl
end

section
```

```
open foo.bar (renaming fzz -> bzz)

example : bzz = 9 := rfl
example : foo.bar.bloo = 11 := rfl
end

export foo (bar.baz)

example : bar.baz = 5 := rfl

export foo.bar

example : fzz = 9 := rfl
```

If `t` is an element of an inductive type or family `foo`, then any function `bar` defined in the namespace `foo` can be treated as a "projection" using the anonymous projector notation described in Section 4.9. Specifically, if the first argument to `foo.bar` is of type `foo`, then `t.bar x y z` abbreviates `foo.bar t x y z`. More generally, as long as `foo.bar` has any argument of type `foo`, then `t.bar x y z` is interpreted as the result of applying `foo.bar` to `x`, `y`, and `z`, inserting `t` at the position of the first argument of type `foo`.

```
variables (xs ys : list ℕ) (f : ℕ → ℕ)

#check xs.length
#check xs.append ys
#check (xs.append ys).length
#check xs.map f
#check xs.reverse.reverse

example : [1, 2, 3].reverse.map (λ x, x + 2) = [5, 4, 3] := rfl
```

5.4 Attributes

Objects in Lean can bear *attributes*, which are tags that are associated to them, sometimes with additional data. You can assign an attribute `foo` to a object by preceding its declaration with the annotation `attribute [foo]` or, more concisely, `@[foo]`.

You can also assign the attribute `foo` to a object `bar` after it is declared by writing `attribute [foo] bar`. You can list more than one attribute and more than one name, in which case all the attributes are assigned to all the objects at once.

Finally, you can assign attributes locally by using `local attribute` instead of `attribute`. In that case, the attribute remains associated with the object until the end of the current section or namespace, or until the end of the current file if the command occurs outside any section or namespace.

The set of attributes is open-ended since users can declare additional attributes in Lean (see Chapter 7. You can ask Lean to give you a list of all the attributes present in the current environment with the command `#print attributes`. Below are some that are commonly used:

- `[class]` : a type class
- `[instance]` : an instance of a type class
- `[priority n]` : sets the class resolution priority to the natural number `n`
- `[refl]` : a reflexivity rule for the `reflexivity` tactic, for the `calc` environment, and for the simplifier
- `[symm]` : a symmetry rule for the `symmetry` tactic

- `[trans]` : a transitivity rule for the `transitivity` tactic, for the `calc` environment, and for the simplifier
- `[congr]` : a congruence rule for the simplifier
- `[simp]`: a simplifier rule
- `[recursor]` : a user-defined elimination principle, used, for example, by the induction tactic

Note that the `class` command, as discussed in Section 4.10, does more than simply assign the attribute.

There are attributes that control how eagerly definitions are unfolded during elaboration:

- `[reducible]` : unfold freely
- `[semireducible]` : unfold when inexpensive (the default)
- `[irreducible]` : do not unfold

There are also attributes used to specify strategies for elaboration:

- `[elab_with_expected_type]` : elaborate the arguments using their expected type (the default)
- `[elab_simple]` : elaborate arguments from left to right without propagating information about their types.
- `[elab_as_eliminator]` : uses a separate heuristic to infer higher-order parameters; commonly used for eliminators like recursors and induction principles

```
def foo (x : ℕ) := x + 5

attribute [simp]
theorem bar₁ (x : ℕ) : foo x = x + 5 := rfl

@[simp] theorem bar₂ (x : ℕ) : foo x = x + 5 := rfl

theorem bar₃ (x : ℕ) : foo x = x + 5 := rfl

theorem bar₄ (x : ℕ) : foo x = x + 5 := rfl

attribute [simp] bar₃ bar₄

#print attributes
```

5.5 Options

Lean maintains a number of internal variables that can be set by users to control its behavior. You can set such an option by writing *set_option <name> <value>*.

One very useful family of options controls the way Lean's pretty-printer displays terms. The following options take a value of `true` or `false`:

- `pp.implicit` : display implicit arguments
- `pp.universes` : display hidden universe parameters
- `pp.coercions` : show coercions
- `pp.notation` : display output using defined notations
- `pp.beta` : beta reduce terms before displaying them

As an example, the following settings yield much longer output:

```
set_option pp.implicit true
set_option pp.universes true
set_option pp.notation false
set_option pp.numerals false

#check 2 + 2 = 4
#reduce (λ x, x + 2) = (λ x, x + 3)
#check (λ x, x + 1) 1
```

5.6 Instructions

Commands that query Lean for information are generally intended to be transient, rather than remain permanently in a theory file. Such commands are typically preceded by a hash symbol.

- `#check t` : check that `t` is well-formed and show its type
- `#print t` : print information about `t`
- `#reduce t` : use the kernel reduction to reduce `t` to normal form
- `#eval t` : use the bytecode evaluator to evaluate `t`

The form of the output of the `#print` command varies depending on its argument. Here are some more specific variations:

- `#print definition` : display definition
- `#print inductive` : display an inductive type and its constructors
- `#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
- `#print prefix <namespace>` : display all declarations in the namespace
- `#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

Here are examples of how these commands are used:

```
def foo (x : ℕ) := x + 2

#check foo
#print foo
#reduce foo
#reduce foo 2
#eval foo 2

#print notation
#print notation + * -
#print axioms
#print options
#print prefix nat
```

```
#print prefix nat.le
#print classes
#print instances ring
#print fields ring
```

In addition, Lean provides the command `run_cmd` to execute an expression of type `tactic unit` on an empty goal. (See Chapter 7.)

5.7 Notation Declarations

Lean's parser is a Pratt-style parser, which means that tokens can serve separate functions at the beginning of an expression and in the middle of an expression, and every expression has a "left-binding power." Roughly, tokens with a higher left-binding power bind more tightly as an expression is parsed from left to right.

The following commands can be used in Lean to declare tokens and assign a left-binding power:

- `reserve infix `tok`:n`
- `reserve infixl `tok`:n`
- `reserve infixr `tok`:n`
- `reserve prefix `tok`:n`
- `reserve postfix `tok`:n`

In each case, `tok` is a string of characters that will become a new token, `n` is a natural number. The annotations `infix` and `infixl` mean the same thing, and specify that the infix notation should associate to the left. The keywords `prefix` and `postfix` are used to declare prefix and postfix notation, respectively.

Instance of the notation can later be assigned as follows:

- `infix tok := t`

where `t` is the desired interpretation, and similarly for the others. Notation can be overloaded.

It is not necessary to `reserve` a token before using it in notation. You can combine the two steps by writing

- `infix `tok`:n := t`

Note that in this case, backticks are needed to delimit the token. If a left binding power has already been assigned using the `reserve` keyword, it cannot be reassigned by an ordinary notation declaration. A later `reserve` command can, however, change the left binding power.

Surrounding the token by spaces in an infix declaration (that is, writing `` ` tok ` ``) instructs Lean's pretty printer to use extra space when displaying the notation. The spaces are not, however, part of the token. For example, all the following declarations are taken from the core library:

```
notation `Prop` := Sort 0
notation f ` $ `:1 a:0 := f a
notation `∅` := has_emptyc.emptyc _
notation h1 ▸ h2 := eq.subst h1 h2
notation h :: t  := list.cons h t
notation `[` l:(foldr `, ` (h t, list.cons h t) list.nil `]`) := l
notation `∃!` binders `, ` r:(scoped P, exists_unique P) := r
```

Note that, here, too, left-binding powers can be assigned on the fly, and backticks need to be used to enclose a token if it has not been declared before.

More examples can be found in the core library, for example in this file, which shows the binding strength of common symbols. The implication arrow binds with strength 25, denoted by `std.prec.arrow` in that file. Application has a high binding power, denoted `std.prec.max`. For postfix notation, you may wish to use the higher value, `std.prec.max_plus`. For example, according to the definition of the `inv` notation there, `f x⁻¹` is parsed as `f (x⁻¹)`.

The last two examples make possible list notation like `[1, 2, 3]` and the exists-unique binder, respectively. In the first, `foldr` specifies that the iterated operation is a right-associative fold, and binds the result to `l`. The four arguments then specify the separation token (in this case a comma, to be followed by a space when pretty printing), the fold operation, the start value, and the terminating token. You can use `foldl` instead for a left-associative fold.

In the last example, `binders` specifies that any number of binders can occur in that position, and the annotation after the comma indicates that these binders are to be iteratively abstracted using `exists_unique`.

Notation declarations can be preceded by the word "local," in which case the notation only remains in use in the current section or namespace, or in the current file if it is declared outside of any namespace.

Remember that you can use the `#print notation` command to show the notation that has been declared in the current environment. Given a token, it shows the notation associated with the token. Without arguments, it displays all notation currently in use. You can also use `set_option pp.notation false` to turn off the pretty-printing of notation.

CHAPTER

SIX

TACTICS

6.1 Tactic Mode

Anywhere an expression is expected, Lean will accept a sequence of instructions bracketed by the keywords `begin` and `end`. The input between these keywords represents a *tactic*, typically a compound sequence of basic tactics, each possibly applied to suitable arguments, separated by commas. When processing such a tactic block, Lean's elaborator executes the compound tactic with the expectation that it will produce an expression of the required type.

Individual tactics act on one or more *goals*, each of the form `a : α ⊢ p`, where `a : α` is a context and `p` is the target type. Tactics are typically used to prove a theorem, in which case `p` is a `Prop`, but they can be used to construct an element of an arbitrary `Type` as well.

At the outset, the elaborator presents the tactic block with a goal that consists of the local context in which the expression is being elaborated together with its expected type. Individual tactics can change goals and introduce new subgoals. A sequence of tactics is done when no subgoals remain, that is, when the compound tactic has succeeded in constructing an expression of the requisite type.

Tactics can fail. For example, a tactic may fail to make progress, or may not be appropriate to the goal. Other tactics can catch or handle those failures (see Section 6.6), but otherwise an error message is presented to the user.

Results produced by tactics are checked by the kernel for correctness. This provides another possible point of failure: a tactic block can, in principle, claim success but produce a term that fails to type check.

Tactics are themselves Lean expressions of a special `tactic` type. This makes it possible to implement Lean tactics in Lean itself; see Chapter 8. Tactics in a `begin ... end` block, however, are parsed in a special *interactive mode* that provides a more convenient manner of expression. In this section, we will focus exclusively on this interactive syntax.

You can use the keyword `by` instead of `begin ... end` to invoke a single tactic rather than a comma-separated sequence.

```
example (p q : Prop) : p ∧ q → q ∧ p :=
begin
  intro h,
  cases h,
  split,
  repeat { assumption }
end

example (p q : Prop) : p ∧ q → q ∧ p :=
assume ⟨h₁, h₂⟩,
and.intro (by assumption) (by assumption)
```

The documentation below coincides with documentation strings that are stored in the Lean source files and displayed by editors. The argument types are as follows:

- `id` : an *identifier*
- `expr` : an *expression*
- `<binders>` : a sequence of identifiers and expressions `(a : α)` where `a` is an identifier and α is a `Type` or a `Prop`.

An annotation `t?` means that the argument `t` is optional, and an annotation `t*` means any number of instances, possibly none. Many tactics parse arguments with additional tokens like `with`, `at`, `only`, `*`, or $\vdash$, as indicated below. The token `*` is typically used to denote all the hypotheses, and $\vdash$ is typically used to denote the goal, with ascii equivalent `|-`.

6.2 Basic Tactics

`intro id?`

If the current goal is a Pi / forall $\forall$ `x : t, u` (resp. `let x := t in u`) then `intro` puts `x : t` (resp. `x := t`) in the local context. The new subgoal target is `u`.

If the goal is an arrow `t` $\rightarrow$ `u`, then it puts `h : t` in the local context and the new goal target is `u`.

If the goal is neither a Pi/forall nor begins with a let binder, the tactic `intro` applies the tactic `whnf` until the tactic `intro` can be applied or the goal is not head reducible. In the latter case, the tactic fails.

`intros id*`

Similar to `intro` tactic. The tactic `intros` will keep introducing new hypotheses until the goal target is not a Pi/forall or let binder.

The variant `intros` h_1 `...` h_n introduces `n` new hypotheses using the given identifiers to name them.

`introv id*`

The tactic `introv` allows the user to automatically introduce the variables of a theorem and explicitly name the hypotheses involved. The given names are used to name non-dependent hypotheses.

Examples:

```
example : ∀ a b : nat, a = b → b = a :=
begin
introv h,
exact h.symm
end
```

The state after `introv h` is

```
a b : ℕ,
h : a = b
⊢ b = a
```

```
example : ∀ a b : nat, a = b → ∀ c, b = c → a = c :=
begin
introv h₁ h₂,
```

```
exact h₁.trans h₂
end
```

The state after `introv h₁ h₂` is

```
a b : ℕ,
h₁ : a = b,
c : ℕ,
h₂ : b = c
⊢ a = c
```

`rename id id`

The tactic `rename h₁ h₂` renames hypothesis `h₁` to `h₂` in the current local context.

`apply expr`

The `apply` tactic tries to match the current goal against the conclusion of the type of term. The argument term should be a term well-formed in the local context of the main goal. If it succeeds, then the tactic returns as many subgoals as the number of premises that have not been fixed by type inference or type class resolution. Non-dependent premises are added before dependent ones.

The `apply` tactic uses higher-order pattern matching, type class resolution, and first-order unification with dependent types.

`fapply expr`

Similar to the `apply` tactic, but does not reorder goals.

`eapply expr`

Similar to the `apply` tactic, but only creates subgoals for non-dependent premises that have not been fixed by type inference or type class resolution.

`apply_with expr (tactic.apply_cfg)`

Similar to the `apply` tactic, but allows the user to provide a `apply_cfg` configuration object.

`apply_instance`

This tactic tries to close the main goal `... ⊢ t` by generating a term of type `t` using type class resolution.

`refine expr`

This tactic behaves like `exact`, but with a big difference: the user can put underscores `_` in the expression as placeholders for holes that need to be filled, and `refine` will generate as many subgoals as there are holes.

Note that some holes may be implicit. The type of each hole must either be synthesized by the system or declared by an explicit type ascription like `(_ : nat → Prop)`.

`assumption`

This tactic looks in the local context for a hypothesis whose type is equal to the goal target. If it finds one, it uses it to prove the goal, and otherwise it fails.

`change expr (with expr)? (at (* | (⊢ | id)*))?`

`change u` replaces the target `t` of the main goal to `u` provided that `t` is well formed with respect to the local context of the main goal and `t` and `u` are definitionally equal.

`change u at h` will change a local hypothesis to `u`.

`change t with u at h1 h2 ...` will replace `t` with `u` in all the supplied hypotheses (or `*`), or in the goal if no `at` clause is specified, provided that `t` and `u` are definitionally equal.

`exact expr`

This tactic provides an exact proof term to solve the main goal. If `t` is the goal and `p` is a term of type `u` then `exact p` succeeds if and only if `t` and `u` can be unified.

`exacts ([expr, ...] | expr)`

Like `exact`, but takes a list of terms and checks that all goals are discharged after the tactic.

`revert id*`

`revert` $\mathtt{h}_1$ `...` $\mathtt{h}_n$ applies to any goal with hypotheses $\mathtt{h}_1$ `...` $\mathtt{h}_n$. It moves the hypotheses and their dependencies to the target of the goal. This tactic is the inverse of *intro*.

`generalize id? : expr = id`

`generalize : e = x` replaces all occurrences of `e` in the target with a new hypothesis `x` of the same type.

`generalize h : e = x` in addition registers the hypothesis `h : e = x`.

`admit`

Closes the main goal using `sorry`.

`contradiction`

The contradiction tactic attempts to find in the current local context an hypothesis that is equivalent to an empty inductive type (e.g. `false`), a hypothesis of the form `c_1 ... = c_2 ...` where `c_1` and `c_2` are distinct constructors, or two contradictory hypotheses.

`trivial`

Tries to solve the current goal using a canonical proof of `true`, or the `reflexivity` tactic, or the *contradiction* tactic.

`exfalso`

Replaces the target of the main goal by `false`.

`clear id*`

`clear` $\mathtt{h}_1$ `...` $\mathtt{h}_n$ tries to clear each hypothesis $\mathtt{h}_i$ from the local context.

`specialize expr`

The tactic `specialize h` $\mathtt{a}_1$ `...` $\mathtt{a}_n$ works on local hypothesis `h`. The premises of this hypothesis, either universal quantifications or non-dependent implications, are instantiated by concrete terms coming either from arguments $\mathtt{a}_1$ … $\mathtt{a}_n$. The tactic adds a new hypothesis with the same name `h :=` `h` $\mathtt{a}_1$ `...` $\mathtt{a}_n$ and tries to clear the previous one.

`by_cases expr (with id)?`

`by_cases p with h` splits the main goal into two cases, assuming `h : p` in the first branch, and `h : ¬ p` in the second branch.

This tactic requires that `p` is decidable. To ensure that all propositions are decidable via classical reasoning, use `local attribute classical.prop_decidable [instance]`.

`by_contradiction id?`

If the target of the main goal is a proposition `p`, `by_contradiction h` reduces to goal to proving `false` using the additional hypothesis `h : ¬ p`. If `h` is omitted, a name is generated automatically.

This tactic requires that `p` is decidable. To ensure that all propositions are decidable via classical reasoning, use `local attribute classical.prop_decidable [instance]`.

`by_contra id?`

An abbreviation for `by_contradiction`.

6.3 Equality and Other Relations

`reflexivity`

This tactic applies to a goal whose goal has the form `t ~ u` where `~` is a reflexive relation, that is, a relation which has a reflexivity lemma tagged with the attribute `[refl]`. The tactic checks whether `t` and `u` are definitionally equal and then solves the goal.

`refl`

Shorter name for the tactic `reflexivity`.

`symmetry`

This tactic applies to a goal whose target has the form `t ~ u` where `~` is a symmetric relation, that is, a relation which has a symmetry lemma tagged with the attribute `[symm]`. It replaces the goal with `u ~ t`.

`transitivity ?expr`

This tactic applies to a goal whose target has the form `t ~ u` where `~` is a transitive relation, that is, a relation which has a transitivity lemma tagged with the attribute `[trans]`.

`transitivity s` replaces the goal with the two subgoals `t ~ s` and `s ~ u`. If `s` is omitted, then a metavariable is used instead.

6.4 Structured Tactic Proofs

Tactic blocks can have nested `begin ... end` blocks and, equivalently, blocks `{ ... }` enclosed with curly braces. Opening such a block focuses on the current goal, so that no other goals are visible within the nested block. Closing a block while any subgoals remain results in an error.

`assume (: expr | <binders>)`

Assuming the target of the goal is a Pi or a let, `assume h : t` unifies the type of the binder with `t` and introduces it with name `h`, just like `intro h`. If `h` is absent, the tactic uses the name `this`. If `T` is omitted, it will be inferred.

`assume` $(h_1 : t_1) \ldots (h_n : t_n)$ introduces multiple hypotheses. Any of the types may be omitted, but the names must be present.

`have id? (: expr)? (:= expr)?`

`have h : t := p` adds the hypothesis `h : t` to the current goal if `p` a term of type `t`. If `t` is omitted, it will be inferred.

`have h : t` adds the hypothesis `h : t` to the current goal and opens a new subgoal with target `t`. The new subgoal becomes the main goal. If `t` is omitted, it will be replaced by a fresh metavariable.

If `h` is omitted, the name `this` is used.

`let id? (: expr)? (:= expr)?`

`let h : T := p` adds the hypothesis `h : t := p` to the current goal if `p` a term of type `t`. If *t* is omitted, it will be inferred.

`let h : t` adds the hypothesis `h : t := ?M` to the current goal and opens a new subgoal `?M : t`. The new subgoal becomes the main goal. If `t` is omitted, it will be replaced by a fresh metavariable.

If `h` is omitted, the name `this` is used.

`suffices id? (: expr)?`

`suffices h : t` is the same as `have h : t, tactic.swap`. In other words, it adds the hypothesis `h : t` to the current goal and opens a new subgoal with target `t`.

`show expr`

`show t` finds the first goal whose target unifies with `t`. It makes that the main goal, performs the unification, and replaces the target with the unified version of `t`.

`from expr`

A synonym for `exact` that allows writing `have/suffices/show ..., from ...` in tactic mode.

```
variables (p q : Prop)

example : p ∧ (p → q) → q ∧ p :=
begin
  assume h : p ∧ (p → q),
  have h₁ : p, from and.left h,
  have : p → q := and.right h,
  suffices : q, from and.intro this h₁,
  show q, from ⟨p → q⟩ h₁
end

example (p q : Prop) : p → p → p :=
begin
  assume h (h' : p),
  from h
end

example : ∃ x, x = 5 :=
begin
  let u := 3 + 2,
  existsi u, reflexivity
end
```

6.5 Inductive Types

The following tactics are designed specifically to work with elements on an inductive type.

`induction expr (using id)? (with id*)? (generalizing id*)?`

Assuming `x` is a variable in the local context with an inductive type, `induction x` applies induction on `x` to the main goal, producing one goal for each constructor of the inductive type, in which the target is replaced by a general instance of that constructor and an inductive hypothesis is added for each recursive argument to the constructor. If the type of an element in the local context depends on `x`, that element is reverted and reintroduced afterward, so that the inductive hypothesis incorporates that hypothesis as well.

For example, given `n : nat` and a goal with a hypothesis `h : P n` and target `Q n`, `induction n` produces one goal with hypothesis `h : P 0` and target `Q 0`, and one goal with hypotheses `h : P (nat.succ a)` and $\texttt{ih}_1$ `: P a` $\rightarrow$ `Q a` and target `Q (nat.succ a)`. Here the names `a` and $\texttt{ih}_1$ ire chosen automatically.

`induction e`, where `e` is an expression instead of a variable, generalizes `e` in the goal, and then performs induction on the resulting variable.

`induction e with` $\texttt{y}_1$ `...` $\texttt{y}_n$, where `e` is a variable or an expression, specifies that the sequence of names $\texttt{y}_1$ `...` $\texttt{y}_n$ should be used for the arguments to the constructors and inductive hypotheses, including implicit arguments. If the list does not include enough names for all of the arguments, additional names are generated automatically. If too many names are given, the extra ones are ignored. Underscores can be used in the list, in which case the corresponding names are generated automatically.

`induction e using r` allows the user to specify the principle of induction that should be used. Here `r` should be a theorem whose result type must be of the form `C t`, where `C` is a bound variable and `t` is a (possibly empty) sequence of bound variables

`induction e generalizing` $\texttt{z}_1$ `...` $\texttt{z}_n$, where $\texttt{z}_1$ `...` $\texttt{z}_n$ are variables in the local context, generalizes over $\texttt{z}_1$ `...` $\texttt{z}_n$ before applying the induction but then introduces them in each goal. In other words, the net effect is that each inductive hypothesis is generalized.

`cases (id :)? expr (with id*)?`

Assuming `x` is a variable in the local context with an inductive type, `cases x` splits the main goal, producing one goal for each constructor of the inductive type, in which the target is replaced by a general instance of that constructor. If the type of an element in the local context depends on `x`, that element is reverted and reintroduced afterward, so that the case split affects that hypothesis as well.

For example, given `n : nat` and a goal with a hypothesis `h : P n` and target `Q n`, `cases n` produces one goal with hypothesis `h : P 0` and target `Q 0`, and one goal with hypothesis `h : P (nat.succ a)` and target `Q (nat.succ a)`. Here the name `a` is chosen automatically.

`cases e`, where `e` is an expression instead of a variable, generalizes `e` in the goal, and then cases on the resulting variable.

`cases e with` $\texttt{y}_1$ `...` $\texttt{y}_n$, where `e` is a variable or an expression, specifies that the sequence of names $\texttt{y}_1$ `...` $\texttt{y}_n$ should be used for the arguments to the constructors, including implicit arguments. If the list does not include enough names for all of the arguments, additional names are generated automatically. If too many names are given, the extra ones are ignored. Underscores can be used in the list, in which case the corresponding names are generated automatically.

`cases h : e`, where `e` is a variable or an expression, performs cases on `e` as above, but also adds a hypothesis `h : e = ...` to each hypothesis, where `...` is the constructor instance for that particular case.

`case id id* { tactic }`

Focuses on the `induction`/`cases` subgoal corresponding to the given introduction rule, optionally renaming introduced locals.

```
example (n : ℕ) : n = n :=
begin
  induction n,
  case nat.zero { reflexivity },
  case nat.succ a ih { reflexivity }
end
```

`destruct expr`

Assuming `x` is a variable in the local context with an inductive type, `destruct x` splits the main goal, producing one goal for each constructor of the inductive type, in which `x` is assumed to be a general instance of that constructor. In contrast to `cases`, the local context is unchanged, i.e. no elements are reverted or introduced.

For example, given `n : nat` and a goal with a hypothesis `h : P n` and target `Q n`, `destruct n` produces one goal with target `n = 0 → Q n`, and one goal with target `∀ (a : ℕ), (λ (w : ℕ), n = w → Q n) (nat.succ a)`. Here the name `a` is chosen automatically.

`existsi`

`existsi e` will instantiate an existential quantifier in the target with `e` and leave the instantiated body as the new target. More generally, it applies to any inductive type with one constructor and at least two arguments, applying the constructor with `e` as the first argument and leaving the remaining arguments as goals.

`existsi [e₁, ..., eₙ]` iteratively does the same for each expression in the list.

`constructor`

This tactic applies to a goal such that its conclusion is an inductive type (say `I`). It tries to apply each constructor of `I` until it succeeds.

`econstructor`

Similar to `constructor`, but only non-dependent premises are added as new goals.

`left`

Applies the first constructor when the type of the target is an inductive data type with two constructors.

`right`

Applies the second constructor when the type of the target is an inductive data type with two constructors.

`split`

Applies the constructor when the type of the target is an inductive data type with one constructor.

`injection expr (with id*)?`

The `injection` tactic is based on the fact that constructors of inductive data types are injections. That means that if `c` is a constructor of an inductive datatype, and if `(c t₁)` and `(c t₂)` are two terms that are equal then `t₁` and `t₂` are equal too.

If `q` is a proof of a statement of conclusion `t₁ = t₂`, then injection applies injectivity to derive the equality of all arguments of `t₁` and `t₂` placed in the same positions. For example, from `(a::b) = (c::d)` we derive `a=c` and `b=d`. To use this tactic `t₁` and `t₂` should be constructor applications of the same constructor.

Given `h : a::b = c::d`, the tactic `injection h` adds two new hypothesis with types `a = c` and `b = d` to the main goal. The tactic `injection h with h₁ h₂` uses the names `h₁` an `h₂` to name the new hypotheses.

`injections (with id*)?`

`injections with h₁ ... hₙ` iteratively applies `injection` to hypotheses using the names `h₁ ... hₙ`.

6.6 Tactic Combinators

Tactic combinators build compound tactics from simpler ones.

`repeat { tactic }`

`repeat { t }` repeatedly applies `t` until `t` fails. The compound tactic always succeeds.

`try { tactic }`

`try { t }` tries to apply tactic `t`, but succeeds whether or not `t` succeeds.

`skip`

A do-nothing tactic that always succeeds.

`solve1 { tactic }`

`solve1 { t }` applies the tactic `t` to the main goal and fails if it is not solved.

`abstract id? { tactic }`

`abstract id { t }` tries to use tactic `t` to solve the main goal. If it succeeds, it abstracts the goal as an independent definition or theorem with name `id`. If `id` is omitted, a name is generated automatically.

`all_goals { tactic }`

`all_goals { t }` applies the tactic `t` to every goal, and succeeds if each application succeeds.

`any_goals { tactic }`

`any_goals { t }` applies the tactic `t` to every goal, and succeeds if at least one application succeeds.

`done`

Fail if there are unsolved goals.

`fail_if_success { tactic }`

Fails if the given tactic succeeds.

`success_if_fail { tactic }`

Succeeds if the given tactic succeeds.

`guard_target expr`

`guard_target t` fails if the target of the main goal is not `t`.

`guard_hyp id := expr`

`guard_hyp h := t` fails if the hypothesis `h` does not have type `t`.

6.7 The Rewriter

`rewrite ([ (←? expr), ... ] | ←? expr) (at (* | (⊢ | id)*))? tactic.rewrite_cfg?`

`rewrite e` applies identity `e` as a rewrite rule to the target of the main goal. If `e` is preceded by left arrow (`←` or `<-`), the rewrite is applied in the reverse direction. If `e` is a defined constant, then the equational lemmas associated with `e` are used. This provides a convenient way to unfold `e`.

`rewrite [`e_1`, ..., `e_n`]` applies the given rules sequentially.

`rewrite e at l` rewrites `e` at location(s) `l`, where `l` is either `*` or a list of hypotheses in the local context. In the latter case, a turnstile ⊢ or `|-` can also be used, to signify the target of the goal.

`rw`

An abbreviation for `rewrite`.

`rwa`

`rewrite` followed by `assumption`.

`erewrite`

A variant of `rewrite` that uses the unifier more aggressively, unfolding semireducible definitions.

`erw`

An abbreviation for `erewrite`.

`subst expr`

Given hypothesis `h : x = t` or `h : t = x`, where `x` is a local constant, `subst h` substitutes `x` by `t` everywhere in the main goal and then clears `h`.

6.8 The Simplifier and Congruence Closure

`simp only? (* | [(* | (- id | expr)), ...]?) (with id*)? (at (* | (⊢ | id)*))? tactic.simp_config_ext?`

The `simp` tactic uses lemmas and hypotheses to simplify the main goal target or non-dependent hypotheses. It has many variants.

`simp` simplifies the main goal target using lemmas tagged with the attribute `[simp]`.

`simp [h₁ h₂ ... hₙ]` simplifies the main goal target using the lemmas tagged with the attribute `[simp]` and the given h_i's, where the h_i's are expressions. These expressions may contain underscores, in which case they are replaced by metavariables that `simp` tries to instantiate. If a h_i is a defined constant `f`, then the equational lemmas associated with `f` are used. This provides a convenient way to unfold `f`.

`simp [*]` simplifies the main goal target using the lemmas tagged with the attribute `[simp]` and all hypotheses.

`simp *` is a shorthand for `simp [*]`.

`simp only [h₁ h₂ ... hₙ]` is like `simp [h₁ h₂ ... hₙ]` but does not use `[simp]` lemmas

`simp [-id₁, ... -idₙ]` simplifies the main goal target using the lemmas tagged with the attribute `[simp]`, but removes the ones named id_i.

`simp at h₁ h₂ ... hₙ` simplifies the non-dependent hypotheses `h₁ : T₁` ... `hₙ : Tₙ`. The tactic fails if the target or another hypothesis depends on one of them. The token ⊢ or `|-` can be added to the list to include the target.

`simp at *` simplifies all the hypotheses and the target.

`simp * at *` simplifies target and all (non-dependent propositional) hypotheses using the other hypotheses.

`simp with attr₁ ... attrₙ` simplifies the main goal target using the lemmas tagged with any of the attributes `[attr₁]`, ..., `[attrₙ]` or `[simp]`.

`dsimp only? (* | [(* | (- id | expr)), ...]?) (with id*)? (at (* | (⊢ | id)*))? tactic.dsimp_config?`

`dsimp` is similar to `simp`, except that it only uses definitional equalities.

`simp_intros id* only? (* | [(* | (- id | expr)), ...]?) (with id*)? tactic.simp_intros_config?`

`simp_intros` h_1 h_2 ... h_n is similar to `intros` h_1 h_2 ... h_n except that each hypothesis is simplified as it is introduced, and each introduced hypothesis is used to simplify later ones and the final target.

As with `simp`, a list of simplification lemmas can be provided. The modifiers `only` and `with` behave as with `simp`.

`unfold id* (at (* | (⊢ | id)*))? tactic.unfold_config?`

Given defined constants e_1 ... e_n, `unfold` e_1 ... e_n iteratively unfolds all occurrences in the target of the main goal, using equational lemmas associated with the definitions.

As with `simp`, the `at` modifier can be used to specify locations for the unfolding.

`unfold1 id* (at (* | (⊢ | id)*))? tactic.unfold_config?`

Similar to `unfold`, but does not iterate the unfolding.

`dunfold id* (at (* | (⊢ | id)*))? tactic.dunfold_config?`

Similar to `unfold`, but only uses definitional equalities.

`delta id* (at (* | (⊢ | id)*))?`

Similar to `dunfold`, but performs a raw delta reduction, rather than using an equation associated with the defined constants.

`unfold_projs`

This tactic unfolds all structure projections.

`trace_simp_set`

Just construct the simp set and trace it. Used for debugging.

`ac_reflexivity`

Proves a goal with target `s = t` when `s` and `t` are equal up to the associativity and commutativity of their binary operations.

`ac_refl`

An abbreviation for `ac_reflexivity`.

`cc`

Tries to prove the main goal using congruence closure.

6.9 Other Tactics

`trace_state`

This tactic displays the current state in the tracing buffer.

`trace a`

`trace a` displays `a` in the tracing buffer.

`type_check expr`

Type check the given expression, and trace its type.

`apply_opt_param`

If the target of the main goal is an *opt_param*, assigns the default value.

`apply_auto_param`

If the target of the main goal is an *auto_param*, executes the associated tactic.

`dedup`

Renames hypotheses with the same name.

6.10 Conversions

6.11 The SMT State

CHAPTER

SEVEN

PROGRAMMING

7.1 The Virtual Machine

7.2 Monads

(Describe instances of monads and monadic notation.)

CHAPTER

EIGHT

METAPROGRAMMING

8.1 Quotations

8.2 User Defined Attributes

CHAPTER

NINE

LIBRARIES

BIBLIOGRAPHY

[Dybjer] Dybjer, Peter, *Inductive Families*. Formal Aspects of Computing 6, 1994, pages 440-465.