An Introduction to Lean

Jeremy Avigad
Leonardo de Moura
Gabriel Ebner
and Sebastian Ullrich

Version 77fdc88, updated at 2017-09-10 21:52:10 -0400

Contents

Contents **3**

1 Overview **5**
1.1 Perspectives on Lean . . . 5
1.2 Where To Go From Here . . . 12

2 Defining Objects in Lean **13**
2.1 Some Basic Types . . . 14
2.2 Defining Functions . . . 17
2.3 Defining New Types . . . 20
2.4 Records and Structures . . . 22
2.5 Nonconstructive Definitions . . . 25

3 Programming in Lean **27**
3.1 Evaluating Expressions . . . 28
3.2 Recursive Definitions . . . 30
3.3 Inhabited Types, Subtypes, and Option Types . . . 32
3.4 Monads . . . 34
3.5 Input and Output . . . 35
3.6 An Example: Abstract Syntax . . . 36

4 Theorem Proving in Lean **38**
4.1 Assertions in Dependent Type Theory . . . 38
4.2 Propositions as Types . . . 39
4.3 Induction and Calculation . . . 42
4.4 Axioms . . . 45

5 Using Automation in Lean **46**

6 Metaprogramming in Lean **47**

Bibliography **48**

1

Overview

This introduction offers a tour of Lean and its features, with a number of examples for you to play around with and explore. If you are reading this in our online tutorial system, you can run examples like the one below by clicking the button that says "try it yourself."

```
#check "hello world!"
```

The response from Lean appears in the small window underneath the editor text, and also in popup windows that you can read when you hover over the indicators in the left margin. Alternatively, if you have installed Lean and have it running in a stand-alone editor, you can copy and paste examples and try them there.

1.1 Perspectives on Lean

Lean is an implementation of a logical foundation known as *dependent type theory*. Specifically, it implements a version of dependent type theory known as the *Calculus of Inductive Constructions*. The *CIC* is a formal language with a small and precise set of rules that governs the formation of expressions. In this formal system, moreover, every expression has a *type*. The type of expression indicates what sort of object the expression denotes. For example, an expression may denote a mathematical object like a natural number, a data type, an assertion, or a proof.

Lean has a small and carefully written kernel, which serves to check that an expression is well-formed and confirm that it has a given type. It is this kernel that gives Lean its special character. Dependent type theory serves as a foundational language, allowing us to describe all sorts of objects and prove things about them. The foundational language fixes

the meaning of the objects we introduce, and the kernel ensures that the things we prove about them are correct.

Put simply, Lean is designed to help you construct, manipulate, and check expressions in this foundational language. This may not sound like much, but what makes the system powerful is the fact that dependent type theory is expressive enough to allow us to define and reason about all sorts of objects. For example, Lean's standard library defines the natural numbers to be the structure generated freely and inductively by a constant, *zero*, and a unary function *succ*:

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

If you copy this definition into the editor window at right you will see that we have wrapped it in a *namespace* to avoid conflicting with the standard definition, which is loaded by default. Even so, choosing the name `nat` means that within the namespace this identifier is overloaded, which can cause confusion. Thus we will do this only sparingly, for purposes of illustration.

Having specified this data type, we can go on to define addition by recursion on the second argument:

```
def add : nat → nat → nat
| m nat.zero     := m
| m (nat.succ n) := nat.succ (add m n)
```

Lean compiles definitions like these down to a single axiomatic primitive that governs use of both induction and recursion on inductively defined structures. The library defines notation for the data type, as well as for `zero` and `add`. (In fact, Lean uses *type classes*, a very handy mechanism used by functional programming languages like Haskell, to share notation and properties across algebraic structures.) Lean uses the Unicode character ℕ as alternative notation for the type `nat`. You can enter this in an editor by writing `\nat`.

Of course, we can also define non-recursive functions by giving an explicit definition:

```
def double (n : ℕ) : ℕ := n + n
```

We can then go on to define other data types like the integers, the rationals, and the real numbers, the booleans, characters and strings, lists, products, disjoint sums, and so on. We can also define algebraic structures like groups, rings, fields, vector spaces, and categories. In fact, dependent type theory was designed to serve as a foundation for all conventional mathematics.

This points to a first intended use of Lean: it serves as a *specification language*, that is, a means to specify and define mathematical objects in precise terms. With these specifications, Lean can interpret basic objects and infer their types:

```
#check (27 + 9) * 33
#check [(1, 2), (3, 4), (5, 6)] ++ [(7, 8), (9, 10)]
```

When there is no other information present to constrain the type of a numeral, Lean assumes it denotes a natural, by default. Thus Lean can recognize that the first expression denotes a natural number, and that the second, a concatenation of two lists of pairs of natural numbers, is again a list of pairs. It also remembers that `double` is a function from the natural numbers to the natural numbers, and can print out the definition when requested to do so:

```
#check double
#print double
```

Lean can reason about abstract objects as well as it can reason about concrete ones. In the following example, we declare a type `G` with a group structure, and variables g_1 and g_2 that range over `G`. With those declarations, Lean knows that the expression $g_2^{-1} * g_1 * g_2$ denotes an element of `G`.

```
section
  variables (G : Type) [group G]
  variables g₁ g₂ : G

  #check g₂⁻¹ * g₁ * g₂
end
```

Putting the declarations in a `section`, as we do here, delimits their scope. In this case, the section declaration is not needed, and no harm would be done if we had declared these variables at the top level.

An important feature of dependent type theory is that every expression has a computational interpretation, which is to say, there are rules that specify how they can be *reduced* to a normal form. Moreover, expressions in a computationally pure fragment of the language evaluate to *values* in the way you would expect. For example, assuming the definition does not depend on nonconstructive components in an essential way, every closed term of type $\mathbb{N}$ evaluates to a numeral. Lean's kernel can carry out this evaluation:

```
#eval (27 + 9) * 33
```

As part of the kernel, the results of this evaluation can be highly trusted. The evaluator is not very efficient, however, and is not intended to be used for substantial computational tasks. For that purpose, Lean also generates bytecode for every definition of a computable object, and can evaluate it on demand. To process the bytecode quickly, it uses an efficient *virtual machine*, similar to the ones used to interpret OCaml and Python.

```
#eval (27 + 9) * 33
#eval (2227 + 9999) * 33
#eval double 9999
#eval [(1, 2), (3, 4), (5, 6)] ++ [(7, 8), (9, 10)]
```

Relying on results from the bytecode evaluator requires a higher level of trust than relying on the kernel. For example, for efficiency, the bytecode evaluator uses the GNU multiple precision library to carry out numerical computations involving the natural numbers and integers, so the correctness of those computations are no longer underwritten by the axiomatic foundation.

This points to a second intended use of Lean, namely, as a *programming language*. Because dependent type theory is so expressive, we can make use of all the usual methods and techniques of functional programming, including higher types, type classes, records, monads, and other abstractions. In fact, we have the entire Lean library at our disposal. With just a few lines of code, we can write a generic sort procedure that sorts elements of a list according to a specified binary relation `r` on an arbitrary type α, assuming only that we can determine computationally when `r` holds.

```
section sort
universe u
parameters {α : Type u} (r : α → α → Prop) [decidable_rel r]
local infix `≼` : 50 := r

def ordered_insert (a : α) : list α → list α
| []       := [a]
| (b :: l) := if a ≼ b then a :: (b :: l) else b :: ordered_insert l

def insertion_sort : list α → list α
| []       := []
| (b :: l) := ordered_insert b (insertion_sort l)

end sort
```

For foundational reasons, types in Lean have to be stratified into a hierarchy of *type universes*, and the definitions above work for any type α in any such universe. We can run the procedure above on a list of natural numbers, using the usual ordering:

```
#eval insertion_sort (λ m n : ℕ, m ≤ n) [5, 27, 221, 95, 17, 43, 7, 2, 98, 567, 23, 12]
```

Substantial programs can be written in Lean and run by the bytecode interpreter. In fact, a full-blown resolution theorem prover for Lean has been written in Lean itself.

You can profile your code by setting the relevant options:

```
set_option profiler true
set_option profiler.freq 10
```

The second option determines the frequency that the virtual machine is polled with. Be careful: if the task you profile is too short, there won't be any output! You can even implement your own debugger in Lean itself.

What makes Lean special as a programming language is that the programs we write define functions in a precise axiomatic framework. Which brings us to third, and central, intended use of Lean: namely we can make assertions about the objects we define and then go on to prove those assertions. We can do this because the language of dependent type theory is rich enough to encode such assertions and proofs. For example, we can express the property that a natural number is even:

```
def even (n : ℕ) : Prop := ∃ m, n = 2 * m
```

As presented, it is not clear that the property of being even is decidable, since we cannot in general test every natural number to determine whether any of them serves as a witness to the given existential statement. But we can nonetheless use this definition to form compound statements:

```
#check even 10
#check even 11
#check ∀ n, even n ∨ even (n + 1)
#check ∀ n m, even n → even m → even (n + m)
```

In each case, the expression has type `Prop`, indicating that Lean recognizes it as an assertion.

Incidentally, of course, we do know that the property of being `even n` is algorithmically decidable. We can develop any algorithm we want for that purpose. Provided we can prove that it behaves as advertised, we can then use Lean's type class mechanism to associate this decision procedure to the predicate. Once we do so, we can use the predicate `even` in conditional statements in any program.

In any case, in order to *prove* assertions like the ones above (at least, the ones that are true), we need a proof language. Fortunately, dependent type theory can play that role: proofs are nothing more than certain kinds of expressions in the formal language. In the encoding used, if `p` is any proposition, a proof of `p` is just an expression `e` of type `p`. Thus, in Lean, checking a proof is just a special case of checking that an expression is well-formed and has a given type. We can prove that 10 is even as follows:

```
example : even 10 := ⟨5, rfl⟩
```

In general, to prove an existential statement, it is enough to present a witness to the existential quantifier and then show that the subsequent claim is true of that witness. The Unicode angle brackets just package this data together; you can enter them in an editor with `\<` and `\>`, or use the ASCII equivalents `(|` and `|)`. The second component, `rfl`,

is short for reflexivity. Lean's kernel can verify that `10 = 2 * 5` by reducing both sides and confirming that they are, in fact, identical. (For longer expressions, Lean's simplifier, which will be discussed below, can do this more efficiently, producing a proof instead that carries out the calculation using binary representations.)

As noted above, dependent type theory is designed to serve as a mathematical foundation, so that any conventional mathematical assertion can be reasonably expressed, and any theorem that can be proved using conventional mathematical means can be carried out formally, with enough effort. Here is a proof that the sum of two even numbers is even:

```
theorem even_add : ∀ m n, even m → even n → even (n + m) :=
take m n,
assume ⟨k, (hk : m = 2 * k)⟩,
assume ⟨l, (hl : n = 2 * l)⟩,
have n + m = 2 * (k + l),
  by simp [hk, hl, mul_add],
show even (n + m),
  from ⟨_, this⟩
```

Again, we emphasize that the proof is really just an expression in dependent type theory, presented with syntactic sugar that makes it look somewhat like any informal mathematical proof. There is also a tiny bit of automated reasoning thrown in: the command `by simp` calls on Lean's built-in simplifier to prove the assertion after the `have`, using the two facts labelled `hk` and `hl`, and the distributivity of multiplication over addition.

Lean supports another style of writing proofs, namely, using *tactics*. These are instructions, or procedures, that tell Lean how to construct the requisite expression. Here is a tactic-style proof of the theorem above:

```
theorem even_add : ∀ m n, even m → even n → even (n + m) :=
begin
  intros m n hm hn,
  cases hm with k hk,
  cases hn with l hl,
  unfold even,
  existsi (k + l),
  simp [hk, hl, mul_add]
end
```

Just as we can prove statements about the natural numbers, we can also reason about computer programs written in Lean, because these, too, are no different from any other definitions. This enables us to specify properties of computer programs, prove that the programs meet their specifications, and run the code with confidence that the results mean what we think they mean.

The use of `simp` in the proof above points to another aspect of Lean, namely, that it can serve as a gateway to the use of automated reasoning. Terms in dependent type theory can be very verbose, and formal proofs can be especially long. One of Lean's strengths is that

it can help you construct these terms, and hide the details from you. We have already seen hints of this: in the examples above, Lean inferred the fact that the natural numbers form an instance of a semiring in order to make use of the theorem `mul_add`, it found a procedure for comparing two natural numbers when we applied `insertion_sort` with the less-than ordering, and it did some work behind the scenes (though in this case, not much) when transforming the recursive specification of addition on the natural numbers to a formal definition. But a central goal of the Lean project is to develop powerful automation that will assist in the verification of programs and the construction of proofs as well.

It is the tactic framework that serves as a gateway to the use of automation. Lean provides means of implementing automated reasoning procedures in such a way that they produce formal proofs that their results are correct. This imposes an extra burden on the implementation, but it comes with benefits as well: automated procedures can make full use of the Lean library and API, and the formal justifications they produce provide a strong guarantee that the results are indeed correct.

Which brings us to yet another aspect of Lean, namely, its role as a *metaprogramming language*. Many of Lean's internal data structures and procedures are exposed and available within the language of Lean itself, via a monadic interface. We refer to the use of these procedures as "metaprogramming" because they take us outside the formal framework: the access points to the API are declared as constants, and the formal framework knows nothing about them, other than their type. Lean keeps track of which objects in the environment are part of the trusted kernel and which make use of this special API, and requires us to annotate the latter definitions with the special keyword `meta`. The virtual machine, however, handles calls to the API appropriately. This makes it possible to write Lean tactics in Lean itself.

For example, the procedure `contra_aux` searches through two lists of expressions, assumed to be hypotheses available in the context of a tactic proof, in search of a pair of the form $\texttt{h}_1$ `:` `p` and $\texttt{h}_2$ `:` $\neg$ `p`. When it finds such a pair, it uses it to produce a proof of the resulting theorem. The procedure `contra` then applies `contra_aux` to the hypotheses in the local context.

```
open expr tactic

private meta def contra_aux : list expr → list expr → tactic unit
| []         hs := failed
| (h₁ :: rs) hs :=
  do t₀ ← infer_type h₁,
     t  ← whnf t₀,
     (do a ← match_not t,
         h₂ ← find_same_type a hs,
         tgt ← target,
         pr ← mk_app `absurd [tgt, h₂, h₁],
         exact pr)
     <|> contra_aux rs hs

meta def contra : tactic unit :=
```

```
do ctx ← local_context,
   contra_aux ctx ctx
```

Having defined this procedure, we can then use it to prove theorems:

```
example (p q r : Prop) (h₁ : p ∧ q) (h₂ : q → r) (h₃ : ¬ (p ∧ q)) : r :=
by contra
```

The results of such a tactic are always checked by the Lean kernel, so they can be trusted, even if the code itself is buggy. If the kernel fails to type check the resulting term, it raises an error, and the resulting theorem is not added to the environment.

Substantial tactics can be written in such a way, even, as noted above, a full-blown resolution theorem prover. Indeed, many of Lean's core tactics *are* implemented in Lean itself. The code from `contra` above is, in fact, part of the `contradiction` tactic that is part of Lean's standard library. Thus Lean offers a language for expressing not just mathematical knowledge, construed as a body of definitions and theorems, but also other kinds of mathematical expertise, namely the algorithms, procedures, and heuristics that are part and parcel of mathematical understanding.

1.2 Where To Go From Here

We have surveyed a number of ways that Lean can be used, namely, as

- a specification language
- a programming language
- an assertion language
- a proof language
- a gateway to using automation with fully verified results, and
- a metaprogramming language.

Subsequent chapters provide a compendium of examples for you to play with and enjoy. These chapters are fairly short on explanation, however, and are not meant to serve as definitive references. If you are motivated to continue using Lean in earnest, we recommend continuing, from here, to either of the following more expansive introductions:

- Theorem Proving in Lean
- Programming in Lean

The first focuses on the use of Lean as a theorem prover, whereas the second focuses on aspects of Lean related to programming and metaprogramming.

2

Defining Objects in Lean

As a foundational framework, the Calculus of Inductive Constructions, or CIC, is flexible enough to define all kinds of mathematical objects. It can define the number systems, ranging from the natural numbers to the complex numbers; algebraic structures, from semigroups to categories and modules over an arbitrary ring; limits, derivatives, and integrals, and other components of real and complex analysis; vector spaces and matrices; measure spaces; and much more.

This flexibility is not so mysterious. Common set-theoretic constructions are mirrored in dependent type theory, so type-theoretic definitions of many mathematical objects are not so different from their set-theoretic counterparts. As the name suggests, the notion of an inductively defined type is central to the CIC, and forms the basis for a number of fundamental constructions.

There are important differences between set theory and type theory, however. In axiomatic set theory, there is only fundamentally one type of object, in that every object is a set. In type theory, a natural number, a function from the reals to the reals, and a vector space are all different types of objects, and it doesn't even make sense to ask whether an object of one type is equal to an object of another. Similarly, it does not make sense to apply a vector operation to a natural number, without (implicitly or explicitly) coercing the natural number to a vector in some way. Giving up the free-wheeling flexibility of set theory has some disadvantages, but it has a number of advantages, too. It makes it possible to communicate more efficiently, since a good deal of information can be inferred from an understanding of the types of objects involved. A typing discipline also makes it possible for a system like Lean to detect and flag errors, such as sending the wrong sorts of arguments to a function.

Other differences between set-theoretic foundations and the type theoretic foundation

implemented by Lean stem from the fact that computation is a central part of the latter. We will see below that in Lean one can define noncomputable functions and operations, as is common in everyday mathematics. But Lean tracks the use of such classical constructions, and any definition in Lean that is not explicitly tagged as noncomputable generates executable code. Below, we will illustrate this, by evaluating computable objects as we go. We will discuss computation in greater detail in the next chapter.

2.1 Some Basic Types

Remember that, in Lean:

- `#check` can be used a check the type of an expression.
- `#print` can be used to print information about an identifier, for example, the definition of a defined constant.
- `#reduce` can be used to normalize a symbolic expression.
- `#eval` can be used to run the bytecode evaluator on any closed term that has a computational interpretation.

Lean's standard library defines a number of data types, such as `nat`, `int`, `list`, and `bool`.

```
#check nat
#print nat

#check int
#print int

#check list
#print list

#check bool
#print bool
```

You can use the unicode symbols ℕ and ℤ for nat and int, respectively. The first can be entered with `\N` or `\nat`, and the second can be entered with `\Z` or `\int`.

The library includes standard operations on these types:

```
#check 3 + 6 * 9
#eval 3 + 6 * 9

#check 1 :: 2 :: 3 :: [4, 5] ++ [6, 7, 8]
#eval 1 :: 2 :: 3 :: [4, 5] ++ [6, 7, 8]

#check tt && (ff || tt)
#eval tt && (ff || tt)
```

Remember that, by default, a numeral denotes a natural number. You can always specify an intended type `t` for an expression `e` by writing `(e : t)`. In that case, Lean does its best to interpret the expression as an object of the given type, and raises an error if it does not succeed.

```
#check (3 : ℤ)
#check (3 : ℤ) + 6 * 9
#check (3 + 6 * 9 : ℤ)

#eval (3 + 6 * 9 : ℤ)
```

We can also declare variables ranging over elements and types.

```
variables m n k : ℕ
variables u v w : ℤ
variable  α : Type
variables l₁ l₂ : list ℕ
variables s₁ s₂ : list α
variable  a : α

#check m + n * k
#check u + v * w
#check m :: l₁ ++ l₂
#check s₁ ++ a :: s₂
```

The standard library adopts the convention of using the Greek letters α, β, and γ to range over types. You can type these with `\a`, `\b`, and `\g`, respectively. You can type subscripts with `\0`, `\1`, `\2`, and so on.

Lean will insert coercions automatically:

```
#check v + m
```

The presence of a coercion is indicated by Lean's output, `v + ↑m : ℤ`. Since Lean infers types sequentially as it processes an expression, you need to indicate the coercion manually if you write the arguments in the other order:

```
#check ↑m + v
```

You can type the up arrow by writing `\u`. This is notation for a generic coercion function, and Lean finds the appropriate one using type classes, as described below. The notations `+`, `*`, `++` similarly denote functions defined generically on any type that supports the relevant operations:

```
#check @has_add.add
#print has_add.add
```

```
#check @has_mul.mul
#print has_mul.mul

#check @append
#print append
```

Here, the `@` symbol before the name of the function indicates that Lean should display arguments that are usually left implicit. These are called, unsurprisingly, *implicit arguments*. In the examples above, type class resolution finds the relevant operations, which are declared in the relevant *namespaces*.

```
#check nat.add
#check nat.mul
#check list.append
#check list.cons
```

When generic functions and notations are available, however, it is usually better to use them, because Lean's automation is designed to work well with generic functions and facts. Incidentally, when infix notation is defined for a binary operation, Lean's parser will let you put the notation in parentheses to refer to the operation in prefix form:

```
#check (+)
#check (*)
#check (≤)
```

Lean knows about Cartesian products and pairs:

```
variables α β : Type
variables (a₁ a₂ : α) (b : β) (n : ℕ)
variables (p : α × β) (q : α × ℕ)

#check α × β
#check (a₁, a₂)
#check (n, b)
#check p.1
#check p.2

#reduce (n, b).1
#reduce (2, 3).1
#eval (2, 3).1
```

It interprets tuples as iterated products, associated to the right:

```
variables α β : Type
variables (a₁ a₂ : α) (b : β) (n : ℕ)

#check (n, a₁, b)
#reduce (n, a₁, b).2
#reduce (n, a₁, b).2.2
```

Lean also knows about subtypes and option types, which are described in the next chapter.

2.2 Defining Functions

In Lean, one can define a new constant with the `definition` command, which can be abbreviated to `def`.

```
definition foo : ℕ := 3

def bar : ℕ := 2 + 2
```

As with the `#check` command, Lean first attempts to elaborate the given expression, which is to say, fill in all the information that is left implicit. After that, it checks to make sure that the expression has the stated type. Assuming it succeeds, it creates a new constant with the given name and type, associates it to the expression after the `:=`, and stores it in the environment.

The type of functions from α to β is denoted $\alpha \to \beta$. We have already seen that a function `f` is applied to an element `x` in the domain type by writing `f x`.

```
variables α β : Type
variables (a₁ a₂ : α) (b : β) (n : ℕ)
variables f : ℕ → α
variables g : α → β → ℕ

#check f n
#check g a₁
#check g a₂ b
#check f (g a₂ b)
#check g (f (g a₂ b))
```

Conversely, functions are introduced using λ abstraction.

```
variables (α : Type) (n : ℕ) (i : ℤ)

#check λ x : ℕ, x + 3
#check λ x, x + 3
#check λ x, x + n
#check λ x, x + i
#check λ x y, x + y + 1
#check λ x : α, x
```

As the examples make clear, you can leave out the type of the abstracted variable when it can be inferred. The following two definitions mean the same thing:

```
def foo : ℕ → ℕ := λ x : ℕ, x + 3
def bar := λ x, x + 3
```

Instead of using a lambda, you can abstract variables by putting them before the colon:

```
def foo (x y : ℕ) : ℕ := x + y + 3
def bar x y := x + y + 3
```

You can even test a definition without adding it to the environment, using the `example` command:

```
example x y := x + y + 3
```

When variables have been declared, functions implicitly depend on the variables mentioned in the definition:

```
variables (α : Type) (x : α)
variables m n : ℕ

def foo := x
def bar := m + n + 3
def baz k := m + k + 3

#check foo
#check bar
#check baz
```

Evaluating expressions involving abstraction and application has the expected behavior:

```
#reduce (λ x, x + 3) 2
#eval (λ x, x + 3) 2

def foo (x : ℕ) : ℕ := x + 3

#reduce foo 2
#eval foo 2
```

Both expressions evaluate to 5.

In the CIC, types are just certain kinds of objects, so functions can depend on types. For example, the following defines a polymorphic identity function:

```
def id (α : Type) (x : α) : α := x

#check id ℕ 3
#eval id ℕ 3

#check id
```

Lean indicates that the type of `id` is Π α `: Type,` $\alpha \to \alpha$. This is an example of a *pi type*, also known as a dependent function type, since the type of the second argument to `id` depends on the first.

It is generally redundant to have to give the first argument to `id` explicitly, since it can be inferred from the second argument. Using curly braces marks the argument as *implicit*.

```
def id {α : Type} (x : α) : α := x

#check id 3
#eval id 3

#check id
```

In case an implicit argument follows the last given argument in a function application, Lean inserts the implicit argument eagerly and tries to infer it. Using double curly braces `{{ … }}`, or the unicode equivalents obtained with `\{{` and `\}}`, tells the parser to be more conservative about inserting the argument. The difference is illustrated below.

```
def id₁ {α : Type} (x : α) : α := x
def id₂ {{α : Type}} (x : α) : α := x

#check (id₁ : ℕ → ℕ)
#check (id₂ : Π α : Type, α → α)
```

In the next section, we will see that Lean supports a hierarchy of type universes, so that the following definition of the identity function is more general:

```
universe u
def id {α : Type u} (x : α) := x
```

If you `#check @list.append`, you will see that, similarly, the append function takes two lists of elements of any type, where the type can occur in any type universe.

Incidentally, subsequent arguments to a dependent function can depend on arbitrary parameters, not just other types:

```
variable vec : ℕ → Type
variable foo : Π {n : ℕ}, vec n → vec n
variable v : vec 3

#check foo v
```

This is precisely the sense in which dependent type theory is dependent.

The CIC also supports recursive definitions on inductively defined types.

```
open nat

def exp (x : ℕ) : ℕ → ℕ
| 0      := 1
| (succ n) := exp n * (succ n)
```

We will provide lots of examples of those in the next chapter.

2.3 Defining New Types

In the version of the Calculus of Inductive Constructions implemented by Lean, we start with a sequence of type universes, `Sort 0`, `Sort 1`, `Sort 2`, `Sort 3`, … The universe `Sort 0` is called `Prop` and has special properties that we will describe later. `Type u` is a syntactic sugar for `Sort (u+1)`. For each `u`, an element `t : Type u` is itself a type. If you execute the following,

```
universe u
#check Type u
```

you will see that each `Type u` itself has type `Type (u+1)`. The notation `Type` is shorthand for `Type 0`, which is a shorthand for `Sort 1`.

In addition to the type universes, the Calculus of Inductive Constructions provides two means of forming new types:

- pi types
- inductive types

Lean provides an additional means of forming new types:

- quotient types

We discussed pi types in the last section. Quotient types provide a means of defining a new type given a type and an equivalence relation on that type. They are used in the standard library to define, for example, the rational numbers, and a computational representation of finite sets (as lists, without duplicates, up to permutation).

Inductive types are suprisingly useful. The natural numbers are defined inductively:

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

So is the type of lists of elements of a given type α:

```
universe u

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

The booleans form an inductive type, as do, indeed, any finitely enumerated type:

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

inductive Beatle : Type
| John   : Beatle
| Paul   : Beatle
| George : Beatle
| Ringo  : Beatle
```

So are the type of binary trees, and the type of countably branching trees in which every node has children indexed by the type of natural numbers:

```
inductive binary_tree : Type
| empty : binary_tree
| cons  : binary_tree → binary_tree → binary_tree

inductive nat_tree : Type
| empty : nat_tree
| sup   : (ℕ → nat_tree) → nat_tree
```

What these examples all have in common is that the associated types are built up freely and inductively by the given *constructors*. For example, we can build some binary trees:

```
#check binary_tree.empty
#check binary_tree.cons (binary_tree.empty) (binary_tree.empty)
```

If we open the namespace `binary_tree`, we can use shorter names:

```
open binary_tree

#check cons empty (cons (cons empty empty) empty)
```

In the Lean library, the identifier `empty` is used as a generic notation for things like the empty set, so opening the `binary_tree` namespaces means that the constant is overloaded. If you write `#check empty`, Lean will complain about the overload; you need to say something like `#check (empty : binary_tree)` to disambiguate.

The `inductive` command axiomatically declares all of the following:

- A constant, to denote the new type.
- The associated constructors.
- A corresponding *eliminator*.

The latter gives rise to the principles of recursion and induction that we will encounter in the next two chapters.

We will not give a precise specification of the inductive data types allowed by Lean, but only note here that the description is fairly small and straightforward, and can easily be given a set-theoretic interpretation. Lean also allows *mutual* inductive types and *nested* inductive types. As an example, in the definition below, the type under definition appears as a parameter to the `list` type:

```
inductive tree (α : Type) : Type
| node : α → list tree → tree
```

Such definitions are *not* among Lean's axoimatic primitives; rather, they are compiled down to more primitive constructions.

2.4 Records and Structures

When computer scientists bundle data together, they tend to call the result a *record*. When mathematicians do the same, they call it a *structure*. Lean uses the keyword `structure` to introduce inductive definitions with a single constructor.

```
structure color : Type :=
mk :: (red : ℕ) (green : ℕ) (blue : ℕ) (name : string)
```

Here, `mk` is the constructor (if ommitted, Lean assumes it is `mk` by default), and `red`, `green`, `blue`, and `name` project the four values that are used to construct an element of `color`.

```
def purple := color.mk 150 0 150 "purple"

#eval color.red purple
#eval color.green purple
#eval color.blue purple
#eval color.name purple
```

Because records are so important, Lean provides useful notation for dealing with them. For example, when the type of the record can be inferred, Lean allows the use of *anonymous constructors* $\langle \ldots \rangle$, entered as `\<` and `\>`, or the ascii equivalents `(|` and `|)`. Similarly, one can use the notation `.1`, `.2`, and so on for the projections.

```
def purple : color := ⟨150, 0, 150, "purple"⟩

#eval purple.1
#eval purple.2
#eval purple.3
#eval purple.4
```

Alternatively, one can use the notation `.` to extract the relevant projections:

```
#eval purple.red
#eval purple.green
#eval purple.blue
#eval purple.name
```

When the type of the record can be inferred, you can also use the following notation to build an instance, explicitly naming each component:

```
def purple : color :=
{ red := 150, blue := 0, green := 150, name := "purple" }
```

You can also use the `with` keyword for *record update*, that is, to define an instance of a new record by modifying an existing one:

```
def mauve := { purple with green := 100, name := "mauve" }

#eval mauve.red
#eval mauve.green
```

Lean provides extensive support for reasoning generically about algebraic structures, in particular, allowing the inheritance and sharing of notation and facts. Chief among these is the use of *class inference*, in a manner similar to that used by functional programming languages like Haskell. For example, the Lean library declares the structures `has_one` and `has_mul` to support the generic notation `1` and `*` in structures which have a one and binary multiplication:

```
universe u
variables {α : Type u}

class has_one (α : Type u) := (one : α)
class has_mul (α : Type u) := (mul : α → α → α)
```

The `class` command not only defines a structure (in the cases above, each storing only one piece of data), but also marks them as targets for *class inference*. The symbol `*` is notation for the identifier `has_mul.mul`, and if you check the type of `has_mul.mul`, you will see there is an implicit argument for an element of `has_mul`:

```
#check @has_mul.mul
```

The sole element of the `has_mul` structure is the relevant multiplication, which should be inferred from the type α of the arguments. Given an expression `a * b` where `a` and `b` have type α, Lean searches through instances of `has_mul` that have been declared to the system,

in search of one that matches the type α. When it finds such an instance, it uses that as the argument to `mul`.

With `has_mul` and `has_one` in place, some of the most basic objects of the algebraic hierarchy are defined as follows:

```
universe u
variables {α : Type u}

class semigroup (α : Type u) extends has_mul α :=
(mul_assoc : ∀ a b c : α, a * b * c = a * (b * c))

class comm_semigroup (α : Type u) extends semigroup α :=
(mul_comm : ∀ a b : α, a * b = b * a)

class monoid (α : Type u) extends semigroup α, has_one α :=
(one_mul : ∀ a : α, 1 * a = a) (mul_one : ∀ a : α, a * 1 = a)
```

There are a few things to note here. First, these definitions are introduced as `class` definitions also. This marks them as eligible for class inference, enabling Lean to find the `semigroup`, `comm_semigroup`, or `monoid` structure associated to a type, α, when necessary. The `extends` keyword does two things: it defines the new structure by adding the given fields to those of the structures being extended, and it declares any instance of the new structure to be an instance of the previous ones. Finally, notice that the new elements of these structures are not data, but, rather, *properties* that the data is assumed to satisfy. It is a consequence of the encoding of propositions and proofs in dependent type theory that we can treat assumptions like associativity and commutativity in a manner similar to data. We will discuss this encoding in a later chapter.

Because any monoid is an instance of `has_one` and `has_mul`, Lean can interpret `1` and `*` in any monoid.

```
variables (M : Type) [monoid M]
variables a b : M

#check a * 1 * b
```

The declaration `[monoid M]` declares a variable ranging over the monoid structure, but leaves it anonymous. The variable is automatically inserted in any definition that depends on `M`, and is marked for class inference. We can now define operations generically. For example, the notion of squaring an element makes sense in any structure with a multiplication.

```
universe u
def square {α : Type u} [has_mul α] (x : α) : α := x * x
```

Because `monoid` is an instance of `has_mul`, we can then use the generic squaring operation in any monoid.

```
variables (M : Type) [monoid M]
variables a b : M

#check square a * square b
```

2.5 Nonconstructive Definitions

Lean allows us to to define nonconstructive functions using familiar classical principles, provided we mark the associated definitions as `noncomputable`.

```
open classical
local attribute [instance] prop_decidable

noncomputable def choose (p : ℕ → Prop) : ℕ :=
if h : (∃ n : ℕ, p n) then some h else 0

noncomputable def inverse (f : ℕ → ℕ) (n : ℕ) : ℕ :=
if h : (∃ m : ℕ, f m = n) then some h else 0
```

In this example, declaring the type class instance `prop_decidable` allows us to use a classical definition by cases, depending on whether an arbitrary proposition is true or false. Given an arbitrary predicate `p` on the natural numbers, `choose p` returns an `n` satisfying `p n` if there is one, and `0` otherwise. For example, `p n` may assert that `n` codes a halting computation sequence for some Turing machine, on a given input. In that case, `choose p` magically decides whether or not such a computation exists, and returns one if it doesn't. The second definition makes a best effort to define an inverse to a function `f` from the natural numbers to the natural numbers, mapping each `n` to some `m` such that `f m = n`, and zero otherwise.

The two previous definitions make use of the `some` function, which in turn depends on a construct known as *Hilbert's epsilon*. The next two definitions have essentially the same effect, although they do not specify the default value in case the given condition fails:

```
open classical
local attribute [instance] prop_decidable

noncomputable def choose (p : ℕ → Prop) : ℕ :=
epsilon p

noncomputable def inverse (f : ℕ → ℕ) (n : ℕ) : ℕ :=
epsilon (λ m, f m = n)
```

These definitions rely on the fact that Lean can infer (again using type class inference) that the natural numbers are nonempty. The `epsilon` operator is, in turn, defined from an even more fundamental choice principle, which is the source of all nonconstructive definitions in the standard library. The dependence is made manifest by the `#print axioms` command.

```
#print axioms choose
#print axioms inverse
```

3

Programming in Lean

Lean aims to support both mathematical abstraction alongside pragmatic computation, allowing both to interact in a common foundational framework. Some users will be interested in viewing Lean as a programming language, and making sure that every assertion has direct computational meaning. Others will be interested in treating Lean as a system for reasoning about abstract mathematical objects and assertions, which may not have straightforward computational interpretations. Lean is designed to be a comfortable environment for both kinds of users.

But Lean is also designed to support users who want to maintain both world views at once. This includes mathematical users who, having developed an abstract mathematical theory, would then like to start computing with the mathematical objects in a verified way. It also includes computer scientists and engineers who, having written a program or modeled a piece of hardware or software in Lean, would like to verify claims about it against a background mathematical theory of arithmetic, analysis, dynamical systems, or stochastic processes.

Lean employs a number of carefully chosen devices to support a clean and principled unification of the two worlds. Chief among these is the inclusion of a type `Prop` of propositions, or assertions. If `p` is an element of type `Prop`, you can think of an element `t : p` as representing evidence that `p` is true, or a proof of `p`, or simply the fact that `p` holds. The element `t`, however, does not bear any computational information. In contrast, if α is an element of `Type u` for any `u` greater than 0 and `t : α`, then `t` contains data, and can be evaluated.

We saw in Section 2.5 that Lean allows us to define functions that produce data from a proposition `∃ x, p x`, but that such functions are marked as `noncomputable`, and do not generate bytecode. Expressions `t : α`, where α is a type of data, can contain subexpres-

sions that are elements of `Prop`, and these can even refer to nonconstructive objects. During the extraction of bytecode, these elements are simply ignored, and do not contribute to the computational content of `t`.

For that reason, abstract elements in Lean's library can have *computational refinements*. For example, for every type, α, there is another type, `set` α, of sets of elements of α and some sets satisfy the property of being `finite`. Saying that a set is finite is equivalent to saying that there exists a list that contains exactly the same elements. But this statement is a proposition, which means that it is impossible to extract such a list from the mere assertion that it exists. For that reason, the standard library also defines a type `finset` α, which is better suited to computation. An element `s : finset` α is represented by a list of elements of α without duplicates. Using quotient types, we can arrange that lists that differ up to permutation are considered equal, and a defining principle of quotient types allows us to define a function on `finset` α in terms of any list that represents it, provided that we show that our definition is invariant under permutations of the list. Computationally, an element of `finset` α *is* just a list. Everything else is essentially a contract that we commit ourselves to obeying when working with elements of `finset` α. The contract is important to reasoning about the results of our computations and their properties, but it plays no role in the computation itself.

As another example of the interaction between propositions and data, consider the fact that we do not always have algorithms that determine whether a proposition is true (consider, for example, the proposition that a Turing machine halts). In many cases, however, we do. For example, assertions `m = n` and `m < n` about natural numbers, and Boolean combinations of these, can be evaluated. Propositions like this are said to be *decidable*. Lean's library uses class inference to infer the decidability, and when it succeeds, you can use a decidable property in an `if … then … else` conditional statement. Computationally, what is going on is that class inference finds the relevant procedure, and the bytecode evaluator uses it.

One side effect of the choice of CIC as a foundation is that all functions we define, computational or not, are total. Once again, dependent type theory offers various mechanisms that we can use to restrict the range of applicability of a function, and some will be described below.

3.1 Evaluating Expressions

When translating expressions to byte code, Lean's virtual machine evaluator ignores type information entirely. The whole elaborate typing schema of the CIC serves to ensure that terms make sense, and mean what we think they mean. Type checking is entirely static: when evaluating a term `t` of type α, the bytecode evaluator ignores α, and simply computes the value of `t`, as described below. As noted above, any subexpressions of `t` whose type is an element of `Prop` are computationally irrelevant, and they are ignored too.

The evaluation of expressions follows the computational rules of the CIC. In particular:

- To evaluate a function application `(λ x, s) t`, the bytecode evaluator evaluates `t`, and then evaluates `s` with `x` instantiated to `t`.
- To evaluate an eliminator for an inductively defined type — in other words, a function defined by pattern matching or recursion — the bytecode evaluator waits until all the arguments are given, evaluates the first one, and, on the basis of the result, applies the relevant case or recursive call.

We have already seen that Lean can evaluate expressions involving natural numbers, integers, lists, and booleans.

```
#eval 22 + 77 * 11
#eval tt && (ff || tt)
#eval [1, 2, 3] ++ 4 :: [5, 6, 7]
```

Lean can evaluate conditional expressions:

```
#eval if 11 > 5 ∧ ff then 27 else 33 + 12

#eval if 7 ∈ [1, 3, 5] then "hooray!" else "awww..."
```

Here is a more interesting example:

```
def craps (roll : ℕ) (come_out : bool) (point : ℕ) : string :=
if (come_out ∧ (roll = 7 ∨ roll = 11)) ∨ (¬ come_out ∧ roll = point) then
  "You win!"
else if (come_out ∧ roll ∈ [2, 3, 12]) ∨ (¬ come_out ∧ roll = 7) then
  "You lose!"
else
  "Roll again."

#eval craps 7 tt 4
#eval craps 11 ff 2
```

The standard library defines a number of common operations on lists:

```
#eval list.range 100

#eval list.map (λ x, x * x) (list.range 100)

#eval list.filter (λ x, x > 50) (list.range 100)

#eval list.foldl (+) 0 (list.range 100)
```

A `char` is a natural number that is less than 255. You can enter the character "A," for example, by typing `'A'`. Lean defines some basic operations on characters:

```
open char

#eval to_lower 'X'
#eval to_lower 'x'
#eval to_lower '!'

#eval to_lower '!'

#eval if is_punctuation '?' then tt else ff
```

In the example above, we have to tell Lean how to define a decision procedure for the predicate `is_punctuation`. We do this simply by unfolding the definition and asking Lean to use the inferred decision procedure for list membership.

Strings can be mapped to lists of characters and back, so operations on lists and be used with strings.

```
namespace string

def filter (p : char → Prop) [decidable_pred p] (s : string) : string :=
((s.to_list).filter p).to_string

def map (f : char → char) (l : string) : string :=
(l.to_list.map f).to_string

def to_lower (s : string) : string := s.map char.to_lower

def reverse (s : string) : string := s.to_list.reverse.to_string

def remove_punctuation (s : string) : string :=
s.filter (λ c, ¬ char.is_punctuation c)

end string
```

We can use these to write a procedure that tests to see whether a given sentence is a palindrome.

```
def test_palindrome (s : string) : bool :=
let s' := to_lower (remove_punctuation s) in
if s' = reverse s' then tt else ff

#eval test_palindrome "A man, a plan, a canal -- Panama!"
#eval test_palindrome "Madam, I'm Adam!"
#eval test_palindrome "This one is not even close."
```

3.2 Recursive Definitions

Lean supports definition of functions by structural recursion on its arguments.

```
open nat

def fact : ℕ → ℕ
| 0        := 1
| (succ n) := (succ n) * fact n

#eval fact 100
```

Lean recognizes that addition on the natural numbers is defined in terms of the `succ` constructor, so you can also use more conventional mathematical notation.

```
def fact : ℕ → ℕ
| 0     := 1
| (n+1) := (n+1) * fact n
```

Lean will compile definitions like these down to the primitives of the Calculus of Inductive Constructions, though in the case of `fact` it is straightforward to define it from the primitive recursion principle directly.

Lean's function definition system can handle more elaborate forms of pattern matching with defaults. For example, the following function returns true if and only if one of its arguments is positive.

```
def foo : ℕ → ℕ → ℕ → bool
| (n+1) _     _     := tt
| _     (m+1) _     := tt
| _     _     (k+1) := tt
| _     _     _     := ff
```

We can define the sequence of Fibonacci numbers in a natural way:

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

#eval fib 100
```

The naive implementation runs the risk of an exponential run time, since the computation of `fib (n+2)` calls for two independent computations of `fib n`, one hidden in the computation of `fib (n+1)`. In fact, the current Lean compilation scheme avoids this, because it joins the recursive falls in a single tuple and evaluates them both at once. We can do this explictly, thereby avoiding reliance on the inner workings of Lean's function definition system, by defining an auxiliary function that computes the values in pairs:

```
def fib_aux : ℕ → ℕ × ℕ
| 0     := (0, 1)
```

```
| (n+1) := let p := fib_aux n in (p.2, p.1 + p.2)

def fib n := (fib_aux n).2

#eval fib 100
```

A similar solution is to use additional arguments to accumulate partial results:

```
def fib_aux : ℕ → ℕ → ℕ → ℕ
| 0     a b := b
| (n+1) a b := fib_aux n b (a+b)

def fib n := fib_aux n 0 1

#eval fib 100
```

Functions on lists are naturally defined by structural recursion. These definitions are taken from the standard library:

```
universe u
variable {α : Type u}

def append : list α → list α → list α
| []       l := l
| (h :: s) t := h :: (append s t)

def mem : α → list α → Prop
| a []       := false
| a (b :: l) := a = b ∨ mem a l

def concat : list α → α → list α
| []     a := [a]
| (b::l) a := b :: concat l a

def length : list α → nat
| []       := 0
| (a :: l) := length l + 1

def empty : list α → bool
| []       := tt
| (_ :: _) := ff
```

3.3 Inhabited Types, Subtypes, and Option Types

In the Calculus of Inductive Constructions, every term denotes something. In particular, if `f` has a function type and `t` has the corresponding argument type, the `f t` denotes some object. In other words, a function defined on a type has to be define on *every* element of that type, so that every function is total on its domain.

It often happens that a function is naturally defined only on some elements of a type. For example, one can take the head of a list only if it is nonempty, and one can divide one

rational number or real number by another as long as the second is nonzero. There are a number of ways of handling that in dependent type theory.

The first, and simplest, is to totalize the function, by assigning an arbitrary or conveniently chosen value where the function would otherwise be undefined. For example, it is convenient to take `x / 0` to be equal to `0`. A downside is that this can run counter to mathematical intuitions. But it does give a precise meaning to the division symbol, even if it is a nonconventional one. (The treatment of undefined values in ordinary mathematics is often ambiguous and sloppy anyhow.)

It helps that the Lean standard library defines a type class, `inhabited` α, that can be used to keep track of types that are known to have at least one element, and to infer such an element. The expressions `default` α and `arbitrary` α both denote the element that is inferred. The second is unfolded less eagerly by Lean's elaborator, and should be used to indicate that you do not want to make any assumptions about the value returned (though ultimately nothing can stop a theory making use of the fact that the arbitrary element of nat, say, is chosen to be zero). The list library defines the `head` function as follows:

```
universe u
variable {α : Type u}

def head [inhabited α] : list α → α
| []       := default α
| (a :: l) := a
```

Another possibility is to add a precondition to the function. We can do this because in the CIC, an assertion can be treated as an argument to a function. The following function explicitly requires evidence that the argument `l` is not the empty list.

```
def first : Π (l : list α), l ≠ [] → α
| []        h := absurd rfl h
| (a :: l₀) h := a
```

This contract ensures that `first` will never be called to evaluate the first element of an empty list. The check is entirely static; the evidence is ignored by the bytecode evaluator.

A closely related solution is to use a `subtype`. This simply bundles together the data and the precondition.

```
def first' : {l₀ // l₀ ≠ []} → α :=
λ l, first l.1 l.2
```

Here, the type `{l₀ // l₀ ≠ []}` consists of (dependent) pairs, where the first element is a list and the second is evidence that the list is nonempty. In a similar way, `{n // (n : ℤ) > 0}` denotes the type of positive integers. Using subtypes and preconditions can be inconvenient at times, because using them requires a mixture of proof and calculation.

But subtypes are especially useful when the constraints are common enough that is pays to develop a library of functions that take and return elements satisfying them — in other words, when the subtype is really worthy of being considered a type in its own right.

Yet another solution is to signal the success or failure of the function on the output, using an `option` type. This is defined in the standard library as follows:

```
inductive option (α : Type u)
| none {} : option
| some    : α → option
```

You can think of the return value `none` as signifying that the function is undefined at that point, whereas `some a` denotes a return value of `a`. (The inscription `{}` after the none constructor indicates that the argument α should be marked implicit, even though it cannot be inferred from other arguments.) For example, then `nth` element function is defined in the list library as follows:

```
def nth : list α → nat → option α
| []       n     := none
| (a :: l) 0     := some a
| (a :: l) (n+1) := nth l n
```

To use an element `oa` of type `option` α, one typically has to pattern match on the cases `none` and `some` α. Doing this manually in the course of a computation can be tedious, but it is much more pleasant and natural using *monads*, which we turn to next.

3.4 Monads

This section assumes that you are familiar with the use of monads in functional programming languages, such as Haskell. There are a number of tutorials on monads available online, including a chapter of Programming in Lean.

Monads are well supported in Lean: they are straightforward to encode in dependent type theory, and class inference can be used to infer monads associated with specific types. Lean knows about the option monad:

```
open list

def foo (l₁ l₂ l₃ : list ℕ) : option (list ℕ) :=
do v₁₀ ← nth l₁ 0,
   v₂₀ ← nth l₂ 0,
   v₂₁ ← nth l₂ 1,
   v₃₀ ← nth l₃ 0,
   v₃₁ ← nth l₃ 1,
   v₃₂ ← nth l₃ 2,
   return [v₁₀, v₂₀, v₂₁, v₃₀, v₃₁, v₃₂]
```

```
#eval foo [1, 2, 3] [4, 5] [6, 7]
#eval foo [1, 2, 3] [4, 5] [6, 7, 8]
```

Here, if an calls to `nth` return `none`, `foo` returns `none` as well. But if all the calls are successful, the function constructs the value on the last line and returns it wrapped with the `some` constructor.

Lean also knows about the list monad:

```
open list

def foo : list string :=
do n ← range 10,
   a ← ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j"],
   repeat a n

#eval foo
```

Think of the body of `foo` is choosing a value `n` nondeterministically from `range 10`, and a value `a` nondeterministically from the given list, and returning the value `repeat a n`, which simply repeats the element `a` `n` times. In fact, `repeat a n` is computed for each choices of `n` and `a`, and the results are concatenated into a single list.

The standard library also defines a state monad, and a special `tactic` monad provides metaprogramming access to an internal tactic state, allowing users to write tactics in Lean. We will return to this in a later chapter.

3.5 Input and Output

Lean can access standard input and output via a special `io` monad. From within the foundational system, the functions `put_str` and `get_str` are simply unaxiomatized constants, but when executed by the virtual machine, the perform the desired operations.

The input faculty can only be used when running Lean from a command line, but standard output is associated to function calls in the editor. The following example uses the `io` monad to output a list of instructions solving the *Tower of Hanoi* game for any number of disks.

```
import system.io
variable [io.interface]
open io

def hanoi_aux : ℕ → string → string → string → io unit
| 0     fr to aux := put_str "nothing to do!\n"
| 1     fr to aux := put_str ("move disk 1 from " ++ fr ++ " to " ++
                              to ++ "\n")
| (n+2) fr to aux := do hanoi_aux (n+1) fr aux to,
                        put_str ("move disk " ++ to_string (n+2) ++
                                 " from " ++ fr ++ " to " ++ to ++ "\n"),
```

```
                        hanoi_aux (n+1) aux to fr

def hanoi (n : ℕ) := hanoi_aux n "A" "B" "C"

#eval hanoi 5
```

3.6 An Example: Abstract Syntax

We close this chapter with an example that nicely illustrates the capacities for algebraic abstraction in Lean and the CIC. We first define an abstract type of arithmetic expressions, each of which is either a variable, a numeral, a sum, or a product.

```
inductive arith_expr
| var   : ℕ → arith_expr
| const : ℕ → arith_expr
| plus  : arith_expr → arith_expr → arith_expr
| times : arith_expr → arith_expr → arith_expr

open arith_expr

def sample_expr := plus (times (const 7) (var 0)) (times (const 2) (var 1))
```

Notice that the variables are indexed by the natural numbers.

We can evaluate an arithmetic expression in any semiring, given an assignment to the variables. We can define a variable assignment to be simply a function from the natural numbers to elements of the carrier type, but it is more convenient to specify an assignment by giving a finite list of values for the initial variables and using a default value elsewhere.

```
universe u

def var_assignment (α : Type u) := ℕ → α

def var_assignment_of_list {α : Type u} [inhabited α] (l : list α) : var_assignment α :=
λ n, match (list.nth l n) with
| some a := a
| none   := arbitrary α
end

section
variables (α : Type u) [inhabited α]
instance : has_coe (list α) (var_assignment α) := ⟨var_assignment_of_list⟩
end
```

The instance declaration at the end declares a `var_assignment_to_list` to be coercion. We need to know that α is inhabited to be able to return a default value for variables whose indices are largere than the length of the list.

We can interpret a numeral in any semiring. The following definition uses well-founded recursion to carry out the cast; we use the `div` and `mod` functions to carry the translation

efficiently. The two `have` statements give facts that are needed to justify the well-founded recursion.

```
universe u

namespace semiring
variables {α : Type u} [semiring α]

def of_nat : ℕ → α
| 0 := 0
| 1 := 1
| (n+2) :=
  have n / 2 < n + 2,
    begin abstract { apply lt_of_le_of_lt, apply nat.div_le_self,
                     apply nat.lt_add_of_pos_right, apply nat.succ_pos } end,
  have n % 2 < n + 2,
    begin abstract { apply nat.lt_add_left, apply nat.mod_lt,
                     apply nat.succ_pos } end,
  2 * of_nat (n / 2) + of_nat (n % 2)

end semiring
```

We these in hand, the evaluation function for arithmetic expressions can now be defined using a straightforward recursion on syntax:

```
variables (α : Type u) [semiring α]

def arith_eval (v : var_assignment α) : arith_expr → α
| (var n)       := v n
| (const n)     := semiring.of_nat n
| (plus e₁ e₂)  := arith_eval e₁ + arith_eval e₂
| (times e₁ e₂) := arith_eval e₁ * arith_eval e₂
```

We can now try it out:

```
#eval arith_eval ℕ ↑[5, 7]        sample_expr
#eval arith_eval ℕ ↑[5, 7, 12]    sample_expr
#eval arith_eval ℕ ↑[5]           sample_expr

#check arith_eval ℤ ↑[(5 : ℤ), 7] sample_expr
```

In these examples, we help out the elaborator by indicating that we intend to coerce the list to a `var_assignment`.

4

Theorem Proving in Lean

4.1 Assertions in Dependent Type Theory

We have seen that dependent type theory is flexible enough to encode a broad array of data types and objects. A simple device makes it possible to encode any assertion you might want to make, as well: there is a type `Prop`, whose elements are taken to be propositions. The usual logical connectives are simply functions that take propositions and return a proposition.

```
variables p q r : Prop

#check and  -- Prop → Prop → Prop
#check or   -- Prop → Prop → Prop
#check not  -- Prop → Prop

#check p ∧ (p → r)
#check p ∧ ¬ (q ∨ ¬ r)
```

There is no corresponding constant for implication: if `p` and `q` are propositions, the arrow in `p → q` is in fact an instance of the arrow used to construct function spaces. We will return to this in the next section.

A predicate on a type α is a function from α to `Prop`, and a binary relation on α is a function that takes two arguments in α and returns an element of `Prop`.

```
variables p q : ℕ → Prop
variable  r : ℕ → ℕ → Prop
variables m n : ℕ

#check p m ∧ q n
```

```
#check p m ∧ ¬ r m n → q n

#check ∀ x, p x → q x
#check ∀ x, ∃ y, r x y
```

The universal quantifier is really an instance of the dependent function space construction; again, more on this in the next section. Lean notation supports bounded quantifiers:

```
#check ∀ x ≤ n, p x → q x

variables (α : Type) (s t : list α) (a b : α)

#check ∀ x ∈ s, x ∈ t
```

With these resources, we can start writing substantial mathematical assertions:

```
def dvd (m n : ℕ) : Prop := ∃ k, n = m * k

instance : has_dvd ℕ := ⟨dvd⟩

def even (n : ℕ) : Prop := 2 ∣ n

def prime (p : ℕ) : Prop := p ≥ 2 ∧ ∀ n, n ∣ p → n = 1 ∨ n = p

def Fermat : Prop := ∀ n > 2, ∀ (a b c : ℕ), a ≠ 0 → b ≠ 0 → a^n + b^n ≠ c^n

def Goldbach : Prop := ∀ n > 2, even n → ∃ p q, prime p ∧ prime q ∧ n = p + q

def twin_primes : Prop := ∀ n, ∃ p > n, prime p ∧ prime (p + 2)
```

Of course, what we really want are means to *prove* such assertions, which is what we turn to next.

4.2 Propositions as Types

Given the expressive power of dependent type theory, it should by now not be too surprisingly that the language is rich enough to encode proofs as well. In fact, the CIC employs a device known as the `Curry-Howard isomorphism`, or `propositions as types`, that makes writing and checking proofs especially convenient. Remember that if φ is any expression of type `Prop`, we are thinking of φ as a proposition, or an assertion. In that case, we think of an term `t : φ` as being a proof of φ. The rules of CIC support this interpretation: given `t : φ → ψ` and `s : φ`, then `t s : ψ` describes the result of using modus ponens. To construct a proof of $\varphi \to \psi$, it suffices to construct a proof of `t : ψ`, assuming hypothetically `x : φ`. The resulting proof is written `λ x, t`, which is to say, the proof is an instance of lambda abstraction.

Similar considerations hold of the universal quantifier. The net effect is that we can use the same notation we use for function application to apply theorems to parameters and hypotheses. For example, the theorem `and.left` in the standard library has the following type:

```
∀ {a b : Prop}, a ∧ b → a
```

Notice that the arguments `a` and `b` are implicit. This means that if `h` is any expression of the form `a ∧ b`, then `and.left h : a` is a proof of `a`. Similarly, `add_comm`, which expresses the commutativity of addition for any type that can be seen as an instance of an additive, commutative semigroup has the following type

```
∀ {α : Type u} [s : add_comm_semigroup α] (a b : α), a + b = b + a
```

The second argument, that is, the relevant algebraic structure, is inferred by class inference. Given, say, `m n : ℕ`, the expression `add_comm m n` then represents the fact that `m + n = n + m`.

Now the task of proving a proposition boils down to the task of constructing an expression of the right type, and Lean is designed to help us do this. We can provide such an expression explicitly:

```
example (a b : Prop) : a ∧ b → b ∧ a :=
λ h, and.intro (and.right h) (and.left h)
```

We can use projections and anonymous constructors to express the proof even more concisely, though somewhat cryptically:

```
example (a b : Prop) : a ∧ b → b ∧ a :=
λ h, ⟨h.right, h.left⟩
```

In the opposite direction, Lean provides syntactic sugar that allows us to annotation assumptions and goals, and build a proof incrementally:

```
example (a b : Prop) : a ∧ b → b ∧ a :=
assume h : a ∧ b,
have ha : a, from and.left h,
have hb : b, from and.right h,
show b ∧ a, from and.intro hb ha
```

You can write proofs incrementally using `sorry` to temporarily fill in any intermediate step.

```
example (a b : Prop) : a ∧ b → b ∧ a :=
assume h : a ∧ b,
have ha : a, from sorry,
have hb : b, from sorry,
show b ∧ a, from and.intro hb ha
```

Lean notices that you are cheating, but will otherwise confirm that the proof is correct modulo the instances of `sorry`. Replacing one of them by an underscore tells Lean that it should infer the value of that expression. Lean's elaborator will not prove propositions for us without explicit instructions to do so, but the error message will show you exactly what needs to be proved, and what hypotheses are available.

Lean supports the use of `tactics`, which are instructions which tell the system how to construct a term or proof.

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

These commands can be used to invoke automation, like the simplifier:

```
example (a b : Prop) : a ∧ b → b ∧ a :=
begin intro, simp_using_hs end
```

Anywhere Lean's parser expects an expression, you can enter tactic mode with a `begin ... end` block, or with the `by` keyword.

```
example (a b : Prop) : a ∧ b → b ∧ a :=
assume h : a ∧ b,
have ha : a, from h.left,
have hb : b, from h.right,
show b ∧ a,
  begin split, repeat { assumption } end
```

Conversely, in a `begin ... end` block, Lean provides various ways of specifying an explicit term:

```
example (a b : Prop) : a ∧ b → b ∧ a :=
begin
  intro h, cases h with ha hb,
  exact and.intro hb ha
end
```

We can even pass back and forth between the two modes freely:

```
example (a b : Prop) : a ∧ b → b ∧ a :=
begin
  intro h, cases h with ha hb,
  exact and.intro (by assumption) (by assumption)
end
```

This lets us write proofs in a manner that lays out the structure explicitly and provides briefer hints and instructions where appropriate, just as in an ordinary mathematical proof.

When writing proof terms explicitly, Lean provides the word `suppose` to introduce an assumption without a label, and the label can be ommitted in the `have` command as well. In this case, we can refer to the anonymous fact that was most recently added to the context with the keyword `this`. We can also refer to them by surrounding the statement of the proposition with French quotes, obtained by typing `\f<` and `\f>`.

```
example (a b : Prop) : a ∧ b → b ∧ a :=
suppose a ∧ b,
have a, from this.left,
have b, from ‹a ∧ b›.right,
show b ∧ a, from and.intro ‹b› ‹a›
```

These anonymous elements of the context are also visible to tactics and automation:

```
example (a b : Prop) : a ∧ b → b ∧ a :=
suppose a ∧ b,
have a, from this.left,
have b, from ‹a ∧ b›.right,
show b ∧ a, begin split, repeat { assumption } end

example (a b : Prop) : a ∧ b → b ∧ a :=
suppose a ∧ b,
have a, from this.left,
have b, from ‹a ∧ b›.right,
show b ∧ a, by simp_using_hs
```

4.3 Induction and Calculation

Because inductive types are so fundamental, Lean's proof language provides a number of ways of carrying out proofs by induction. Suppose, for example, we define exponentiation generically in any monoid.

```
universe u
variable {α : Type u}
variable [monoid α]

open nat

def pow (a : α) : ℕ → α
```

```
| 0       := 1
| (n + 1) := a * pow n

infix `^` := pow

theorem pow_zero (a : α) : a^0 = 1 := rfl

theorem pow_succ (a : α) (n : ℕ) : a^(succ n) = a * a^n := rfl
```

We use the rewrite tactic `rw` to rewrite an expression with a sequence of identities.

The theorem `pow_succ` states that `a^(succ n) = a * a^n`. The monoid in question is not assumed to be commutative, so it requires a proof by induction to show that `a^(succ n) = a^n * a`.

```
theorem pow_succ' (a : α) (n : ℕ) : a^(succ n) = a^n * a :=
nat.rec_on n
  (show a^(succ 0) = a^0 * a,
    by simp [pow_zero, one_mul, pow_succ])
  (take n,
    assume ih : a^(succ n) = a^n * a,
    show a^(succ (succ n)) = a^(succ n) * a,
      by rw [pow_succ, ih, -mul_assoc, -pow_succ, ih])
```

By propositions as types, the same principle `nat.rec_on` governs both induction and recursion on the natural numbers, and works as you would expect: you prove the base case, and then carry out the induction step. Lean has a special proof mode, `calc`, that facilitates writing calculational proofs. It can be used in this case to make the argument more readable:

```
theorem pow_succ' (a : α) (n : ℕ) : a^(succ n) = a^n * a :=
nat.rec_on n
  (show a^(succ 0) = a^0 * a,
    by simp [pow_zero, one_mul, pow_succ])
  (take n,
    assume ih : a^(succ n) = a^n * a,
    show a^(succ (succ n)) = a^(succ n) * a, from
      calc
        a^(succ (succ n)) = a * a^(succ n) : by rw pow_succ
                      ... = a * (a^n * a)  : by rw ih
                      ... = (a * a^n) * a  : by rw mul_assoc
                      ... = a^(succ n) * a : by rw -pow_succ)
```

The `calc` mode can be used with inequalities and transitive relations that have been registered with the system.

By the propositions-as-types correspondence, induction is just a form of recursion, and so the function definition system can be used to write proofs by induction as well.

```
theorem pow_succ' (a : α) : ∀ n, a^(succ n) = a^n * a
| 0       := by simp [pow_zero, one_mul, pow_succ]
```

```
| (succ n) := by rw [pow_succ, pow_succ' n, -mul_assoc,
                     -pow_succ, pow_succ' n]
```

Here the rewrite tactic uses the inductive hypothesis `pow_succ' n`. In an inductive proof like this, structurally decreasing calls can be used.

Finally, one can write a tactic proof using the `induction` tactic, which will revert any hypotheses in the context that depend on the induction variable and then generalize them again. The `with` clause names the variable used in the inductive step, as well as the inductive hypothesis.

```
theorem pow_succ' (a : α) (n : ℕ) : a^(succ n) = a^n * a :=
begin
  induction n with n ih,
  { simp [pow_zero, one_mul, pow_succ] },
  rw [pow_succ, ih, -mul_assoc, -pow_succ, ih]
end
```

Here is another example of proof that uses the induction tactic.

```
theorem pow_add (a : α) (m n : ℕ) : a^(m + n) = a^m * a^n :=
begin
  induction n with n ih,
  { simp [add_zero, pow_zero, mul_one] },
  rw [add_succ, pow_succ', ih, pow_succ', mul_assoc]
end
```

Recall the recursive definitions of the `append` and `length` functions for lists from Section 3.2.

```
def append : list α → list α → list α
| []       l := l
| (h :: s) t := h :: (append s t)

def length : list α → nat
| []       := 0
| (a :: l) := length l + 1
```

The natural way to prove things about these is to use induction on lists. Here are some examples.

```
theorem append_nil (t : list α) : t ++ [] = t :=
begin induction t with a t ih, reflexivity, simp [nil_append, cons_append, ih] end

theorem append.assoc (s t u : list α) : s ++ t ++ u = s ++ (t ++ u) :=
begin induction s with a s ih, reflexivity, simp [cons_append, ih] end

theorem length_append (s t : list α) : length (s ++ t) = length s + length t :=
begin
```

```
  induction s with a s ih,
  simp [nil_append, length],
  simp [length_cons, cons_append, ih]
end

theorem eq_nil_of_length_eq_zero : ∀ {l : list α}, length l = 0 → l = []
| []     h := rfl
| (a::s) h := by contradiction

theorem ne_nil_of_length_eq_succ : ∀ {l : list α} {n : nat}, length l = succ n → l ≠ []
| []     n h := by contradiction
| (a::l) n h := begin intro leq, contradiction end
```

The first three are tactic-style proofs, whereas the last two use the function definition package

4.4 Axioms

[To do: describe all the axioms of Lean, including classical axioms.]

5

Using Automation in Lean

[Under construction.]

6

Metaprogramming in Lean

[Under construction.]

Bibliography