Programming in Lean

Jeremy Avigad
Leonardo de Moura
Jared Roesch

Version 527531a, updated at 2017-06-27 09:33:44 +0200

Contents

1

Introduction

Warning. This book is still under construction. It aims to serve as both an introduction and a reference manual for programming in the Lean theorem prover.

We are making this material public now because currently it is the only existing documentation for many of the specifics of the Lean programming language and its API, and we are hoping that the information will be useful to brave souls experimenting with it at this early stage. Most of the chapters are currently only stubs, but comments and feedback on the material that is available will be helpful.

1.1 Lean as a Programming Language

This book can be viewed as a companion to Theorem Proving in Lean, which presents Lean as a system for building mathematical libraries and stating and proving mathematical theorems. From that perspective, the point of Lean is to implement a formal axiomatic framework in which one can define mathematical objects and reason about them.

As noted in that book, however, expressions in Lean have a computational interpretation, which is to say, they can be *evaluated.* As long as it is defined in the computational fragment of Lean's foundational language, any closed term of type `nat` – that is, any term of type `nat` without free variables – evaluates to a numeral. Similarly, any closed term of type `list nat` evaluates to a list of numerals, and any closed term of type `bool` evaluates either to the boolean value `tt`, for "true," or `ff`, for "false."

This provides another perspective on Lean: instead of thinking of it as a theorem prover whose language just happens to have a computational interpretation, think of it as a programming language that just happens to come equipped with a rich specification language and an interactive environment for proving that programs meet their specifications. The specification language and proof system are quite powerful, rich enough, in fact, to include all conventional mathematics.

We will see that Lean's underlying logical framework, the Calculus of Inductive Constructions, constitutes a surprisingly good programming language. It is expressive enough to define all sorts of data structures, and it supports powerful abstractions. Programs written in the language can be evaluated efficiently by Lean's virtual-machine interpreter or translated automatically to C++ and compiled.

Viewed from a computational perspective, the Calculus of Inductive Constructions is an instance of a purely functional programming language. This means that a program in Lean is simply an expression whose value is determined compositionally from the values of the other expressions it refers to, independent of any sort of ambient state of computation. There is no notion of storing a result in memory or changing the value of a global variable; computation is just evaluation of expressions. This paradigm makes it easier to reason about programs and verify their correctness. At the same time, we will see that Lean incorporates concepts and abstractions that make it feasible to use this paradigm in practice.

The underlying foundational framework imposes one restriction that is alien to most programming languages, namely, that every program is terminating. So, for example, every "while" loop has to be explicitly bounded, though, of course, we can consider the result of iterating an arbitrary computation `n` times for any given natural number `n`. We will see that Lean provides flexible mechanisms for structural and well-founded recursion, allowing us to define functions in natural ways. At the same, the system provides complementary mechanisms for proving claims, using inductive principles that capture the structure of the function definitions.

1.2 Examples

For example...

[Define something like factorial. Evaluate (use both `reduce` and `eval`).]

[Define operations on lists.]

[Prove things, like `length (reverse l) = length l` or `reverse (reverse l) = l`.]

1.3 Input and Output

People often want to write programs that interact with the outside world, querying users for input and presenting them with output during the course of a computation. Lean's foundational framework has no model of "the real world," but Lean declares `get_str` and `put_str` commands to get an input string from the user and write an input string to output, respectively. Within the foundational system, these are treated as black box operations. But when programs are evaluated by Lean's virtual machine or when they are translated to C++, they have the expected behavior. Here, for example, is a program that prints "hello world":

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

def hello_world : io unit :=
put_str "hello world\n"

#eval hello_world
```

The next example prints the first 100 squares:

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

def print_squares : ℕ → io unit
| 0        := return ()
| (succ n) := print_squares n >>
              put_str (nat.to_string n ++ "^2 = " ++
                       nat.to_string (n * n) ++ "\n")

#eval print_squares 100
```

We will explain the data type `io unit` in Chapter Monads. Although this program has a real world side effect of sending output to the screen when run, that effect is invisible to the formal foundation. From the latter's perspective, the type constructor `io` and the functions `put_str` and `get_str` are entirely opaque, objects about which that the axiomatic system has nothing to say. The `print axioms` command shows that the expression `hello world` depends on the constants `io` and `put_str`, which have been forcibly added to the axiomatic system.

```
#print axioms hello_world
```

In this way, we can prove properties of programs involving `io` that do not depend in any way on the particular results of the input and output.

1.4 Metaprogramming in Lean

Lean also allows *metaprograms*, which are Lean programs that involve objects and constructs that are not part of the axiomatic foundation. In particular:

- Metaprograms can use arbitrary recursive calls, with no concern for termination.

- Metaprograms can access *metaconstants*, that is, primitive functions and objects that are implemented internally in Lean and are not meant to be trusted by the foundational framework.

Such definitions can be introduced using the keywords `meta def` instead of `def` and are marked for special treatment. In particular, because they are not part of the axiomatic foundation, they cannot appear as part of ordinary Lean definitions and theorems.

For example, the following definition computes McCarthy's 91 function, without verifying that the computation terminates on all inputs (though, in fact, it does):

```
meta def m91 : ℕ → ℕ
| n := if n > 100 then n - 10 else m91 (m91 (n + 11))

#eval m91 10
#eval m91 100
#eval m91 1000
```

We can print out the first 120 values of `m91`:

```
meta def print_m91 : ℕ → io unit
| 0        := return ()
| (succ n) := print_m91 n >>
              put_str ("m91 " ++ nat.to_string n ++ " = " ++
                       nat.to_string (m91 n) ++ "\n")

#eval print_m91 120
```

Of course, such uses of recursion are dangerous.

```
meta def foo : ℕ → ℕ
| n := foo n + 1

#reduce foo
-- #eval foo 0
```

Evaluating `foo` using the kernel evaluator shows that the implementation is a bit of a hack; the term in the definition includes a macro which names `foo` itself. The virtual machine that evaluates foo goes further, and carries out the recursive call, repeating this until the process runs out of memory. It is a good thing that Lean will not allow `foo` to appear in a `theorem` or in an ordinary `definition`; if we could prove `foo = foo + 1` then, substracting `foo` from both sides, we could prove `0 = 1`, and hence a contradiction.

Although metaprograms can be used in various ways, its primary purpose is to provide a means of extending the functionality of Lean, within Lean itself. For example, we can use metaprograms to write new procedures, known as *tactics*, which help us construct proofs.

This next example assumes you are familiar with the notion of a tactic, as described in *Theorem Proving in Lean*.

The following code implements a tactic that, given any goal, repeatedly finds a hypothesis `h` of the form `a ∧ b`, and replaces it by hypotheses (with fresh names) for `a` and `b`.

```
meta def destruct_conjunctions : tactic unit :=
repeat (do
  l ← local_context,
  first $ l^.for (λ h, do
    ht ← infer_type h >>= whnf,
    match ht with
    | `(and %%a %%b) := do
      n ← mk_fresh_name,
      mk_mapp ``and.left [none, none, some h] >>= assertv n a,
      n ← mk_fresh_name,
      mk_mapp ``and.right [none, none, some h] >>= assertv n b,
      clear h
    | _ := failed
    end))
```

We will explain the details in Chapter 8 but, roughly speaking, the code repeats the following action until there is nothing left to do: get the list of hypotheses in the local context, find a hypothesis `h` whose type is a conjunction, add new hypotheses justified by `and.left h` and `and.right h` to the local context, and then delete `h`. We can then use `destruct_conjunctions` like any other Lean tactic.

```
example (a b c : Prop) (h : (a ∧ b) ∧ (c ∧ a)) : c :=
begin destruct_conjunctions >> assumption end
```

Note that the reason we can use such code to prove theorems without compromising the integrity of the formal system is that Lean's kernel always certifies the result. From a foundational point of view, we don't have to worry about the integrity of the code, only the integrity of the resulting proofs.

1.5 Overview of the contents

To summarize, we can use Lean in any of the following ways:

- as a programming language
- as a system for verifying properties of programs
- as a system for writing metaprograms, that is, programs that extend the functionality of Lean itself

Chapters 2 to 7 explain how to use Lean as a programming language. It will be helpful if you have some familiarity with the syntax and meaning of dependent type theory, for example, as presented in *Theorem Proving in Lean* (henceforth *TPL*). But, if not, it is likely that you will be able to pick up the details as we proceed. Similarly, if you are familiar with functional programming, you will be able to move through the material more quickly, but we will try to keep the presentation below self contained.

Chapter 4 in particular deals with the task of proving things about programs. Once again, it will be helpful if you are familiar with the use of Lean as an interactive theorem prover as described in *TPL*, but if not you are encouraged to forge ahead and refer back to *TPL* as necessary.

Finally, Chapter 8 and Chapter 9 deal with metaprogramming aspects of Lean, and, in particular, writing tactics and automation.

2 Programming Basics

[This chapter should be a straightforward introduction to functional programming, with examples using natural numbers, lists, strings, and so on. Cover if ... then ... else statements, simple instances of matching and recursion. A fuller treatment of matching and recursion will be given in Chapter ??]

3
Data Structures

[Discuss the most common and useful structures, pairs, lists, enumeration types, structures. Introduce subtypes with some simple examples, without including too much detail about propositions and proofs.]

4

Verifying Properties of Programs

[This chapter will have to assume some familiarity with Lean as a proof assistant. Give some natural examples, for example, proving properties of functions of lists, sorting routines, properties of the extended gcd. Discuss two styles: separating functions and properties, and combining them, using subtypes.]

5 Recursion

[Explain Lean's function definition package, how to use well-founded recursion, and how to prove things about general recursive functions. Explain how to define and use new inductive types.]

6

Type Classes

[Explain how to make things instances of common type classes, including arithmetic operations, string formatting operations, etc.]

7

Monads

In this chapter, we will describe a powerful abstraction known as a *monad*. A monad is a type constructor `m : Type` $\to$ `Type` that comes equipped with two special operations, `return` and `bind`. If α is any type, think of `m` α as being a "virtual α," or, as some people describe it, "an α inside a box."

For a given monad `m`, the function `return` has type Π `{`α `: Type},` $\alpha \to$ `m` α. The idea is that for any element `a :` α, `return a` produces the virtual version of `a`, or puts `a` inside the box.

Once we are inside the box, we cannot get out; there is no general way of taking an element of `m` α and obtaining an element of α. But the `bind` operation gives us a way of turning some operations on α into operations inside the monad. Specifically, for a given monad `m`, the function `bind` has type Π `{`α β `: Type}, m` $\alpha \to$ `(`$\alpha \to$ `m` β`)` $\to$ `m` β. Suppose we have a function `f` that, given any element `a :` α, produces a virtual element of β; in more prosaic terms, `f` has type $\alpha \to$ `m` β. Suppose also that we have a virtual element of α, that is, `ma : m` α. If we could extract from `ma` a corresponding element `a` of α, we could apply `f` to it to get a virtual element of β. We cannot do that in general, but `bind` gives us a way of simulating the compound operation: it applies `f` directly "inside the box," and gives us an element of `m` β.

As an example of how `bind` and `return` can be used, given any function `f :` $\alpha \to \beta$, we can get a function `map f : m` $\alpha \to$ `m` β by defining `map f ma` to be `bind ma (`λ `a, return (f a)`. Roughly, given `ma : m` α, the `bind` reaches into the box, finds an associated `a`, and then puts `f a` back into the box.

For another example, given any element `mma : m (m` α`)`, the expression `monad.bind mma id` has type `m` α. This means that even though we cannot in general extract an element of β from `m` β, we *can* do it when β itself is a virtual type, `m` α. The expression `monad.bind mma id` reaches into the `m (m α)` box, catches hold of an element of `m α`, and simply leaves it in the `m α` box.

If you have never come across the notion of a monad before, these operations will seem quite mysterious. But instances of `return` and `bind` arise in many natural ways, and the goal of this chapter is to show you some examples. Roughly, they arise in situations where `m` is a type construction with the property that functions in the ordinary realm of types can be transported, unformly, into functions in the realm of `m`-types. This should sound quite general, and so it is perhaps not that surprising that monads be instantiated in many different ways. The power of the abstraction is not only that it provides general functions and notation the can be used in all these various instantiations, but also that it provides a helpful way of thinking about what they all have in common.

Lean implements the following common notation. First, we have the infix notation

```
ma >>= f
```

for `bind ma f`. Think of this as saying “take an element `a` out of the box, and send it to `f`.” Remember, we are allowed to do that as long as the return type of `f` is of the form `m β`. We also have the infix notation,

```
ma >> mb
```

for `bind ma (λ a, mb)`. This takes an element `a` out of the box, ignores it entirely, and then returns `mb`. These two pieces of notation are most useful in situations where the act of taking an element of the box can be viewed as inducing a change of state. In situations like that, you can think of `ma >>= f` as saying “do `ma`, take the result, and then send it to `f`.” You can then think of of `ma >> mb` more simply as “do `ma`, then do `mb`.” In this way, monads provide a way of simulating features of imperative programming languages in a functional setting. But, we will see, they do a lot more than that.

Thinking of monads in terms of performing actions while computing results is quite powerful, and Lean provides notation to support that perspective. The expression

```
do a ← ma, t
```

is syntactic sugar for `ma >>= (λ a, t)`. Here `t` is typically an expression that depends on `a`, and it should have type `m β` for some `β`. So you can read `do a ← ma, t` as reaching into the box, extracting an `a`, and then continuing the computation with `t`. Similarly, `do s, t` is syntactic sugar for `s >> t`, supporting the reading “do `s`, then do `t`.” The notation supports iteration, so, for example,

```
do a ← s,
   b ← t,
   f a b,
   return (g a b)
```

is syntactic sugar for

```
bind s (λ a, bind t (λ b, bind (f a b) (λ c, return (g a b)))).
```

It supports the reading "do `s` and extract `a`, do `t` and extract `b`, do `f a b`, then return the value `g a b`."

Incidentally, as you may have guessed, a monad is implemented as a type class in Lean. In other words, `return` really has type

```
Π {m : Type → Type} [monad m] {α : Type}, α → m α},
```

and `bind` really has type

```
Π {m : Type → Type} [monad m] {α β : Type}, m α → (α → m β) → m β.
```

In general, the relevant monad can be inferred from the expressions in which `bind` and `return` appear, and the monad structure is then inferred by type class inference.

There is a constraint, namely that when we use monads all the types we apply the monad to have to live in the same type universe. When all the types in question appear as parameters to a definition, Lean's elaborator will infer that constraint. When we declare variables below, we will satisfy that constraint by explicitly putting them in the same universe.

7.1 The option monad

The `option` constructor provides what is perhaps the simplest example of a monad. Recall that an element of `option` α is either of the form `some a` for some element `a :` α, or `none`. So an element `a` of `option` α is a "virtual α" in the sense of being either an element of α or an empty promise.

The associated `return` is just `some`: given an element `a` of α, `some a` returns a virtual α. It is also clear that we cannot go in the opposite direction: given an element `ma : option` α, there is no way, in general, of producing an element of α. But we can simulate extraction of such an element as long as we are willing to stay in the virtual land of `option`s, by defining `bind` as follows:

```
def bind {α β : Type} (oa : option α) (f : α → option β) :
  option β :=
match oa with
| (some a) := f a
| none     := none
end
```

If the element `oa` is `some a`, we can simply apply `f` to `a`, and otherwise we simply return `none`. Notice how the `do` notation allows us to chain these operations:

```
universe u
variables {α β γ δ : Type.{u}} (oa : option α)
variables (f : α → option β) (g : α → β → option γ)
          (h : α → β → γ → option δ)

example : option β :=
do a ← oa,
   b ← f a,
   return b

example : option δ :=
do a ← oa,
   b ← f a,
   c ← g a b,
   h a b c
```

Think of `f`, `g`, and `h` as being partial functions on their respective domains, where a return value of `none` indicates that the function is undefined for the given input. Intuitively, the second example above returns `h a (f a) (g a (f a))`, assuming `oa` is `some a` and all the subterms of that expression are defined. The expression `h a (f a) (g a (f a))` does not actually type check; for example, the second argument of `h` should be of type β rather than `option` β. But monadic notation allows us to simulate the computation of a possibly undefined term, where the bind operation serves to percolate a value of `none` to the output.

7.2 The list monad

Our next example of a monad is the `list` monad. In the last section we thought of a function `f :` $\alpha \to$ `option` β as a function which, on input α, possibly returns an element of β. Now we will think of a function `f :` $\alpha \to$ `list` β as a function which, on input α, returns a list of possible values for the output. This monad is sometimes also called the `nondeterministic` monad, since we can think of `f` as a computation which may nondeterministically return any of the elements in the list.

It is easy to insert a value `a :` α into `list` α; we define `return a` to be just the singleton list `[a]`. Now, given `la : list` α and `f :` $\alpha \to$ `list` β, how should we define the bind operation `la >>= f`? Intuitively, `la` represents any of the possible values occurring in the list, and for each such element `a`, `f` may return any of the elements in `f a`. We can then gather all the possible values of the virtual application by applying `f` to each element of `la` and merging the results into a single list:

```
def bind {α β : Type} (la : list α) (f : α → list β) : list β :=
join (map f la)
```

Since the example in the previous section used nothing more than generic monad operations, we can replay it in the `list` setting:

```
universe u
variables {α β γ δ : Type.{u}} (la : list α)
variables (f : α → list β) (g : α → β → list γ)
          (h : α → β → γ → list δ)

example : list δ :=
do a ← la,
   b ← f a,
   c ← g a b,
   h a b c
```

Now think of the computation as representing the list of all possible values of the expression `h a (f a) (g a (f a))`, where the bind percolates all possible values of the subexpressions to the final output.

Notice that the final output of the expression is a list, to which we can then apply any of the usual functions that deal with lists:

```
open list

variables {α β γ δ : Type} (la : list α)
variables (f : α → list β) (g : α → β → list γ) (h : α → β → γ → list δ)

example : ℕ :=
length
  (do a ← la,
      b ← f a,
      c ← g a b,
      h a b c)
```

We can also move `length` inside the `do` expression, but then the output lives in ℕ instead of a `list`. As a result, we need to use `return` to put the result in a monad:

```
open list

variables {α β γ δ : Type} (la : list α)
variables (f : α → list β) (g : α → β → list γ)
          (h : α → β → γ → list δ)

example : list ℕ :=
do a ← la,
   b ← f a,
   c ← g a b,
   return (length (h a b c))
```

7.3 The state monad

Let us indulge in science fiction for a moment, and suppose we wanted to extend Lean's programming language with three global registers, `x`, `y`, and `z`, each of which stores a natural number. When evaluating an expression `g (f a)` with `f : α → β` and `g : β → γ`, `f` would start the computation with the registers initialized to `0`, but could read and write values during the course of its computation. When `g` began its computation on `f a`, the registers would be set they way that `g` left them, and `g` could continue to read and write values. (To avoid questions as to how we would interpret the flow of control in terms like `h (k₁ a) (k₂ a)`, let us suppose that we only care about composing unary functions.)

There is a straightforward way to implement this behavior in a functional programming language, namely, by making the state of the three registers an explicit argument. First, let us define a data structure to hold the three values, and define the initial settings:

```
structure registers : Type := (x : ℕ) (y : ℕ) (z : ℕ)

def init_reg : registers := registers.mk 0 0 0
```

Now, instead of defining `f : α → β` that operates on the state of the registers implicitly, we would define a function f_0 `: α × registers → β × registers` that operates on it explicitly. The function f_0 would take an input `a : α`, paired with the state of the registers at the beginning of the computation. It could the do whatever it wanted to the state, and return an output `b : β` paired with the new state. Similarly, we would replace `g` by a function g_0 `: β × registers → γ × registers`. The result of the composite computation would be given by `(`g_0 `(`f_0 `(a, init_reg))).1`. In other words, we would pair the value `a` with the initial setting of the registers, apply f_0 and then g_0, and take the first component. If we wanted to lay our hands on the state of the registers at the end of the computation, we could do that by taking the second component.

The biggest problem with this approach is the annoying overhead. To write functions this way, we would have to pair and unpair arguments and construct the new state explicitly. A key virtue of the monad abstraction is that it manages boilerplate operations in situations just like these.

Indeed, the monadic solution is not far away. By currying the input, we could take the input of f_0 equally well to be `α → registers → β × registers`. Now think of f_0 as being a function which takes an input in α and returns an element of `registers → β × registers`. Moreover, think of this output as representing a computation which starts with a certain state, and returns a value of β and a new state. Lo and behold, *that* is the relevant monad.

To be precise: for any type α, the monad `m α` we are after is `registers → α × registers`. We will call this the state monad for `registers`. With this notation, the function f_0 described above has type `α → m β`, the function g_0 has type `β → m γ`, and the composition of the two on input `a` is `f a >>= g`. Notice that the result is an element of `m` γ, which is to say, it is a computation which takes any state and returns a value of γ paired with a new state. With `do` notation, we would express this instead as `do b ← f a, g b`. If we want to leave the monad and extract a value in γ, we can apply this expression to the initial state `init_reg`, and take the first element of the resulting pair.

The last thing to notice is that there is nothing special about `registers` here. The same trick would work for any data structure that we choose to represent the state of a computation at a given point in time. We could describe, for example, registers, a stack, a heap, or any combination of these. For every type `S`, Lean's library defines the state monad `state S` to be the monad that maps any type α to the type `S` $\to$ α $\times$ `S`. The particular monad described above is then simply `state registers`.

Let us consider the `return` and `bind` operations. Given any `a :` α, `return a` is given by `λ s, (a, s)`. This represents the computation which takes any state `s`, leaves it unchanged, and inserts `a` as the return value. The value of `bind` is tricker. Given an `sa : state S` α and an `f :` α $\to$ `state S` β, remember that `bind sa f` is supposed to "reach into the box," extract an element `a` from `sa`, and apply `f` to it inside the monad. Now, the result of `bind sa f` is supposed to be an element of `state S` β, which is really a function `S` $\to$ β $\times$ `S`. In other words, `bind sa f` is supposed to encode a function which operates on any state to produce an element of β tonad a new state. Doing so is straightforward: given any state `s`, `sa s` consists of a pair `(a, s₀)`, and applying `f` to `a` and then `s₀` yields the required element of β $\times$ `S`. Thus the def of `bind sa f` is as follows:

```
λ s, match (sa s) with (a, s₀) := b a s₀
```

The library also defines operations `read` and `write` as follows:

```
def read {S : Type} : state S S :=
λ s, (s, s)

def write {S : Type} : S → state S unit :=
λ s₀ s, ((), s₀)
```

With the argument `S` implicit, `read` is simply the state computation that does not change the current state, but also returns it as a value. The value `write s₀` is the state computation which replaces any state `s` by `s₀` and returns `unit`. Notice that it is convenient to use `unit` for the output type any operation that does not return a value, though it may change the state.

Returning to our example, we can implement the register state monad and more focused read and write operations as follows:

```
def init_reg : registers :=
registers.mk 0 0 0
```

```
@[reducible] def reg_state := state registers

def read_x : reg_state ℕ :=
do s ← read, return (registers.x s)

def read_y : reg_state ℕ :=
do s ← read, return (registers.y s)

def read_z : reg_state ℕ :=
do s ← read, return (registers.z s)

def write_x (n : ℕ) : reg_state unit :=
do s ← read,
   write (registers.mk n (registers.y s) (registers.z s))

def write_y (n : ℕ) : reg_state unit :=
do s ← read,
   write(registers.mk (registers.x s) n (registers.z s))

def write_z (n : ℕ) : reg_state unit :=
do s ← read,
   write (registers.mk (registers.x s) (registers.y s) n)
```

We can then write a little register program as follows:

```
open nat

def foo : reg_state ℕ :=
do write_x 5,
   write_y 7,
   x ← read_x,
   write_z (x + 3),
   y ← read_y,
   z ← read_z,
   write_y (y + z),
   y ← read_y,
   return (y + 2)
```

To see the results of this program, we have to "run" it on the initial state:

```
#reduce foo init_reg
```

The result is the pair `(17, {x := 5, y := 15, z := 8})`, consisting of the return value, `y`, paired with the values of the three registers.

7.4 The IO monad

We can finally explain how Lean handles input and output: the constant `io` is axiomatically declared to be a monad with certain supporting operations. It is a kind of state monad, but in contrast to the ones discussed in the last section, here the state is entirely opaque to

Lean. You can think of the state as "the real world," or, at least, the status of interaction with the user. Lean's axiomatically declared constants include the following:

```
import system.io
open io

#check (@put_str : Π [ioi : io.interface], string → io unit)
#check (@get_line : Π [ioi : io.interface], io string)
```

Here `io.interface` is a type class packing information needed to interpret the input output interface. Users can instantiate that type class in different ways, but they can also leave these variables uninstantiated in calls to Lean's virtual machine, which then substitutes the usual terminal io operations.

The expression `put_str s` changes the `io` state by writing `s` to output; the return type, `unit`, indicates that no meaningful value is returned. The expression `put_nat n` does the analogous thing for a natural number, `n`. The expression `get_line`, in contrast; however you want to think of the change in `io` state, a `string` value is returned inside the monad. When we use the native virtual machine interpretation, thinking of the `io` monad as representing a state is somewhat heuristic, since within the Lean language, there is nothing that we can say about it. But when we run a Lean program, the interpreter does the right thing whenever it encounters the bind and return operations for the monad, as well as the constants above. In particular, in the example below, it ensures that the argument to `put_nat` is evaluated before the output is sent to the user, and that the expressions are printed in the right order.

```
#eval put_str "hello " >> put_str "world!" >> put_str (to_string (27 * 39))
```

[TODO: somewhere – probably in a later chapter? – document the format type and operations.]

7.5 Related type classes

In addition to the monad type class, Lean defines all the following abstract type classes and notations.

```
universe variables u v

class functor (F : Type u → Type v) : Type (max u+1 v) :=
(map : Π {α β : Type u}, (α → β) → F α → F β)

@[inline] def fmap {F : Type u → Type v} [functor F] {α β : Type u} : (α → β) → F α → F β :=
functor.map

infixr ` <$> `:100 := fmap
```

```
class applicative (F : Type u → Type v) extends functor F : Type (max u+1 v):=
(pure : Π {α : Type u}, α → F α)
(seq  : Π {α β : Type u}, F (α → β) → F α → F β)

@[inline] def pure {F : Type u → Type v} [applicative F] {α : Type u} : α → F α :=
applicative.pure F

@[inline] def seq_app {α β : Type u} {F : Type u → Type v} [applicative F] : F (α → β) → F α → F β :=
applicative.seq

infixr ` <*> `:2 := seq_app

class alternative (F : Type u → Type v) extends applicative F : Type (max u+1 v) :=
(failure : Π {α : Type u}, F α)
(orelse  : Π {α : Type u}, F α → F α → F α)

@[inline] def failure {F : Type u → Type v} [alternative F] {α : Type u} : F α :=
alternative.failure F

@[inline] def orelse {F : Type u → Type v} [alternative F] {α : Type u} : F α → F α → F α :=
alternative.orelse

infixr ` <|> `:2 := orelse

@[inline] def guard {F : Type → Type v} [alternative F] (P : Prop) [decidable P] : F unit :=
if P then pure () else failure
```

The `monad` class extends both `functor` and `applicative`, so both of these can be seen as even more abstract versions of `monad`. On the other hand, not every `monad` is `alternative`, and in the next chapter we will see an important example of one that is. One way to think about an alternative monad is to think of it as representing computations that can possibly fail, and, moreover, Intuitively, an alternative monad can be thought of supporting definitions that say "try `a` first, and if that doesn't work, try `b`." A good example is the `option` monad, in which we can think of an element `none` as a computation that has failed. If `a` and `b` are elements of `option` α for some type α, we can define `a <|> b` to have the value `a` if `a` is of the form `some` $\mathtt{a}_0$, and `b` otherwise.

8

Writing Tactics

8.1 A First Look at the Tactic Monad

The canonical way to invoke a tactic in Lean is to use the `by` keyword within a Lean expression. Suppose we write the following:

```
open tactic

variables a b : Prop

example : a → b → a ∧ b :=
by _
```

Lean expects something of type `tactic unit` to fill the underscore, where `tactic` refers to the tactic monad. When the elaborator processes this definition, it elaborates everything outside the `by` invocation first (in this case, just the statement of the theorem), and then calls on the Lean virtual machine to execute the tactic. When doing so, the virtual machine interprets any axiomatically declared `meta constant` as references to internal Lean functions that implement the functionality of the tactic monad.

The tactic monad can be thought of as a combination of a state monad (where the internal "state" as accessed and acted on by the `meta constant` primitives) and the option monad. Because it is a monad, we have the usual `do` notation. So, if `r`, `s`, and `t` are tactics, you should think of

```
do a ← r,
   b ← s,
   t
```

as meaning “apply tactic `r` to the state, and store the return result in `a`; apply tactic `s` to the state, and store the return result in `b`; then, finally, apply tactic `t` to the state.” Moreover, any tactic can *fail*, which is analogous to a return value of `none` in the option monad. In the example above, if any of `r`, `s`, or `t` fail, then the compound expression has failed.

There is an additional, really interesting feature of the tactic monad: it is an *alternative* monad, in the sense described at the end of the last chapter. This is used to implement backtracking. If `s` and `t` are monads, the expression `s <|> t` can be understood as follows: “do `s`, and if that succeeds, return the corresponding return value; otherwise, undo any changes to the state that `s` may have produced, and do `t` instead.” This allows us to try `s` and, if it fails, go on to try `t` as though the first attempt never happened.

When the tactic expression after a `by` is invoked, the tactic wakes up and says “Whoa! I’m in a monad!” This is just a colorful way of saying that the virtual machine expects a function that acts on the tactic state in a certain way, and interprets primitive operations on the tactic state in terms of functions that are implemented internal to Lean. At any rate, when it wakes up, it can start to look around and assess its current state. The goal of this section is to give you a first look at of some of the things it can do there. Don’t worry if some of the expressions seem mysterious; they will be explained in the sections that follow.

One thing the tactic can do is print a message to the outside world:

```
example : a → b → a ∧ b :=
by do trace "Hi, Mom!"
```

When the file is executed, Lean issues an error message to the effect that the tactic has failed to fill the relevant placeholder, which is what it is supposed to do. But the `trace` message is printed during the execution, providing us with a glimpse of its inner workings. We can actually trace value of any type that Lean can coerce to a string output, and we will see that this includes a number of useful types.

Another thing we can do is trace the current tactic state:

```
example : a → b → a ∧ b :=
by do trace "Hi, Mom!",
      trace_state
```

Now the output includes the list of *goals* that are active in the tactic state, each with a local context that includes the local variables and hypotheses. In this case there is only one:

```
Hi, Mom!
a b : Prop
⊢ a → b → a ∧ b
```

This points to an important fact: the internal, and somewhat mysterious, tactic state includes at least a list of goals. In fact, it includes much more: every tactic is invoked in a rich *environment* that includes all the objects and declarations that are present when the tactic is invoked, as well as notations, option declarations, and so on. In most cases, however, the list of goals is most directly relevant to the task at hand.

Let us dispense with the trace messages now, and start to prove the theorem by introducing the first two hypotheses.

```
example : a → b → a ∧ b :=
by do eh1 ← intro `h1,
      eh2 ← intro `h2,
      skip
```

The backticks indicate that `h1` and `h2` are *names*; we will discuss these below. The tactic `skip` is a do-nothing tactic, simply included to ensure that the resulting expression has type `tactic unit`.

We can now do some looking around. The `meta_constant` called `target` has type `tactic expr`, and returns the type of the goal. The type `expr`, like `name`, will be discussed below; it is designed to reflect the internal representation of Lean expressions, so, roughly, via meta-programming glue, the `expr` type allows us to manipulate Lean expressions in Lean itself. In particular, we can ask the tactic to print the current goal:

```
example : a → b → a ∧ b :=
by do eh1 ← intro `h1,
      eh2 ← intro `h2,
      target >>= trace
```

In this case, the output is `a ∧ b`, as we would expect. We can also ask the tactic to print the elements of the local context.

```
example : a → b → a ∧ b :=
by do eh1 ← intro `h1,
      eh2 ← intro `h2,
      local_context >>= trace
```

This yields the list `[a, b, h1, h2]`. We already happen to have representations of `h1` and `h2`, because they were returned by the `intro` tactic. But we can extract the other expressions in the local context given their names:

```
example : a → b → a ∧ b :=
by do intro `h1,
      intro `h2,
      ea ← get_local `a,
      eb ← get_local `b,
      trace (to_string ea ++ ", " ++ to_string eb),
      skip
```

Notice that `ea` and `eb` are different from `a` and `b`; they have type `expr` rather than `Prop`. They are the internal representations of the latter expressions. At present, there is not much for us to do with these expressions other than print them out, so we will drop them for now.

In any case, to prove the goal, we can proceed to invoke any of the Lean's standard tactics. For example, this will work:

```
example : a → b → a ∧ b :=
by do intro `h1,
      intro `h2,
      split,
      repeat assumption
```

We can also do it in a more hands-on way:

```
example : a → b → a ∧ b :=
by do eh1 ← intro `h1,
      eh2 ← intro `h2,
      mk_const ``and.intro >>= apply,
      exact eh1,
      exact eh2
```

The double-backticks will also be explained below, but the general idea is that the third line of the tactic builds an `expr` that reflects the `and.intro` declaration in the Lean environment, and applies it. The `applyc` tactic combines these two steps:

```
example : a → b → a ∧ b :=
by do eh1 ← intro `h1,
      eh2 ← intro `h2,
      applyc ``and.intro,
      exact eh1,
      exact eh2
```

We can also finish the proof as follows:

```
example : a → b → a ∧ b :=
by do eh1 ← intro `h1,
      eh2 ← intro `h2,
      e ← to_expr ```(and.intro h1 h2),
      exact e
```

Here, the construct ```` ```(...) ```` is used to build a *pre-expression*, the tactic `to_expr` elaborates it and converts it to an expression, and the `exact` tactic applies it. In the next section, we will see even more variations on constructions like these, including tactics that would enable us to construct the expression `and.intro h1 h2` more explicitly.

The `do` block in this example has type `tactic unit`, and can be broken out as an independent tactic.

```
meta def my_tactic : tactic unit :=
do eh1 ← intro `h1,
   eh2 ← intro `h2,
   e ← to_expr ``(and.intro %%eh1 %%eh2),
   exact e

example : a → b → a ∧ b :=
by my_tactic
```

Of course, `my_tactic` is not a very exciting tactic; we designed it to prove one particular theorem, and it will only work on examples that have the very same shape. But we can write more intelligent tactics that inspect the goal, the local hypotheses, and the environment, and then do more useful things. The mechanism is exactly the same: we construct an expression of type `tactic unit`, and ask the virtual machine to execute it at elaboration time to solve the goal at hand.

8.2 Names and Expressions

Suppose we write an ordinary tactic proof in Lean:

```
example (a b : Prop) (h : a ∧ b) : b ∧ a :=
begin
  split,
  exact and.right h,
  exact and.left h
end
```

This way of writing the tactic proof suggests that the `h` in the tactic block refers to the expression `h : a ∧ b` in the list of hypotheses. But this is an illusion; what `h` *really* refers to is the first hypothesis *named* `h` that is in the local context of the goal in the state when the tactic is executed. This is made clear, for example, by the fact that earlier lines in the proof can change the name of the hypothesis:

```
example (a b : Prop) (h : a ∧ b) : b ∧ a :=
begin
  revert h,
  intro h',
  split,
  exact and.right h',
  exact and.left h'
end
```

Now writing `exact and.right h` would make no sense. We could, alternatively, contrive to make `h` denote something different from the original hypothesis. This often happens with the `cases` and `induction` tactics, which revert hypotheses, peform an action, and then reintroduce new hypotheses with the same names.

Metaprogramming in Lean requires us to be mindful of and explicit about the distinction between expressions in the current environment, like `h : a ∧ b` in the hypothesis of the example, and the Lean objects that we use to act on the tactic state, such as the name "h" or an object of type `expr`. Without using the `begin...end` front end, we can construct the proof as follows:

```
example (a b : Prop) (h : a ∧ b) : b ∧ a :=
by do split,
   to_expr ```(and.right h) >>= exact,
   to_expr ```(and.left h) >>= exact
```

This tells Lean to elaborate the expressions `and.right h` and `and.left h` in the context of the current goal, and then apply them. The `begin...end` construct is essentially a front end that interprets the proof above in these terms.

To understand what is going on in situations like this, it is important to know that Lean's metaprogramming framework provides three distinct Lean types that are relevant to constructing syntactic expressions:

- the type `name`, representing *hierarchical names*
- the type `expr`, representing *expressions*
- the type `pexpr`, representing *pre-expressions*

Let us consider each one of them, in turn.

Hierarchical names are denoted in ordinary .lean files with expressions like `foo.bar.baz` or `nat.mul_comm`. They are used as identifiers that reference defined constants in Lean, but also for local variables, attributes, and other objects. Their Lean representations are defined in `init/meta/name.lean`, together with some operations that can be performed on them. But for many purposes we can be oblivious to the details. Whenever we type an expression that begins with a backtick that is not followed by an open parenthesis, Lean's parser translates this to the construction of the associated name. In other words, `` `nat.mul_comm `` is simply notation for the compound name with components `nat` and `mul_comm`.

When metaprogramming, we often use names to refer to definitions and theorems in the Lean environment. In situations like that, it is easy to make mistakes. In the example below, the tactic definition is accepted, but its application fails:

```
open tactic

namespace foo

theorem bar : true := trivial

meta def my_tac : tactic unit :=
```

```
mk_const `bar >>= exact

-- example : true := by my_tac -- fails

end foo
```

The problem is that the proper name for the theorem is `foo.bar` rather than `bar`; if we replace `` `bar `` by `` `foo.bar ``, the example is accepted. The `mk_const` tactic takes an arbitrary name and attempts to resolve it when the tactic is invoked, so there is no error in the definition of the tactic. The error is rather that when we wrote `` `bar `` we had in mind a particular theorem in the environment at the time, but we did not identify it correctly.

For situations like these, Lean provides double-backtick notation. The following example succeeds:

```
open tactic

namespace foo

theorem bar : true := trivial

meta def my_tac : tactic unit :=
mk_const ``bar >>= exact

example : true := by my_tac -- fails

end foo
```

It also succeeds if we replace ``` ``bar ``` by ``` ``foo.bar ```. The double-backtick asks the parser to resolve the expression with the name of an object in the environment *at parse time*, and insert the relevant name. This has two advantages:

- if there is no such object in the environment at the time, the parser raises an error; and
- assuming it does find the relevant object in the environment, it inserts the full name of the object, meaning we can use abbreviations that make sense in the context where we are writing the tactic.

As a result, it is a good idea to use double-backticks whenever you want to refer to an existing definition or theorem.

When writing tactics, it is often necessary to generate a fresh name. You can use `mk_fresh_name` for that:

```
example (a : Prop) : a → a :=
by do n ← mk_fresh_name,
      intro n,
      hyp ← get_local n,
      exact hyp
```

The type `expr` reflects the internal representation of Lean expressions. It is defined inductively in the file `expr.lean`, but when evaluating expressions that involve terms of type `expr`, the virtual machine uses the internal C++ representations, so each constructor and the eliminator for the type are translated to the corresponding C++ functions. Expressions include the sorts `Prop`, `Type`$_1$, `Type`$_2$, …, constants of each type, applications, lambdas, Pi types, and let definitions. The also include de Bruijn indices (with constructor `var`), metavariables, local constants, and macros.

The whole purpose of tactic mode is to construct expressions, and so this data type is fundamental. We have already seen that the `target` tactic returns the current goal, which is an expression, and that `local_context` returns the list of hypotheses that can be used to solve the current goal, that is, a list of expressions.

Returning to the example at the start of this section, let us consider ways of constructing the expressions `and.left h` and `and.right h` more explicitly. The following example uses the `mk_mapp` tactic.

```
example (a b : Prop) (h : a ∧ b) : b ∧ a :=
by do split,
   eh ← get_local `h,
   mk_mapp ``and.right [none, none, some eh] >>= exact,
   mk_mapp ``and.left [none, none, some eh] >>= exact
```

In this example, the invocations of `mk_mapp` retrieve the definition of `and.right` and `and.left`, respectively. It makes no difference whether the arguments to those theorems have been marked implicit or explicit; `mk_mapp` ignores those annotations, and simply applies that theorem to all the arguments in the subsequent list. Thus the first argument to `mk_mapp` is a name, while the second argument has type `list (option expr)`. Each `none` entry in the list tells `mk_mapp` to treat that argument as implicit and infer it using type inference. In contrast, an entry of the form `some t` specifies `t` as the corresponding argument.

The tactic `mk_app` is an even more rudimentary application builder. It takes the name of the operator, followed by a complete list of its arguments.

```
example (a b : Prop) (h : a ∧ b) : b ∧ a :=
by do split,
      ea ← get_local `a,
      eb ← get_local `b,
      eh ← get_local `h,
      mk_app ``and.right [ea, eb, eh] >>= exact,
      mk_app ``and.left [ea, eb, eh] >>= exact
```

You can send less than the full list of arguments to `mk_app`, but the arguments you send are assumed to be the *final* arguments, with the earlier ones made implicit. Thus, in the example above, we could send instead `[eb, eh]` or simply `[eh]`, because the earlier arguments can be inferred from these.

```
example (a b : Prop) (h : a ∧ b) : b ∧ a :=
by do split,
      eh ← get_local `h,
      mk_app ``and.right [eh] >>= exact,
      mk_app ``and.left [eh] >>= exact
```

Finally, as indicated in the last section, you can also use `mk_const` to construct a constant expression from the corresponding name:

```
example (a b : Prop) (h : a ∧ b) : b ∧ a :=
by do split,
      eh ← get_local `h,
      mk_const ``and.right >>= apply,
      exact eh,
      mk_const ``and.left >>= apply,
      exact eh
```

We have also seen above that it is possible to use `to_expr` to elaborate expressions at executation time, in the context of the current goal.

```
example (a b : Prop) (h : a ∧ b) : b ∧ a :=
by do split,
   to_expr ```(and.right h) >>= exact,
   to_expr ```(and.left h) >>= exact
```

Here, the expressions ` ```(and.right h) ` and ` ```(and.left h) ` are pre-expressions, that is, objects of type `pexpr`. The interface to `pexpr` can be found in the file `pexpr.lean`, but the type is largely opaque from within Lean. The canonical use is given by the example above: when Lean's parser encounters an expression of the form ` ```(...) `, it constructs the corresponding `pexpr`, which is simply an internal representation of the unelaborated term. The `to_expr` tactic then sends that object to the elaborator when the tactic is executed.

Note that the backtick is used in two distinct ways: an expression of the form `` `n ``, without the parentheses, denotes a `name`, whereas an expression of the form `` `(...) ``, with parentheses, denotes a `pexpr`. Though this may be confusing at first, it is easy to get used to the distinction, and the notation is quite convenient.

Lean's pre-expression mechanism also supports the use of *anti-quotation*, which allows a tactic to tell the elaborator to insert an expression into a pre-expression at runtime. Returning to the example above, suppose we are in a situation where instead of the name `h`, we have the corresponding *expression*, `eh`, and want to use that to construct the term. We can insert it into the pre-expression by preceding it with a double-percent sign:

```
example (a b : Prop) (h : a ∧ b) : b ∧ a :=
by do split,
   eh ← get_local `h,
```

```
  to_expr ``(and.right %%eh) >>= exact,
  to_expr ``(and.left %%eh) >>= exact
```

When the tactic is executed, Lean elaborates the pre-expressions given by ``` ``(...) ```, with the expression `eh` inserted in the right place. The difference between ``` ``(...) ``` and ```` ```(...) ```` is that the first resolves the names contained in the expression when the tactic is defined, whereas the second resolves them when the tactic is executed. Since the only name occurring in `and.left %%eh` is `and.left`, it is better to resolve it right away. However, in the expression `and.right h` above, `h` only comes into existence when the tatic is executed, and so we need to use the triple backtick.

[TODO: describe `` `(...) ``, and improve the explanations]

8.3 Examples

When it comes to writing tactics, you have all the computable entities of Lean's standard library at your disposal, including lists, natural numbers, strings, product types, and so on. This makes the tactic monad a powerful mechanism for writing metaprograms. Some of Lean's most basic tactics are implemented internally in C++, but many of them are defined from these in Lean itself.

The entry point for the tactic library is the file `init/meta/tactic.lean`, where you can find the details of the interface, and see a number of basic tactics implemented in Lean. For example, here is the definition of the `assumption` tactic:

```
meta def find_same_type : expr → list expr → tactic expr
| e []         := failed
| e (h :: hs) :=
  do t ← infer_type h,
     (unify e t >> return h) <|> find_same_type e hs

meta def assumption : tactic unit :=
do ctx ← local_context,
   t   ← target,
   h   ← find_same_type t ctx,
   exact h
<|> fail "assumption tactic failed"
```

The expression `find_same_type t es` tries to find in `es` an expression with type definitionally equal to `t` in the list of expressions `es`, by a straightforward recursion on the list. The `infer_type` tactic calls Lean's internal type inference mechanism to infer to the type of an expression, and the `unify` tactic (which will be discussed further in Section Section 8.6 tries to unify two expressions, instantiating metavariables if necessary. Note the use of the `orelse` notation: if the unification fails, the procedure backtracks and continues to try the remaining elements on the list. The `fail` tactic announces failure with a given string. The `failed` tactic simply fails with a generic message, "tactic failed."

One can even manipulate data structures that include tactics themselves. For example, the `first` tactic takes a list of tactics, and applies the first one that succeeds:

```
meta def first {α : Type} : list (tactic α) → tactic α
| []      := fail "first tactic failed, no more alternatives"
| (t::ts) := t <|> first ts
```

It fails if none of the tactics on the list succeeds. Consider the example from Section 1.4 of the Introduction:

```
meta def destruct_conjunctions : tactic unit :=
repeat (do
  l ← local_context,
  first $ l.map (λ h, do
    ht ← infer_type h >>= whnf,
    match ht with
    | `(and %%a %%b) := do
      n ← get_unused_name `h none,
      mk_mapp ``and.left [none, none, some h] >>= assertv n a,
      n ← get_unused_name `h none,
      mk_mapp ``and.right [none, none, some h] >>= assertv n b,
      clear h
    | _ := failed
    end))
```

The `repeat` tactic simply repeats the inner block until it fails. The inner block starts by getting the local context. The expression `l.map ...` is just shorthand for `list.map ... l`; it applies the function in `...` to each element of `l` and returns the resulting list, in this case a list of tactics. The `first` function then calls each one sequentially until one of them succeeds. Note the use of the dollar-sign for function application. In general, an expression `f $ a` denotes nothing more than `f a`, but the binding strength is such that you do not need to use extra parentheses when `a` is a long expression. This provides a convenient idiom in situations exactly like the one in the example.

Some of the elements of the body of the main loop will now be familiar. For each element `h` of the context, we infer the type of `h`, and reduce it to weak head normal form. (We will discuss weak head normal form in the next section.) Assuming the type is an `and`, we construct the terms `and.left h` and `and.right h` and add them to the context with a fresh name. The `clear` tactic then deletes `h` itself.

Remember that when writing `meta defs` you can carry out arbitrary recursive calls, without any guarantee of termination. You should use this with caution when writing tactics; if there is any chance that some unforseen circumstance will result in an infinite loop, it is wiser to use a large cutoff to prevent the tactic from hanging. Even the `repeat` tactic is implemented as a finite iteration:

```
meta def repeat_at_most : nat → tactic unit → tactic unit
| 0        t := skip
| (succ n) t := (do t, repeat_at_most n t) <|> skip

meta def repeat : tactic unit → tactic unit :=
repeat_at_most 100000
```

But 100,000 iterations is still enough to get you into trouble if you are not careful.

8.4 Reduction

[This section still under construction. It will discuss the various types of reduction, the notion of weak head normal form, and the various transparency settings. It will use some of the examples that follow.]

```
open tactic

set_option pp.beta false

section
  variables {α : Type} (a b : α)

  example : (λ x : α, a) b = a :=
  by do goal ← target,
        match expr.is_eq goal with
        | (some (e₁, e₂)) := do trace e₁,
                                whnf e₁ >>= trace,
                                reflexivity
        | none            := failed
        end

  example : (λ x : α, a) b = a :=
  by do goal ← target,
        match expr.is_eq goal with
        | (some (e₁, e₂)) := do trace e₁,
                                whnf e₁ transparency.none >>= trace,
                                reflexivity
        | none            := failed
        end

  attribute [reducible]
  definition foo (a b : α) : α := a

  example : foo a b = a :=
  by do goal ← target,
        match expr.is_eq goal with
        | (some (e₁, e₂)) := do trace e₁,
                                whnf e₁ transparency.none >>= trace,
                                reflexivity
        | none            := failed
        end

  example : foo a b = a :=
  by do goal ← target,
        match expr.is_eq goal with
        | (some (e₁, e₂)) := do trace e₁,
                                whnf e₁ transparency.reducible >>= trace,
                                reflexivity
        | none            := failed
        end
end
```

8.5 Metavariables and Unification

[This section is still under construction. It will discuss the notion of a metavariable and its local context, with the interesting bit of information that goals in the tactic state are nothing more than metavariables. So the goal list is really just a list of metavariables, which can help us make sense of the `get_goals` and `set_goals` tactics. It will also discuss the `unify` tactic.]

9

Writing Automation

The goal of this chapter is to provide some examples that illustrate the ways that metaprogramming in Lean can be used to implement automated proof procedures.

9.1 A Tableau Prover for Classical Propositional Logic

In this section, we design a theorem prover that is complete for classical propositional logic. The method is essentially that of tableaux theorem proving, and, from a proof-theoretic standpoint, can be used to demonstrate the completeness of cut-free sequent calculi.

The idea is simple. If `a`, `b`, `c`, and `d` are formulas of propositional logic, the sequent `a, b, c ⊢ d` represents the goal of proving that `d` follows from `a`, `b` and `c`, and `d`. The fact that they are propositional formulas means that they are built up from variables of type `Prop` and the constants `true` and `false` using the connectives $\wedge$ $\vee$ $\to$ $\leftrightarrow$ $\neg$. The proof procedure proceeds as follows:

- Negate the conclusion, so that the goal becomes `a, b, c, ¬ d ⊢ false`.
- Put all formulas into *negation-normal form.* In other words, eliminate $\to$ and $\leftrightarrow$ in terms of the other connectives, and using classical identities to push all equivalences inwards.
- At that stage, all formulas are built up from *literals* (propositional variables and negated propositional variables) using only $\wedge$ and $\vee$. Now repeatedly apply all of the following proof steps:
  - Reduce a goal of the form `Γ, a ∧ b ⊢ false` to the goal `Γ, a, b ⊢ false`, where Γ is any set of propositional formulas.

– Reduce a goal of the form Γ, `a ∨ b ⊢ false` to the pair of goals Γ, `a ⊢ false` and Γ, `b ⊢ false`.
– Prove any goal of the form Γ, `a, ¬ a ⊢ false` in the usual way.

It is not hard to show that this is complete. Each step preserves validity, in the sense that the original goal is provable if and only if the new ones are. And, in each step, the number of connectives in the goal decreases. If we ever face a goal in which the first two rules do not apply, the goal must consist of literals. In that case, if the last rule doesn't apply, then no propositional variable appears with its negation, and it is easy to cook up a truth assignment that falsifies the goal.

In fact, our procedure will work with arbitrary formulas at the leaves. It simply applies reductions and rules as much as possible, so formulas that begin with anything other than a propositional connective are treated as black boxes, and act as propositional atoms.

First, let us open the namespaces we will use:

```
open expr tactic classical
```

The next step is to gather all the facts we will need to put formulas in negation-normal form.

```
section logical_equivalences
  local attribute [instance] prop_decidable
  variables {a b : Prop}

  theorem not_not_iff (a : Prop) : ¬¬a ↔ a :=
  iff.intro classical.by_contradiction not_not_intro

  theorem implies_iff_not_or (a b : Prop) : (a → b) ↔ (¬ a ∨ b) :=
  iff.intro
    (λ h, if ha : a then or.inr (h ha) else or.inl ha)
    (λ h, or.elim h (λ hna ha, absurd ha hna) (λ hb ha, hb))

  theorem not_and_of_not_or_not (h : ¬ a ∨ ¬ b) : ¬ (a ∧ b) :=
  assume ⟨ha, hb⟩, or.elim h (assume hna, hna ha) (assume hnb, hnb hb)

  theorem not_or_not_of_not_and (h : ¬ (a ∧ b)) : ¬ a ∨ ¬ b :=
  if ha : a then
    or.inr (show ¬ b, from assume hb, h ⟨ha, hb⟩)
  else
    or.inl ha

  theorem not_and_iff (a b : Prop) : ¬ (a ∧ b) ↔ ¬a ∨ ¬b :=
  iff.intro not_or_not_of_not_and not_and_of_not_or_not

  theorem not_or_of_not_and_not (h : ¬ a ∧ ¬ b) : ¬ (a ∨ b) :=
  assume h₁, or.elim h₁ (assume ha, h^.left ha) (assume hb, h^.right hb)

  theorem not_and_not_of_not_or (h : ¬ (a ∨ b)) : ¬ a ∧ ¬ b :=
  and.intro (assume ha, h (or.inl ha)) (assume hb, h (or.inr hb))
  theorem not_or_iff (a b : Prop) : ¬ (a ∨ b) ↔ ¬ a ∧ ¬ b :=
  iff.intro not_and_not_of_not_or not_or_of_not_and_not
end logical_equivalences
```

We can now use Lean's built-in simplifier to do the normalization:

```
meta def normalize_hyp (lemmas : list expr) (hyp : expr) : tactic unit :=
do try (simp_at hyp lemmas)

meta def normalize_hyps : tactic unit :=
do hyps ← local_context,
   lemmas ← monad.mapm mk_const [``iff_iff_implies_and_implies,
         ``implies_iff_not_or, ``not_and_iff, ``not_or_iff, ``not_not_iff,
         ``not_true_iff, ``not_false_iff],
   monad.for' hyps (normalize_hyp lemmas)
```

The tactic `normalize_hyp` just simplifies the given hypothesis with the given list of lemmas. The `try` combinator ensures that the tactic is deemed successful even if no simplifications are necessary. The tactic `normalize_hyps` gathers the local context, turns the list of names into a list of expressions by applying the `mk_const` tactic to each one, and then calls `normalize_hyp` on each element of the context with those lemmas. The `for`' tactic, like the `for` tactic, applies the second argument to each element of the first, but it returns unit rather than accumulate the results in a list.

We can test the result:

```
example (p q r : Prop) (h₁ : ¬ (p ↔ (q ∧ ¬ r))) (h₂ : ¬ (p → (q → ¬ r))) : true :=
by do normalize_hyps,
      trace_state,
      triv
```

The result is as follows:

```
p q r : Prop,
h₁ : p ∧ (r ∨ ¬q) ∨ q ∧ ¬p ∧ ¬r,
h₂ : p ∧ q ∧ r
⊢ true
```

The next five tactics handle the task of splitting conjunctions.

```
open tactic expr

meta def add_fact (prf : expr) : tactic unit :=
do nh ← get_unused_name `h none,
   p ← infer_type prf,
   assertv nh p prf,
   return ()
```

```
meta def is_conj (e : expr) : tactic bool :=
do t ← infer_type e,
   return (is_app_of t `and)

meta def add_conjuncts : expr → tactic unit | e :=
do e₁ ← mk_app `and.left [e],
   monad.cond (is_conj e₁) (add_conjuncts e₁) (add_fact e₁),
   e₂ ← mk_app `and.right [e],
   monad.cond (is_conj e₂) (add_conjuncts e₂) (add_fact e₂)

meta def split_conjs_at (h : expr) : tactic unit :=
do monad.cond (is_conj h)
     (add_conjuncts h >> clear h)
     skip

meta def split_conjs : tactic unit :=
do l ← local_context,
   monad.for' l split_conjs_at
```

The tactic `add_fact prf` takes a proof of a proposition `p`, and adds `p` the the local context with a fresh name. Here, `get_unused_name `h none` generates a fresh name of the form `h_n`, for a numeral `n`. The tactic `is_conj` infers the type of a given expression, and determines whether or not it is a conjunction. The tactic `add_conjuncts e` assumes that the type of `e` is a conjunction and adds proofs of the left and right conjuncts to the context, recursively splitting them if they are conjuncts as well. The tactic `split_conjs_at h` tests whether or not the hypothesis `h` is a conjunction, and, if so, adds all its conjuncts and then clears it from the context. The last tactic, `split_conjs`, applies this to every element of the context.

We need two more small tactics before we can write our propositional prover. The first reduces the task of proving a statement `p` from some hypotheses to the task of proving falsity from those hypotheses and the negation of `p`.

```
meta def deny_conclusion : tactic unit :=
do refine ```(classical.by_contradiction _),
   nh ← get_unused_name `h none,
   intro nh,
   return ()
```

The refine tactic applies the expression in question to the goal, but leaves any remaining metavariables for us to fill. The theorem `classical.by_contradiction` has type `∀ {p : Prop}, (¬p → false) → p`, so applying this theorem proves the goal but leaves us with the new goal of proving `¬p → false` from the same hypotheses, at which point, we can use the introduction rule for implication. If we omit the `return ()`, we will get an error message, because `deny_conclusion` is supposed to have type `tactic unit`, but the `intro` tactic returns an expression.

The next tactic finds a disjunction among the hypotheses, or returns the `option.none` if there aren't any.

```
meta def find_disj : tactic (option expr) :=
do l ← local_context,
   (first $ l.map
     (λ h, do t ← infer_type h,
              cond (is_app_of t `or)
                (return (option.some h)) failed)) <|>
   return none
```

Our propositional prover can now be implemented as follows:

```
meta def prop_prover_aux : ℕ → tactic unit
| 0            :=  fail "prop prover max depth reached"
| (nat.succ n) :=
  do split_conjs,
     contradiction <|>
     do (option.some h) ← find_disj |
          fail "prop_prover failed: unprovable goal",
        cases h,
        prop_prover_aux n,
        prop_prover_aux n

meta def prop_prover : tactic unit :=
do deny_conclusion,
   normalize_hyps,
   prop_prover_aux 30
```

The tactic `prop_prover` denies the conclusion, reduces the hypotheses to negation-normal form, and calls `prop_prover_aux` with a maximum splitting depth of 30. The tactic `prop_prover_aux` executes the following simple loop. First, it splits any conjunctions in the hypotheses. Then it tries applying the `contradiction` tactic, which will find a pair of contradictory literals, `p` and `¬ p`, if there is one. If that does not succeed, it looks for a disjunction `h` among the hypotheses. At this stage, if there aren't any disjunctions, we know that the goal is not propositionally valid. On the other hand, if there is a disjunction, `prop_prover_aux` calls the `cases` tactic to split the disjunction, and then applies itself recursively to each of the resulting subgoals, decreasing the splitting depth by one.

Notice the pattern matching in the `do` notation:

```
(option.some h) ← find_disj |
         fail "prop_prover failed: unprovable goal"
```

This is shorthand for the use of the `bind` operation in the tactic monad to extract the result of `find_disj`, together with the use of a `match` statement to extract the result. The expression after the vertical bar is the value returned for any other case in the pattern match; in this case, it is the value returned if `find_disj` returns `none`. This is a common idiom when writing tactics, and so the compressed notation is handy.

All this is left for us to do is to try it out:

```
section
  variables a b c d : Prop

  example (h₁ : a ∧ b) (h₂ : b ∧ ¬ c) : a ∨ c :=
  by prop_prover

  example (h₁ : a ∧ b) (h₂ : b ∧ ¬ c) : a ∧ ¬ c :=
  by prop_prover

  -- not valid
  -- example (h₁ : a ∧ b) (h₂ : b ∧ ¬ c) : a ∧ c :=
  -- by prop_prover

  example : ((a → b) → a) → a :=
  by prop_prover

  example : (a → b) ∧ (b → c) → a → c :=
  by prop_prover

  example (α : Type) (x y z w : α) :
    x = y ∧ (x = y → z = w) → z = w :=
  by prop_prover

  example : ¬ (a ↔ ¬ a) :=
  by prop_prover
end
```

10

The Tactic Library

[Document all the useful things available for writing tactics: the simplifier API, red black trees, pattern matching, ...]

Bibliography