Programming in Lean

Jeremy Avigad
Leonardo de Moura
Jared Roesch

Version 4b8fbb8, updated at 2016-08-23 20:10:10 -0700

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 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.

Most of the examples here will not run under the current version of Lean (which is what is running in the online version). To try them out, you will have to build the version of Lean found in the lean3 branch of the Github repository. Remember that you can run the examples from Emacs using C-c C-x.

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 Construtions 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 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 `eval` and `vm_eval`).]

[Define operations on lists.]

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

definition hello_world : IO unit :=
put_str "hello world\n"

vm_eval hello_world
```

The next example prints the first 100 squares:

```
import system.IO
open nat

definition print_squares : ℕ → IO unit
| 0        := return ()
| (succ n) := print_squares n >>
              put_str (to_string n ++ "^2 = " ++
                       to_string (n * n) ++ "\n")

vm_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 contants `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 keyword `metadefinition` instead of `definition` 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_definition m91 (n : ℕ) : ℕ :=
if n > 100 then n - 10 else m91 (m91 (n + 11))

vm_eval m91 10
vm_eval m91 100
vm_eval m91 1000
```

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

```
meta_definition print_m91 : ℕ → IO unit
| 0        := return ()
| (succ n) := print_m91 n >>
              put_str ("m91 " ++ to_string n ++ " = " ++
                       to_string (m91 n) ++ "\n")

vm_eval print_m91 120
```

Of course, such uses of recursion are dangerous.

```
meta_definition foo : nat := foo + 1

eval foo
-- vm_eval foo
```

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_definition destruct_conjunctions : tactic unit :=
repeat
  (do l ← local_context,
      first $ forl l (λ h,
      do ht ← infer_type h,
         if head_symbol ht = ``and then do
            mk_mapp ``and.left [none, none, some h] >>= assert_fact,
            mk_mapp ``and.right [none, none, some h] >>= assert_fact,
            clear h
         else failed))
```

We will explain the details in Chapter ?, 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.

```
-- TODO: replace this by a begin...end block when they exist
example (A B C : Prop) (H : (A ∧ B) ∧ (C ∧ A)) : C :=
by do destruct_conjunctions >> assumption
```

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 ??-?? 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.

Chapters ?? and ?? deal 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.

Finall, chapters ?? to ?? 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 → Type` that comes equipped with two special operations, `return` and `bind`. If `A` is any type, think of `m A` as being a "virtual `A`," or, as some people describe it, "an `A` inside a box."

For a given monoid `m`, the function `return` has type Π `{A : Type}, A → m A`. The idea is that for any element `a : A`, `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 A` and obtaining an element of `A`. But the `bind` operation gives us a way of turning some operations on `A` into operations inside the monad. Specifically, for a given monad `m`, the function `bind` has type Π `{A B : Type}, m A → (A → m B) → m B`. Suppose we have a function `f` that, given any element `a : A`, produces a virtual element of `B`; in more prosaic terms, `f` has type `A → m B`. Suppose also that we have a virtual element of `A`, that is, `ma : m A`. If we could extract from `ma` a corresponding element `a` of `A`, we could apply `f` to it to get a virtual element of `B`. 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 B`.

As an example of how `bind` and `return` can be used, given any function `f : A → B`, we can get a function `map f : m A → m B` by defining `map f ma` to be `bind ma (λ a, return (f a))`. Roughly, given `ma : m A`, 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 A)`, the expression `monad.bind mma id` has type `m A`. This means that even though we cannot in general extract an element of `B` from `m B`, we *can* do it when `B` itself is a virtual type, `m A`. The expression `monad.bind`

`mma id` reaches into the `m (m A)` box, catches hold of an element of `m A`, and simply leaves it in the `m A` 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 B`. 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`. This is 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 B` for some `B`. 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] {A : Type}, A → m A},
```

and `bind` really has type

```
Π {m : Type → Type} [monad m] {A B : Type}, m A → (A → m B) → m B.
```

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 A` is either of the form `some a` for some element `a : A`, or `none`. So an element `a` of `option A` is a "virtual `A`" in the sense of being either an element of `A` or an empty promise.

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

```
definition bind {A B : Type} (oa : option A) (f : A → option B) :
  option B :=
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 {A B C D : Type.{u}} (oa : option A)
variables (f : A → option B) (g : A → B → option C)
          (h : A → B → C → option D)

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

example : option D :=
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 `B` rather than `option B`. 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 : A → option B` as a function which, on input `A`, possibly returns an element of `B`. Now we will think of a function `f : A → list B` as a function which, on input `A`, 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 : A` into `list A`; we define `return a` to be just the singleton list `[a]`. Now, given `la : list A` and `f : A → list B`, how should we define the bind operation `la >>= f`? Intuitively, `la` represents any of the possible values occuring 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:

```
open list
```

```
definition bind {A B : Type} (la : list A) (f : A → list B) : list B :=
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 {A B C D : Type.{u}} (la : list A)
variables (f : A → list B) (g : A → B → list C)
          (h : A → B → C → list D)

example : list D :=
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

universe u
variables {A B C D : Type.{u}} (la : list A)
variables (f : A → list B) (g : A → B → list C) (h : A → B → C →
    list D)

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 {A B C D : Type.{1}} (la : list A)
variables (f : A → list B) (g : A → B → list C)
          (h : A → B → C → list D)

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 : A → B` and `g : B → C`, `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 straightfoward 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)

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

Now, instead of defining `f : A → B` that operates on the state of the registers implicitly, we would define a function `f₀ : A × registers → B × registers` that operates on it explicitly. The function `f₀` would take an input `a : A`, 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 : B` paired with the new state. Similarly, we would replace `g` by a function `g₀ : B × registers → C × registers`. The result of the composite computation would be given by `(g₀ (f₀ (a, init_reg))).1`. In other words, we would pair the value `a` with the initial setting of the registers, apply `f₀` and then `g₀`, 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 `f₀` equally well to be `A → registers → B × registers`. Now think of `f₀` as being a function which takes an input in `A` and returns an element of `registers → B × registers`. Moreover, think of this output as representing a computation which starts with a certain state, and returns a value of `B` and a new state. Lo and behold, *that* is the relevant monad.

To be precise: for any type `A`, the monad `m A` we are after is `registers → A × registers`. We will call this the state monad for `registers`. With this notation, the function `f₀` described above has type `A → m B`, the function `g₀` has type `B → m C`, and the composition of the two on input `a` is `f a >>= g`. Notice that the result is an element of `m C`, which is to say, it is a computation which takes any state and returns a value of `C` 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 `C`, 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 `A` to the type `S → A × S`. The particular monad described above is then simply `state registers`.

Let us consider the `return` and `bind` operations. Given any `a : A`, `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 A` and an `f : A → state S B`, 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 B`, which is really a function `S → B × S`. In other words, `bind sa f` is supposed to encode a function which operates on any state to produce an element of `B` 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 `B × S`. Thus the definition 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:

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

definition 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` s_0 is the state computation which replaces any state `s` by s_0 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:

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

abbreviation reg_state := state registers

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

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

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

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

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

definition 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

definition 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:

```
eval foo init_reg
```

The result is the pair `(15, registers.mk 5 15 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:

```
constant put_str : string → IO unit
constant put_nat : nat → IO unit
constant get_line : IO string
```

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. Again, 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.

```
vm_eval do put_str "hello " >> put_str "world!" >> put_nat (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.

```
structure [class] functor (f : Type → Type) : Type :=
(map : Π {a b: Type}, (a → b) → f a → f b)

definition fmap {F : Type → Type} [functor F] {A B : Type}
  (f : A → B) (a : F A) : F B :=
functor.map f a

infixr ` <$> `:100 := fmap

structure [class] applicative.{u₁ u₂} (f : Type.{u₁} → Type.{u₂})
  extends functor f : Type.{max u₁+1 u₂} :=
(pure : Π {A : Type.{u₁}}, A → f A)
(seq  : Π {A B : Type.{u₁}}, f (A → B) → f A → f B)

definition pure {f : Type → Type} [applicative f] {A : Type} (a : A) : f
    A :=
applicative.pure f a

definition seq_app {A B : Type} {f : Type → Type} [applicative f]
  (g : f (A → B)) (a : f A) : f B :=
applicative.seq g a

infixr ` <*> `:2 := seq_app

structure [class] alternative (f : Type → Type) extends applicative f :=
(failure : Π {A : Type}, f A)
(orelse  : Π {A : Type}, f A → f A → f A)

definition failure {f : Type → Type} [alternative f] {A : Type} : f A :=
alternative.failure f

definition orelse {f : Type → Type} [alternative f] {A : Type} : f A →
    f A → f A :=
alternative.orelse

infixr ` <|> `:2 := orelse
```

```
definition guard {f : Type₁ → Type} [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 A` for some type `A`, we can define `a <|> b` to have the value `a` if `a` is of the form `some` a_0, and `b` otherwise.

8

Writing Tactics

[Start with exprs, pexprs, names, … Getting context and target, basic tactics, repeat, first, etc.]

9

Writing Automation

[Give a hands-on example, writing a simple tableau prover for classical propositional logic].

10

The Tactic Library

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

Bibliography