Monads in Lean

Tanner Duve, Lean Reading Seminar

10/08/2024

Introduction

Monads are mathematical structures originating from category theory and used in algebra to model various constructs. In functional programming, particularly in pure functional languages like Lean, monads provide a way to handle side effects within a pure functional framework.

A *pure function* is one that, given the same inputs, always produces the same outputs and does not cause any observable side effects. A *side effect* refers to any interaction with the outside world or alteration of state that persists beyond the function's scope, such as modifying a variable, reading user input, logging outputs, or changing global state.

Functional programming languages like Lean enforce purity to ensure that functions are predictable and composable. However, real-world programs often need to perform actions that involve side effects. Monads allow us to model these side effects by encapsulating them within monadic types. This enables us to sequence computations involving side effects while keeping the core functions pure and the code maintainable.

A side effect that's important to us as Lean programmers is being able to alter the proof state when writing a proof, and we do this using *tactics* - this is in fact defined by a tactic monad, which will be discussed in a future presentation.

Category Theory Basics

Definition: A *category* $\mathfrak{C}$ consists of:

- A collection of *objects* $\mathrm{Ob}(\mathfrak{C})$.
- For any pair of objects $A, B \in \mathrm{Ob}(\mathfrak{C})$, a set of *morphisms* (or *arrows*) $\mathrm{Hom}_{\mathfrak{C}}(A, B)$.

These satisfy:

- (**Composition**) For any $f : A \to B$ and $g : B \to C$, there exists a composition $g \circ f : A \to C$.

- (**Associativity**) For all composable morphisms f, g, h, we have $h \circ (g \circ f) = (h \circ g) \circ f$.
- (**Identity**) For each object A, there exists an identity morphism $\mathrm{id}_A : A \to A$ such that for any morphism $f : A \to B$, we have $\mathrm{id}_B \circ f = f = f \circ \mathrm{id}_A$.

Examples of Categories:

- **Set**: Objects are sets, morphisms are functions between sets.
- **Grp**: Objects are groups, morphisms are group homomorphisms.
- **Mon**: Any monoid can be viewed as a category with a single object where morphisms are the elements of the monoid, and composition is given by the monoid operation.

Definition: A *functor* $F : \mathfrak{C} \to \mathfrak{D}$ between categories $\mathfrak{C}$ and $\mathfrak{D}$ consists of:

- For each object $A \in \mathrm{Ob}(\mathfrak{C})$, an object $FA \in \mathrm{Ob}(\mathfrak{D})$.
- For each morphism $f : A \to B$ in $\mathfrak{C}$, a morphism $Ff : FA \to FB$ in $\mathfrak{D}$.

Such that:

- For all composable morphisms $f : A \to B$ and $g : B \to C$, we have $F(g \circ f) = Fg \circ Ff$.
- For all objects $A \in \mathrm{Ob}(\mathfrak{C})$, $F(\mathrm{id}_A) = \mathrm{id}_{FA}$.

Examples of Functors:

- The *forgetful functor* $U : \mathbf{Grp} \to \mathbf{Set}$ maps a group to its underlying set and group homomorphisms to their underlying functions.
- The *free group functor* $F : \mathbf{Set} \to \mathbf{Grp}$ assigns to each set the free group generated by that set.

Definition: Given functors $F, G : \mathfrak{C} \to \mathfrak{D}$, a natural transformation $\tau : F \to D$ is a family of morphisms $\{\tau_A : FA \to GA\}_{A \in Ob(\mathfrak{C})}$, such that for any morphism $f : A \to B$:

$$Gf \circ \tau_A = \tau_A \circ Ff$$

Monads in Category Theory

Definition: A *monad* (T, η, μ) on a category $\mathfrak{C}$ consists of:

- An endofunctor $T : \mathfrak{C} \to \mathfrak{C}$.
- A natural transformation $\eta_A : \mathbf{1} \to T$ (called the *unit*).
- A natural transformation $\mu : T^2 \to T$ (called the *multiplication*).

These satisfy the following coherence conditions:

- (**Associativity**) The diagram

$$\begin{array}{ccc} T^3A & \xrightarrow{T\mu_A} & T^2A \\ {\scriptstyle \mu_{TA}}\downarrow & & \downarrow{\scriptstyle \mu_A} \\ T^2 & \xrightarrow[\mu_A]{} & TA \end{array}$$

  commutes.

- (**Unit Laws**) The diagrams

$$\begin{array}{ccc} TA & \xrightarrow{T\eta_A} & T^2A \\ {\scriptstyle \eta_{TA}}\downarrow & & \downarrow{\scriptstyle \mu_A} \\ T^2A & \xrightarrow[\mu_A]{} & TA \end{array}$$

  commute.

These conditions ensure that T behaves like a monoid in the category of endofunctors on $\mathfrak{C}$, with μ as the multiplication and η as the unit.

Examples

The Powerset Monad Consider the endofunctor $P : \mathbf{Set} \to \mathbf{Set}$, which maps a set X to its powerset $P(X)$, and a function $f : X \to Y$ to the function $Pf : P(X) \to P(Y)$ defined by $Pf(S) = f[S]$ for $S \subseteq X$.

We define the monad structure:

- The unit $\eta_X : X \to P(X)$ is $\eta_X(x) = \{x\}$.
- The multiplication $\mu_X : P(P(X)) \to P(X)$ is $\mu_X(\mathcal{S}) = \bigcup \mathcal{S}$.

Monads in Functional Programming

Type Classes

In functional programming, *type classes* define a set of operations that a type must implement. They are similar to interfaces in object-oriented programming and allow for polymorphism and code reuse. For example, a `Functor` type class in Lean can be defined as:

```
class Functor (f : Type → Type) :=
  (map : (a → b) → f a → f b)
```

A common instance of `Functor` is the `List` type, which implements the `map` operation to apply a function to each element.

Monads

Definition: In functional programming, a `Monad` is defined by the following type class:

```
class Monad (m : Type → Type) :=
  (pure : a → m a)
  (bind : m a → (a → m b) → m b)
```

Here, `pure` corresponds to the unit η, but you may notice that `bind` looks different from μ. Monads in functional programming actually use an alternative but equivalent definition called a Kleisli triple or extension system.

Kleisli Triples and Extension Systems

Equivalence to Monads Any Kleisli triple defines a monad by setting μ in terms of `bind`:

```
mu : m (m a) → m a
mu x = bind x (fun y => y)
```

Conversely, given a monad (T, η, μ), we can define `bind` as:

```
bind x f = mu (map f x)
```

This shows that the functional programming definition of a monad is equivalent to the categorical one. The Kleisli triple emphasizes the way we can chain computations, which is particularly useful in programming.

Why Use Bind?

We often work with functions of the form $f : A \to TB$ in monadic contexts, where T is a monad. These functions represent computations that produce a result of type B along with some monadic effects encapsulated by T.

When we try to compose two such functions $f : A \to TB$ and $g : B \to TC$, we encounter a problem: standard function composition doesn't work because f outputs a value of type TB, but g expects an input of type B.

To resolve this, we use the monadic `bind` operation to sequence the computations while properly handling the effects. The composition is defined as:

$$(f \gg g)(a) = \texttt{bind}(f(a), g)$$

Here, `bind` takes the monadic value $f(a)$, unwraps it to obtain a value of type B, applies g to it, and wraps the result back into the monadic context TC.

This allows us to chain computations $A \to TB$ and $B \to TC$ into a new computation $A \to TC$, effectively composing the functions within the monadic context. This method of composition forms what is known as a *Kleisli category* for the monad T.

Example

The Writer Monad

The Writer monad allows us to augment computations with a log or additional output. Let M be a monoid (e.g., strings with concatenation). We consider functions $f : A \to B$ representing computations. In the Kleisli category for the Writer monad, we lift these functions to Kleisli morphisms $f' : A \to B \times M$, which perform the computation f and produce an additional output in M (e.g., a log message).

Monad Structure in Lean:

We define the monadic structure of the Writer monad in Lean as follows:

```
instance : Monad (M : A -> A × M) where
    pure a := (a, e)
    bind x f :=
        let (a, m1) := x
        let (b, m2) := f a
        (b, m1 <> m2)
```

Here:

- `e` is the identity element of the monoid M.
- `<>` represents the monoid operation in M (e.g., string concatenation).

Composition:

In the Kleisli category, composition of morphisms $f' : A \to B \times M$ and $g' : B \to C \times M$ is defined using `bind`:

$$(f' >=> g')(a) = \texttt{bind}(f'(a), g')$$

This composition combines both the computational results and accumulates the logs using the monoid operation. In Haskell this is often called the "fish operator"

Application: Calculations with Logging

Using the Writer monad, we can perform calculations while logging messages. For example, suppose are performing arithmetic calculations and with each calculation we write a message to the screen. By composing these computations using the monadic structure, we can arrive at a final calculation with a sequence of annotations at each step.

This allows us to model side effects like logging within the pure functional framework of Lean, utilizing the monadic structure to sequence computations and manage the accumulation of logs.

1 The Tactic Monad

When we write our Lean proofs, there is a proof state which gets updated as we apply tactics, and tactics can either fail or succeed. Both of these features

appear to be impure - we have mutable state and functions that may not have an output. You've likely guessed then that these features are handled by monads. As we've seen, monadic programming allows us to simulate side effects while also composing effectful computations in a pure functional way.

The type of a function that can inspect the proof state, modify it, and potentially return something of type α (or fail) is called `tactic` α. In this section we will talk about the `tactic` monad by introducing at a high level how it is designed, which will require some discussion of *monad transformers*, as well as how it works and what we can do with it by writing our own tactics. Using the tactic monad to write custom tactics is called *metaprogramming* and is another interesting feature of Lean - you can write new Lean features in Lean itself.

1.1 Monad Transformers

Note some of the features of tactic mode, we have access to global state, including definitions, theorems, inductive types, notations, etc. Tactics also behave like the option monad, by failing or succeeding. Further they can be used to display messages, like the writer monad, and most importantly the provide access to the list of goals.

The question is then, how do we package together all these monadic effects into a single monad? The answer is *monad transformers*. A monad transformer is a way to supplement a monad with additional effects. Monads can be built up by composing monad transformers, resulting in monads packaged with multiple effects.

The tactic monad needs to be able to read global context, and mutate state - this is done with the reader monad transformer and the state monad transformer.

1.1.1 Reader Monad

In Lean, certain computations require consistent access to shared, immutable context—such as configuration settings or global definitions—throughout their execution. Rather than manually passing this context to every function, the reader monad encapsulates the ability to implicitly read from a shared environment.

Now, to make any monad context-aware, we can use the ReaderT monad transformer. ReaderT takes a type for the environment and adds it to any existing monad. It is defined as follows:

```
def ReaderT (p : Type u) (m : Type u → Type v) (a : Type u) : Type (max u v) :=
  p -> m a
```

Its arguments are as follows:

- `p` is the environment that is accessible to the reader
- `m` is the monad that is being transformed, such as IO

- `a` is the type of values being returned by the monadic computation
- Both `a` and `p` are in the same universe because the operator that retrieves the environment in the monad will have type m `p`.

The Monad instance of ReaderT is given as follows:

```
instance [Monad m] : Monad (ReaderT p m) where
    pure x := fun _ => pure x
    bind result next := fun env => do
        let v ← result env
        next v env
```

ReaderT Monad Example To demonstrate how the `ReaderT` monad transformer allows access to shared context within a computation in Lean, consider the following example. We define a configuration that provides a multiplier factor and use it in a computation without explicitly passing the configuration to every function.

Defining the Configuration

First, we define a simple structure to hold our configuration data:

```
structure Config where
  factor : Int
```

The `Config` structure contains a single field, `factor`, which represents the multiplier factor used in our computations. This structure serves as the shared environment that our monadic computations will access.

Understanding the Identity Monad (`Id`)

In this example, we use the `Id` monad as the base monad for `ReaderT`. The `Id` monad is a fundamental monad in Lean that does not introduce any additional computational effects. It is defined simply as:

```
def Id ( : Type) : Type :=


instance : Monad Id where
  pure x := x
  bind x f := f x
```

Explanation: The `Id` monad allows us to utilize monadic operations without introducing extra effects such as IO, state, or error handling. It essentially acts as a wrapper that passes values through unchanged. By using `Id` as the base monad in `ReaderT Config Id`, we create a reader monad that solely provides access to the shared environment (`Config`) without layering additional effects. This simplifies our example, focusing purely on the Reader functionality.

Creating a ReaderT Computation

Next, we define a computation within the `ReaderT` monad that accesses the configuration and performs a multiplication:

```
def multiplyByFactor : ReaderT Config Id Int :=
  do
    config ← ReaderT.get
    pure (config.factor * 10)
```

In this computation:

- `ReaderT.get` retrieves the current `Config` from the environment.
- We then multiply the `factor` by 10 and return the result using `pure`, which wraps the value back into the monad.

Running the Computation
Finally, we execute the computation by providing a specific configuration:

```
def example : Int :=
  let config := { factor := 5 }
  multiplyByFactor.run config  -- Evaluates to 50
```

Here:

- We create an instance of `Config` with `factor = 5`.
- `multiplyByFactor.run config` executes the computation using the provided `config`, resulting in `50`.

This example demonstrates how the `ReaderT` monad transformer allows access to shared environment data seamlessly. By defining computations within the `ReaderT` monad, we avoid the need to pass the configuration explicitly to each function, resulting in cleaner and more maintainable code.

1.1.2 The State Monad

The `StateT` monad transformer allows us to incorporate mutable state into our computations in Lean. It enables functions to access and modify a shared state without explicitly passing the state around. This is particularly useful for modeling scenarios where state needs to be threaded through a series of computations.

Defining the StateT Monad
First, we define the `StateT` monad transformer. It takes three parameters:
- `s`: The type of the state.
- `m`: The underlying monad.
- `a`: The type of the result.

```
def StateT (s : Type u) (m : Type u → Type v) (a : Type u) : Type (max u v) :=
  s → m (a × s)
```

The `StateT` monad transformer is essentially a function that takes an initial state of type `s` and returns a computation in the monad `m` that produces a result of type `a` along with a new state.

Monad Instance for StateT

Next, we define the `Monad` instance for `StateT`. This allows us to use monadic operations such as `pure` and `bind` within the `StateT` monad.

```
instance [Monad m] : Monad (StateT s m) where
  pure x := fun s => pure (x, s)

  bind result next := fun s => do
    let (v, s') ← result s
    next v s'
```

Here, `pure x` creates a stateful computation that returns the value `x` without modifying the state. The `bind` operation sequences two stateful computations by passing the updated state from the first computation to the second.

State Operations: get and set

To interact with the state within the `StateT` monad, we define two essential operations: `get` and `set`.

```
def get : StateT s m s :=
  fun s => pure (s, s)

def set (s' : s) : StateT s m Unit :=
  fun _ => pure ((), s')
```

The `get` function retrieves the current state, while `set s'` updates the state to `s'`.

Example: Incrementing a Counter

Consider a simple example where we increment a counter within a stateful computation.

```
structure Counter where
  count : Nat
  deriving Repr

def increment : StateT Counter Id Unit :=
  do
    counter ← get
    let newCount := counter.count + 1
    set { count := newCount }

def runIncrement : Counter :=
  let initialState := { count := 0 }
  let (_, finalState) := increment { count := 0 }
  finalState
```

In this example: - `Counter` is a structure that holds a single field `count`. - `increment` is a stateful computation that retrieves the current counter, increments it by one, and updates the state. - `runIncrement` executes the `increment` computation starting with an initial state where `count` is zero, resulting in a final state where `count` is one.

This toy example demonstrates how the `StateT` monad transformer in Lean facilitates stateful computations. By using `StateT`, we can thread state through our functions without manually passing it, leading to cleaner and more maintainable code.

1.2 Tactic Monad Roughly Defined

The tactic monad is defined in terms of these two monad transformers, so that it can read global context as well as manipulate state. We won't go too far into the details of the definition but here is how it is specified:

```
abbrev TacticM := ReaderT Context $ StateT State TermElabM
abbrev Tactic  := Syntax → TacticM Unit
```

Where Context is a metaprogram which captures the gloabal context of the program, and State is the proof state. TermElabM is a monad which is also built similarly out of monad transformers. There is in fact a hierarchy of monads that is built up by iteratively applying `TacticM := ReaderT Context $ StateT State`. At the base of the hierarchy is the BasicIO monad, and TacticM is at the top.

1.2.1 Building A Custom Tactic

In Lean, tactics allow us to manipulate the proof state, automating repetitive proof steps and enhancing Lean's proof capabilities. Writing custom tactics can make proofs more efficient and readable. To do this, we work within the `tactic` monad, which manages the proof state, enabling state inspection, modification, and handling of potential failures.

In this example, we'll create a custom tactic, `my_assumption`, that searches the local context for an assumption that can close the current goal.

Basic Tactic Structure A tactic in Lean is defined as a function of type `tactic` α, where α is the type of the return value. Tactics that only modify the proof state without returning a specific value use `tactic unit`. We use the `meta` keyword to indicate these are for metaprogramming purposes, so they don't need to satisfy the standard checks for non-meta functions.

```
meta def my_first_tactic : tactic unit := tactic.trace "Hello, World."
```

In this basic example, `my_first_tactic` prints "Hello, World" to the message buffer.

Creating a Useful Tactic: `my_assumption` To build our `my_assumption` tactic, we first define a helper function, `find_matching_type`, which checks if any assumption in the local context has a type that matches the current goal. This function will traverse the list of hypotheses, attempting to unify each hypothesis's type with the target type of the goal.

```
meta def find_matching_type (e : expr) : list expr → tactic expr
| []         := tactic.failed
| (H :: Hs)  := do t ← tactic.infer_type H,
                   (tactic.unify e t >> return H) <|> find_matching_type Hs
```

Here's a breakdown of the control flow:

- `find_matching_type` takes an expression `e` (the target type) and a list of expressions (hypotheses).
- If the list is empty, the tactic fails.
- For each hypothesis `H`:
  - Infer `H`'s type `t` using `tactic.infer_type H`.
  - Attempt to unify `e` with `t`. If successful, return `H`.
  - If unification fails, recurse on the remaining hypotheses.

Next, we define the `my_assumption` tactic itself, using `find_matching_type` to locate a hypothesis in the local context that matches the current goal's type and apply it.

```
meta def my_assumption : tactic unit :=
do { ctx ← tactic.local_context,
     t   ← tactic.target,
     find_matching_type t ctx >>= tactic.exact }
<|> tactic.fail "my_assumption tactic failed"
```

Explanation of each part:

- `tactic.local_context`: Retrieves the list of current hypotheses.
- `tactic.target`: Obtains the current goal, which we aim to prove.
- `find_matching_type t ctx`: Searches for a hypothesis in `ctx` with a type that unifies with `t`.
- `>>= tactic.exact`: If a matching hypothesis `H` is found, applies `tactic.exact H` to close the goal.
- `<|> tactic.fail "..."`: If no matching hypothesis is found, the tactic fails with an error message.

Example Usage The `my_assumption` tactic can be used in proofs as follows:

```
example (a b j k : Z) (h1 : a = b) (h2 : j = k) :
  a + j = b + k :=
begin
  my_assumption,
  trivial
end
```

In this example:

- `my_assumption` searches the local context for hypotheses that could match the goal, applying them if found.
- If successful, the proof state is updated; if not, an error is returned.

Conclusion By creating `my_assumption`, we've seen how to manipulate the proof state, retrieve and unify expressions, and handle potential failures. Writing custom tactics in Lean allows us to automate proof steps, making proofs more efficient and maintainable. This example serves as a foundation for more advanced tactics, opening possibilities for complex automated proof strategies within Lean.