50.054 Monad

ISTD, SUTD

Learning Outcomes

1. Describe and define Monads
2. Apply Monad to in design and develop highly modular and resusable software.

Recall the example

```
enum MathExp {
    case Plus(e1:MathExp, e2:MathExp)
    case Minus(e1:MathExp, e2:MathExp)
    case Mult(e1:MathExp, e2:MathExp)
    case Div(e1:MathExp, e2:MathExp)
    case Const(v:Int)
}
```

Recall the example

```
import MathExp.*
def eval(e:MathExp):Option[Int] = e match {
    case Plus(e1, e2)  => eval(e1) match {
        case None     => None
        case Some(v1) => eval(e2) match {
            case None     => None
            case Some(v2) => Some(v1 + v2)
        }
    }
    // cases for Mult and Minus are omitted.
    case Div(e1, e2)   => eval(e1) match {
        case None     => None
        case Some(v1) => eval(e2) match {
            case None     => None
            case Some(0)  => None
            case Some(v2) => Some(v1 / v2)
        }
    }
    case Const(i)      => Some(i)
}
```

Recall the example

```
import MathExp.*
def eval(e:MathExp):Either[ErrMsg, Int] = e match {
    case Plus(e1, e2)  => eval(e1) match {
        case Left(m)    => Left(m)
        case Right(v1) => eval(e2) match {
            case Left(m)    => Left(m)
            case Right(v2) => Right(v1 + v2)
        }
    }
    // cases for Mult and Minus are omitted.
    case Div(e1, e2)   => eval(e1) match {
        case Left(m)    => Left(m)
        case Right(v1) => eval(e2) match {
            case Left(m)    => Left(m)
            case Right(0)   => Left(s"div by zero caused by ${e.toString}")
            case Right(v2) => Right(v1 / v2)
        }
    }
    case Const(i)       => right(i)
}
```

Two Complaints

- Verbose code
- Switching from `Option[Int]` to `Either[ErrMsg,Int]` requires a lot of modification.

Recap Applicative

```scala
trait Applicative[F[_]] extends Functor[F] {
    def ap[A, B](ff: F[A => B])(fa: F[A]): F[B]
    def pure[A](a: A): F[A]
    def map[A, B](fa: F[A])(f: A => B): F[B] = ap(pure(f))(fa)
}

given optApp:Applicative[Option] = new Applicative[Option] {
    def pure[A](a:A):Option[A] = Some(a)
    def ap[A, B](ff: Option[A => B])(fa: Option[A]):Option[B] =
        ff.flatMap(fa => fa.map(f))
        // because .map and .flatMap are methods of Option
}

val o1 = optApp.pure(1)
val of = optApp.pure((x:Int) => x + 1)
optApp.ap(of)(o1) // Some(2)
```

Monad

```scala
trait Monad[F[_]] extends Applicative[F] {
    def bind[A,B](fa:F[A])(f:A => F[B]):F[B]
    def pure[A](v:A):F[A]
    def ap[A, B](ff: F[A => B])(fa: F[A]): F[B] =
        bind(ff)((f:A=>B) => bind(fa)((a:A)=> pure(f(a))))
}

given optMonad:Monad[Option] = new Monad[Option] {
    def bind[A,B](fa:Option[A])(f:A=>Option[B]):Option[B] = fa match {
        case None => None
        case Some(a) => f(a)
    }
    def pure[A](v:A):Option[A] = Some(v)
}
```

Comparison

- ap has type `F[A] => F[A => B] => F[B]`
  - e.g. `optApp.ap` has type `Option[A] => Option[A => B] => Option[B]`
- bind has type `F[A] => (A => F[B]) => F[B]`
  - e.g. `optMonad.bind` has type `Option[A] => (A => Option[B]) => Option[B]`

Monad

Rewriting `eval` using Option Monad.

```
def eval(e:MathExp)(using i:Monad[Option]):Option[Int] = e match {
    case Plus(e1, e2)  =>
        i.bind(eval(e1))( v1 => {
            i.bind(eval(e2))( v2 => i.pure(v1+v2))
        })
    // cases for Mult and Minus are omitted.
    case Div(e1, e2)   =>
        i.bind(eval(e1))( v1 => {
            i.bind(eval(e2))( v2 =>
                if (v2 == 0) {
                    None
                } else {i.pure(v1/v2)})
        })
    case Const(v)      => i.pure(v)
}
```

Definition of Either

```scala
enum Either[+A, +B] {
    case Left(v: A)
    case Right(v: B)
    // to support for comprehension
    def flatMap[A,D](f: B => Either[A,D]):Either[A,D] = this match {
        case Left(a) => Left(a)
        case Right(b) => f(b)
    }
    def map[D](f:B => D):Either[A,D] = this match {
        case Left(a) => Left(a)
        case Right(b) => Right(f(b))
    }
}
```

- Scala OOP and Generic squabble
  - A appears as co- and contra- variants in `flatMap` thus also in `Either`
  - because Subtyping among function type

$$\frac{A <: C \qquad D <: B}{A \Rightarrow B <: C \Rightarrow D}$$

Definition of Either

```scala
enum Either[+A, +B] {
    case Left(v: A)
    case Right(v: B)
    // to support for comprehension
    def flatMap[C>:A,D](f: B => Either[C,D]):Either[C,D] = this match {
        case Left(a) => Left(a)
        case Right(b) => f(b)
    }
    def map[D](f:B => D):Either[A,D] = this match {
        case Right(b) => Right(f(b))
        case Left(e) => Left(e)
    }
}
```

- Scala OOP and Generic squabble
    - `C>:A` introduces a generic type `C` which has a lower bound `A`.

Either Monad

```
type ErrMsg = String
type EitherErr = [B] =>> Either[ErrMsg,B]

given eitherErrMonad:Monad[EitherErr] = new Monad[EitherErr] {
    def bind[A,B](fa:EitherErr[A])(f:A=>EitherErr[B]):EitherErr[B] = fa match {
        case Left(a) => Left(a)
        case Right(b) => f(b)
    }
    // same as
    // def bind[A,B](fa:EitherErr[A])(f:A=>EitherErr[B]):EitherErr[B] = fa.flatMap(f)
    def pure[A](v:A):EitherErr[A] = Right(v)
}
```

- `[B] =>> Either[ErrMsg, B]` defines a type lambda
- `EitherErr[_]` is a single argument type constructor

Either Monad

Rewriting `eval` using EitherErr Monad.

```
def eval(e:MathExp)(using i:Monad[EitherErr]):EitherErr[Int] = e match {
    case Plus(e1, e2)  =>
        i.bind(eval(e1))( v1 => {
            i.bind(eval(e2))( v2 => i.pure(v1+v2))
        })
    // cases for Mult and Minus are omitted.
    case Div(e1, e2)   =>
        i.bind(eval(e1))( v1 => {
            i.bind(eval(e2))( v2 =>
                if (v2 == 0) {
                    Left(s"div by zero caused by ${e.toString}")
                } else {i.pure(v1/v2)})
        })
    case Const(v)      => m.pure(v)
}
```

Recall Option Monad

`eval` using Option Monad.

```
def eval(e:MathExp)(using i:Monad[Option]):Option[Int] = e match {
    case Plus(e1, e2)  =>
        i.bind(eval(e1))( v1 => {
            i.bind(eval(e2))( v2 => i.pure(v1+v2))
        })
    // cases for Mult and Minus are omitted.
    case Div(e1, e2)   =>
        i.bind(eval(e1))( v1 => {
            i.bind(eval(e2))( v2 =>
                if (v2 == 0) {
                    None
                } else {i.pure(v1/v2)})
        })
    case Const(v)      => i.pure(v)
}
```

Monad Laws

Similar to Functor and Applicative, all instances of Monad must satisfy the following three Monad Laws.

1. Left Identity: `bind(pure(a))(f)` $\equiv$ `f(a)`
2. Right Identity: `bind(m)(pure)` $\equiv$ `m`
3. Associativity: `bind(bind(m)(f))(g)` $\equiv$ `bind(m)(x => bind(f(x))(g))`

Left Identity Law

1. Left Identity: `bind(pure(a))(f)` $\equiv$ `f(a)`

- Intuitively speaking, a `bind` operation is to *extract* results of type `A` from its first argument with type `F[A]` and apply `f` to the extracted results.
- Left identity law enforces that binding a lifted value to `f`, is the same as applying `f` to the unlifted value directly, because the lifting and the *extraction* of the bind cancel each other.

Right Identity Law

2. Right Identity: `bind(m)(pure)` $\equiv$ `m`

- Right identity law enforces that binding a lifted value to `pure`, is the same as the lifted value, because *extracting* results from `m` and `pure` cancel each other.

Associativity Law

Similar to Functor and Applicative, all instances of Monad must satisfy the following three Monad Laws.

3. Associativity: `bind(bind(m)(f))(g)` $\equiv$ `bind(m)(x => bind(f(x))(g))`

- The Associativity law enforces that binding a lifted value `m` to `f` then to `g` is the same as binding `m` to a monadic bind composition `(x => bind(f(x))(g))`

Bind and FlatMap

In general, we also find that

`bind(o)(f)` is equivalent to `o.flatMap(f)`, if `flatMap` is a defined method in `o`.

More Syntactic Perks

Recall Scala allow us to write

```
e1.flatMap( v1 => e2.flatMap( v2 => ... en.map(vn => e ... )))
```

as

```
for {
    v1 <- e1
    v2 <- e2
    ...
    vn <- en
} yield (e)
```

More Syntactic Perks

```
def eval(e:MathExp)(using i:Monad[Option]):Option[Int] = e match {
    case Plus(e1, e2)  => for {
        v1 <- eval(e1)
        v2 <- eval(e2)
    } yield (v1+v2)
    // cases for Mult and Minus are omitted.
    case Div(e1, e2)   => for {
        v1 <- eval(e1)
        v2 <- eval(e2)
        // Something not right!
        // How to check whether v2 is 0?
    } yield (v1/v2)
    case Const(x)      => i.pure(x)
}
```

To allow us to check and stop the yield `(v1/v2)` as result, we need a special type of Monad.

Monad Error

```
trait ApplicativeError[F[_], E] extends Applicative[F] {
    def raiseError[A](e:E):F[A]
}

trait MonadError[F[_], E] extends Monad[F] with ApplicativeError[F, E] {
    override def raiseError[A](e:E):F[A]
}

given optMonadErr:MonadError[Option,ErrMsg] = new MonadError[Option, ErrMsg] {
    def bind[A,B](fa:Option[A])(f:A=>Option[B]):Option[B] = fa.flatMap(f)
    def pure[A](v:A):Option[A] = Some(v)
    def raiseError[A](e:ErrMsg):Option[A] = None
}
```

- `raiseError(error_message)` signals an error into the monadic functor.
    - Though in this case the message is ignored.

More Syntactic Perks

```
def eval(e:MathExp)(using i:MonadError[Option]):Option[Int] = e match {
    case Plus(e1, e2)  => for {
        v1 <- eval(e1)
        v2 <- eval(e2)
    } yield (v1+v2)
    // cases for Mult and Minus are omitted.
    case Div(e1, e2)   => for {
        v1 <- eval(e1)
        v2 <- eval(e2)
        _  <- if (v2 == 0) {
            i.raiseError(s"div by zero caused by ${e.toString}")
            } else {
                i.pure(())
            }
    } yield (v1/v2)
    case Const(x)      => i.pure(x)
}
```

EitherErr as Monad Error

```
given eitherErrMonadErr:MonadError[EitherErr,ErrMsg] = new MonadError[EitherErr, ErrMsg] {
    def bind[A,B](fa:EitherErr[A])(f:A=>EitherErr[B]):EitherErr[B] = fa.flatMap(f)
    def pure[A](v:A):EitherErr[A] = Right(v)
    def raiseError[A](e:ErrMsg):EitherErr[A] = Left(e)
}
```

More Syntactic Perks

```
def eval(e:MathExp)(using i:MonadError[EitherErr]):EitherErr[Int] = e matc
    case Plus(e1, e2)  => for {
        v1 <- eval(e1)
        v2 <- eval(e2)
    } yield (v1+v2)
    // cases for Mult and Minus are omitted.
    case Div(e1, e2)   => for {
        v1 <- eval(e1)
        v2 <- eval(e2)
        _  <- if (v2 == 0) {
            i.raiseError(s"div by zero caused by ${e.toString}")
            } else {
                i.pure(())
            }
    } yield (v1/v2)
    case Const(x)      => i.pure(x)
}
```

Commonly Used Monads

1. List Monad
2. Reader Monad
3. State Monad

List Monad

```scala
given listMonad:Monad[List] = new Monad[List] {
    def pure[A](v:A):List[A] = List(v)
    def bind[A,B](fa:List[A])(f:A => List[B]):List[B] =
        fa.flatMap(f)
}
```

List Monad Example

```scala
import java.util.Date
import java.util.Calendar
import java.util.GregorianCalendar
import java.text.SimpleDateFormat
case class Staff(id:Int, dob:Date)

def mkStaff(id:Int, dobStr:String):Staff = {
    val sdf = new SimpleDateFormat("yyyy-MM-dd")
    val dobDate = sdf.parse(dobStr)
    Staff(id, dobDate)
}
val staffData = List(
    mkStaff(1, "1076-01-02"),
    mkStaff(2, "1986-07-24")
)
```

```scala
def ageBelow(staff:Staff, age:Int): Boolean =
    staff match {
    case Staff(id, dob) => {
        val today = new Date()
        val calendar = new GregorianCalendar();
        calendar.setTime(today)
        calendar.add(Calendar.YEAR, -age)
        val ageYearsAgo = calendar.getTime()
        dob.after(ageYearsAgo)
    }
}

def query(data:List[Staff]):List[Staff] = for {
    staff <- data           // from data
    if ageBelow(staff, 40) // where staff.age < 40
} yield staff               // select *
```

List Monad allows us to define high-level queries and data manipulation operations

Reader Monad

```scala
case class Reader[R, A] (run: R=>A) {
    // we need flatMap and map for for-comprehension
    def flatMap[B](f:A =>Reader[R,B]):Reader[R,B] = this match {
        case Reader(ra) => Reader (
            r => f(ra(r)) match {
                case Reader(rb) => rb(r)
            }
        )
    }
    def map[B](f:A=>B):Reader[R, B] = this match {
        case Reader(ra) => Reader (
            r => f(ra(r))
        )
    }
}
```

- Reader is an algebraic data type that "prescribes" some computation `R=>A`
    - Input `R` is some shared information.
    - `A` is the result.

Reader Monad

```scala
case class Reader[R, A] (run: R=>A) {
    // we need flatMap and map for for-comprehension
    def flatMap[B](f:A =>Reader[R,B]):Reader[R,B] = this match {
        case Reader(ra) => Reader (
            r => f(ra(r)) match {
                case Reader(rb) => rb(r)
            }
        )
    }
    def map[B](f:A=>B):Reader[R, B] = this match {
        case Reader(ra) => Reader (
            r => f(ra(r))
        )
    }
}
```

- `map` takes a function `f:A=>B` and applies to the result of returned by the current computation `ra`.
- `flatMap` takes a function `f:A=>Reader[R,B]`
  1. it runs the current computation `ra`
  2. applies `f` to the results which yields another computation `rb`
  3. it runs `rb` with the same shared input `r`.

Reader Monad

```scala
type ReaderM = [R] =>> [A] =>> Reader[R, A]

trait ReaderMonad[R] extends Monad[ReaderM[R]] {
    override def pure[A](v:A):Reader[R, A] = Reader (r => v)
    override def bind[A,B](fa:Reader[R, A])(f:A=>Reader[R,B]):Reader[R,B] =
        fa.flatMap(f)
    def ask:Reader[R,R] = Reader( r => r)
    def local[A](f:R=>R)(r:Reader[R,A]):Reader[R,A] = r match {
        case Reader(ra) => Reader( r => {
            val localR = f(r)
            ra(localR)
        })
    }
}
```

- `ReaderMonad[R]` is a derived type class of `Monad[ReaderM[R]]`
- function `ask` queries for the shared input.
- function `local` temporarily run the local computation with an updated share envrionment.

API caller via Reader Monad

```scala
case class API(url:String)

given APIReader:ReaderMonad[API] =
    new ReaderMonad[API] {}

def get(path:String)(using pr:ReaderMonad[API])
    :Reader[API,Unit] = for {
    r <- pr.ask
    s <- r match {
        case API(url) =>
            pr.pure(println(s"${url}${path}"))
    }
} yield s
```

```scala
def authServer(api:API):API =
    API("https://127.0.0.10/")

def test1(using pr:ReaderMonad[API])
    :Reader[API, Unit] = for {
    a <- pr.local(authServer)(get("auth"))
    t <- get("time")
    j <- get("job")
} yield (())

def runtest1():Unit = test1 match {
    case Reader(run) =>
        run(API("https://127.0.0.1/"))
}
```

- Authentication request is sent to `https://127.0.0.10/`
- time and job requests are sent to `https://127.0.0.1/`

State Monad

```
case class State[S,A]( run:S=>(S,A)) {
    def flatMap[B](f: A => State[S,B]):State[S,B] = this match {
        case State(ssa) => State(
            s=> ssa(s) match {
                case (s1,a) => f(a) match {
                    case State(ssb) => ssb(s1)
                }
            }
        )
    }
    def map[B](f:A => B):State[S,B] = this match {
        case State(ssa) => State(
            s=> ssa(s) match {
                case (s1, a) => (s1, f(a))
            }
        )
    }
}
```

- data type `State` prescribes a stateful computation
  - `S` is the state type
  - `A` is the result type
  - States are updated in a computation `S=>(S,A)`

State Monad

```
case class State[S,A]( run:S=>(S,A)) {
    def flatMap[B](f: A => State[S,B]):State[S,B] = this match {
        case State(ssa) => State(
            s=> ssa(s) match {
                case (s1,a) => f(a) match {
                    case State(ssb) => ssb(s1)
                }
            }
        )
    }
    def map[B](f:A => B):State[S,B] = this match {
        case State(ssa) => State(
            s=> ssa(s) match {
                case (s1, a) => (s1, f(a))
            }
        )
    }
}
```

- `map` runs the current stateful computation `ssa`, which returns the result `a` and the updated state `s1`, then it applies `f` to the result.
- `flatMap` runs the current stateful computation `ssa`, which returns the result `a` and the updated state `s1`, then it applies `f` to the result which generates the new stateful computation `ssb`. Finally it runs `ssb` with the updated state `s1`.

State Monad

```
type StateM = [S] =>> [A] =>> State[S,A]

trait StateMonad[S] extends Monad[StateM[S]] {
    override def pure[A](v:A):State[S,A] = State( s=> (s,v))
    override def bind[A,B](fa:State[S,A])(ff:A => State[S,B])
        :State[S,B] = fa.flatMap(ff)
    def get:State[S, S] = State(s => (s,s))
    def set(v:S):State[S,Unit] = State(s => (v,()))
}
```

- `StateMonad[S]` is a derived type class of `Monad[StateM[S]]`
- function `get` queries for the current state.
- function `set` (permenantly) updates the state for the following computation. (`Unit` is like `void` in Java)
  - `()` is the only value having type `Unit`

Counter with State Monad

```scala
case class Counter(c:Int)

given counterStateMonad:StateMonad[Counter] =
    new StateMonad[Counter] {}

def incr(using csm:StateMonad[Counter]):State[Counter,Unit] = for {
    Counter(c) <- csm.get
    _ <- csm.set(Counter(c+1))
} yield ()

def app(using csm:StateMonad[Counter]):State[Counter, Int] = for {
    _ <- incr
    _ <- incr
    Counter(v) <- csm.get
} yield v
```

In the above example, we define an counter application that increase the counter in the state when `incr` is called.

Monads not covered, but worth a read

- Writer Monad
- Monad Transformer

Summary

In this class, we covered

1. A Monad Functor is a sub-class of Applicative Functor, with the method `bind`.
2. The three laws of Monad Functor.
3. A few commonly used Monad such as, List Monad, Option Monad, Reader Monad and State Monad.

Further Reading

https://medium.com/beingprofessional/understanding-functor-and-monad-with-a-bag-of-peanuts-8fa702b3f69e
https://www.continuum.be/en/blog/a-gentle-introduction-to-monads/