50.054 Applicative

ISTD, SUTD

Learning Outcomes

- Describe and define derived type class
- Describe and define Applicative Functors

Recap

Functor type class

```
trait Functor[T[_]] {
    // Functor is not built-in
    def map[A,B](t:T[A])(f:A => B):T[B]
}

given listFunctor:Functor[List] = new Functor[List] {
    def map[A,B](l:List[A])(f:A => B):List[B] = l.map(f)
}
```

Option type - builtin

```
enum Option[+A] {
    case None
    case Some(v:A)
}
```

Can `Option[_]` be a functor instance?

Recap

```
given optionFunctor:Functor[Option] = new Functor[Option] {
    def map[A,B](o:Option[A])(f:A => B):Option[B] = ???
}
```

How about `Either`?

Recap Ordering

Recall that

```
// Ordering is builtin
trait Ordering[A] {
    def compare(x:A,y:A):Int // less than: -1, equal: 0, greater than 1
}

given intOrd:Ordering[Int] = new Ordering[Int] {
    def compare(x:Int, y:Int):Int =
        if (x == y) { 0 }
        else if (x > y) { 1 }
        else { -1 }
}

intOrder.compare(10, 9) // yields 1
```

BTW, there's a new and shorter syntax for anonymous given

Recall that

```
// Ordering is builtin
trait Ordering[A] {
    def compare(x:A,y:A):Int // less than: -1, equal: 0, greater than 1
}

given Ordering[Int] {
    def compare(x:Int, y:Int):Int =
        if (x == y) { 0 }
        else if (x > y) { 1 }
        else { -1 }
}

given_Ordering_Int.compare(10, 9) // yields 1
```

Beyond builtin type classes

- The community needs some predefined type classes (and functions).
- Some popular implementations such as
  - https://typelevel.org/cats/
  - https://scalaz.github.io/7/

Derived Type Class

- Code snippet from Cats

```
trait Eq[A] {
    def eqv(x:A, y:A):Boolean
}

trait Order[A] extends Eq[A] {
    def compare(x:A, y:A):Int
    def eqv(x:A, y:A):Boolean = compare(x,y) == 0
    def gt(x:A,  y:A):Boolean = compare(x,y) > 0
    def lt(x:A,  y:A):Boolean = compare(x,y) < 0
}
```

- `Order` is a *derived type class* of `Eq`
  - An instance of `Order` is also an instance of `Eq` automatically.
- Methods `eqv`, `gt` and `lt` are given some default implementaitons..

Derived Type Class Instances

```
given eqInt:Eq[Int] = new Eq[Int] { // ok, but redundant
    def eqv(x:Int, y:Int):Boolean = x == y
}

given orderInt:Order[Int] = new Order[Int] {
    def compare(x:Int, y:Int):Int = x - y
}

eqInt.eqv(1,1) // true,
orderInt.eqv(1,1) // true

def g(x:Int, y:Int)(using i:Eq[Int]) = i.eqv(x,y)
```

Derived Type Class Instances (Alternative)

```scala
trait Order[A] extends Eq[A] {
    def compare(x:A, y:A):Int
    def gt(x:A,  y:A):Boolean = compare(x,y) > 0
    def lt(x:A,  y:A):Boolean = compare(x,y) < 0
}

given orderInt(using eqInt:Eq[Int]):Order[Int] = new Order[Int] {
    def eqv(x:Int,y:Int):Boolean = x == y
    def compare(x:Int, y:Int):Int = x - y
}

def g(x:Int, y:Int)(using i:Eq[Int]) = i.eqv(x,y)
```

Applicative Functor

The `Applicative` Functor is a derived type class of `Functor`, which is defined as follows

```
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)
}
```

- pure function "lifts" a value of type `A` into functor `F[A]`
- ap function "applies" a lifted function `F[A => B]` to a functor value `F[A]`
- Overridden function `map` is defined in terms of ap and pure.

Applicative Example

We define the Applicative instance of `List` functor as follows,

```
given listApplicative:Applicative[List] = new Applicative[List] {
    def pure[A](a:A):List[A] = List(a)
    def ap[A, B](ff: List[A => B])(fa: List[A]):List[B] =
        ff.flatMap( f => fa.map(f))
}

val l = List(1,2,3)
listApplicative.map(l)(x => x + 1) // List(2,3,4)
```

In the above we use `listApplicative` as if it is the instance of `Functor`.

Applicative Example

Why `map(l)(f)` can be defined as `ap(pure(f))(l)`?

```
map(l)(x=>x+1)  // by defn of `map` in listApplicative
ap(pure(x=>x+1))(l)  // by defn of `pure` in listApplicative
ap(List(x=>x+1))(l)  // by defn of `ap` in listApplicative
List(x=>x+1).flatMap(f=>l.map(f)) // by defn of `flatMap` of list
List( (f=>l.map(f))(x=>x+1) ).flatten  // beta
List( l.map(x=>x+1) ).flatten // flatten destroys the outer list
l.map(x=>x+1)
```

Applicative Example

We define the Applicative instance of `List` functor as follows,

```
given listApplicative:Applicative[List] = new Applicative[List] {
    def pure[A](a:A):List[A] = List(a)
    def ap[A, B](ff: List[A => B])(fa: List[A]):List[B] =
        ff.flatMap( f => fa.map(f))
}
val l = List(1,2,3)
val fs:List[Int => Int] = List(x => x + 1, x => x * 2)
listApplicative.ap(fs)(l) // List(2, 3, 4, 2, 4, 6)
```

Applicative Example

We define the Applicative instance of `Option` functor as follows,

```
given optionApplicative: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 match {
        case None => None
        case Some(f) => fa match {
            case None => None
            case Some(v) => Some(f(v))
        }
    }
}
val o1 = optionApplicative.pure(1)
val of = optionApplicative.pure((x:Int) => x + 1)
optionApplicative.ap(of)(o1) // Some(2)
```

Applicative Laws

1. Identity: `ap(pure(x=>x))` $\equiv$ `y=>y`
2. Homomorphism: `ap(pure(f))(pure(x))` $\equiv$ `pure(f(x))`
3. Interchange: `ap(u)(pure(y))` $\equiv$ `ap(pure(f=>f(y)))(u)`
4. Composition: `ap(ap(ap(pure(f=>f.compose))(u))(v))(w)` $\equiv$ `ap(u)(ap(v)(w))`

Identity and Homomoprhism Laws

1. Identity: `ap(pure(x=>x))` $\equiv$ `y=>y`
2. Homomorphism: `ap(pure(f))(pure(x))` $\equiv$ `pure(f(x))`

- Identity law states that applying a lifted identity function of type `A=>A` is same as an identity function of type `F[A] => F[A]` where `F` is the applicative functor.
- Homomorphism says that applying a lifted function (which has type `A=>A` before being lifted) to a lifted value, is equivalent to applying the unlifted function to the unlifted value directly and then lift the result.

Interchange Law

3. Interchange: `ap(u)(pure(y))` $\equiv$ `ap(pure(f=>f(y)))(u)`

- To understand Interchange law let's consider the following simplified equation in lambda calculus

$$u\ y \equiv (\lambda f.(f\ \ y))\ u$$

  - Interchange law says that the above equation remains valid when u is already lifted, as long as we also lift y.

Composition law

4. Composition: `ap(ap(ap(pure(f=>f.compose))(u))(v))(w)` $\equiv$ `ap(u)(ap(v)(w))`

- To understand the Composition law, we consider the following equation in lambda calculus

$$(((\lambda f.(\lambda g.(f \circ g)))\ u)\ v)\ w \equiv u\ (v\ \ w)$$

$$\begin{array}{rl}
(\underline{((\lambda f.(\lambda g.(f \circ g)))\ u)}\ v)\ w & \longrightarrow_{\beta} \\
(\underline{(\lambda g.(u \circ g))\ v})\ w & \longrightarrow_{\beta} \\
(u \circ v)\ w & \longrightarrow_{\texttt{composition}} \\
u\ (v\ \ w) &
\end{array}$$

The Composition Law says that the above equation remains valid when u, v and w are lifted, as long as we also lift $\lambda f.(\lambda g.(f \circ g))$.

Relationship between Applicative and Functor

Let `T[_]` be a type constructor such that it satisfies the applicative laws. Then `T[_]` satisfies the Functor laws.

Another Applicative Example

```scala
type EitherStr = [C] =>> Either[String, C]
given eitherStrFunctor:Functor[EitherStr] = new Functor[EitherStr] {
    def map[A,B](t:EitherStr[A]) (f : A => B) : EitherStr[B] = t match {
        case Left(msg) => Left(msg)
        case Right(a)  => Right(f(a))
    }
}

given eitherStrApplicative:Applicative[EitherStr] = new Applicative[EitherStr] {
    def pure[A](a:A):EitherStr[A] = Right(a)
    def ap[A, B](ff: EitherStr[A => B])(fa: EitherStr[A]):EitherStr[B] = ff match {
        case Left(msg) => Left(msg)
        case Right(f) => fa match {
            case Left(msg) => Left(msg)
            case Right(v) => Right(f(v))
        }
    }
}
val em1 = eitherStrApplicative.pure(1) // EitherStr[Int]
val emf = eitherStrApplicative.pure( (x : Int)  => x + 1 ) // EitherStr[Int => Int]
eitherStrApplicative.ap(emf)(em1) // yields Right(2)
```

- `[C] =>> Either[String, C]` defines a type lambda
- `EitherStr[_]` is a single argument type constructor

Little Syntactic Perk

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)
```

Little Syntactic Perk

```scala
given listApplicative:Applicative[List] = new Applicative[List] {
    def pure[A](a:A):List[A] = List(a)
    def ap[A, B](ff: List[A => B])(fa: List[A]):List[B] =
        ff.flatMap( f => fa.map(f))
}
```

can be rewritten

```scala
given listApplicative:Applicative[List] = new Applicative[List] {
    def pure[A](a:A):List[A] = List(a)
    def ap[A, B](ff: List[A => B])(fa: List[A]):List[B] = for {
        f <- ff
        a <- fa
    } yield f(a)
}
```

Little Syntactic Perk

Now the builtin `Option[A]` type has `map` and `flatMap` predefined too

```
enum Option[+A] {
    case None
    case Some(v)
    def map[B](f:A=>B):Option[B] = this match {
        case None => None
        case Some(v) => Some(f(v))
    }
    def flatMap[B](f:A=>Option[B]):Option[B] = this match {
        case None => None
        case Some(v) => f(v) match {
            case None => None
            case Some(u) => Some(u)
        }
    }
}
```

Little Syntactic Perk

```scala
def ap[A, B](ff: Option[A => B])(fa: Option[A]):Option[B] = ff match {
    case None => None
    case Some(f) => fa match {
        case None => None
        case Some(v) => Some(f(v))
    }
}
```

can be simplifed

```scala
def ap[A, B](ff: Option[A => B])(fa: Option[A]):Option[B] =
    ff.flatMap( f => fa.map(f))
```

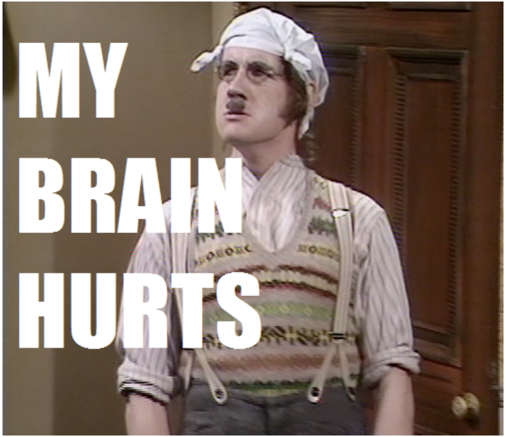
Or the for-comprehension.

Quick Summary

We have discussed

- Derived type classes
- A special kind of Functor, namely Applicative functor.

When



it means I am getting a hang of it.