50.054 Type Class, Functor and Error Handling

ISTD, SUTD

Learning Outcomes

- develop generic programming style code using `Functor` type class.
- make use of `Option` and `Either` to handle and manipulate errors and exceptions.

Recap - Type Class

```
trait JS[A] {
    def toJS(v:A):String
}
given toJSInt:JS[Int] = new JS[Int]{
    def toJS(v:Int):String = v.toString
}
given toJSString:JS[String] = new JS[String] {
    def toJS(v:String):String = s"'${v}'"
}
given toJSBoolean:JS[Boolean] = new JS[Boolean] {
    def toJS(v:Boolean):String = v.toString
}
def g(x:Boolean)(using i:JS[Boolean]):String = i.toJS(x)
g(true)(toJSBoolean) // "true"
g(true) // "true"
```

Recap - Type Class

```
given toJSList[A](using jsa:JS[A]):JS[List[A]] = new JS[List[A]] {
    def toJS(as:List[A]):String = {
        val j = as.map(a=>jsa.toJS(a)).mkString(",")
        s"[${j}]"
    }
}
def gg(xs:List[Boolean])(using i:JS[List[Boolean]]):String = i.toJS(xs)
gg(List(false,true)) // "[false,true]"
```

Recall

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

Can we define a `map` function for `BTree`?

```
enum BTree[+A] {
    case Empty
    case Node(v:A, lft:BTree[A], rgt:BTree[A])
}
```

What about `MyList[A]`?

Let's go Type Class!

```
trait Functor[T[_]] {
    def map[A,B](t:T[A])(f:A => B):T[B]
}
```

- `Functor` is a type class defined for a type constructor `T[_]` instead of `T`.
- `T[_]` is a generic type constructor that is taking another type `A` and return `T[A]`.
- `T[_]` is said to have *kind* `* => *`
  - c.f. `JS` is a type class defined for a type `A`, where `A` is not a type constructor, `A` has kind `*`.
- This is also known as *Higher-kinded Type*.

The Functor type class

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

given btreeFunctor:Functor[BTree] = new Functor[BTree] {
    import BTree.*
    def map[A,B](t:BTree[A])(f:A => B):BTree[B] = t match {
        case Empty => Empty
        case Node(v, lft, rgt) => Node(f(v), map(lft)(f), map(rgt)(f))
    }
}
```

The Functor type class

```
enum MyList[+A] {
    case Nil
    case Cons(x:A, xs:MyList[A])

    def map[B](f:A => B):MyList[B] = this match {
        case Nil => Nil
        case Cons(hd, tl) => Cons(f(hd), tl.map(f))
    }
}


given myListFunctor:Functor[MyList] = new Functor[MyList] {
    def map[A,B](ml:MyList[A])(f:A => B):MyList[B] = ml.map(f)
}
```

The Functor type class

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

val t = BTree.Node(2, BTree.Node(1, BTree.Empty, BTree.Empty),
        BTree.Node(3, BTree.Empty, BTree.Empty))
btreeFunctor.map(t)((x:Int) => x + 1)
// BTree.Node(3, BTree.Node(2, BTree.Empty, BTree.Empty),
//          BTree.Node(4, BTree.Empty, BTree.Empty))

val ml = MyList.Cons(1,MyList.Nil)
myListFunctor.map(ml)((x:Int) => x + 1)
//  MyList.Cons(2,MyList.Nil)
```

Functor Laws

- For functor implementations to be predictable, some laws must be followed.
  1. Identity: `i => map(i)(x => x)` $\equiv$ `x => x`
  2. Composition Morphism:

`i=> map(i)(f.compose(g))` $\equiv$ `(i => map(i)(f)).compose(j => map(j)(g))`

- It's the programmer's duty to verify them, not the compiler.
  - Such an algorithm does not exist in general.

Showing `myListFunctor` satisfying Identity Law

```
(i:MyList[A]) => map(i)(x => x) // by defn of myListFunctor
(i:MyList[A]) => i.map(x=>x) // by defn of the map method in MyList
(i:MyList[A]) => i // by alpha renaming
(x:MyList[A]) => x
```

Showing `myListFunctor` satisfying Composition Morphism Law

```
// LHS
(i:MyList[A]) => map(i)(f.compose(g))
// by defintion of myListFunctor
(i:MyList[A]) => i.map(f.compose(g))
// by definition of compose
(i:MyList[A]) => i.map(x => f(g(x)))

// RHS
((i:MyList[A]) => map(i)(f)).compose(j => map(j)(g))
// by definition of compose
(l:MyList[A]) => ((i:MyList[A]) => map(i)(f))((j => map(j)(g))(l)) // beta
(l:MyList[A]) => ((i:MyList[A]) => map(i)(f))(map(l)(g))  // beta
(l:MyList[A]) => map(map(l)(g))(f) // definition of myListFunctor
(l:MyList[A]) => map(l.map(g))(f)  // definition of myListFunctor
(l:MyList[A]) => l.map(g).map(f)
```

RHS = LHS for all lists (can be proven by induction, e.g. firstly `Nil`, then a non-empty `Cons x xs`, the I.H. applied to `xs`).

The Foldable type class

Similar to defining the Functor type class for a set of overloaded map functions, we can define the Foldable type class for a set of overloaded foldLeft functions.

```
trait Foldable[T[_]]{
    def foldLeft[A,B](t:T[B])(acc:A)(f:(A,B)=>A):A
}

given listFoldable:Foldable[List] = new Foldable[List] {
    def foldLeft[A,B](t:List[B])(acc:A)(f:(A,B)=>A):A = t.foldLeft(acc)(f)
}
```

The Foldable type class

```
given btreeFoldable:Foldable[BTree] = new Foldable[BTree] {
    import BTree.*
    def foldLeft[A,B](t:BTree[B])(acc:A)(f:(A,B)=>A):A = t match {
        case Empty => acc
        case Node(v, lft, rgt) => {
            val acc1 = f(acc,v)
            val acc2 = foldLeft(lft)(acc1)(f)
            foldLeft(rgt)(acc2)(f)
        }
    }
}

listFoldable.foldLeft(l)(0)((x:Int,y:Int) => x + y) // 6
btreeFoldable.foldLeft(t)(0)((x:Int,y:Int) => x + y) // 6
```

Handling Error

Recall

```
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)
}
def eval(e:MathExp):Int = e match {
    case MathExp.Plus(e1, e2)  => eval(e1) + eval(e2)
    case MathExp.Minus(e1, e2) => eval(e1) - eval(e2)
    case MathExp.Mult(e1, e2)  => eval(e1) * eval(e2)
    case MathExp.Div(e1, e2)   => eval(e1) / eval(e2)
    case MathExp.Const(i)      => i
}
```

Handling Error

Recall

```
import MathExp.*
eval(Div(Const(1), Minus(Const(2), Const(2))))
```

yields a run-time error

```
java.lang.ArithmeticException: / by zero
  at rs$line$2$.eval(rs$line$2:5)
  ... 41 elided
```

Try-catch?

```
try {
    import MathExp.*
    eval(Div(Const(1), Minus(Const(2), Const(2))))
}
catch {
    case e:java.lang.ArithmeticException =>
        println("handinging div by zero")
}
```

- Not preferred
  - it is hard to track the unhandled exceptions, (in particular with the presence of Java unchecked exceptions.)
  - i.e. more test cases required

Option Type

- Scala comes with Option type builtin

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

Error Handling with Option Type

```
def eval(e:MathExp):Option[Int] = e match {
    case MathExp.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 omitted for Minus and Mult
    case MathExp.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 MathExp.Const(i)      => Some(i)
}

eval(Div(Const(1), Minus(Const(2), Const(2))))
// yields None
```

Error Handling with Either Type

- Handle Errors with `Option` Type has one remaining issue.
  - i.e. there is no information with the error in `None`
- Let's consider another buildin type

```
enum Either[+A, +B] {
    case Left(v:A)
    case Right(v:B)
}

type ErrMsg = String
```

Error Handling with Either Type

```
def eval(e: MathExp): Either[ErrMsg,Int] = e match {
    case MathExp.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 omitted for Minus and Mult
    case MathExp.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 MathExp.Const(i) => Right(i)
}
eval(Div(Const(1), Minus(Const(2), Const(2))))
// yields Left(div by zero caused by Div(Const(1),Minus(Const(2),Const(2))))
```

Summary

In this lesson, we have discussed

- how to develop generic programming style code using `Functor` type class.
- how to make use of `Option` and `Either` to handle and manipulate errors and exceptions.