50.054 Parametric Polymorphism and Adhoc Polymorphism

ISTD, SUTD

Learning Outcomes

- Develop parametrically polymorphic Scala code using Generic, Algebraic Datatype
- Safely mix parametric polymoprhism with adhoc polymoprhism (overloading) using type classes

Recap

Recall that during the exercise, we want to have a span function, let's try to implement it (though there exists a builtin definition)

```
def span[A](l:List[A], chk:A => Boolean):(List[A], List[A]) = l match {
    case Nil => (Nil, Nil)
    case (x::xs) => ??? // what should go here?
}
```

Can span be written using tailrec?

Currying

Imagine

```
def sum(x:Int, y:Int):Int = x + y
val result = sum(1,2)
```

Why not?

```
def sum(x:Int)(y:Int):Int = x + y
val result = sum(1)(2)
```

The second version is called the curry form of the first one.

Why Currying?

It promotes better resusability.

```
def plusone(x:Int):Int = sum(1)(x) // or
val plusone = sum(1)
List(1,2,3).map(plusone) // yields List(2,3,4)
```

What about?

Recall in lambda calculus, $\lambda x.\lambda y.x + y$.

Why not?

```
def sum(x:Int) = (y:Int) => x + y
```

What is the returned type of `sum`?

How about?

```
val sum = (x:Int) => (y:Int) => x + y
```

What is the returned type of `sum`?

Function Composition

Functions and methods in Scala are *composable*.

In math, let g and f be functions, then

$$(g \circ f)(x) \equiv g(f(x))$$

In Scala, let `g` and `f` be functions (or methods), then

```
g.compose(f) // or
g compose f
```

is equivalent to

```
x => g(f(x))
```

Function Composition

For example

```
def f(x:Int):Int = 2 * x + 3
def g(x:Int):Int = x * x

assert((g.compose(f))(2) == g(f(2)))
```

Why function compostion? Promote code reusability!

Generics Again

Recall

```
def reverse(l:List[Int]):List[Int] = l match {
    case Nil => Nil
    case (hd::tl) => reverse(tl) ++ List(hd)
}
```

vs

```
def reverse[A](l:List[A]):List[A] = l match {
    case Nil => Nil
    case (hd::tl) => reverse(tl) ++ List(hd)
}
```

- We only need to put `[A]` after a method/function name if `A` is a generic type / polymoprhic type.
- We don't need to put a monomorphic type in the `[ ]` after the function/method name.

Generic / Polymorphic Algebraic Data Type

Suppose we want to redefine the List datatype.

```
enum MyList[A] {
    case Nil // type error
    case Cons(x:A, xs:MyList[A])
}

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

-- Error: ------------------------------------------------------------
2 |    case Nil
  |    ^^^^^^^^
  |    cannot determine type argument for enum parent class MyList,
  |    type parameter type A is invariant
1 error found
```

Subtyping inside Enum

The desugared version of `MyList`

```
enum MyList[A] {
    case Nil extends MyList[Nothing] // type error
    case Cons(x:A, xs:MyList[A]) extends MyList[A]
}
```

- `Nil`'s case can't be a subtype of `MyList[A]`, since `A` is not used in `Nil`.
- The best we can infer is `Nothing` which is the least type in Scala subtyping hierachy.
- The above is valid only if `MyList[Nothing] extends MyList[A]`.
- Only if `A` is a covariant of `MyList[A]`.

What is Invariant, What is Covariant?

- Given a type constructor `TC[_]`, a type parameter `A` is a covariant of `TC[_]` iff
  - for all types `S` and `U` such that `S extends U` implies `TC[S] extends TC[U]`.
  - i.e. Subtyping relation is maintained via type application `TC[_]`.
- Given a type constructor `TC[_]`, a type parameter `A` is a contravariant of `TC[_]` iff
  - for all types `S` and `U` such that `S extends U` implies `TC[U] extends TC[S]`.
  - i.e. Subtyping relation is reversed via type application `TC[_]`.
- Given a type constructor `TC[_]`, a type parameter `A` is a invariant of `TC[_]` iff
  - `A` is neither covariant or contravariant.

Back to the Example

```
enum MyList[+A] {
    case Nil extends MyList[Nothing] // type error is fixed.
    case Cons(x:A, xs:MyList[A]) extends MyList[A]
}

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

- The + sign preceding `A` declares that `A` is covariant of `MyList[_]`, i.e.
    - for all types `S` and `U` such that `S extends U` implies `MyList[S] extends MyList[U]`.
    - since `Nothing extends A` for all `A`, hence `MyList[Nothing] extends MyList[A]`.
    - the `extends` clauses are optional.
- A - sign declares an contravariant.

Function overloading

Function overloading is to use a same function name for multiple implementations in different type context.

```
def toJS(v:Int):String = v.toString
def toJS(v:String):String = s"'${v}'"
def toJS(v:Boolean):String = v.toString
```

- `s"'${v}'"` is a string interpolation, just like f-string in python.
- given `v = "hello"`, `s"'${v}'"` yields `"'hello'"`

Function overloading - issue # 1

```
enum Contact {
    case Email(e:String)
    case Phone(ph:String)
}
import Contact.*
def toJS(c:Contact):String = c match {
    case Email(e) => s"'email': ${toJS(e)}"  // error
    case Phone(ph) => s"'phone': ${toJS(ph)}"  // error
}
```

```
-- [E007] Type Mismatch Error: -------------------------------------------------
3 |    case Email(e) => s"'email': ${toJS(e)}"
  |                                        ^
  |                                        Found:    (e : String)
  |                                        Required: Contact
  |
  | longer explanation available when compiling with `-explain`
```

toJS refers to toJS(c:Contact) or toJS(v:String)?

Scala Case class

```
case class Person(name:String, contacts:List[Contact])
case class Team(members:List[Person])
```

- `case class` defines a pattern-matchable single alternative algebraic data type.

```
import Contact.*
val myTeam = Team( List(
    Person("kenny", List(Email("kenny_lu@sutd.edu.sg"))),
    Person("simon", List(Email("simon_perrault@sutd.edu.sg")))
))
```

Function Overloading - issue # 2

```scala
def toJS(p:Person):String = p match {
    case Person(name, contacts) =>
        s"'person':{ 'name':${toJS(name)}, 'contacts':${toJS(contacts)} }"
}
def toJS(cs:List[Contact]):String = { // code duplicate
    val j = cs.map(c=>toJS(c)).mkString(",")
    s"[${j}]"
}
def toJS(t:Team):String = t match {
    case Team(members) => s"'team':{ 'members':${toJS(members)} }"
}
def toJS(ps:List[Person]):String = {  // code duplicate
    val j = ps.map(p=>toJS(p)).mkString(",")
    s"[${j}]"
}
```

Type Class

- Type Class is supported by a few FP languages such as Scala and Haskell
- Safely mix parametric (generics) and adhoc polymoprhism (overloading)
- In Scala, a type class is defined as a polymorphic trait.

```scala
trait JS[A] {
    def toJS(v:A):String
}
```

Type Class instances

An instance of a type class defines an implementation of a type class for a specific type.

```
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
}
```

- `given` defines a type class instance in Scala 3.
- `toJSInt`, `toJSString` and `toJSBoolean` are the instance names.
  - In Haskell, type class instances are unamed.

Type Class Instances

```scala
def f(x:Boolean)(i:JS[Boolean]):String = i.toJS(x)
f(true)(toJSBoolean) // "true"
```

- Scala resolves type class instances by the type context, not by name (mostly).

```scala
def g(x:Boolean)(using i:JS[Boolean]):String = i.toJS(x)
g(true) // "true"
```

- `using` declares an implicit argument which will be resolved as a type class instance based on the given type.
- `using` modifies the single currying argument.

Type Class Instances

- Type class instances can be generic
- Type class instances will be synthesised by the Scala's type class resolution

```
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]"
```

- `i:JS[List[Boolean]]` is synthesised by combining `toJSList` insance with and `toJSBoolean` instance

Type Class Instance the full example

```
given toJSContact(using jsstr:JS[String]):JS[Contact] = new JS[Contact] {
    import Contact.*
    def toJS(c:Contact):String = c match {
        case Email(e) => s"'email': ${jsstr.toJS(e)}"
        case Phone(ph) => s"'phone': ${jsstr.toJS(ph)}"
    }
}
given toJSPerson(using jsstr:JS[String], jsl:JS[List[Contact]]):JS[Person] = new JS[Person] {
    def toJS(p:Person):String = p match {
        case Person(name, contacts) =>
            s"'person':{ 'name':${jsstr.toJS(name)}, 'contacts':${jsl.toJS(contacts)} }"
    }
}
given toJSTeam(using jsl:JS[List[Person]]):JS[Team] = new JS[Team] {
    def toJS(t:Team):String = t match {
        case Team(members) => s"'team':{ 'members':${jsl.toJS(members)} }"
    }
}
toJSTeam.toJS(myTeam)
```

yields

```
'team':{ 'members':['person':{ 'name':'kenny','contacts':['email': 'kenny_lu@sutd.edu.sg'] },
         'person':{ 'name':'simon', 'contacts':['email': 'simon_perrault@sutd.edu.sg'] }] }
```

Quick Summary

We have discussed

1. Currying
2. Function Composition
3. Generic (Polymorphic) Algebraic Data Types
4. Type Classes