Advanced typing

MyList: Invariant

```
sealed abstract class MyList[A] {
  def head: A
  def tail: MyList[A]

  def :::(other: MyList[A]): MyList[A]

  def map[B](f: A => B): MyList[B]

  def flatMap[B](f: A => MyList[B]): MyList[B]
}
```

MyList: Invariant

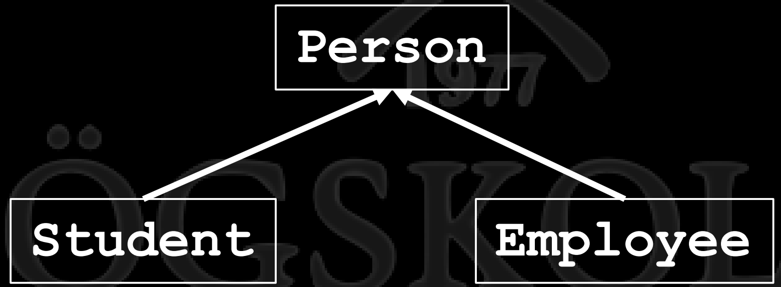
```
case class NonEmpty[A](head: A, tail: MyList[A]) extends MyList[A] {
  def :::(other: MyList[A]): MyList[A] = other match {
    case Empty() => this
    case NonEmpty(h, t) => NonEmpty[A](h, :::(t))
  }

  def map[B](f: A => B): MyList[B] =
    NonEmpty[B](f(head), tail.map[B](f))

  def flatMap[B](f: A => MyList[B]): MyList[B] =
    f(head) ::: tail.flatMap[B](f)
}
```

MyList: Invariant

```scala
case class Empty[A]() extends MyList[A] {
  def head: A = throw new Exception("Empty has no head")
  def tail: MyList[A] = throw new Exception("Empty has no tail")

  def :::(e: MyList[A]): MyList[A] = e
  def map[B](f: A => B): MyList[B] = Empty[B]()
  def flatMap[B](f: A => MyList[B]): MyList[B] = Empty[B]()
}
```

Example hierarchy

```
class Person(val name: String)

class Student(name: String, val id: Int) extends Person(name)

class Employee(name: String, val salary: Double) extends Person(name)
```



MyList: Invariant

```
val students: MyList[Student] =              // = MyList(S1, S2)
  NonEmpty[Student](new Student ("S1", 1),
    NonEmpty[Student](new Student ("S2", 2),
      Empty[Student]()))

val p: Person = students.head ✓
val persons: MyList[Person] = students  ✗
val persons: MyList[Person] = Empty[Nothing]() ✗
```

```
Note: Student <: Person, but class MyList is invariant in type A.
You may wish to define A as +A instead. (SLS 4.5)

Note: Nothing <: Person, but class MyList is invariant in type A.
You may wish to define A as +A instead. (SLS 4.5)
```

MyList: Invariant

```
val students: MyList[Student] =           // = MyList(S1, S2)
  NonEmpty[Student](new Student ("S1", 1),
    NonEmpty[Student](new Student ("S2", 2),
      Empty[Student]()))

val employees: MyList[Employee] =         // = MyList(E1, E2)
  NonEmpty[Employee](new Employee ("E1", 10000),
    NonEmpty[Employee](new Employee ("E2", 20000),
      Empty[Employee]()))

students ::: employees ✗
```

```
error: type mismatch;
 found   : MyList[Student]
 required: MyList[Employee]
```

MyList: Covariant

```
sealed abstract class MyList[+A] {
  def head: A
  def tail: MyList[A]

  def :::(other: MyList[A]): MyList[A]    // ERROR

  def map[B](f: A => B): MyList[B]

  def flatMap[B](f: A => MyList[B]): MyList[B]
}
```

```
covariant type A occurs in contravariant position in type MyList[A] of
value other
```

MyList: Covariant

```
sealed abstract class MyList[+A] {
  def head: A
  def tail: MyList[A]

  def :::[B >: A](other: MyList[B]): MyList[B]

  def map[B](f: A => B): MyList[B]

  def flatMap[B](f: A => MyList[B]): MyList[B]
}
```

B should be equal to A, or a super class of A

MyList: Covariant

```scala
object Empty extends MyList[Nothing] {
  def head: Nothing = throw new Exception("Empty has no head")
  def tail: MyList[Nothing] = throw new Exception("Empty has no tail")

  def :::[B >: Nothing](e: MyList[B]): MyList[B] = e
  def map[B](f: Nothing => B): MyList[B] = this
  def flatMap[B](f: A => MyList[B]): MyList[B] = this
}
```

MyList: Covariant

```
val students: MyList[Student] =           // = MyList(S1, S2)
  NonEmpty[Student](new Student ("S1", 1),
    NonEmpty[Student](new Student ("S2", 2),
      Empty))

val p: Person = students.head  ✓
val persons: MyList[Person] = students ✓
val persons: MyList[Person] = Empty ✓
```

MyList: Covariant

```
val students: MyList[Student] =              // = MyList(S1, S2)
  NonEmpty[Student](new Student ("S1", 1),
    NonEmpty[Student](new Student ("S2", 2),
      Empty))

val employees: MyList[Employee] =            // = MyList(E1, E2)
  NonEmpty[Employee](new Employee ("E1", 10000),
    NonEmpty[Employee](new Employee ("E2", 20000),
      Empty))
```

```
students :::[Person] employees   ✓

students :::[Student] employees  ✗
```

```
type arguments [Student] do not conform to method :::'s type
parameter bounds [B >: Employee]
```

Contravariant

```
trait Stringify[A] {
  def apply(x: A): String
}

val personName = new Stringify[Person] {
  def apply(x: Person): String = x.name
}

personName(new Person("P1"))  // = P1
val studentName: Stringify[Student] = personName ✗
```

```
Note: Person >: Student, but trait Stringify is invariant in type A.
You may wish to define A as -A instead. (SLS 4.5)
```

Contravariant

```
trait Stringify[-A] {
  def apply(x: A): String
}

val personName = new Stringify[Person] {
  def apply(x: Person): String = x.name
}

personName(new Person("P1"))  // = P1
val studentName: Stringify[Student] = personName ✓
```

Contravariant

```
trait Callable[A, B] {   // similar to Function1[A, B] in Scala
  def apply(x: A): B
}

sealed abstract class MyList[+A] {...
  def map[B](f: Callable[A, B]): MyList[B]  ✗
  ...
}
```

```
error: covariant type A occurs in invariant position in type
Callable[A,B] of value f
```

Contravariant

```
trait Callable[-A, B] {   // similar to Function1[A, B] in Scala
  def apply(x: A): B
}

sealed abstract class MyList[+A] {...
  def map[B](f: Callable[A, B]): MyList[B]  ✓
  ...
}
val employ = new Callable[Person, Employee] {
  def apply(x: Person): Employee = new Employee(x.name, 10000)
}

val ps: MyList[Student] = NonEmpty(new Student("S1", 1), Empty)


  def map[B](f: Callable[Student, B]): MyList[B] ...
```

Contravariant

```
trait Callable[-A, B] {  // similar to Function1[A, B] in Scala
  def apply(x: A): B
}

sealed abstract class MyList[+A] {...
  def map[B](f: Callable[A, B]): MyList[B]
  ...
}
val employ = new Callable[Person, Employee] {
  def apply(x: Person): Employee = new Employee(x.name, 10000)
}

val ps: MyList[Student] = NonEmpty(new Student("S1", 1), Empty)
ps.map[Employee](employ)          // = MyList[Employee]


  def map[Employee](f: Callable[Student, Employee]): MyList[Employee] ...
```

Implicit parameters

```
case class Context(sideDish: String)
val myContext = Context("smashed potatoes")


def makeDish(main: String, c: Context): String =
  s"$main with ${c.sideDish}"

makeDish("Meatballs", myContext)    // = Meatballs with smashed potatoes
makeDish("Meatballs", Context("rice")) // = Meatballs with rice
```

Implicit parameters

```
case class Context(sideDish: String)
implicit val myContext = Context("smashed potatoes")


def makeDish(main: String)(implicit c: Context): String =
  s"$main with ${c.sideDish}"

makeDish("Meatballs")               // = Meatballs with smashed potatoes
makeDish("Meatballs")(Context("rice")) // = Meatballs with rice
implicit val myContext = Context("beans")
makeDish("Meatballs")               // = Meatballs with beans
```

Implicit parameters

```
sealed abstract class MyList[+A] {...
  def isSorted[B >: A](o: Ordering[B]): Boolean
}

case class NonEmpty[A](head: A, tail: MyList[A]) extends MyList[A] {...
  def isSorted[B >: A](o: Ordering[B]): Boolean = tail match {
    case Empty => true
    case NonEmpty(h, t) => o.gteq(h, head) && tail.isSorted(o)
  }
}

object Empty extends MyList[Nothing] {...
  def isSorted[B >: A](o: Ordering[B]): Boolean = true
}
```

Implicit parameters

```
val intOrdering = new Ordering[Int] {
  def compare(a: Int, b: Int): Int = a - b
}
MyList(1, 2, 3).isSorted(intOrdering)     // = true
MyList(1, 3, 2).isSorted(intOrdering)     // = false
```

Implicit parameters

```
sealed abstract class MyList[+A] {...
  def isSorted[B >: A](implicit o: Ordering[B]): Boolean
}

case class NonEmpty[A](head: A, tail: MyList[A]) extends MyList[A] {...
  def isSorted[B >: A](implicit o: Ordering[B]): Boolean = tail match {
    case Empty => true
    case NonEmpty(h, t) => o.gteq(h, head) && tail.isSorted(o)
  }
}

object Empty extends MyList[Nothing] {...
  def isSorted[B >: A](implicit o: Ordering[B]): Boolean = true
}
```

Implicit parameters

```
~~val intOrdering = new Ordering[Int] {~~
~~  def compare(a: Int, b: Int): Int = a - b~~
}
MyList(1, 2, 3).isSorted            // = true
MyList("a", "c", "b").isSorted      // = false
students.isSorted // = ERROR ✗
```

```
error: No implicit Ordering defined for Student.
```

Implicit parameters

```
implicit val studentOrdering = new Ordering[Student] {
  def compare(a: Student, b: Student): Int = if (a.name > b.name) 1 else 0
}
MyList(1, 2, 3).isSorted            // = true
MyList("a", "c", "b").isSorted      // = false
students.isSorted // = true
```

Implicit conversions

```
def toMyList[A](l: List[A]): MyList[A] =
  l.foldRight[MyList[A]](Empty){
    case (x, acc) => NonEmpty(x, acc)
  }
```

```
toMyList(List(1, 2, 3))    // = NonEmpty(1,NonEmpty(2,NonEmpty(3,Empty)))
```

Implicit conversions

```
implicit class WrappedList[A](l: List[A]) {
  def toMyList: MyList[A] =
    l.foldRight[MyList[A]](Empty){
      case (x, acc) => NonEmpty(x, acc)
    }
}

List(1, 2, 3).toMyList     // = NonEmpty(1,NonEmpty(2,NonEmpty(3,Empty)))
"2".toDouble               // = 2.0
"abcd".toList              // = List(a, b, c, d)
```