58

Types, Parametric types, Kinds, Variance, etc.

Statically Typed Languages

(repeated from week 1)

- When you write a program (in most languages), you have to let the compiler know five things about any object:
  - its identifier (how do we refer to it);
  - what *type* of thing it is;
  - what its *value* is;
  - whether that value can be changed after initialization;
  - when it is evaluated.
- Scala is <u>strictly typed</u>. And, as we shall see, types themselves have “kinds”

“Container” types

- There are many container types: types that “wrap” one or more other objects or one or more types.
  - some of these are in the collections framework:
    - List, Map, Tree
    - http://docs.scala-lang.org/overviews/collections/introduction.html
  - also Tuple(N), Option, Either, Future, Try…
- Just about everything you ever do will use one or more of these so it makes sense to understand them!

List of Ints

```scala
trait List {
  override def toString = {
    @tailrec def tos(as: List[Int], r: StringBuffer): CharSequence = as match {
      case Nil => r
      case Cons(hd, tl) => tos(tl, r.append((if (r.length > 1) ", " else "") + s""""$hd""""))
    }
    tos(this, new StringBuffer("(")) + ")"
  }
  def ++(x: List[Int]): List[Int] = this match {
    case Nil => x
    case Cons(hd, tl) => Cons(hd, tl ++ x)
  }
  def length: Int = {
    @tailrec def len(as: List[Int], x: Int): Int = as match {
      case Nil => x
      case Cons(hd, tl) => len(tl, x + 1)
    }
    len(this, 0)
  }
  def isEmpty: Boolean = this match {
    case Nil => true;
    case _ => false
  }
  def head: Int = this match {case Nil => throw new Exception("logic error"); case Cons(hd,tl) => hd}
  def tail: List[Int] = this match {
    case Nil => Nil;
    case Cons(hd, tl) => tl
  }
  def headOption: Option[Int] = this match {
    case Nil => None;
    case Cons(hd, tl) => Some(hd)
  }
  def apply(idx: Int): Int = {
    @tailrec def internal(as: List[Int], x: Int): Int = as match {
      case Nil => throw new IndexOutOfBoundsException(s"index out of bounds: $idx")
      case Cons(hd, tl) => if (x == 0) hd else internal(tl, x - 1)
    }
    if (idx >= 0) internal(this, idx) else throw new IndexOutOfBoundsException(s"index out of bounds: $idx")
  }
  def filter(f: Int => Boolean): List[Int] = this match {
    case Cons(hd, tl) => val ftl = tl.filter(f); if (f(hd)) Cons(hd, ftl) else ftl
    case Nil => Nil
  }
  def find(f: Int => Boolean): Option[Int] = this match {
    case Cons(hd, tl) => if (f(hd)) Some(hd) else tl.find(f)
    case Nil => None
  }
  def map[B](f: Int => B): List[B] = this match {
    case Cons(hd, tl) => Cons(f(hd), tl.map(f));
    case Nil => List[B]()
  }
  def flatMap[B](f: Int => List[B]): List[B] = this match {
    case Cons(hd, tl) => f(hd) ++ tl.flatMap(f)
    case Nil => List[B]()
  }
  def foreach(f: Int => Unit): Unit = this match {
    case Cons(hd, tl) => f(hd); tl.foreach(f)
    case Nil =>
  }
  def count(f: Counter[Int]): List[Int] = this map f
  def foldLeft[B](z: B)(f: (B, Int) => B): B = this match {
    case Nil => z
    case Cons(hd, tl) => tl.foldLeft(f(z, hd))(f)
  }
}
case object Nil extends List[Nothing]
```

Parametric List - part one

- But surely we don't want to re-invent all of that for *String*, *Double* and every other conceivable type? — of course not! in other words a *List* is a *List* regardless of what it is a list *of*. Let's make it *List[A]*.

```
trait List[A] extends Int=>A {
  def length: Int
  def isEmpty: Boolean = this match {case Nil => true; case _ => false }
  def head: List[A] = this match {case Nil => throw new Exception("logic error"); case Cons(hd,tl) => tl}
  def apply(idx: Int): A = {
    @tailrec def internal(as: List[A], x: Int): A = as match {
      case Nil => throw new IndexOutOfBoundsException(s"index out of bounds: $idx")
      case Cons(hd,tl) => if (x==0) hd else internal(tl,x-1)
    }
    if (idx>=0) internal(this,idx) else throw new IndexOutOfBoundsException(s"index out of bounds: $idx")
  }
  def ++(x: List[A]): List[A] = this match {
    case Nil => x
    case Cons(hd, tl) => Cons(hd, tl ++ x)
  }
  // etc.
}
case object Nil extends List[Nothing] {
  def length: Int = 0
  override def toString = "()"
}
case class Cons[A] (head: A, tail: List[A]) extends List[A] {
  def equals(z: List[A]): Boolean = tail match {
    case Nil => false
    case Cons(x,xs) => z match {
      case Cons(y,ys) => x==y && xs==ys
    }
  }
  def length: Int = this match {
    case Cons(h,t) => 1 + t.length
  }
  override def toString = {
    @tailrec def tos(as: List[A], r: StringBuffer): CharSequence = as match {
      case Nil => r
      case Cons(hd,tl) => tos(tl,r.append((if (r.length>1) ", " else "") + s""""$hd""""))
    }
    tos(this,new StringBuffer("("))+")"
  }
}
```

For historical(?) reasons, *List* extends *Int=>A* instead of *Int=>Option[A]* so that (in my opinion) *get* and *apply* are backwards. See my [blog](#).

Invariance

- Suppose we have a type hierarchy where *Dog* is a subtype of *Animal,* and *Chihuahua* is a subtype of *Dog*.
- What about *List[Dog]*? Is this a subtype of *List[Animal]*? That's to say, if a method/function takes a parameter *List[Animal]*, can we pass it a *List[Dog]* and all will be well?
- Well, for our *List*, this is NOT OK, because *List[A]* is **invariant** in A*.
- *ListBuffer[A]* and *Array[A]* are ***invariant***. so you <u>cannot</u> pass an instance of *Array[Dog]* where it expects an *Array[Animal].*

* but don't worry, the real list is not invariant

Parametric List - part two

- In part one, we saw a possible set of operations on *List[A].* Very much like an *Array*, in fact. Now, let's think about a *covariant* list.

```scala
package edu.neu.coe.scala
package list

trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A] (head: A, tail: List[A]) extends List[A]

object List {
  def sum(ints: List[Int]): Int = ints match {
    case Nil => 0
    case Cons(x,xs) => x + sum(xs)
  }

  def apply[A](as: A*): List[A] =
    if (as.isEmpty) Nil
    else Cons(as.head, apply(as.tail: _*))
}
```

By preceding the polymorphic type by "+" we say that it is *covariant*.

as before, returns the sum but works only for *Int* (or, possibly, any type *A* that also defines the "+" operator).

Covariance

- Suppose we again have a type hierarchy where *Dog* is a subtype of *Animal,* and *Chihuahua* is a subtype of *Dog*.
- What about *List[Dog]*? Is this a subtype of *List[Animal]*? That's to say, if a method/function takes a parameter *List[Animal]*, we really ought to be able to pass it a *List[Dog].*
- Well, for *List*, this is OK, because *List[+A]* extends *Seq[A]*. That's to say: *List* is ***covariant*** on *A*.
- Technically, *List[+A]* is a **type function** that takes a class *A* and creates a list class such that if *A* is a subtype of *B*, then *List[A]* will be a subtype of *List[B]*.

List — actual definition

```
sealed abstract class List[+A] extends AbstractSeq[A] with LinearSeq[A]
with Product with GenericTraversableTemplate[A, List] with
LinearSeqOptimized[A, List[A]] with java.io.Serializable
```

```
/** Adds an element at the beginning of this list.
 *  @param x the element to prepend.
 *  @return  a list which contains `x` as first element and
 *           which continues with this list.
 *
 *  @usecase def ::(x: A): List[A]
 *    @inheritdoc
 *
 *    Example:
 *    {{{1 :: List(2, 3) = List(2, 3).::(1) = List(1, 2, 3)}}}
 */
def ::[B >: A] (x: B): List[B] = new scala.collection.immutable.::(x, this)
```

Values of type *A* can only appear in *covariant position*, that's to say as the result of a method (not as a parameter). So, here we must create a new parametric type *B* which is a super-class of *A*. Parameters are in *contravariant position.*

Collections: detail from API

- `trait Seq[+A] extends Iterable[A] with collection.Seq[A] with GenericTraversableTemplate[A, Seq] with SeqLike[A, Seq[A]] with Parallelizable[A, ParSeq[A]]`
- `trait Map[A, +B] extends Iterable[(A, B)] with GenMap[A, B] with MapLike[A, B, Map[A, B]]`
- `trait GenTraversable[+A] extends GenTraversableLike[A, GenTraversable[A]] with GenTraversableOnce[A] with GenericTraversableTemplate[A, GenTraversable]`
- `trait Iterable[+A] extends Traversable[A] with GenIterable[A] with GenericTraversableTemplate[A, Iterable] with IterableLike[A, Iterable[A]]`
- This looks pretty weird, right? But, unlike in Java, every different behavior (method) is defined in a separate trait.
- In Scala, there is no wildcard (?) as in Java generics, not as such (but there is "_" which we will talk about soon)

Expressions and Types

- *val y: Y = ???*
- *val x: X = y*

X >: Y
X: a
Y: a, b

- In the second variable declaration, *y* is of type *Y* and, according to the **Liskov substitution principle**, the type *Y* can be any sub-type of *X* (or X itself).
- Why? Because all properties of *X* (*a*) must be defined by *y*, otherwise *x* would be somehow incomplete. Because *Y* must be a sub-type of *X* (or *X*), it follows that *Y* supports all properties of *X*, as well as some that *X* does not support (*b*).

Functions and Types

- *def f(z: Z): Y = z*
- *val z: Z = ???*
- *val x: X = f(z)*

X >: Y >: Z

X: a

Y: a, b

Z: a, b, c

- In the method declaration, *z* is of type *Z* and, according to the **Liskov substitution principle**, the type *Y* can be any super-type of *Z*.
- Why? Because the properties promised the caller of *f*, i.e. *a* and *b*, must be supported by *z*. But the type *Z* might well have other properties (*c*) not supported by *Y* (which are of no interest to the caller).
- As before, *Y* can be any sub-type of *X*.

Liskov Substitution Principle

- The subtype requirement:
  - Let *ϕ(x)* be a property provable about objects *x* of type *T*. Then *ϕ(y)* should be true for objects *y* of type *S* where *S* is a subtype of *T*.
  - Example:

```
trait Vertebrate extends Animal {
  def vertebra: Int
}

trait Mammal extends Vertebrate {
  def sound: String
}

trait Dog extends Mammal {
  def sound = woof
  def woof: String
}

trait Cat extends Mammal {
  def sound = miao
  def miao: String
}
```

  - *sound* is a provable property of an instance of *Mammal*. Therefore, since *Dog* is a subtype of *Mammal*, therefore *sound* must be a provable property of an instance of *Dog* or *Cat*.
  - *vertebra* is a provable property of the type *Vertebrate*. Therefore since *Mammal* is a subtype of *Vertebrate*, therefore *vertebra* must be a provable property of a *Mammal*; and so, *vertebra* must be a provable property of a *Dog* or *Cat*.

Pets

```scala
trait Base { val name: String }
trait Organelle
trait Organism { def genotype: Seq[Base] }
trait Eukaryote extends Organism { def organelles: Seq[Organelle] }
trait Animal extends Eukaryote { def female: Boolean; def organelles: Seq[Organelle] = Nil }
trait Vertebrate extends Animal { def vertebra: Int; def sound: Sound }
trait Sound { def sound: Seq[Byte] }
trait Voice extends Sound with (() => String) { def sound: Seq[Byte] = apply().getBytes }
trait Bear extends Mammal { def sound: Sound = Growl; def growl: String }
case object Woof extends Voice { def apply(): String = “Woof!" }
case object Growl extends Sound { def sound: Seq[Byte] = “growl".getBytes }

trait Mammal extends Vertebrate {
  def vertebra: Int = 33
}
trait Pet extends Animal {
  def name: String
}
trait Dog extends Mammal with Pet {
  def sound: Sound = Woof
  def genotype: Seq[Base] = Nil
}
case class Chihuahua(name: String, female: Boolean, color: String) extends Dog
case class Pets[+X <: Pet with Mammal, -Y <: Sound](xs: Seq[X]) {
  def identify(s: String): X = xs find (_.name == s) get
  def sounders(y: Y): Seq[X] = xs filter (_.sound == y)
}
```

Types and variance in practice

```
object Pets extends App {
  def create[X <: Pet with Mammal, Y <: Sound](xs: X*): Pets[X, Y] = Pets(xs)
  // This method takes a Chihuahua and returns it as a Dog which works because Chihuahua is a
subtype of Dog.
  // All of the required properties of Dog are specified by any instance of Chihuahua
  def asDog(x: Chihuahua): Dog = x
  val bentley = Chihuahua("Bentley", female = false, "black")
  val gingerSnap = Chihuahua("GingerSnap", female = true, "ginger")
  val ralphie = Chihuahua("Ralphie", female = true, "white")
  // List[Chihuahua] is a subtype of Seq[Dog] because A is covariant in Seq[A] and because
List is a subtype of Seq
  val dogs: Seq[Dog] = List(bentley, gingerSnap, ralphie)
  // Pets[Chihuahua, Sound] is a subtype of Pets[Dog,Voice] because Chihuahua is a subtype of
Dog (and covariant)
  // while Sound is a supertype of Voice (and contravariant)
  val pets: Pets[Dog, Voice] = Pets.create[Chihuahua, Sound](bentley, gingerSnap, ralphie)
  // Dog is a subtype of Mammal: all of the required properties of Mammal are specified by
any instance of Dog
  val m: Mammal = asDog(bentley)
  val ps = pets.sounders(Woof)
}
```

Variance explained

- First, let's define an arbitrary type with both covariant and contravariant parametric types:
  - *X[+S,-T]*
  - For example, we could declare:
    - *type* X[+S, -T] = (T) => S
- Now, what's the relationship between any two *X* types?
- *X1[S1,T1]* is a sub-type of *X2[S2,T2]* provided that:
  - *X1* is a <u>sub-type</u> of *X2*
  - *S1* is a <u>sub-type</u> of *S2*
  - *T1* is a <u>super-type</u> of *S2*

Summarizing variance

- In the following, *S* is a sub-type of *T*
- Invariance, e.g. for *Array*:
  - if we expect an *Array[T]*, we **cannot** give an Array[S] because *Array[S]* is **not** a sub-type of *Array[T]*
- Covariance, e.g. for *abstract class List[+A]*:
  - if we expect a *List[T]*, we **can** give a *List[S]* because *List[S]* **is** a sub-type of *List[T]*
- Contravariance, e.g. for *Function1[-T,+R]*:
  - if we expect a *T=>Unit*, we **cannot** give a *S=>Unit,* but…
  - if we expect a *S=>Unit*, we **can** give a *T=>Unit* because *T=>Unit* is a sub-type of *S=>Unit*.

More on variance

```
trait Function1[-T1, +R] extends AnyRef
```

- “-“ defines *T1* to be contra-variant
- “+” defines *R* to be co-variant
- That's to say if *T2* is a super-type of *T1* and if *S* is a sub-type of *R* then *Function1[T2,S]* is a sub-type of *Function1[T1,R]*
- Let's say you need a function *f* which takes an object of type *T1* and transforms it into an object of type *R*. If you can find a function *g* that takes an object of type *T2* and transforms it into an object of type *S*, then you can say *f* = *g*, that's to say *g* is a sub-type of *f* (that’s required for assignment).
- For example, we have an *x: CharSequence* and we want to turn it into a different *y: CharSequence* by writing *val y = f(x)*.
- We have a function *g* which takes an *Any* and turns it into a *String*:
  - `val g: Any=>String = _.toString`
- We can use *g* for *f*, that's to say *val f = g* (where *g* is a sub-type of *f*). This works because *T2=Any* (a supertype of *T1*) and *S=String* (a subtype of *R*)

Help on variance

- There are some good resources on the internet to help:
  - Variances (from Tour of Scala at scala-lang.org)
  - Covariance and contravariance in Scala (blog)
  - Covariance and contravariance in Scala (another blog—at Atlassian)
  - Scala by Example (section 8.2—at scala-lang.org)
  - Scala's pesky contravariant position problem (my blog)