Continuing

Yinghui Wang,Mehrdad Alemzadeh

CAS 706

November 4, 2010

Covariant Subtyping

- Should stack[A] be stack[B]'s subtype if A is B's subtype?

Covariant Subtyping

- Should stack[A] be stack[B]'s subtype if A is B's subtype?
- Generic type in Scala non-Covariant by default

Covariant Subtyping

- Should stack[A] be stack[B]'s subtype if A is B's subtype?
- Generic type in Scala non-Covariant by default
- Class stack[+A](co) or class stack[-A](contra)

Covariant Subtyping

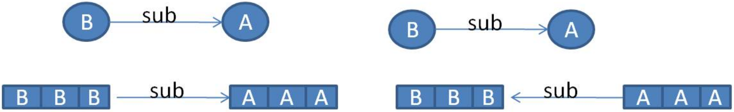
- Should stack[A] be stack[B]'s subtype if A is B's subtype?
- Generic type in Scala non-Covariant by default
- Class stack[+A](co) or class stack[-A](contra)
- 

Figure: subtyping

lower bound and upper bound

- Covariant type parameters of a class are only allowed appear in positions:
  - types of values in the class
  - the result types of methods in the class
  - type arguments to other covariant types

lower bound and upper bound

- Covariant type parameters of a class are only allowed appear in positions:
  - types of values in the class
  - the result types of methods in the class
  - type arguments to other covariant types
- So:

```
class Array[+A] {
    def apply(index: Int): A
    def update(index: Int, elem: A)
```

lower bound and upper bound

- Covariant type parameters of a class are only allowed appear in positions:
  - types of values in the class
  - the result types of methods in the class
  - type arguments to other covariant types
- So:

  class Array[+A] {
      def apply(index: Int): A
      def update(index: Int, elem: A)
- **class** stack[+A]{def push(x:A): Stack[A]}

class Stack[+A] {def push[B >: A](x: B): Stack[B] = new NonEmptyStack(x, this)}

Here $B >: A$ denotes push can accept parameterized type B which is restricted over the superType of A

Now, we can push any element of supertype of A, and the type of stack will change accordingly

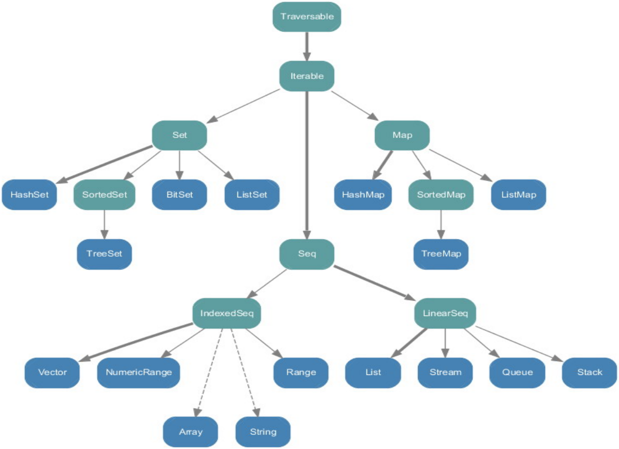
Least Type

Nothing is subtype of any type.

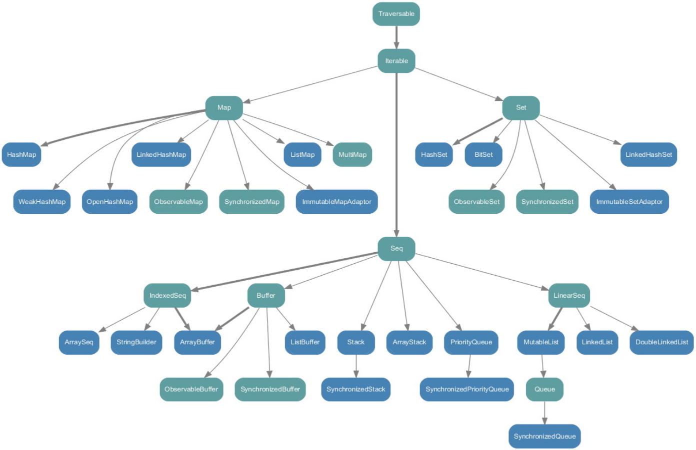
- object EmptyStack extends Stack[Nothing] ...
- val x = EmptyStack.push("abc")

Figure: type covariant of push

Immutable Hierarchy



Mutable Hierarchy



List

There are immutable and mutable list in scala. By default, list is immutable.

so: you can not use List(i) in the left hand of "="

Switch from immutable to mutable List?

- Should worrying about making copies of mutable list
- Explicitly import scala.collection.mutalbe or declare a list variable using "var"
  - var ls = List(3,4);ls = ls::: List(4,5)

List

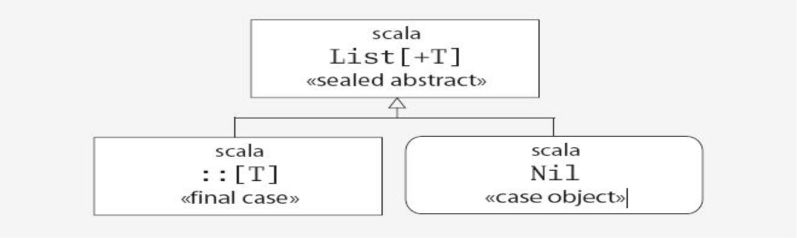
There are immutable and mutable list in scala. By default, list is immutable.
so: you can not use List(i) in the left hand of "="
Switch from immutable to mutable List?

- Should worrying about making copies of mutable list
- Explicitly import scala.collection.mutalbe or declare a list variable using "var"
    - var ls = List(3,4);ls = ls::: List(4,5)
    - val ls = List(3,4)

Declaration and Initialization

- val a = List("abc", "hello") % a immutable list of Type String
- val a: List[Int] = List()
- val a: List[List[Int]]=List(List(0,2,4),List(2,3,4))
- val a: List[Int] = 3::4::5::Nil

Constructor



All lists are built from fundamental constructors, **Nil** and **::**. And :: operator associate from right.

So val a: List[Int] = 3::4::5::Nil == 3::(4::(5::Nil))

Operations on List

Name	Form	Function
head:A	xs.head	returns the first element of a list
tail:List[A]	xs.tail	returns the list consisting of all elements except the first
isEmpty:Booelan	xs.isempty	check empty
take(n:Int):List[A]	xs take n	return first n elems or the whole list
drop(n:Int):List[A]	xs drop n	return elems except first n elems
apply(n: Int): A	xs.apply(n) or xs(n)	return nth elems
:::[B>:A](List[B]): List[B]	xs:::ys	concatenating lists

Cont

- No append operation appending single element to a list.Because the time it takes to append to a list grows linearly with the size of the list.
- List buffers can solve the problem.
  val buf = new ListBuffer[Int]; buf+= elem; buf.toList
- ::: also associate to the right, and takes time proportional to the length of its first operand
- You can use pattern matching in list:

```
def isort(xs: List[Int]): List[Int] = xs match
    case List() => List()
    case x :: xs1 => insert(x, isort(xs1))
```

Higher Order methods

- Mapping
  - map **def** map[B](f: A => B):List[B]this match{
    case Nil => this
    case x :: xs => f(x) :: xs.map(f)}
    e.g. xs map(x => *x*∗factor)
  - foreach: xs foreach (x => println(x))
  - flatmap:The combination of mapping and then concatenating sublists resulting from the map
    def flatmap[B](f:A=>List[B]):List[B] **this** match{
    case Nil => this
    case x :: xs => f(x):::xs.map(f)}

Higher Order methods

- Mapping
  - map **def** map[B](f: A => B):List[B]this match{
    case Nil => this
    case x :: xs => f(x) :: xs.map(f)}
    e.g. xs map(x => $x*$factor)
  - foreach: xs foreach (x => println(x))
  - flatmap:The combination of mapping and then concatenating sublists resulting from the map
    def flatmap[B](f:A=>List[B]):List[B] **this** match{
    case Nil => this
    case x :: xs => f(x):::xs.map(f)}
- Filterring: filter(p: A => Boolean): List[A]
  def posElems(xs: List[Int]): List[Int] = xs filter (x => $x > 0$)

Cont.

- Folding: Applies a binary operator to a start value z and all elements of this sequence, according to some association rule.
  def foldLeft[B](z: B)(op: (B, A) => B): B
  (List(x1, ..., xn) foldLeft z)(op) = ((z op x1) op ...) op xn
  Also known as operator / :. So xs foldLeft z (op) = z /: xs (op)

Other collection type

- **Array**: Array is mutable by default. Unlike List, you can efficiently access an element at an arbitrary position by using the index in parenthesis.
  Apply of List for indexing however is much more costly than in the case of arrays
  - **val** fiveInts = **new** Array[Int](5)
  - **val** fiveInts = Array(1,3,4,5,6)
- **Set**: By default you get an immutable object.
- **Map**: Maps let you associate a value with each element of the collection.
  By importing scala.collection.immutable.TreeSet or TreeMap, one can get sorted set and map.

Tuple

Tuple combines a fixed number of items of different types together so that they can be passed around as a whole.
This is helpful when you want to define a function returning two or more values.For example,under the definition of:

package scala
  case class Tuple2[A, B](_1 : *A*, _2 : *B*)

One can define:

def divmod(x: Int, y: Int) = **new** $Tuple2[Int, Int](x/y, x\%y)$

And then access the element in tuple:

val xy = divmod(x, y)

println("*quotient* : " + *xy*._1 + ", *rest* : " + *xy*._2)

Implicit Conversion

- Want to convert a String in Java to a RandomAccessSeq[Char] and use the method say "exist" in it. However, Java's String class does not extend Scala's RandomAccessSeq trait.

Implicit Conversion

- Want to convert a String in Java to a RandomAccessSeq[Char] and use the method say "exist" in it. However, Java's String class does not extend Scala's RandomAccessSeq trait.
- Now What should we do?

Implicit Conversion

- Want to convert a String in Java to a RandomAccessSeq[Char] and use the method say "exist" in it. However, Java's String class does not extend Scala's RandomAccessSeq trait.
- Now What should we do?
- **Implicit Conversion**

Implicit Conversion

- Want to convert a String in Java to a RandomAccessSeq[Char] and use the method say "exist" in it. However, Java's String class does not extend Scala's RandomAccessSeq trait.
- Now What should we do?
- **Implicit Conversion**
- **implicit def** stringWrapper(s: String) =
    new RandomAccessSeq[Char] {
        def length = s.length
        def apply(i: Int) = s.charAt(i) }

Cont.

Now with the implicit conversion function, one can:

- stringWrapper("abc123") exists (_.isDigit)

Cont.

Now with the implicit conversion function, one can:

- stringWrapper("abc123") exists (_.isDigit)
- "abc123" exists (_.isDigit)

Cont.

Now with the implicit conversion function, one can:

- stringWrapper("abc123") exists (_.isDigit)
- "abc123" exists (_.isDigit)
- scala compiler did the conversion for you.

Cont.

Now with the implicit conversion function, one can:

- stringWrapper("abc123") exists (_.isDigit)
- "abc123" exists (_.isDigit)
- scala compiler did the conversion for you.
- Through doing implicit conversion, class StringWrapper gets every method in RandomAccessSeq for free. This means in scala all implicit conversions pick up newly added method automatically.

Rules for conversion

- **Marking Rule**: Only definitions marked implicit are available. The Functions, Objects, Variables definition are all can be marked as implict
  For example: **implicit def** IntToDouble(x:Int)

Rules for conversion

- **Marking Rule**: Only definitions marked implicit are available. The Functions, Objects, Variables definition are all can be marked as implict
  For example: **implicit def** IntToDouble(x:Int)

- **Scope**:An inserted implicit conversion must be in scope as a single identifier, or be associated with the source or target type of the conversion.
  One exception, the compiler will look for implicit definition in the the companion object of source or target type.
  object Dollar {
  implicit def dollarToEuro(x:): Euro = ...}
  class Dollar ...

Rules for conversion

- **Non-Ambiguity Rule:** An implicit conversion is only inserted if there is no other possible conversion to insert

  $scala > val \quad i : Int = 3 + 3.5$
  This will cause ambiguous conversion error cause the compiler will get two implicit definition of function accepting int as source type: int2double, int2float

Rules for conversion

- **Non-Ambiguity Rule:** An implicit conversion is only inserted if there is no other possible conversion to insert

  $scala > val \quad i : Int = 3 + 3.5$

  This will cause ambiguous conversion error cause the compiler will get two implicit definition of function accepting int as source type: int2double, int2float
- **Where implicit are tried.**:
  - Implicit conversion to an expected type
  - conversions of the receiver of a selection
  - implicit parameters

To expected type

Whenever compiler need type X but see a Y, it search for a implicit conversion that converts Y to X

scala >**implicit def** doubleToInt(x: Double) = x.toInt
scala > val i: Int = 3.5
i: Int = 3

Conversion of receiver

Applying conversion to a receiver of certain method call.

class Rational(n: Int, d: Int) {
def + (that: Rational): Rational...
def + (that: Int): Rational ...
} Suppose we want to compute the expression $1 + Rational(1, 2)$, where the receiver of plus,'1', dose not have the corresponding + operator.

implicit def intToRational(x: Int)=
    new Rational(x,1)

Then $1 + Rational(1, 2) = 3/2$

Implicit Parameters I

The compilers will replace some function call somecall(a) with somecall(a)(b)or (a)(b,c,d),by adding the missing parameters to complete a function call.
Both the last parameter of the function definition and the inserted identifiers should be marked as implicit

class PrePrompt(val pre: String)
class PreDrink(val pre: String)
object Greeter {
    def greet(name: String)
        (implicit prompt: PrePrompt, drink: PreDrink) {
            println("Welcome, "+ name +". The system is ready.")

Implicit Parameters II

```
        println("why not enjoy a cup of "+ drink.pre +"?")
        println(prompt.pre)
}
}
object Prefs {
implicit val prompt = new PrePrompt("Yinghui> ")
implicit val drink = new PreDrink("Tea") }
```

Cont.

If use: import Prefs._, now we can call the greet function without giving the implicit parameters
scala > Greeter.greet("Jane") print
"while you work, why not enjoy a cup of Tea? *Yinghui >*"

ViewBounds I

Here the implicit parameter function orderer allows the whole function can be applied to T which is not the subtype of Ordered[T]

```
def maxList[T](elements: List[T])
        (implicit orderer: T => Ordered[T]): T =
    elements match {
        case List() =>
            throw new IllegalArgumentException("empty list!")
        case List(x) => x
        case x :: rest =>
            val maxRest = maxList(rest)(orderer)
            if (orderer(x) > maxRest) x
```

ViewBounds II

else maxRest }

Because this pattern is common, Scala lets you leave out the name of this parameter and shorten the method header by using a view bound:

def maxList[T< % Ordered[T]](elements: List[T])

You can pass List[Int] to the maxList function even that Int is not the subtype of Ordered[Int] as long the implicit conversion is available

Signals and Monitors I

Every instance of class AnyRef can be used as a monitor by calling one or more of the methods below:

- **def** synchronized[A] (e: => A): A execute in mutual exclusive mode
- **def** wait()
- **def** wait(msec: Long)
- **def** notify()
- **def** notifyAll()

Signals and Monitors II

These methods as well as class Monitor are primitive in scala, we can use them to solve basic concurrent problems.

```
class BoundedBuffer[A](N: Int) {
    var in = 0, out = 0, n = 0
    val elems = new Array[A](N)
    def put(x: A) = synchronized {
        while (n >= N) wait()
        elems(in) = x ; in = (in + 1)%N ; n = n + 1
        if (n == 1) notifyAll() }
    def get: A = synchronized {
        while (n == 0) wait()
        val x = elems(out) ; out = (out + 1)%N ; n = n − 1
```

Signals and Monitors III

```
        if (n == N-1) notifyAll();x}
```

Now we can use this synchronized buffer to communicate between producers and consumers:

```
val buf = new BoundedBuffer[String](10)
spawn { while (true) { val s = produceString ; buf.put(s) } }
spawn { while (true) { val s = buf.get ; consumeString(s) } }

def spawn(p: => Unit) {
    val t =new Thread() { override def run() = p }} t.start()
```

SynVars

A Synchronized variable offers get and set methods to read and set variable. Get block until the variable is set, and after setting the value, set notify all blocked thread who want to read the value of variable to wake up.

Futures

A future is a value which is computed in parallel to some other client thread, to be used by the client thread at some future time.

```
def future[A](p: => A): Unit => A = {
    val result = new SyncVar[A]
    fork { result.set(p)}
    (() => result.get)}
```

Future generate a guard result which is a synchronized variable. Then it forks another thread to compute the result. In parallel to this thread, the function returns a anonymous function. When called, this function will wait until the result guard is invoked. Once this happen, return the result argument.

Mailbox and Actors I

Mailboxes are high-level, constructs for process synchronization and communication.

```
class MailBox {
def send(msg: Any)
def receive[A](f: PartialFunction[Any, A]): A
def receiveWithin[A](msec: Long)(f: PartialFunction[Any, A]): A}
```

The state of mailbox consists of a multiset of messages. Send method adds msg within mailbox, while receive remove the msg. An actor is a thread-like entity that has a mailbox for receiving messages. You can import scala.actor._, then subclass Actor and then implement its act method to implement an actor:

```
import scala.actors._
```

Mailbox and Actors II

```
object myActor extends Actor {
    def act() {
        for (i < 1 to 5) {
            println("Acting!")
            Thread.sleep(1000)}}}
```

Or using utility method actor: **val** someActor = actor{...}

- You can pass a message to an actor by someActor ! msg

Mailbox and Actors III

- An actor will only process messages matching one of the cases in the partial function passed to receive.

  ```
  val intActor = actor {
          receive {
              case x: Int =>
              println("Got an Int: "+ x) }}
  ```

  intActor ! "hello", then the actor will ignore the message

Treat Thread as Actor

The real model of scala actor is more complex than one thread one actor. It can be understood as all the actors share a single thread pool. Whenever an actor start, the system assign a thread to it. If the actor use receive model(mailbox), then the thread always belong to it. If the actor use react model(Future), then scala throw an exception when finish react and the thread can be used by other actors.

If you want to use an thread as an actor,you cannot use Thread.current directly, because it does not have the necessary methods. Instead, you should use Actor.self if you want to view the current thread as an actor.

General rule

Scala is implemented as a translation to standard Java bytecodes. As much as possible, Scala features map directly onto the equivalent Java features.Scala classes, methods, strings, exceptions, for example, are all compiled to the same in Java bytecode as their Java counterparts.

Classes are classes

Scala classes are real JVM classes.

In Java:

```
public class Person {
    public String getName() {
    return "Daniel Spiewak"; } }
```

The same as in scala:

```
class Person {
    def getName() = "Daniel Spiewak" }
```

So one can extend a Java class within Scala, overriding some methods. Or in turn extend this Scala class from within Java

Traits are interfaces I

Because traits allow method definitions, while interfaces must be purely-abstract. Code cannot be mapped directly to a Java construct. Scala is still able to compile traits into interfaces at the bytecode level with some minor enhancements.

In scala:

trait Model {
    def value: Any }

Then it will generate bytecode actually equivalent to Java code below:

public interface Model {
    public Object value(); }

Traits are interfaces II

When comes to traits with method definition, Scala solves this problem by introducing an ancillary class which contains all of the method definitions for a given trait:

The following scala code:

```
trait Model {
    def value: Any
        def printValue(){println(value)
    }
}
```

Will be translated into bytecode equivalent to the Java code below:

```
public interface Model extends ScalaObject {
    public Object value();
    public void printValue(); }
```

Traits are interfaces III

```
public class Model$class {
    public static void printValue(Model self) {
        System.out.println(self.value());}
}
```

So you can implement the Model trait as:

```
public class StringModel implements Model {
    public Object value() {          return "Hello, World!";}
    public void printValue() {
        Model$class.printValue(this);}
    ...
}
```

Generics in Scala

The code in Scla:
abstract class List[$+A$] { ...}
will be translated by type erasure to Java:
public abstract class List$< A >$ { ...}
The variance annotation is gone, but Java wouldnt be able to make anything of it anyway.

Resources I

- Scala Org
  http://www.scala-lang.org/
- Martin Odersky
  Scala By Example.
  PROGRAMMING METHODS LABORATORY, SWITZERLAND, 2009.
- Martin Odersky, Lex Spoon, Bill Venners
  Programming in Scala.
  ARTIMA PRESS, CALIFORNIA, 2007.

Resources II

Dean Wampler
Interop Between Java and Scala.
http://www.codecommit.com/blog/java/interop-between-java-and-scala