L13: The Type System Part 1

Rachata Ausavarungnirun

(rachata.a@tggs.kmutnb.ac.th)

February 25th, 2020

Architecture Research Group

Software System Engineering

Thai-German Graduate School, KMUTNB

Midterm Is Graded

- Place note that the current score is your raw score
  - I will apply a curve to this
- I am definitely making the Ackermann CPS an extra credit
- In general, if you can handle Scala, you did well
  - Otherwise you get stuck on the entire Huffman encoding
- Suggestion for the final exam: Practice

The Type System

(String, Int)

- Type: The prediction of the outcome of an expression and its property
  - This is known during the compile time

- If the expression is a string type, it will eventually evaluate to a string
  - This allows compiler to make more assumption
  - This allows programmers to make more assumption

Ex def foo(n:Int):String

Define a New Type in Scala

- We went through multiple of these examples

- Define a class, trait and objects - Create new type of object
  - class ClassName - blueprint
  - trait TraitName - Abstract of class (Interface) -> Follow certain guideline
  - object ObjectName - Physical item
  - What are the differences between these three?

- You can also use "type" keyword
  - type Mytype = …

Using the Type After Declarations

- You can then utilize this newly declared type using:
- def fooA(x: ClassName) = x
- def fooB(x: TraitName) = x
- def fooC(x: ObjectName.type) = x
- def fooD(x: MyType) = x

Access the type (object or class)

- The .type for the objectName allows you to distinguish between a class and an object

Rules/Constraints

- Defining a type allows you to assert rules - new rule

- There are generally two kinds

lower ≤ P ≤ upper

- Upper bounds extends
  - A <: B means B → A in the type hierarchy (A extends B)

- Lower bounds
  - A >: B means A → B in the type hierarchy (B extends A)

Upperbound Examples

- Let’s first makes some nested declarations
- abstract class Animal {
        def name: String
}
abstract class Pet extends Animal {}
- class Cat extends Pet {
        override def name: String = "Cat“
        def meow: String = "Say meow“
}
- class Dog extends Pet {
        override def name: String = "Dog“
        def bark: String = "Woof woof“
}
- class Lion extends Animal {
        override def name: String = "Lion“
        def growl: String = "(muffled)“
}

$$
\begin{tikzcd}
\text{Animal} \arrow[d] \arrow[dr] & \\
\text{Pet} \arrow[d] & \text{Lion} \\
\text{Dog, Cat} &
\end{tikzcd}
$$

Upperbound Examples

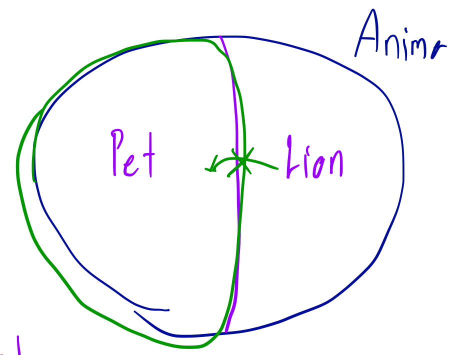
- Let's create a kennel to
- class BadKennel(p: Pet) {
  def pet: Pet = p
  }

Cover Pet (Pet become super set)

↳ Prevent lion

- class Kennel[P <: Pet](p: P) {
  def pet: P = p
  }
- Why is the second version better?
- val dogKennel = new Kennel[Dog](new Dog)
  - This create a new Kennel to keep one pet
- Can you do val lionKennel = new Kennel[Lion](new Lion)

↳ No, Lion doesn't pet

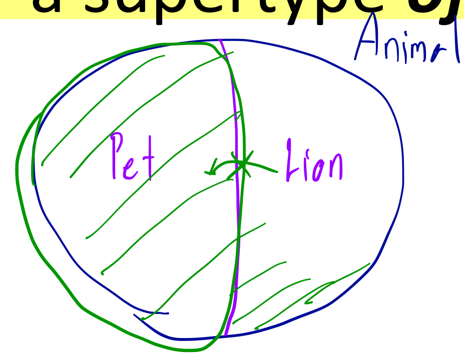


Upperbound Limits

- There is also an infinite type
- Any → Everything → Nothing

Lowerbound

- The type selected must be equal or a supertype ***of the lower bound*** - Everything plus some more



- class A {

  List of Int as lowerbound

  type B >: List[Int]

  def foo(a: B) = a

  }

  Take number, and can traverse (Tree, Iterator)

- val x = new A { type B = Traversable[Int] }
  - This is ok because Traversable -> List

Set[Int] < List[Int] < Traversable[Int]

more specific more general more general

- Then you can use
  - x.foo(List(1,2)) // obviously
  - x.foo(Set(1,2)) // because Traversable[Int] -> Set[Int]
- How about val y = new A {type B = Set[Int]}

↳ In the property of set

Type Parameters

- Let's look at

  Sequence of type T

  - def pickRandom[T] (x: Seq[T]): T

- What does this do?

  ↳ Giving type parameter call T

Type Parameters: Example

- Let’s ties this to the animal/kennel example:
  - def kennelName(animal: …): (String, Kennel[…]) = {
    Val name = animal.name
    (name, new Kennel(animal))
    }

Inside pet

- To use the same thing for Pet, we need to
  - def kennelName**[T <: Pet]**(animal: T): (String, Kennel[T]) = {
    val name = animal.name
    (name, new Kennel(animal))
    }

Variance

- Definite: The ability of type parameters to vary on high-kinded types
  - Say we can C[A]. A higher-kinded type C[A] is said to conform to C[B] if you can assign C[B] to C[A] with no error

- Three types of variance
  - Invariance: if A==B then C[A] will conform to C[B]
  - Covariance: if A→B, then C[A] → C[B] B is smaller than A
  - Contravariance: if A→B, then C[B] → C[A]

Variance Example

- val catKennel = new Kennel(new Cat)
  val dogKennel = new Kennel(new Dog)
  def getName(k: Kennel[Pet]) = k.pet.name

- Why is this not working? - We don't tell the Kennel [Kennel Cat < Kennel Pet]

- Anything missing?

Variance Example Cont.

- Let's fix the Kennel class

Variance of p (p is a pet), whatever go to p is pet

- class Kennel[+P <: Pet](p: P) {

```
    def pet: P = p
}
```

Binary Tree Example

- We can make the left passable/understandable to the binary tree
  - Basically passing the left as any tree type T

Don't have BT[Nothing]

- sealed trait BT[T]
  - This must in fact be +T because the Leaf (i.e. BT[Nothing]) should be passable as BT[any T]

- object Leaf extends BT[Nothing]

- case class Node[T](l: BT[T],k: T,r: BT[T]) extends BT[T]

Create Your Own Function

- Let's say we have f: A→B
- What we learn so far is that it might be ok to pass in a ***wider range of input than A*** and produce and ***output that does not use all of B***

- What if we want to create a function that applies its argument

↳ Not only return type B, but subset of B

Create Your Own Function

- trait MyFunction[Arg, Return] {
      def apply(arg: Arg): Return
  }

My Function

$f : A \to B$

Arg Return

- What is wrong with the code below? - We need exact type

only List[Int] only Any

```
val f: MyFunction[List[Int], Any] = new
MyFunction[Seq[Int], Double] {
      def apply(arg: Seq[Int]): Double = arg.sum
}
```

Create Your Own Function: Fix

- trait MyFunction[-Arg, +Return] { This is works
  def apply(arg: Arg): Return
  }

Withing the scope of Any

- val f: MyFunction[List[Int], Any] = new MyFunction[Seq[Int], Double] {
  def apply(arg: Seq[Int]): Double = arg.sum
  }

We Will Continue on Thursday

- There is more to the type systems and its core to PL

In-Class Exercise 12

- Let’s use this time to finish up in-class exercise 12