Type Conformance and Type Variance

- **Type conformance** governs when things of one type can be used in places defined using other types
  - A key feature of object-oriented languages, but not uniquely so
  - Allows code for functions/methods to be more general

- **Type variance** governs the relationship between basic types and basic type conformance and parameterized (generic) types and their type conformance
  - Determines whether or not basic type conformance relations transfer to the more complex types
  - Determines how exactly this works if possible

Different Types of Type Variance

1. Structure $S$ is invariant if and only if:
   $$(A \preceq B) \land \neg((S[A] \preceq S[B]) \lor (S[B] \preceq S[A]))$$

2. Structure $S$ is covariant if and only if:
   
   $$(A \preceq B) \Rightarrow (S[A] \preceq S[B])$$

3. Structure $S$ is contravariant if and only if:
   
   $$(A \preceq B) \Rightarrow (S[B] \preceq S[A])$$

Data-Structure Type Variance in Java

- In Java, **arrays** are covariant: conformance relationships between types implies the same conformance between arrays of those types
- The following is thus legal in Java:

```java
public static void main(String[] args) {
    String[] strings = new String[10];
    for (int i = 0; i < strings.length; i++) {
        strings[i] = "String_" + i;
    }
    printObjects(strings);
}

private static void printObjects(Object[] objects) {
    for (Object o : objects) {
        System.out.println(o);
    }
}
```

Legal: `String[]` conforms to `Object[]`
Covariance in Java

- The fact that Java arrays are covariant means that there is some flexibility in their use.
- Programmers still need to respect actual types, however:

```java
public static void main(String[] args)
{
    String[] strings = new String[10];
    for (int i = 0; i < strings.length; i++) {
        strings[i] = "String_" + i;
    }
    Object[] objects = strings;
    objects[3] = Integer.valueOf(3);   
}
```

The last line will compile, but will fail at run-time.

Data-Structure Type Variance in Java

- In Java, generic classes are invariant: conformance relationships don’t carry over.
- The following is thus illegal in Java:

```java
public static void main(String[] args)
{
    ArrayList<String> strings = new ArrayList<>();
    for (int i = 0; i < 10; i++) {
        strings.add("String_" + i);
    }
    printObjects(strings);
}

private static void printObjects(ArrayList<Object> objects)
{
    for (Object o : objects) {
        System.out.println(o);
    }
}
```

Use-Site Variance for Java Methods

- Java does allow methods to define some type-variance into their own generic parameters, allowing bounding of types:

```java
public abstract class Animal {...}
public class Cow extends Animal {...}
public class Pen<T> {...}
private static <T extends Animal> void fun(Pen<T> pen) {...}

While a Pen<Cow> does not conform to Pen<Animal> the generic method parameter T allows us to call fun() on both types (and any other containing a sub-class of Animal).
```

Use-Site Variance for Scala

- By default, a Scala generic type parameter is interpreted in an invariant fashion (as in Java):

```scala
abstract class Animal {...}
case class Cow() extends Animal {...}
case class Pen[T] {...}
```

A default Pen[Cow] does not conform to Pen[Animal]

- Scala also allows bounding of generic method parameters to extend the abilities of a function to run on different types of Pen:

```scala
def fun[T <: Animal](pen: Pen[T]): Unit = {...}
```
Declare-Site Variance for Scala

Scala also allows something that Java doesn’t: we can declare generic type parameters in a **covariant** fashion instead:

```scala
abstract class Animal {...}
case class Cow() extends Animal {...}
class Pen[+T]() {...}
```


Any function written to use parameters of type `Pen[Animal]` can be called with inputs of `Pen[Cow]`, and similarly for any other types that extend `Animal`.

---

Covariance and Contravariance

Recall that a generic/parameterized structure `S` is **covariant** if and only if basic conformance transfers in the **same way** to the more complex type:

\[ (A < B) \Rightarrow (S[A] < S[B]) \]

`S` is **contravariant** if and only if the **opposite** relationship holds between the basic and parameterized types:

\[ (A < B) \Rightarrow (S[B] < S[A]) \]

**Why** would this be useful?

---

Contravariant Types in Scala

Scala has a contravariant notation `[−T]` to match covariant option `[+T]`.

In some cases, a contravariant type could allow some **bad things** to happen:

```scala
abstract class Animal {...}
case class Cat() extends Animal {...}
case class Snake() extends Animal {...}
class Pen[−T] (penned: T) {
  var inpen: T = penned
  def penUp(penned: T) = { inpen = penned }
  def getPenned: T = inpen
}
val catPen: Pen[Cat] = new Pen(Cat("Fluffy"))
catPen.penUp(Snake("Kaa"))
val thinkItsCat = catPen.getPenned
```

**Problem**: such a mutable and contravariant `Pen[−T]` could store something of an incorrect type if it meant that we could store any type of `Animal` in a `Pen[Cat]`.

**In fact**: compiler won’t allow this: it can tell that such a `Pen` class is a bad idea.

**What is contravariance good for, then?**

---

Functional Conformance

In a **functional** programming language like Scala, functions are **first-class objects**: they can be the inputs to, and return types of, other functions, just like any other object.

This allows us, for example, to write a function that takes in a list of type `A` and a function from type `A` to type `B`, and produces a list of type `B`:

```scala
def mapList(list: => List[A], fun: A => B): List[B] = list match {
  case head :: tail => fun(head) :: mapList(tail, fun)
  case Nil => Nil
}
```

This then leads to a question: **what** functions actually conform to the type `(A => B)`?
The Liskov Substitution Principle (LSP)

- Introduced by Barbara Liskov (left), and described by Liskov and Jeannette Wing (right) in their paper, “A behavioral notion of subtyping” (1994)
- Says that type $S$ conforms to type $T$ if and only if replacing any object of type $T$ with one of type $S$ does not affect program properties, basic behavior, and correctness

The LSP for Functions

- Type $S$ conforms to type $T$ if and only if replacing any object of type $T$ with one of type $S$ does not affect program properties, basic behavior, and correctness
- Applied to functions, we can say that function $F_1$ conforms to the type of function $F_2$ if and only if:
  1. $F_1$ can handle the same types of inputs as $F_2$, and (maybe) more general types, too
  2. $F_1$ produces the same types of outputs as $F_2$, or (maybe) more specific types, instead

Functional Conformance in Scala

- In order to honor the LSP, Scala defines any function of type $(A \Rightarrow B)$ as having generic type $[-A, +B]$
- That is, a function is contravariant in input type $[-A]$, and is covariant in output type $[+B]$
- This means that if we have functions:
  $$F_1: (S_1 \Rightarrow T_1) \quad F_2: (S_2 \Rightarrow T_2)$$
- Then $F_1$ conforms to $F_2$ ($F_1 \preceq F_2$) if:
  1. $(S_2 \preceq S_1)$: $S_1$ is a supertype of $S_2$
  2. $(T_1 \preceq T_2)$: $T_1$ is a subtype of $T_2$
This Week

- **Topic:** Functional languages (Text, chapter 11)
- **Meet:** usual schedule
- **Homework 05:** due Friday, 10 May, 5:00 PM
- **Office Hours:** Wing 210
  - Monday: 9:00 AM – 10:30 AM
  - Tuesday: 3:00 PM – 4:00 PM
  - Wednesday: 9:00 AM – 10:30 AM
  - Thursday: 2:00 PM – 3:00 PM
  - Friday: 9:00 AM – 10:30 AM

Monday, 29 Apr. 2019

Programming Languages (CS 421/521) 17