**Covariant generics. Patterns of danger and how to detect them statically?**

Many programming languages have to deal with the variance in case of generic types. Java has introduced wildcards (use-site variance), Kotlin, C# and Scala support both declaration-site variance as well as use-site one. And while working with generic types programmers have to understand what co-variance and contra-variance is. Have to distinguish between read and write functions (consumer and producer ones) for the declaration-site variance. And covariance which is used for ordinary (non-generic) types for the assignment or arguments passing is aligned well with the principle that reference to the base class may be assigned with the reference to the derived one. But in case of generics the opposite may work. So, what was wrong with the conformance for generic types based on covariance and why variance for generic type parameters became a rather hard-to-understand concept? The answer is simple – covariant generics are not type-safe! So, if compiler cannot detect the type violation error during compilation then this error is to be detected at runtime – appropriate exception is to be thrown. And thus implies that lack of compilation techniques to detect such cases statically (at compiler time) forced programming language designers to introduce notation for 3 kinds of variance (Java: default-invariance, ‘? extends T’ covariant, ‘? super T’ - contra-variant; Kotlin&C#: default-invariance, ‘out T’ covariant, ‘in T’ contra-variant; Scala: default-invariance, ‘+’ covariant, ‘-‘ contra-variant) and limit operations for covariant generic type parameters as producer (read) and for contravariant ones as consumer (write). So, let’s first have a look at what is wrong with unlimited covariance for generic types

Statement #1: It applies only to the variables or attributes of the reference kind!

// Synthetic example to highlight the issue

**class** A[G] // We have generic class A

attr: G

/\* It has an attribute called ‘attr’ and we can directly assign the value of this attribute using default setter \*/

**end**

Statement #2: Generic class should have at least one attribute of the generic type parameter

// Now let’s consider some variables ‘a’ and ‘b’

**var** a: A[Base]

**var** b: A[Derived] /\* Derived conforms to Base, Derived is a descendant of Base \*/

a := b /\* every polymorphic (covariant) assignment leads to potential extension of the dynamic types set for the assignment target. Existence of such assignment adds A[Derived] into dynamic types set for ‘a’. DynamicTypeSet: {…, A[Derived], …}

The target is always a named entity (variable or class attribute), but source can be an object in place – either new T or literal and then target will be the only reference to it \*/

a.attr := **new** T

attr **is** b.attr /\* What is the type of attr? Static type of attr is equal to the static type of b.attr -> Derived but dynamic is Base. So, the breakage of type system occurred \*/

a := **new** A[Derived] /\* such assignment is always type-safe !!! As we have no direct way how to access created object of type A[Derived]. Object reference aliasing does not occur here \*/

**if** a **is** A[Derived] **do** /\* this type check must dynamically return false if polymorphic assignment was not executed \*/

attr **is** a.attr // attr is of static type U

**end** // if

// More concrete example

a: Array [Any] // a is initialized as an empty array

b: Array [String] is (“s1”, “s2”, “s3”)

a := b /\* That is the polymorphic assignment which creates the basis for a headache \*/

a(1) := Integer.5 // Put integer value into the 1st element

a(2) := Real.3.1415492653589 // Put real value into the 2nd element

a(3) := (“string1”, Boolean.true) // Put tuple into the 2rd one

**var** myString: String **is** b(1)

myString := b(2)

myString := b(3)

Call is valid if it is static-type valid and valid for every dynamic type of the call target

target.foo (argument1, argument2, … , argumentn)

Let static type of target is D0

And dynamic types set of the target is D1, … Dm D0 can be part of this set or not. (D0 can be abstract type for example or have constructor hidden)

Static-type validity

1. Covariant polymorphic assignment A[] <- A[]
2. Write object of non-conforming type
3. Read object of static type