**Conformance and conversion routines – the compatibility of types**

It is essential to define well when assignments are valid and when overriding while inheriting works. The latter is described by the signature conformance while the assignment is driven by the following rule. The type of the expression on the right side of the assignment should either conform to the type of the writable on the left side or have a proper conversion routine be in place. So, type A is compatible with type B is A conforms to B or objects of type A can be converted into the objects of type B.

Let’s start with conformance. The simplest case of conformance that every type conforms to itself.

a: A **is new** A

The next case is when there is a path in the inheritance graph between the current unit type and another one. And this path consists only of conformant inheritance edges.

**unit** A **end**

**unit** B **extend** A **end** // That is a conformant inheritance

**unit** C **extend** ~A **end** // That is a non-conformant inheritance

a: A **is new** **B** // OK! As B conforms to A

a: A **is new** C // Compile-time error as C does not conform to A

Next is when the type is generic instantiation then in addition to unit type conformance it is necessary to take into account type by type conformance of all elements of the instantiation.

**unit** A[U, V] **end**

**unit** B[X, Y] **extend** A [X, Y] **end**

**unit** T1 **end**

**unit** T2 **end**

**unit** S1 **extend** T1 **end**

**unit** A[A, B, C] **end**

a: A[T1, T2] **is new** A[T1, T2]// OK!

a: A[T1, T2] **is new** A[S1, T2]// OK!

a: A[T1, T2] **is new** A[T1, S1]// Compile time error as S1 does not conform to T2

a: A[T1, T2] **is new** B[T1, T2]// OK!

a: A[T1, T2] **is new** B[S1, T2]// OK!

a: A[T1, T2] **is new** B[T1, S1]// Compile time error as S1 does not conform to T2

a: A[T1, T2] **is new** A[T1, T2, S1]/\* Compile time error as A with 3 generic parameters does not conform to A with 2 generic parameters \*/

And last but not least is tuple conformance. All tuples are of the same type – tuple type and it means that we need to consider (similar to generic instantiations) by-element conformance of element types.

a: (T1, T2) **is** (**new** T1, **new** T2)// OK!

a: (T1, T2) **is** (**new** S1, **new** T2)// OK!

a: (T1, T2) **is** (**new** T1, **new** S1)// Compile time error as S1 does not conform to T2

a: (T1, T2) **is** (**new** S1, **new** T2, **new** S1)/\* OK! as all elements of the longer tuple, which has corresponding elements in the shorter tuple, conform to them \*/

And now let’s consider conversion routines as they also play important roles in assignments. There are two types of conversion routines: from-conversion and to-conversion. The first one is a procedure with one parameter and the second one is a function with no arguments. Let’s examine the following example

**unit** A

:= (other: T) **do end**

/\* That is a from-conversion procedure, which has some algorithm how to perform a conversion from objects of type T into the objects of current type A \*/

:= (): T **do end**

/\* That is a to-conversion function which creates a proper object of type T and

works well for assignments too \*/

foo (arg: T) **do end**

**end**

**unit** T **end**

**var** a **is new** A /\* We need to create a valid object of type A first. To ensure A invariant is held \*/

a := **new** T /\* And then we can assign to an object of type A the object of type T using the from-convertor procedure \*/

a: A **is new** T /\* That is incorrect (compile-time error) as object a of type A was not created yet \*/

A.foo (**new** T) // That is Ok as T conforms to T

A.foo (**new** A) /\* That is OK as unit A has to-conversion function to type T, the semantics of any routine call is that arguments passing is an assignment of arguments to parameters so conformance and conversion functions will work and that is why conversion functions are marked with the := sign \*/

And let's consider one more example which combines inheritance with tuples.

**unit** PolarCoordinates **extend** (radius: Real; angle: Real)

**new** make (r: **as** radius; a: **as** angle) **do**

radius := r

angle := a

**end**

**end**

**unit** CartesianCoordinates **extend** (x: Real; y: Real)

**new** create (h: **as** x; v: **as** y) **do**

x := h

y := v

**end**

**end**

**var** point1 **is new** PolarCoordiantes.make (5.0, 30.0)

**var** point2 **is new** CartesianCoordinates.create (4.6, 7.7)

point1 := point2 /\* Compile time error as type CartesianCoordinates does not confrom to PolarCoordiantes \*/

point2 := point1 /\* Compile time error as type PolarCoordiantes does not confrom to CartesianCoordinates \*/

**var** tuple : (Real, Real) **is** point1 // It works! As point1 is descendant of tuple

tuple := point2 // It works too as popint2 is descendant fo tupel as well

point1 := (5.5, 6.6) // Compile-time error as point1 is not a tuple

point2 := (4.4, 7.7) // Compile-time error as point2 is not a tuple

And if one likes to extend basic types functionality then try something like this

**val unit** MyInt **extend** Integer

:= (that: Integer) **do** … **end**

**end**

**var** x **is new** MyInt

x := 6 // That is OK only because there is a from-conversion procedure defined

x: MyInt **is** 7 // That is a compile-time error as Integer does not conform to MyInt

And now we can review signature conformance which also has similarity with generic instantiation conformance and uses tuple conformance. If we have routine foo with signature S1 and routine goo with signature S2 then S2 conforms to S1 if they have the same number of elements and every type element of signature S1 conforms to the appropriate element of signature S1. Let’s consider the following example

**unit** A

foo (T1; T2; T3): T4

**end**

**unit** B **extend** A

**override** foo (U1; U2; U3): U4

**end**

So, in this example the signature of foo from A is ((T1, T2, T3), T4) and foo from B has ((U1, U2, U3), U4) and the task is equal to tuple conformance. Tuple ((U1, U2, U3), U4) conforms to the tuple ((T1, T2, T3): T4) as they have the same number of elements – 2 in this case (for the procedure we may just drop the return type) and for the first element we again have tuples conformance case - whether (U1, U2, U3) conforms to (T1, T2, T3) and check if U4 conforms to T4.