**“Up-casting” to resolve ambiguity in case of overloading.**

In case of overloading, there could be cases when the compiler is not able to determine uniquely which routine is to be called. See the example below. Note that U and V are different types but they may conform to each other

foo (p: U) **do** … **end**

foo (p: V) **do** … **end**

foo (**new** T)

If T conforms to both U and V compiler has to state the problem – call ambiguity in case of overloading. The solution is rather straightforward

a: U is **new** T

foo (a)

Introducing extra local attribute compiler is notified which version of foo is to be called. But if we have several parameters for foo and for some subset of them we have to introduce such local attributes only to denote the right version of foo to be called. So, we do not need these locals we just need to specify the type helping the compiler to identify the right signature.

foo ({U} **new** T)

Generalizing we will have the following form of the call

some\_routine ({U1} exprT1, {U2} exprT2, …, {Un} exprTn)

where Ti conforms to Ui for i in 1..n

The alternative notation could be in the postfix form, but it looks less expressive

foo (**new** T: U)

some\_routine (exprT1: U1, exprT2: U2, …, exprTn: Un)

If we have similar ambiguity with the function call

goo(): U **do** … **end**

goo(): V **do** … **end**

x **is** goo () /\* which goo is to be called and what type should be deduced for x \*/

x: S **is** goo () // where U and V both conform to S

x: S **is** {U} goo () // prefixing resolves ambiguity

So, any expression can be prefixed with the explicit unit type specification to help the compiler to resolve ambiguity in case of overloading. The typical tactic is a programmer runs the compiler, it reports call ambiguity, and then the programmer inserts type specifications.