Type conformance for the modern general-purpose programing language

**Abstract**

The paper presents an overview of the type system which supports the convergence of procedural, object-oriented, functional, and concurrent programming paradigms relying on static type checking with smart type inference support and the ability to ensure dynamic type safety as well.

**Keywords**

Object, type, unit, class, module, interface, conformance, compatibility, type conversions, setters, reference and value objects, immutability.

[CCS](https://dl.acm.org/ccs/ccs.cfm?id=0&lid=0&CFID=1009033506&CFTOKEN=16747759) →  [Software and its engineering](https://dl.acm.org/ccs/ccs.cfm?id=10011007&lid=0.10011007&CFID=1009033506&CFTOKEN=16747759) →  [Software notations and tools](https://dl.acm.org/ccs/ccs.cfm?id=10011006&lid=0.10011007.10011006&CFID=1009033506&CFTOKEN=16747759) →  [General programming languages](https://dl.acm.org/ccs/ccs.cfm?id=10011008&lid=0.10011007.10011006.10011008&CFID=1009033506&CFTOKEN=16747759) →  [Language types](https://dl.acm.org/ccs/ccs.cfm?id=10011009&lid=0.10011007.10011006.10011008.10011009&CFID=1009033506&CFTOKEN=16747759) →  Multi-paradigm programming languages

# INTRODUCTION

The type system sets the basis for the reliable programming language and allows programmers to effectively express software design solutions using the power of the particular programming language raising the productivity of the software development process.

The modern tendency of convergence of different programming paradigms (merging procedural programming, structured programming, object-oriented programming, functional programming, and concurrent programming) forces the type system to support this.

Next is to define the notion of type as an important characteristic of every object during execution time (runtime). The type fixes the number of operations and their properties (signatures) as well as the size of memory required to store the object (number, valid values, and types of object attributes). So, a type is an abstraction used to describe the structure and behavior of objects.

Authors rely on concepts that are well-known by a broad audience of programmers and terms like class or variable will be used without formal definitions. Some definitions will be given right now to simplify the understanding of examples.

The unit is a named set of members, where a member can be a routine or an attribute. Routines stand for actions while attributes stand for data. If a routine returns some value as a result of its execution we call it a function otherwise a procedure. If an attribute can change its value during the program execution we call it a variable attribute (or simply variable) otherwise we call it a constant attribute (or simply constant or immutable attribute). Unit is very similar to class and the difference is that the unit incorporates characteristics of classes and modules (The term module is used like it was introduced in Ada (package) [2], Modula-2 (module) [4] – a generally available collection of data and routines with initialization) in one concept and the foundation for types.

### **GENERIC CONFORMANCE**

maxElement [G] (container: Array[G]): G

**require**

container.count >0

/// Non-empty container

**do**

**return** := container(1)

**while** index in 2..container.count **do**

**if** return < container(index) **then**

**return** := container(index)

**end**

**end**

**end**

versus polymorphic variant

maxElement (container: Array [Any]): Any **=>** maxElement[Any] (container)

### **TYPE CONFORMANCE**

The simplest case of conformance is that each type conforms to itself.

a: A **is new** A

**Unit conformance** is based on the idea to check if there is a path in the inheritance graph between the current unit type and another one. And this path should consist only of conformant inheritance edges.

**unit** A **end  
unit** B **extends** A **end**

That is a conformant inheritance.

**unit** C **extend** ~A **end**

That is a non-conformant inheritance.

a: A **is new** **B**

Valid as B conforms to A.

a: A **is new** C

Not valid as C does not conform to A.

When a type is a 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. Notice that square brackets are used to highlight generics. Access to tuples and arrays is done using parentheses as these are function calls with parameters.

**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]

Valid as types are identical.

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

Valid as S1 conforms to T1.

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

Not valid as S1 does not conform to T2.

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

Valid as B conforms to A and has identical instantiation types.

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

Valid as B conforms to A and has conformant instantiation types.

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

Not valid as S1 does not conform to T2.

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

Not valid as A with 3 generic parameters does not conform to A with 2 generic parameters.

**Tuple conformance**. All tuples are of the same type – tuple type. 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)  
 // Valid as types are identical.

a: (T1, T2) **is** (**new** S1, **new** T2)  
 // Valid as S1 conforms to T1.

a: (T1, T2) **is** (**new** T1, **new** S1)  
 // Not valid as S1 does not  
 // conform to T2.

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

Last but not least is **unit type conformance**. All unit types are of the same type – ‘unit’, similar to tuple conformance. So, we need to look at a member after a member to check if they conform to each other. The difference from tuples is that tuples have an order of their elements but unit types don’t. But every member of the unit type has a name. And search by name identifies the subset of members that will define the conformance. So, if we have two unit types A and B then A conforms to B if for every member of A there is a member with the same name in B and its signature in A conforms to the signature of the corresponding member in B and B has not other members. Common sense logic brings the idea that to an empty unit any unit type will conform. Any ‘thinner’ unit type will always accommodate in terms of conformance the ‘thicker’ one. Empty unit means any unit!

**var** A **is unit end**

**var** B **is unit** foo (T1, T2): T3  
 goo (T3)  
 **var** attr: T1 := (T1)  
 // It has a setter with an argument  
 // of type T1  
**end**

**var** C **is unit** foo (S1, T2): T3  
 goo (T3)  
**end**

**var** D **is unit** foo (S1, T2): T3  
 goo (T3)  
 **var** attr: T1 := (S1)  
 // it has setter with an argument  
 // of type S1  
 too (T1, T2, T3)  
**end**

A := B // Valid as any type conforms  
 // to an empty type

B := C // Not valid as C lacks  
 // a member called attr

B:= D // Valid as all D members fit   
 // all B members in terms of   
 // conformance and D has extra  
 // members; it is thicker than  
 // B

### **TYPE CONVERTABILITY**

Here, conversion routines are considered as they also play important roles in assignments. There are two types of conversion routines: from-conversions and to-conversions. 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 with units A and T.

**unit** A  
 := (other: T) **do ... end** // This is a from-conversion  
 // procedure, which has some   
 // algorithm how to perform a   
 // conversion from objects of type T  
 // into the objects of the current  
 // type A. T is just some empty  
 // type.

:= (): T **do ... end** //This is a to-conversion function   
 // that creates a proper object of   
 // type T and works well for   
 // assignments too.

foo (arg: T) **do end** // Procedure ‘foo’ will be used to   
 // show how converters work

**end**

**unit** T **end**

At first, let’s create a valid object of type A, and then different conversions will be done using an assignment statement.

**var** a **is new** A   
a := **new** T

Here, a can be assigned with an object of type T as it has a from-converter procedure.

a.foo (**new** A)

This call is valid as well as unit A has a to-conversion function to type T.

Here is a brief review of routines’ 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**

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 – two 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.

Some notes about the name and structural type equivalence. Below is an example in Ada[2], which presents name equivalence – type Integer\_1 is not compatible with type Integer\_2 as they have different names! But structurally they are identical.

// Ada  
**type** Integer\_1 **is range** 1..10;  
**type** Integer\_2 **is range** 1..10;  
A: Integer\_1 := 8;  
B: Integer\_2 := A; -- illegal!

We can choose between two different approaches. The first one is right below

a : 1 .. 10 **is** 8  
b : 1 .. 10 **is** a

Here, a and b have the same type: range type 1..10 and a can be assigned to b.

In the second case when one likes to introduce new types, type Integer\_1 is different from Integer\_2 and they are not compatible.

**unit** Integer\_1 **extend** Integer  
**require  
 this in** 1 .. 10  
**end**

**unit** Integer\_2 **extend** Integer  
**require  
 this in** 1 .. 10  
**end**

**var** a **is new** Integer\_1  
**var** b: Integer\_2 **is** a

Declaration of b leads to compile-time error as the type of a is not compatible with the type of b.

So, support of name equivalence is in place but the term name is treated a bit wider. 1..10 is treated as the type name, A | B is the type name too, and (T1, T2, T3) is also a type and its name is a tuple (T1, T2, T3), type “**as this**” is compatible to the type of the unit where an attribute of such type was declared.

# Conclusion

The paper presents the uniform type system which supports the convergence of different models of programming, allows to have static typing with type inference, to have all types and values to be explicitly and fully defined using the same programming language. For that, the concept of the unit is used and it is defined as a combination of class and module concepts. Types compatibility if fully and explicitly defined using type conformance and type conversion. Both conformance and conversions are fully defined too. The approach which allows treating manifest constants as immutable objects of the proper type is introduced, it works well for basic types and user-defined ones. It superceeds enumerations and sets the background to have the programming language which is fully defined using the language itself.

# REFERENCES

1. Clemens A. Szyperski: Import is Not Inheritance. Why We Need Both: Modules and Classes, ECOOP 1992.
2. International Standard: ISO/IEC 8652:2012 Information technology – Programming Languages – Ada.
3. Bertrand Meyer: Object-Oriented Software Construction, Second Edition. Prentice-Hall. ISBN 0-13-629155-4.

[4] International Standard: ISO/IEC 10514-2:1998 Information technology – Programming Languages – Modula-2.