14

Draft 5.02.00-0, 15 August 2005 (Santa Barbara). Extracted from ongoing work on future third edition of "Eiffel: The Language". Copyright Bertrand Meyer 1986-2005. Access restricted to purchasers of the first or second printing (Prentice Hall, 1991). Do not reproduce or distribute.

Conformance

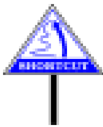
14.1 OVERVIEW

Conformance is the most important characteristic of the Eiffel type system: it determines when a type may be used in lieu of another.

The most obvious use of conformance is to make assignment and argument passing type-safe: for *x* of type *T* and *y* of type *V*, the instruction *x* := *y*, and the call *some_routine* (*y*) with *x* as formal argument, will only be valid if *V* is *compatible* with *T*, meaning that it either *conforms* or *converts* to *T*. Conformance also governs the validity of many other constructs, as discussed below.

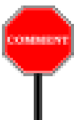
Conformance, as the rest of the type system, relies on inheritance. The basic condition for *V* to conform to *T* is straightforward:

- The base class of *V* must be a descendant of the base class of *T*.
- If *V* is a generically derived type, its actual generic parameters must conform to the corresponding ones in *T*: *B* [*Y*] conforms to *A* [*X*] only if *B* conforms to *A* and *Y* to *X*.
- If *T* is expanded, inheritance is not involved: *V* can only be *T* itself.



If this is your first reading, this simple explanation is probably sufficient to understand the references to conformance in the rest of this book, and you may want to move on right away to the next chapter.

A full understanding of conformance requires the formal rules explained below, which take into account the details of the type system: constrained and unconstrained genericity, special rules for predefined arithmetic types, tuple types, anchored types.



The following discussion introduces the various conformance rules of the language as "DEFINITIONS". Although not validity constraints themselves, these rules play a central role in many of the constraints, so that language processing tools such as compilers may need to refer to them in their error messages. For that reason each rule has a validity code of the form *VNC*x.

These rules appear in the index with other validity codes under "validity constraints", as well as separately under "conformance rules".

14.2 CONVERTIBILITY AND COMPATIBILITY

To permit assignment or argument passing, conformance is only one of the two possibilities; the other is **convertibility**, allowing reattachment operations — assignment and argument passing — that convert the source to the type of the target. Convertibility is particularly useful for arithmetic types, allowing us to rely on standard mathematical conventions when assigning, for example, an integer value to a real target.

Often, as in the rules for assignment and argument passing, we must state that the type of an expression *either* conforms or converts to that of an entity. We need a term that covers both mechanisms:

DEFINITION

Compatibility between types

A type is **compatible** with another if it either conforms or converts to it.

It is also useful to extend this notion to expressions, so that we can say "*y* is compatible with *x*" rather than "the type of *y* is compatible with that of *x*"::

DEFINITION

Compatibility between expressions

An expression *b* **is compatible with** an expression *a* if and only if *b* either conforms or converts to *a*.

For conformance we may define the notion now:

Expression conformance

An expression *exp* of type *SOURCE* **conforms to** an expression *ent* of type *TARGET* if and only if they satisfy the following conditions:

1 • *SOURCE* conforms to *TARGET*.

2 • If *TARGET* is attached, so is *SOURCE*.

3 • If *SOURCE* is expanded, its version of the function *cloned* from *ANY* is available to the base class of *TARGET*.

So conformance of expressions is more than conformance of their types. Both conditions 2 and 3 are essential. Condition 2 guarantees that execution will never attach a void value to an entity declared of an attached type — a declaration intended precisely to rule out that possibility, so that the entity can be used as target of calls. Condition 3 allows us, in the semantics of attachment, to use a cloning operation when attaching an object with "copy semantics", without causing inconsistencies.

A later definition will state what it means for an expression *b* to *convert* to another *a*. As a special case these properties also apply to entities.

Conformance and convertibility are exclusive of each other, so we study the two mechanisms separately. The rest of the present discussion is devoted to conformance.

→ *"Conversion principle", page 400.*

14.3 APPLICATIONS OF CONFORMANCE

Conformance governs the validity of many language constructs. For any of the following to be valid, *V* must conform to *T*, with *x* of type *T* and *y* of type *V*:

→ In the first two cases (but none of the others) V may also **convert** *to T. See chapter 15.*

- The assignment *x* := *y*.
- The routine call *r* (…, *y*, …), where *x* is the formal argument declared in *r* at the position that *y* has in the call.
- The creation instruction **create** {*V*} *x* …, which creates an instance of *V* and attaches *x* to it.
- The redeclaration of *x* as being of type *V* in a proper descendant, where *x* is an attribute, a function, or a routine argument.
- Any use of *C* […, *V*, …] with *V* as actual generic parameter, where the corresponding formal generic parameter of *C* is constrained by *T* — in other words, the class is declared as *C* […, *G* –> *T*, …].

As these examples indicate, conformance is originally a relation between **types**: the language's rules specify when a type *V* conforms to a type *T*.

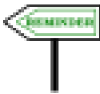
The rest of this chapter starts with a generalization of the notion of conformance, originally defined for types, to signatures. The discussion then covers conformance rules for the various kinds of type studied in the last three chapters: class types, first without genericity, then with genericity added to the picture; formal generic parameters; expanded types; tuple types; anchored types (including expression conformance).

14.4 EXPRESSION AND SIGNATURE CONFORMANCE

We have already generalized the notion of conformance from types to *expressions*. Another useful generalization is to *signatures*. A signature gives the full type information for a feature: the types of its arguments, if any, and of its result, if any. Conformance of signatures is important because it governs redeclaration: whenever you redeclare a feature, the signature of the new version must conform to the signature of the original.

← The conformance constraint for signatures is clause 2 of the Redeclaration rule, page 307

The definition of conformance for signatures will follow immediately from the definition for types: a signature *t* conforms to a signature *s* if and only if every element of *t* (the type of an argument or result) conforms to the corresponding element of *s*.



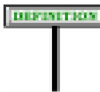
More precisely, recall that a signature is a pair of sequences of the form

← Signatures were defined in "THE SIGNATURE OF A FEATURE", 5.13, page 148.

$[A_1, \ldots A_n], [R]$

where all elements involved are types; the A_i are the types of the formal arguments (for a routine) and R is the result type (for a function or an attribute). Either component of the pair, or both, may be empty (the first is empty for an attribute or a routine without arguments; the second, for a procedure). The second component has at most one element, but remember that this element may be a tuple type, so for all practical purposes we can deal with multiple-result functions.

Then from a definition of type conformance, as explored in the rest of this chapter, we immediately infer a definition of signature conformance:



Signature conformance *VNCS*

A signature $t = [B_1, \ldots B_n], [S]$ **conforms to** a signature $s = [A_1, \ldots A_n], [R]$ if and only if it satisfies the following conditions:

1 • Each of the two components of *t* has the same number of elements as the corresponding component of *s*.

2 • Each type in each of the two components of *t* conforms to the corresponding type in the corresponding component of *s*.

3 • Any B_i not identical to the corresponding A_i is detachable.

For a signature to conform: the argument types must conform (for a routine); the two signatures must both have a result type or both not have it (meaning they are both queries, or both procedures); and if there are result types, they must conform.

A "query" is a function or attribute, i.e. a feature returning a result.

Condition 3 adds a particular rule for "covariant redefinition" of arguments as defined next.

Covariant argument

In a redeclaration of a routine, a formal argument is **covariant** if its type differs from the type of the corresponding argument in at least one of the parents' versions.

From the preceding signature conformance rule, the type of a covariant argument will have to be declared as *detachable*: you cannot redefine *f* (*x*: *T*) into *f* (*x*: *U*) even if *U* conforms to *T*; you may, however, redefine it to *f* (*x*: **?***U*). This forces the body of the redefined version, when applying to *x* any feature of *f*, to ensure that the value is indeed attached to an instance of *U* by applying an Object_test, for example in the form

if {*x*: *U*} *y* **then** *y*.*feature_of_U* **else** … **end**

This protects the program from *catcalls* — wrongful uses, of a redefined feature, through polymorphism and dynamic binding, to an actual argument of the original, pre-covariant type.

The rule only applies to *arguments*, not results, which do not pose a risk of catcall.

This rule is the reason why the Feature Declaration rule requires that if any routine argument is of an anchored type, that type must be detachable, since anchored declaration is a shorthand for explicit covariance.

14.5 DIRECT AND INDIRECT CONFORMANCE

Conformance is, with one restriction, a reflexive and transitive relation: any type conforms to itself, and if *V* conforms to *U* and *U* to *T*, then *V* conforms to *T*. (The restriction is that *T* must not be expanded; see below.)

Also, replacing an actual generic parameter by a conforming type yields a conforming type: if *Y* conforms to *X*, then *B* [*Y*] conforms to *B* [*X*] for a class *B* with one generic parameter; this generalizes to any number of parameters.

We may use these properties to simplify the study of conformance rules. By considering the relation **direct conformance**, which only covers the case of a class conforming to a different one through no intermediary, we can define general conformance by reflexive transitive closure:

DEFINITION

General conformance *VNCC*

Let *T* and *V* be two types. *V* **conforms to** *T* if and only if one of the following conditions holds:

1 • *V* and *T* are identical.

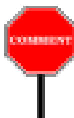
2 • *V* conforms directly to *T*.

3 • *V* is *NONE* and *T* is a detachable reference type.

4 • *V* is $B [Y_1,\dots Y_n]$ where *B* is a generic class, *T* is $B [X_1,\dots X_n]$, and for every X_i the corresponding Y_i is identical to X_i or, if the corresponding formal parameter does not specify **frozen**, conforms (recursively) to X_i.

5 • *T* is a reference type and, for some type *U* (recursively), *V* conforms to *U* and *U* conforms to *T*.

6 • *T* or *V* or both are anchored types appearing in the same class *C*, and the deanchored form of *V* in *C* (recursively) conforms to the deanchored form of *T*.



Cases 1 and 2 are immediate: a type conforms to itself, and direct conformance is a case of conformance.

Case 3 introduces the class *NONE* describing void values for references. You may assign such a value to a variable of a reference type not declared as attached (as the role of such declarations is precisely to exclude void values); an expanded target is also excluded since it requires an object.

← *See "NONE", 6.7, page 175.*

Case 4 covers the replacement of one or more generic parameters by conforming ones, keeping the same base class: *B* [*Y*] conforms to *B* [*X*] if *Y* conforms to *X*. (This does not yet address conformance to *B* [*Y1*, … *Yn*] of a type *CT* based on a class *C* different from *B*.) Also note that the **frozen** specification is precisely intended to preclude conformance other than from the given type to itself.

Case 5 is indirect conformance through an intermediate type *U*. Note the restriction that *T* be a reference type; this excludes indirect conformance through an expanded type, as explained in later discussions.

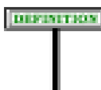
Finally, case 6 allows us to treat any anchored type, for conformance as for its other properties, as an abbreviation — a "macro" in programmer terminology — for the type of its anchor.

Thanks to this definition of conformance in terms of direct conformance, the remainder of the discussion of conformance only needs to define **direct** conformance rules for the various categories of type.

The general conformance rules follow: for any type *T*, direct conformance rules will yield the (possibly empty) set *ST* of types which conform directly to *T*; then the types that conform to *T* are *T* itself, the members of *ST*, and, recursively, if *T* is a reference type, any type conforming to a member of *ST*.

Before we move on, let's give a name, "conformance path", to the sequence of types appearing implicitly in case 5 of the definition. This notion will be useful in particular in the discussion of repeated inheritance:

→ "THE REPEATED INHERITANCE CONSISTENCY CONSTRAINT", 16.13, page 455.



Conformance path

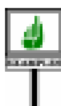
A **conformance path** from a type U to a type T is a sequence of types $T_0, T_1, \ldots T_n$ ($n \geq 1$) such that T_0 is U, T_n is T, and every T_i (for $0 \leq i < n$) conforms to T_{i+1}. This notion also applies to **classes** by considering the associated current types.

← "CURRENT TYPE, FEATURES OF A TYPE", 12.11, page 357

14.6 CONFORMANCE TO A NON-GENERIC REFERENCE TYPE

Let us begin with the simple but common and important case of conformance to a reference type *B* obtained directly from a non-generic class. Then direct conformance is essentially inheritance: *C* conforms directly to *B* if *C* is an **heir** of *B*. (As a consequence, *D* conforms to *B* if *D* is a *descendant* of *B*.) *C* (and *D*) may be generically derived or not.

Assume for example class declarations beginning with



```
class C1… inherit A1 …
class C2 [G] … inherit A2 …
expanded class C3 [G, H –> HASHABLE] … inherit A3 …
```

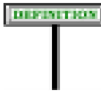
Then, if *X* is any type, and *Y* any type conforming to *HASHABLE*:

- *C1* conforms directly to *A1*.
- *C2* [*X*] conforms directly to *A2*.
- *C3* [*X*, *Y*] conforms directly to *A3*.

C3 is expanded, C1 and C2 are not; this has no influence on the discussion. But A1, A2 and A3 must not be expanded. *See 14.9, page 386, on conformance to expanded types.*

These examples assume that all the types involved are **attached** (the default). Indeed if the target type is attached the source type must be attached too; otherwise — in an attachment made valid by the conformance — we could end up assigning a void value to an attached entity

Here then is the rule:

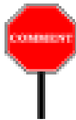


Direct conformance: reference types *VNCN*

A Class_type *CT* of base class *C* **conforms directly** to a reference type *BT* if and only if it satisfies the following conditions:

1 • Applying *CT*'s generic substitution to one of the conforming parents of *C* yields *BT*.

2 • If *BT* is attached, so is *CT*.



The restriction to a reference type in this rule applies only to the target of the conformance, *BT*. The source, *CT*, may be expanded.

As in the examples.

The basic condition, 1, is inheritance. To handle genericity it applies the "generic substitution" associated with every type: for example with a class *C* [*G*, *H*] inheriting from *D* [*G*], the type *C* [*T*, *U*] has a generic substitution associating *T* to *G* and *U* to *H*. So it conforms to the result of applying that substitution to the Parent *D* [*G*]: the type *D* [*T*].

Condition 2 guarantees that we'll never attach a value of a detachable type — possibly void — to a target declared of an attached type; the purpose of such a declaration is to avoid this very case. The other way around, an attached type may conform to a detachable one.

This rule is the foundation of the conformance mechanism, relying on the inheritance structure as the condition governing attachments and redeclarations. The other rules cover refinements (involving in particular genericity), iterations of the basic rule (as with "general conformance") and adaptations to special cases (such as expanded types).

14.7 GENERICALLY DERIVED REFERENCE TYPES

---- SECTION TO BE REWRITTEN OR REMOVED ---

*Conformance **of** a generically derived type such as BT to a non-generic one raises no particular problem and is covered by the above rule on non-generic conformance.*

The next typing mechanism to take into account is genericity. A generic class such as

```
class B [G, H –> DT, I] … end
```

raises two kinds of conformance issues. One is the conformance to a generically derived type *BT* based on *B*, of the form *B* [*TK*, *TL*, *TM*]. The other is conformance properties of the formal parameters *G*, *H*, *I* … themselves, which within the class text represent types. This section deals with the first issue; the next one will cover formal parameters.

When does a type *CT* conforms to *BT* of the form *B* [*TK*, *TL*, *TM*]? For identical *B* and *C* we already have the answer from case 4 of the General Conformance rule: it tells us that in this case *CT* must be of the form *B* [*TR*, *TS*, *TT*] with the same number of parameters, with *TR* conforming to *TK*, *TS* to *TL* and *TT* to *TM*.

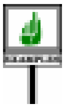
← *Page 380*.

Thanks to this first rule, it suffices to examine the case of different base classes, but the same actual generic parameters. Then to check conformance of, say, *C* [*Y*] to *B* [*X*], you will check separately the conformance of *C* [*Y*] to *B* [*Y*], using the rule given below, and then show that *Y* conforms to *X*, which will complete the deduction thanks to case 4 of General Conformance.

See "EXPANDED TYPE CONFORMANCE", 14.9, page 386 about the expanded case, for which conformance possibilities are very limited.

This is the basic idea for the Generic Substitution rule given below. The full rule is a little more delicate because of all the parameterization involved, but the idea is easy to understand intuitively.

The reason we must be careful in stating the rule is that the two classes involved, here *C* and *B*, may have different formal generic parameters: different in role, number or both. For example, given the above declaration for *B* and



```
class C [P –> DT, Q] inherit
    B [TK, P, TM]
    …
end
```

The constraining type DT plays no role in this example.

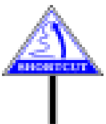
we will want the type *CT* defined as

```
C [TL, TN]
```

to conform to the type *BT* defined above as

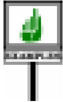
```
B [TK, TL, TM]
```

even though the number of generic parameters is different for each class. Why *CT* should conform to *BT* is intuitively clear: if we interpret the text of *C* for the actual generic parameters *TL*, corresponding to *P*, and *TN*, corresponding to *Q*, the Parent *B* [*TK*, *P*, *TM*] listed in its Inheritance clause really stands for *B* [*TK*, *TL*, *TM*], which is precisely *BT*.



On first reading, if you find this example sufficient to give an intuitive understanding of conformance in such cases, you may wish to skip to the next section.

As the example shows, we will need to use substitutions (of actual to formal generic parameters) to ascertain direct conformance rigorously. If {*x1*, … , *xn*} and {*y1*, … , *yn*} are sets with the same number of elements, a substitution from the first set to the second is a one-to-one correspondence between them, associating a different element of the second to every element of the first. For example, a substitution σ (among six possible ones) from {*T*, *U*, *V*} to {*L*, *M*, *N*} is given by the table



σ maps:	to:
T	*M*
U	*N*
V	*L*

The number of possible substitutions between two sets of n *elements is the factorial of* n, *here 3*! = *6.*

-----The notion of substitution serves to specify actual-formal correspondence rigorously.

As in the non-generic case, you can **disable** conformance of *CT* to *BT* even in the presence an inheritance link by using non-conforming inheritance.

← *"NON-CONFORMING INHERITANCE", 6.8, page 178. The observation for non-generic reference types was in 14.6, page 381.*

To see that the rule is in fact easy to apply, let us use it to check that the type *CT* defined in the above example as *C* [*TL*, *TN*] indeed conforms to *BT*, defined as *B* [*TK*, *TL*, *TM*]. The assumption is that *B* is declared as *B* [*G*, *H*, *I*], with three formal parameters, and that *C* [*P*, *Q*], with two formal parameters, lists *B* [*TK*, *P*, *TM*] as Parent.

The application is straightforward. Here *n* = 3 and *m* = 2; the types and Formal_generic names appearing in the definition are:

X1 : *TK*	*X2* : *TL*	*X3* : *TM*
G1 : *G*	*G2* : *H*	*G3* : *I*
Y1 : *TL*	*Y2* : *TN*	
H1 : *P*	*H2* : *Q*	
Z1 : *TK*	*Z2* : *P*	*Z3* : *TM*

The substitution σ associates *Y1* to *H1* and *Y2* to *H2*. In other words, it defines the associations

σ maps:	to:
P	*TL*
Q	*TN*

and leaves other elements unchanged. So applying σ to the *Zj* yields

σ maps:	to:	Comments
Z_1	*TK*	σ leaves Z_1, i.e. *TK*, unchanged.

Z_2	*TL*	This is the result of applying σ to Z_2, i.e. *P*.
Z_3	*TM*	σ leaves Z_3, i.e. *TM*, unchanged.

The three resulting types *TK*, *TL* and *TM* are indeed, respectively, *X1*, *X2* and *X3*, showing that *C* [*TL, TN*] does conform directly to *B* [*TK, TL, TM*].

To show that *C* [*TL, TN*] conforms to *B* [*SK, SL, SM*] if *TK* conforms to *SK*, *TL* to *SL* and *TM* to *SM*, we would first use the Generic Substitution rule, as was just done, to show conformance to *B* [*TK, TL, TM*], and then apply case 4 of the General Conformance rule to obtain the required actual generic parameters.

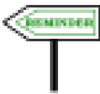
14.8 FORMAL GENERIC PARAMETER CONFORMANCE

The next case is a Formal_generic_name type: a formal generic parameter to the enclosing class. In the text of a generic class

This assumes a single constraints. Multiple constraints are addressed below.

```
class C [G, H –> CT,…] … end
```

you may use the formal generic parameters *G* and *H* as types. *G* and *H* illustrate the two kinds of generic parameters, placing different requirements on the types to be used as the corresponding actual generic parameters: *G*, unconstrained, stands for arbitrary types; *H*, constrained by *CT*, stands for types that conform to *CT*.

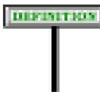


As noted in the discussion of genericity, the base type of a constrained Formal_generic_name such as *H* is the constraining type, here *CT*. An unconstrained generic Formal_generic_name such as *G* is considered to be constrained by the universal class *ANY*, which serves as its base type.

←12.3, page 343 presented unconstrained, and 12.6, page 346 constrained, genericity. 12.10, page 355 addressed the use of generic parameters as types, defining their base type on page 356 (page 361 for the multi-constraint case).

In both cases, the Formal_generic_name will conform directly to its constraining type (*CT* or *ANY*). In the reverse direction, however, no direct conformance is possible: if we allowed assigning to an entity of type *G* or *H* an expression of a different type, we would have no way of guaranteeing that this type is compatible for every possible actual generic parameter.

The rule for conformance to and from generic parameters follows from these observations:

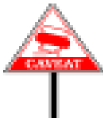


Direct conformance: formal generic *VNCF*

Let *G* be a formal generic parameter of a class *C*, which in the text of *C* may be used as a Formal_generic_name type. Then:

1 • No type **conforms directly** to *G*.

2 • *G* **conforms directly** to every type listed in its constraint, and to no other type.



Remember from the general definition of conformance that every type conforms (not directly) to itself.

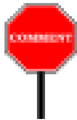
← Case 1 of General Conformance rule, page380.

The last clause of the rule mentions "one or more" constraining types. This is because we allow more than one constraint, as in

← "THE CASE OF MULTIPLE CONSTRAINTS", 12.13, page 359.

```
class D [G –> {CONST1, CONST2, CONST3}] …
```

where *G* will conform to every one of *CONST1*, *CONST2*, *CONST3*. As noted in the discussion of genericity, this occurs only rarely; most uses of constrained genericity limit themselves to one constraint as with *C* above.



In the case of recursive generic constraints, as in

← "RECURSIVE GENERIC CONSTRAINTS", 12.9, page 354.

```
class C [G, H –> ARRAY [G]] …
```

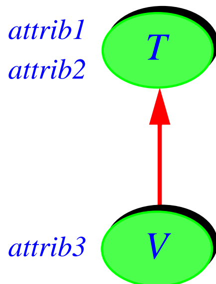
the rule is applicable without any need for a special clause: within the text of *C*, *H* represents a type that only conforms directly to *ARRAY* [*G*]. This corresponds to the property, ensured by the Constrained Genericity rule, that in a generic derivation *C* [*T*, *U*] the type *U* must conform to *ARRAY* [*T*].

← Page 351

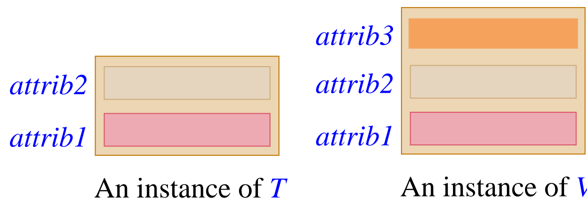
14.9 EXPANDED TYPE CONFORMANCE

---------- TO BE REWRITTEN -----------

.



Two classes



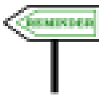
Typical instances

Cannot copy bigger object onto smaller one.

What about the reverse direction — conformance of an expanded type *ET* to a reference type *RT*? Here there is no implementation constraint since it is always physically possible to reattach a reference to an object of arbitrary size. But of course the attachment must be compatible with the type system: the base type of *ET* must conform to *RT*.

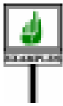
	(Case 1)	(Case 2)
BEFORE	y (OBJ1) with ref semantics (Case 1)	y (OBJ1) with copy semantics (Case 2)
AFTER	x y (OBJ1) with ref semantics	x (OBJ2) with copy semantics y (OBJ1) with copy semantics

These are the essential ideas. Now the details.



First let's review the two forms of expanded type. Examples of the first are ---- REMOVED ---------

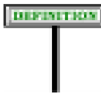
Examples of the second case are



```
A
B [X, Y, Z]
```

where *A* and *B* have been declared as **expanded class** (*A* non-generic, *B* generic). These types are their own base types; the base classes are *A* and *B*. The basic types *BOOLEAN*, *CHARACTER*, *INTEGER*, *REAL* and *POINTER* fall into this category.

---- REMOVED ------------

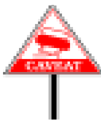


Direct conformance: expanded types *VNCE*

No type **conforms directly** to an expanded type.

→ .

From the definition of general conformance, an expanded type *ET* still *conforms*, of course, to itself. *ET* may also conform to reference types as allowed by the corresponding rule (*VNCN*); the corresponding assignments will use copy semantics. But no other type (except, per General Conformance, for *e* of type *ET*, the type **like** *e*, an abbreviation for *ET*) conforms to *ET*.



This rule might seem to preclude mixed-type operations of the kind widely accepted for basic types, such as *f (3)* where the routine *f* has a formal argument of type *REAL*, or *your_integer_64 := your_integer_16* with a target of type *INTEGER_64* and a source of type *INTEGER_16*. Such attachments, however, involve **conversion** from one type to another. What makes them valid is not conformance but **convertibility**, which does support a broad range of safe mixed-type assignments.

→ *Chapter 15*.

14.10 TUPLE TYPE CONFORMANCE

Next consider tuple types. A tuple type is of the form

$TUPLE\ [label_1: T_1; \ldots; label_n: T_n]$

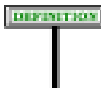
where: the part in square brackets may be absent, giving the most general tuple type, *TUPLE*; the *Ti*, if present, are types, called the "**parameters**" of the tuple type; and any of the label parts *label_i*: may be absent. In fact, the labels will play no role in the conformance rules: *TUPLE* [*A*], *TUPLE* [*x*: *A*] and *TUPLE* [*y*: *A*] are all equivalent for conformance purposes. We saw that the only role of the labels is to define assignable attributes in the corresponding types.

← *Chapter 13 overed tuples.*

There are two sources of conformance for tuple types:

- A tuple type conforms to any other having the same initial sequence of parameters T_1, …, T_n, regardless of the labels present or not in either one.
- In addition, a special rule relates tuples to arrays. If we have a tuple expression, especially in the form of a manifest tuple [*x1*, …, *xn*], it is useful to treat it as an array. This will provide the equivalent of manifest arrays — arrays defined by a list of their items — and is permitted by a rule stating that a tuple type conforms to *ARRAY* [*T*] if all of its parameters conform to *T*.

The conformance rule for tuple types follows



Direct conformance: tuple types *VNCT*

A Tuple_type *U*, of type sequence *us*, **conforms directly** to a type *T* if and only if *T* satisfies the following conditions:

1 • *T* is a tuple type, of type sequence *ts*.

2 • The length of *us* is greater than or equal to the length of *ts*.

3 • For every element *X* of *ts*, the corresponding element of *us* is identical to *X* or, if *X* is not specified **frozen**, conforms to *X*.

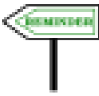
No type conforms directly to a tuple type except as implied by these conditions.

Labels, if present, play no part in the conformance.

----FOLLOWING NOT TRUE, REPLACE BY DISCUSSION OF CONVERTIBIITY ---- allows tuples to be treated as arrays. So if every one of the types of *x1*, *x2*, …, *xn* conforms to *T*:

- You can write an assignment *a* := [*x1*, *x2*, …, *xn*] where *a* is of type *ARRAY* [*T*]. This provides a simple means of array initialization, as in *ia* := [*1*, *2*, *3*] for *ia* of type *ARRAY* [*INTEGER*].
- You can write a call *some_routine* ([*x1*, *x2*, …, *xn*]) where the corresponding formal argument in *some_routine* is of type *ARRAY* [*T*].

14.11 ANCHORED TYPE CONFORMANCE



Anchored types greatly simplify, as you will remember, the management of groups of entities that must keep the same type in redeclarations. An anchored type is of the form

← "ANCHORED TYPES", 11.10, page 331.

like *anchor*

where *anchor* is the name of an attribute, function or formal argument, or *Current*. Such a declaration describes a type which is the same as the type of *anchor* but will automatically follow any redefinition of the type of *anchor* in a proper descendant. Using *Current* as anchor means that the type will be the current type (class name with generic parameters if any).

← "CURRENT TYPE, FEATURES OF A TYPE", 12.11, page 357.

There is no need for a special conformance rule, as the last clause of the General Conformance rule already told us that, for the purpose of conformance, we should simply look at the "deanchored form" of an anchored type.