Genericity

12.1 OVERVIEW

The types discussed so far were directly defined by classes. The *genericity* mechanism, still based on classes, gives us a new level of flexibility through **type parameterization**. You may for example define a class as *LIST* [*G*], yielding not just one type but many: *LIST* [*INTEGER*], *LIST* [*AIRPLANE*] and so on, parameterized by *G*.

Parameterized classes such as *LIST* are known as **generic classes**; the resulting types, such as *LIST* [*INTEGER*], are **generically derived**. "Genericity" is the mechanism making generic classes and generic derivations possible.

Two forms of genericity are available: with *unconstrained* genericity, *G* represents an arbitrary type; with *constrained* genericity, you can demand certain properties of the types represented by *G*, enabling you to do more with *G* in the class text.

The discussion of generically derived types will proceed as with other kinds of type in the previous chapter: to define the semantics of a type, it suffices to say whether it is reference or expanded, and to define its **base type**, always a Class_or_tuple_type. If the type is itself a Class_or_tuple_type, we must also define its **base class**, which determines its instances. For example *LIST* [*INTEGER*] has *LIST* as its base class, and is its own base type.

A subsequent chapter discusses the related *conformance* properties. → *Chapter 14.*

12.2 GENERIC CLASSES

To obtain generically derived types, we start from **generic classes** such as *LIST*, with one or more *formal generic parameters* such as *G*.

Generic classes describe flexible structures having variants parameterized by types. Often these are **container data structures**, used to gather objects of various possible types; examples include lists, stacks, arrays and the like, which contain objects of arbitrary type. The generic parameters of such classes specify the types of objects to be kept in the container structures, such as the elements of an array.

Container data structures were mentioned in 10.8, page 262.

The following examples from EiffelBase show beginnings (Class_header followed by Formal_generics) of classes with unconstrained generic parameters:

```
deferred class TREE [G] ...
class LINKED_LIST [G] ...
class ARRAY [G] ...
```

In each case, *G* is a **formal generic parameter** of the class, representing the types of objects to be kept in an instance of the class – a tree, a linked stack, an array. Classes may have more than one formal generic parameter; the next section will give an example with two parameters.

To derive a type from a generic class — called the **base class** of the derivation — you must provide a type, called an **actual generic parameter**, for each of the formal generic parameters of the base class. This will yield a **generically derived** type. The derived type is expanded if the base class is expanded, a reference type otherwise. Generic derivation, applied to the above base classes, will yield types such as

```
TREE [INTEGER]
TREE [PARAGRAPH]
LINKED_LIST [PARAGRAPH]]
TREE [TREE [PARAGRAPH]]
ARRAY [LINKED_LIST [TREE [LINKED_LIST [PARAGRAPH]]]]
```

Instances of the first type represent trees of integers; instances of the second one represent trees of paragraphs (that is to say, trees of instances of the reference type *PARAGRAPH*); and so on. The base classes are, respectively, *TREE, TREE, LINKED_LIST, TREE* and *ARRAY*.

Since all these base classes have exactly one formal generic parameter, each of the above generically derived types is obtained by providing one actual generic parameter. The actual generic parameters are *INTEGER* for the first example, *PARAGRAPH* for the second and third, *TREE* [*TREE* [*PARAGRAPH*]] for the fourth, *LINKED_LIST* [*TREE* [*LINKED_LIST* [*PARAGRAPH*]]] for the last.

The actual generic parameter is a type; it may itself be generically derived; the last two examples illustrate this possibility, which leads to nested genericity without any limit on the depth of nesting.

In the syntax specification, the place where all this appears is the Class_type construct, introduced earlier as

Class_type ≜ Class_name [Actual_generics]

← *This was in the previous chapter, page 320.*

Actual_generics hasn't been defined until now. Here it is:

Actual generic parameters

Actual_generics ≜ "**[**" Type_list "**]**"
Type_list ≜ {Type "**,**" …}$^{+}$

12.3 GENERIC CLASSES AND GENERIC DERIVATIONS

The construct that makes a class generic is Formal_generics, optionally appearing after the Class_header of a Class_declaration, with this structure:

← Class_declaration *was given in chapter 4, which on page 128 previewed the syntax shown here.*

Formal generic parameters

Formal_generics ≜ "[" Formal_generic_list "]"

Formal_generic_list ≜ {Formal_generic ","…}+

Formal_generic ≜ [**frozen**] Formal_generic_name [Constraint]

Formal_generic_name ≜ [**?**] Identifier

and a straightforward validity constraint:

Formal Generic rule *VCFG*

A Formal_generics part of a Class_declaration is valid if and only if every Formal_generic_name *G* in its Formal_generic_list satisfies the following conditions:

1 • *G* is different from the name of any class in the universe.

2 • *G* is different from any other Formal_generic_name appearing in the same Formal_generics_part.

→ *A rule also applies to the* Constraint *part*: *"Generic Constraint rule", page 349.*

Adding the **frozen** qualification to a formal generic, as in *D* [**frozen** *G*] rather than just *C* [*G*], means that conformance on the corresponding generically derived classes requires identical actual parameters: whereas *C* [*U*] conforms to *C* [*T*] if *U* conforms to *T*, *D* [*U*] does not conform to *D* [*T*] if *U* is not *T*.

Adding the **?** mark to a Formal_generic_name, as in **?** *G*, means that the class may declare *self-initializing* variables (variables that will be initialized automatically on first use) of type *G*; this requires that any actual generic parameter that is an attached type must also be self-initializing, that is to say, make *default_create* from *ANY* available for creation.

The optional Constraint part of the form –> *CONSTRAINING_TYPE* puts a requirement on acceptable actual generic parameters: they must conform to the *CONSTRAINING_TYPE*. If it is not present, any type will do.

Let's make the basic terminology precise:

Generic class; constrained, unconstrained

Any class declared with a Formal_generics part (constrained or not) is a **generic class**.

If a formal generic parameter of a generic class is declared with a Constraint, the parameter is **constrained**; if not, it is **unconstrained**.

A generic class is itself **constrained** if it has at least one constrained parameter, **unconstrained** otherwise.

A generic class does not describe a type but a template for a set of possible types. To obtain an actual type, you must provide an Actual_generics list, whose elements are themselves types. This has a name too, per the following definition.

Generic derivation, non-generic type

The process of producing a type from a generic class by providing actual generic parameters is **generic derivation**.

A type resulting from a generic derivation is a **generically derived type**, or just **generic type**.

A type that is not generically derived is a **non-generic type**.

It is preferable to stay away from the term "generic instantiation" (sometimes used in place of "generic derivation") as it creates a risk of confusion with the normal meaning of "instantiation" in object-oriented development: the *run-time* process of obtaining an object from a class.

Among the above examples, *PARAGRAPH* is non-generic, as a class and as a type (any non-generic class is also a type). *LINKED_LIST*, *TREE* and *ARRAY* are generic classes; to produce a generic derivation from one of them, you choose a suitable actual generic parameter, and get a generically derived type such as *LINKED_LIST* [*INTEGER*].

The expansion status of a generically derived type *T* follows from its base class, independently of the actual generic parameters: *T* is expanded if its base class is an expanded class; otherwise it is a reference type.

12.4 SELF-INITIALIZING FORMALS

---- EXPLAIN

Self-initializing formal

A Formal_generic_parameter is **self-initializing** if and only if its declaration includes the optional **?** mark.

This is related to the notion of self-initializing *type*: a type which makes *default_create* from *ANY* available for creation. The rule will be that an actual generic parameter corresponding to a self-initializing formal must itself, if attached, be a self-initializing type.

12.5 CONSTRAINED AND UNCONSTRAINED GENERICITY

If a formal generic parameter is constrained, appearing as *G* –> *T*, the constraint *T* determines what operations are applicable, in the class to an entity of type *G*: the features of *T*. This will be formalized very simply by defining the base type of *G*, in this case, as being *T*.

In the case of unconstrained genericity, we don't know anything about future actual generic parameters: in *C* [*G*], *G* can represent any type. The only operations that we can apply in this case are those of *ANY*, since we know every Eiffel class conforms to *ANY*. We'll in fact allow these operations, and treat *G* as if it were constrained by *ANY*.

← *"Universal Conformance principle", page 173*.

This observation allows us to simplify the validity and semantics by treating in the same way all formal generic parameters, constrained and unconstrained, thanks to the following convention, which also allows us in every case to talk about "the constraint" of a formal generic parameter:

Constraint, constraining types of a Formal_generic

The **constraint** of a formal generic parameter is its Constraint part if present, and otherwise *ANY*.

Its **constraining types** are all the types listed in its Constraining_types if present, and otherwise just *ANY*.

--- For the language description, it's convenient to avoid treating

A straightforward constraint applies to unconstrained generic derivations: a generically derived type of the form *C* [*T*, …], where *C* does not declare any constraints for its generic parameters if any, is valid if and only if:

- *C* is indeed a generic class.
- The number of Type components *T*, … in the Actual_generics list is the same as the number of Formal_generic parameters in the Formal_generic_list of *C*'s declaration.

This property does not appear as a separate validity constraint since, thanks to the ----- REWRITE notion of unfolded form, it will follow as a special case of the validity rule for the constrained case, where we treat unconstrained genericity as constrained by *ANY*.

This is a "constraint" on "unconstrained" genericity. Sometimes language meets metalanguage.

12.6 CONSTRAINED GENERICITY

In the above unconstrained examples of genericity, any type was acceptable as actual generic parameter; this is because we do not require any special property of the objects to be entered into an array, inserted into a tree or pushed onto a stack. As long as operations applicable to all objects (such as assignment, copying or equality testing) are available, we can write the generic class, for example *TREE* [*T*], without any specific knowledge about the actual types to be used for *T*.

In some cases, however, you will need a guarantee that these types possess specific properties, so that the class text may apply certain operations to the corresponding objects. A typical example is a generic class *VECTOR* [*T* …] describing vectors, which must support an addition operation. To add two vectors, you need the ability to add two vector elements; in other words, you need an addition operation on *T*. Then *T* cannot be an arbitrary type.

With constrained genericity, you can guarantee that *T* supports addition, by requiring any actual generic parameter for *T* to be based on a descendant of a class that includes an addition routine. The class will appear as

```
class VECTOR [G –> NUMERIC]…
```

in reference to class *NUMERIC* of the Kernel Library, describing numerical values and having among others a feature *plus* **alias** "+" representing addition. Numerical classes such as *INTEGER* and *REAL* are descendants of *NUMERIC*, as will be any class that you want to declare as providing a number or number-like facility (for example class *VECTOR* itself). The Constraint part *–> NUMERIC* indicates that a generic derivation *VECTOR* [*SOME_TYPE*] will be valid if and only if *SOME_TYPE* conforms to *NUMERIC*. So you may use *VECTOR* [*INTEGER*], *VECTOR* [*REAL*], or even *VECTOR* [*VECTOR* [*INTEGER*]] if you have made VECTOR itself inherit from NUMERIC, but not *VECTOR* [*PARAGRAPH*] if class *PARAGRAPH* is not a descendant of *NUMERIC*.

Class *HASH_TABLE* of EiffelBase provides another example of constrained generic class. This class describes tables of elements, retrievable through associated keys. Its text begins with

```
class HASH_TABLE [G; KEY –> HASHABLE]...
```

The class has two generic parameters. The first one, *G*, plays the same role as those encountered in the previous section; it stands for the type of table elements, and is unconstrained. The second one, *KEY*, is constrained by the Kernel Library class *HASHABLE*.

The constraint means that the base class of any actual generic parameter used for *KEY* must be a descendant of the constraining class, *HASHABLE*. *HASHABLE* is a simple Kernel Library class introducing a function

The –> symbol is reminiscent of the arrow used in inheritance diagrams.

```
hash_code: INTEGER
        --Hash_code value
    deferred
    end
```

In other words, keys must be "hashable" into integer values. An example of a class that inherits from *HASHABLE* is the Kernel Library class *STRING*, describing character strings, for which a standard *hash_code* function is provided. An example of a type generically derived from *HASH_TABLE* is

```
HASH_TABLE [PARAGRAPH; STRING]
```

As illustrated by these examples, the basic syntax for constrained formal generic parameters includes, after the parameter, a Constraint of the form


```
–> Class_or_tuple_type
```

The effect of such a Constraint, if present, is to restrict allowable actual generic parameters to types that conform to the given Class_or_tuple_type.

Recall that a type *C* conforms to a type *B* if the base class of *C* is a descendant of the base class of *B*; also, if *C* is generically derived, its actual generic parameters must (recursively) conform to those of *B*. In the *HASH_TABLE* case conformance is ensured by the property that *STRING*, as specified by the Kernel Library, inherits from *HASHABLE*.

The next chapter covers conformance.

Two supplementary facilities are available:

- You can require certain creation procedures.
- You can use multiple constraints.

→ *See "CREATING INSTANCES OF FORMAL GENERICS", 20.9, page 535 for a full discussion.*

The first of these enables you to write something like

```
class D [G –> CONST create cp1, cp2, … end] …
```

where *cp1*, *cp2*, … must be procedures of type *CONST*. The purpose — as explained in detail in the chapter on creation — is to allow creation of objects of type *G*: within the class, with *x* declared of type *G*, you can use a creation instruction of the form **create** *x*.*cp1* (*actuals*) and similarly for the other listed procedures, assuming valid *actuals* arguments. A generic derivation *D* [*T*] will then require *T* to declare its versions of *cp1*, *cp2*, … as creation procedures. In *CONST* itself, *cp1*, *cp2*, … must be procedures, but they do not have to be *creation* procedures, since what matters is to be able to use them to create instances of actual generic parameters such as *T*.

The second facility enables you to specify multiple constraints, as in

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

meaning that any actual parameter *T* in a generic derivation *D* [*T*] must conform to **all** of *CONST1*, *CONST2*, *CONST3*.

It is in fact possible to combine both of these two facilities, as in

```
class D [G –> {CONST1; CONST2 create cp1, cp2, … end}] …
```

More generally, as the rest of this chapter will show, multiple constraints significantly complicate the syntax and validity of generic constraints, as well as the definition of the base type for formal generic parameters. This is a typical "borderline" facility, whose presence in the language is subject to criticism.

Most developments do not need it, but there are cases, mostly involving libraries, when working without multiple constraints would make things awkward. When you have control over all classes involved, you can in principle get away with single constraints only, achieving the effect of *D* [*G* –> {*CONST1*, *CONST2*}] by using *D* [*G* –> {*CONST12*}], after writing a class *CONST12* that inherits from *CONST1* and *CONST2*. But this doesn't work with pre-existing classes, especially those from the Kernel Library and other fundamental libraries: with multiple constraints you can write *D* [*G* –> {*COMPARABLE*, *NUMERIC*}], which will accept *INTEGER* or *REAL* as an actual generic parameter; but defining a class *COMPARABLE_NUMERIC* that inherits from *COMPARABLE* and *NUMERIC* won't help you since *INTEGER*, *REAL* and the like do not know it. And you cannot define new classes — *NUMERIC_HASHABLE* and so on — for every potentially useful combination.

So even though multiple constraints are useful for only a minority of cases and users, they are *very* desirable to these users for those cases, explaining why the language supports them in spite of the added complication.

12.7 RULES ON CONSTRAINED GENERICITY

Now for the precise syntax and validity of constrained genericity. The construct that remains to be specified is Constraint:

Generic constraints

Constraint ≜ "–>" Constraining_types
[Constraint_creators]

Constraining_types ≜ Single_constraint | Multiple_constraint

Single_constraint ≜ Type [Renaming]

Renaming ≜ Rename **end**

Multiple_constraint ≜ "**{**" Constraint_list "**}**"

Constraint_list ≜ {Single_constraint "**,**" …}$^{+}$

Constraint_creators ≜ **create** Feature_list **end**

There are two validity rules. One governs the Constraint part of the declaration of a constrained generic class; the other, complementing the Unconstrained Genericity rule, governs the validity of a type derived from In addition, of course, the base class must exist in the universe; this is a consequence of the Class Type rule. First:

Generic Constraint rule *VTGC*

A Constraint part appearing in the Formal_generics part of a class *C* is valid if and only if it satisfies the following conditions for every Single_constraint listing a type *T* in its Constraining_types:

1 • *T* involves no anchored type.

2 • If a Renaming clause **rename** *rename_list* **end** is present, a class definition of the form **class** *NEW* **inherit** *T* **rename** *rename_list* **end** (preceded by **deferred** if the base class of *T* is deferred) would be valid.

This is a validity "constraint" on the generic "constraints" of Eiffel classes. Sometimes language meets metalanguage.

There is no requirement here on the Constraint_creators part, although in most cases it will list names (after Renaming) of creation procedures of the constraining types. The precise requirement is captured by other rules.

Condition 2 implies that the features listed in the Constraint_creators are, after possible Renaming, names of features of one or more of the constraining types, and that no clash remains that would violated the rules on inheritance. In particular, you can use the Renaming either to merge features if they come from the same seeds, or (the other way around) separate them.

If *T* is based on a deferred class the fictitious class *NEW* should be declared as **deferred** too, otherwise it would be invalid if *T* has deferred features. On the other hand, *NEW* cannot be valid if *T* is based on a frozen class; in this case it is indeed desirable to disallow the use of *T* as a constraint, since the purpose of declaring a class **frozen** is to prevent inheritance from it

The last observation suggests a name for the resulting features:

Constraining creation features

If *G* is a formal generic parameter of a class, the **constraining creators of** *G* are the features of *G*'s Constraining_types, if any, corresponding after possible Renaming to the feature names listed in the Constraining_creators if present.

Constraining creators should be creation procedures, but not necessarily (as seen below) in the constraining types themselves; only their instantiatable descendants are subject to this rule.

The usual situation, with a declaration involving a Constraint_creators

class *D* [*G* –> *CONST* **create** *cp1, cp2,* … **end**] …

the names listed, *cp1, cp2,* …, should denote procedures of *CONST*. They don't have to be *creation* procedures of *CONST* — this will be required only in the actual generic parameter, as stated by the next rule — and can in fact be deferred, but they have to be known procedures of *CONST* so that we can assess the validity of a creation call such as **create** *x*.*cp1* (*actuals*), especially the validity of the chosen *actuals*.

--- EXPLAIN CLAUSE 4 (INCLUDING CASE OF TUPLES ---

Next, the rule for generically derived types. The two properties already cited for the unconstrained case still apply: the number and types of actual parameters must match those of the formal parameters. In addition:

- Wherever a formal generic parameter is constrained, the corresponding actual parameter must conform to the constraining type or types.

- If there is a Constraint_creators part requiring some creation procedures, these must indeed be creation procedures in the actual generic parameter.

Here is the precise formulation:

Generic Derivation rule *VTGD*

Let *C* be a generic class. A Class_type *CT* having *C* as base class is valid if and only if it satisfies the following conditions for every actual generic parameter *T* and every Single_constraint *U* appearing in the constraint for the corresponding formal generic parameter *G*:

1 • The number of Type components in *CT*'s Actual_generics list is the same as the number of Formal_generic parameters in the Formal_generic_list of *C*'s declaration.

2 • *T* conforms to the type obtained by applying to *U* the generic substitution of *CT*.

3 • If *C* is expanded, *CT* is generic-creation-ready.

4 • If *G* is a self-initializing formal and *T* is attached, then *T* is a self-initializing type.

In the case of unconstrained generic parameters, only condition 1 applies, since the constraint in that case is *ANY*, which trivially satisfies the other two conditions.

Condition 3 follows from the semantic rule permitting "lazy" creation of entities of expanded types on first use, through *default_create*. Generic-creation-readiness (defined next) is a condition on the actual generic parameters that makes such initialization safe if it may involve creation of objects whose type is the corresponding formal parameters.

→ *Page 352.*

Condition 4 guarantees that if *C* relies, for some of its variables of type *G*, on automatic initialization on first use, *T* provides it, if attached (remember that this includes the case of expanded types), by making *default_create* from *ANY* available for creation. If *T* is detachable this is not needed, since *Void* will be a suitable initialization value.

At first the phrasing of clause 2 seems more complicated than necessary: why must the actual generic parameter conform not just to *D* but to "the type obtained by applying to *U* by ..."? This is to permit *recursive generic constraints*, as detailed next. For most practical cases, however, you can understand clause 2 as if it just read

The role of this condition is to make sure that if the operations of a class *C* [..., *G*, ...] may include creations on targets of the formal generic type *G*, any associated actual generic parameter *T* will support such creations. The basic

"*T conforms to the constraining type*"

12.8 CONSTRAINTS AND CREATION

Consider a formal generic parameter *G*; under what conditions can we create an object of type *G*, for example through an instruction **create** *x*.*make* (…) with *x* of type *G* in one of the routines of the class? Including a Constraint_creators enables you to specify the applicable creation procedures for *G*, as in *G* –> *CONST* **create** *make* **end**. The corresponding actual generic parameters will then have to provide the listed features, here *make*, as creation procedures when needed. This is not a constraint on all generic derivations, however; only on those raising the possibility of a creation on the corresponding parameter. So at this stage we don't have a constraint, just a definition:

Generic-creation-ready type

A type is **generic-creation-ready** if and only if every actual generic parameter *T* of its deanchored form satisfies the following conditions:

1 • If the specification of the corresponding formal generic parameter includes a Constraint_creators, the versions in *T* of the constraining creators for the corresponding formal parameter are creation procedures, and *T* is (recursively) generic-creation-ready.

2 • If *T* is expanded, it is (recursively) generic-creation-ready.

Although phrased so that it is applicable to any type, the condition is only interesting for generically derived types of the form *C* [*…, T, …*]. Non-generically-derived types satisfy it trivially since there is no applicable *T*.

The role of this condition is to make sure that if class *C* [*…, G , …*] may cause a creation operation on a target of type *G* — as permitted only if the class appears as *C* [*…, G –> CONST* **create** *cp1, …* **end**, *…*] — then the corresponding actual parameters, such as *T*, will support the given features — the "constraining creators" — as creation procedures.

It might then appear that generic-creation-readiness is a validity requirement on *any* actual generic parameter. But this would be more restrictive than we need. For example *T* might be a deferred type; then it cannot have any creation procedures, but that's still OK because we cannot create instances of *T*, only of its effective descendants. Only if it is possible to **create** an actual object of the type do we require generic-creation-readiness. Overall, we need generic-creation-readiness only in specific cases, including:

- For the creation type of a creation operation: conditions 4 of the Creation Instruction rule and 3 of the Creation Expression rule. → *Pages 545 and 553.*
- For a Parent in an Inheritance part: condition 6 of the Parent rule. ← *Page 176.*
- For an expanded type: condition 3 of the just seen Generic Derivation rule. ← *Page 351.*

--------- NEXT SECTIONS OBSOLETE

Clause --- covers the case of a Constraint including a Constraint_creators and complements the preceding rule (Generic Constraint). In

class *D* [*G –> CONST* **create** *cp1, cp2, …* **end**] …

the Generic Constraint rule required *cp1, cp2, …* to be procedures of *CONST*. In a generic derivation *D* [*T*], T must be, as per clause 2, a type conforming to *CONST*; in addition, clause --- tells us that *T* must make sure to specify *cp1, cp2, …* as creation procedures. (As a consequence, they cannot for example be deferred.)

Both of these clauses apply to every constraining type in the case of a multiple constraint *D* [*G –> {CONST1, CONST2}*]

No specific validity rule applies to the generic constraints themselves (*CONST*, *CONST1*, *CONST2*). A generic constraint must simply be a valid type. It might even involve a generic parameter, or even *be* a generic parameter; this is the case of "recursive generic constraints", the topic of the next section.

12.9 RECURSIVE GENERIC CONSTRAINTS

(The case described in this section does not arise in elementary uses, and may be skipped in a first reading.)

To understand the last part of clause 2 of the Constrained Genericity rule, assume you want to define a class as

```
class C [G; H –> ARRAY [G]] ...
```

This makes perfect sense and the intent is clear: you want to allow any type of the form *C* [*T, U*] where *T* is an arbitrary type and *U* is *ARRAY* [*T*] or a type conforming to *ARRAY* [*T*]. So the following will be valid

The Class Type rule appeared on page 325.

```
C [INTEGER; ARRAY [INTEGER]]
C [POLYGON; ARRAYED_LIST [POLYGON]]
      -- Where ARRAYED_LIST is a descendant of ARRAY
```

But for example *C* [*INTEGER, REAL*] is not valid. Similarly, you should be able to define

```
class C [G –>H; H –> G] ...
```

meaning: the first actual generic parameter must conform to the first, and conversely. Only derivations of the form *C* [*T, T*], using the same type as actual generic parameter, will be valid. Unlike the first example, this scheme seems useless, but there is no reason to disallow it.

This explains the phrasing of clause 2 of the Constrained Genericity rule. The simpler phrasing

> "*T conforms to the constraining type*"

is appropriate in ordinary, non-recursive cases; but in our first example *ARRAY* [*INTEGER*] does not conform to *ARRAY* [*G*]; actually this conformance question is meaningless since there usually won't even be a type *G* in the class that wants to use *C* [*INTEGER; ARRAY* [*INTEGER*]]. Similarly, in the *C* [*G –>H; H –> G*] example, if we want to use *C* [*T, T*] in a certain class other than *C*, the questions "does *T* conform to *G*?" and "does *T* conform to *H*?" are meaningless in that class.

For such conformance questions to become meaningful, we must first replace, in the constraint, any occurrence of a formal parameter by the corresponding actual parameter. Hence the rephrased clause:

> "*T conforms to the type obtained from the constraining type by replacing every ocurrence of a formal generic parameter of C by the corresponding actual generic parameter in CT.*"

12.10 SEMANTICS OF GENERIC TYPES

We must now define the semantics of types involving genericity. This includes both generically derived types and Formal_generic_name (covering the formal generic parameters themselves, when used as types within the class text).

As noted in the previous chapter, defining the semantics of a type involves saying whether it is expanded or reference, and specifying its base type, as well as its base class if it is a Class_type.

← "Type Semantics rule", page 325.

For a generically derived Class_type the definition is an immediate generalization of the non-generic case:

← "Non-generic class type semantics", page 327.

Generically derived class type semantics

A generically derived Class_type of the form *C* [...], where *C* is a generic class, is expanded if *C* is an expanded class, reference otherwise. It is its own base type, and its base class is *C*.

So *LINKED_LIST* [*POLYGON*] is its own base type, and its base class is *LINKED_LIST*.

The other case is formal generics. In a generic class *C* [..., *G*, ...], a formal parameter *G*, constrained or unconstrained (syntactically known as a Formal_generic_name), stands for any type to be provided as actual parameter in generic derivations of the class. Within the text of *C*, you may use *G* wherever the syntax requires a type.

For example, the EiffelBase text of

```
class HASH_TABLE [G; KEY –> HASHABLE]...
```

declares a number of features using *G* or *KEY* as type of an argument, result or local variable. Typical is the function

```
item (access_key: KEY): G
            -- Item associated with access_key, if present;
            -- otherwise default value of type G.
        do... Routine body omitted ... end
```

The type of the function result in the actual class is not exactly G but **like** *last_put', where 'last_put' is an attribute of type G. See 11.10, page 331 below, on such "anchored" types.*

which uses both of the formal generic parameters as Formal_generic_name types.

It is in fact easier to start with the constrained case. For a constrained parameter such as *KEY*, the only available information is provided by the constraining type, here *HASHABLE*; the features of that type's base class are the only operations that we know can be applied to entities of the Formal_generic_name type. The rule follows, applicable to the case of a single constraint (the next section will address multiple constraints):

Base type of a single-constrained formal generic

The base type of a constrained Formal_generic_name *G* having as its constraining types a Single_constraint listing a type *T* is:

1 • If *T* is a Class_or_tuple_type: *T*.

2 • Otherwise (*T* is a Formal_generic_name): the base type of *T* if it can be determined by (recursively) case 1, otherwise *ANY*.

The definition is never cyclic since the only recursive part is the use of case 1 from case 2.

Case 1 is the common one: for *C* [*G* –> *T*] we use as base type of *G*, in *C*, the base type of *T*. We need case 2 to make sure that this definition is not cyclic, because we permit cases such as *C* [*G, H*–> *D* [*G*]], and as a consequence cases such as *C* [*G* –> *H, H*–> *G*] or even *C* [*G* –> *G*] even though they are not useful; both of these examples yield *ANY* as base types for the parameters.

As a result of the definition of "constraining types", the base type of an unconstrained formal generic, such as *G* in *C* [*G*], is also *ANY*.

In *C* [*G* –> *I, I* –> *T, H*–> *E* [*G, H*]] (unlikely to arise in practice):

- The constraint of *I* is *T* (case 1).
- Applying constraint of *G* is *T*, the base type of *I*: case 2 first gives .
- Without the substitutio

As usual for a type that is not a Class_or_tuple_type, the base class of a Formal_generic_name type is its base type's base class. So *HASHABLE*, in the class *HASH_TABLE* [*G; KEY* –> *HASHABLE*], is both the base type and the base class of *KEY*.

← *See the Base rule, page 324.*

What about an unconstrained Formal_generic_name such as *G* in *HASH_TABLE*? Every object ever manipulated by a system is an instance of some class, and every developer-written class is a descendant of the universal library class *ANY*. In other words, *HASH_TABLE* could be equivalently declared as

← *"ANY", 6.6, page 172; see also chapter 35 for more details.*

```
class HASH_TABLE [G –> ANY; KEY –> HASHABLE]...
```

This is a general rule: we consider an unconstrained generic parameter as if it were constrained by *ANY*. Hence the definition of the base type for unconstrained generics, a special case of the preceding rule:

Base type of an unconstrained formal generic

The base type of an unconstrained Formal_generic_name type is *ANY*.

This also enables us to consider that every formal generic parameter has a **constraining type**, taking it to be *ANY* for an unconstrained parameter.

The last definitions do not give the full semantics of a Formal_generic_name type, but only its base type. We also need to specify whether the type is reference or expanded. This is, however, the one case in which we can't know for sure: only the actual parameter passed in a particular generic derivation will tell.

Reference or expanded status of a formal generic

A Formal_generic_name represents a reference type or expanded type depending on the corresponding status of the associated actual generic parameter in a particular generic derivation.

12.11 CURRENT TYPE, FEATURES OF A TYPE

This discussion of genericity — now complete except for the special case of multiple constraints covered below — leads to a notion that will be convenient in future discussions. The presentation of classes noted that every Eiffel construct is part of a class, the *current class*. Often, what we will need is not just a class but a type. Hence the notion of *current type*. As long as classes were not generic, the current type was the same as the current class; but now the notion becomes more interesting, although straightforward.

← *"THE CURRENT CLASS", 4.5, page 117*.

Assume that we are asked "what is the type of valid targets for *f*?", where *f* is a feature of a generic class *C* [*G*]. The answer is, of course, *C* [*G*] itself. Answering "the current class" would not do, since *C* by itself is not a type — only a type template, which yields a type if we provide an appropriate generic parameter.

This is one of the lessons of this chapter: the concepts of class and type — although closely related since every type is based on a class— are not identical. The difference comes not only from genericity but also from anchoring; the answer to the question "what is the base type of **like** *Current* in *C*?" would also be *C* [*G*].

This will be called the *current type*:

Current type

Within a class text, the **current type** is the type obtained from the current class by providing as actual generic parameters, if required, the class's own formal generic parameters.

Clearly, the base class of the current type is always the current class.

In the same vein, since the type will often be our first source of information — before the underlying class — it is also useful to allow ourselves to extend the notion of "features of a class":

Features of a type

The features of a type are the features of its base class.

These are the features applicable to the type's instances (which are also instances of its base class).

You may note in particular that with genericity we often need to refer to the type rather than the class. If a generic class *C* [*G*] has a feature

```
f (x: G)
```

then for a of type C [T] (a type generically derived from C by using T as actual generic parameter) a call of the form

```
a.f (y)
```

requires an argument y of type T (not G, which is only a placeholder within the text of class C). This means that to understand f and its type properties fully we need to consider not just as a *feature of a certain class* (the class *C*) but as a *feature of a certain type* (the type *C* [*T*]).

12.12 APPLYING GENERICITY TO TYPES

Genericity means that we must be careful when using terms such as "the type of an expression" or "the type of a feature" if a generic derivation is involved. Consider a generic class

```
class C [G] feature
        some_query: G
        some_routine (arg: G) is do … end
        … Other features …
end
```

and a client *D* with a declaration

```
x: C [T]
```

for some type *T*, for example *INTEGER* or *ARRAY* [*REAL*]. In the context of class *D* we may ask the questions:

- What is the type of *x.some_query*?
- What expressions *y* are valid in a call *x.some_routine* (*y*)?

Viewed from within *C* the type of *some_query* is *G*, but this makes no sense in the context of *x*.*some_query* as used in *D*, where we may consider that *G* is simply a placeholder for the actual generic parameter, *T* in this case ----- .

Generic substitution

Every type *T* defines a mapping σ from names to types known as its **generic substitution**:

1 • If *T* is generically derived, σ associates to every Formal_generic_name the corresponding actual parameter.

2 • Otherwise, σ is the identity substitution.

Similarly, an argument *y* that we will pass in the call *x*.*some_routine* (*y*) must be of type *G* or conforming.

These observations lead to the following substitution rule:

Generic Type Adaptation rule

The signature of an entity or feature *f* of a type *T* of base class *C* is the result of applying *T*'s generic substitution to the signature of *f* in *C*.

The signature include both the type of an entity or query, and the argument types for a routine; the rule is applicable to both parts.

In the examples cited this yields the type *T*, as desired, both as the result type of *x*.*some_query* and as the type to which arguments to *x*.*some_routine* (…) must conform.

12.13 THE CASE OF MULTIPLE CONSTRAINTS

(This last section covers an advanced technique needed only in special cases. On first reading you may skip to the next chapter.)

As noted, it is possible for a Formal_generic_name to have several constraints, as in

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

The role of the base class and type, as usual, is to tell us what features *f* we may use for a call *x*.*f* (...) for *x* of type *G*. In the case of a single constraint *CONST1*, the answer was simply: those of *CONST1*.

Here, the basic idea is just as straightforward: we will accept *f* as long as it denotes a feature in *any* of the constraining types.

Of course, the same feature name might denote features in several of these types. That's not to scare us, since we know that any valid actual generic parameter *T* for *G* will have to inherit from all of the $CONST_i$ and hence resolve the conflicts according to the rules of the preceding chapters (renaming, sharing or select under repeated inheritance). But this may still leave some ambiguities as to what *x*.*f* (...) means for *x* of type *G* in class *D*. To keep matters simple we take the following rule:

----- TO BE UPDATED TO ACCOUNT FOR NEW RENAMING ----

1 • If all the matching *f* in the constraining types have a common seed (meaning they all come from a single feature of a common ancestor, as would be the case if *f* is *equal* or *print* from class *ANY*), there won't be any problem: in any acceptable actual generic parameter *T* for *G*, either the corresponding version of *f* will be shared, or a **select** will designate one of the versions as the official one for *T*.

• Otherwise, there will be a name clash that *T* will have to resolve through renaming, but we don't want to go into this. We just renounce the feature for entities of type *G*.

• ----------

•

Generically constrained feature name

Consider a generic class *C*, a constrained Formal_generic_name *G* of *C*, a type *T* appearing as one of the Constraining_types for *G*, and a feature *f* of name *fname* in the base class of *T*. The **generically constrained names** of *f* for *G* in *C* are:

1 • If one or more Single_constraint clauses for *T* include a Rename part with a clause *fname* **as** *ename*, where the Feature_name part of *ename* (an Extended_feature_name) is *gname*: all such *gname*.

2 • Otherwise: just *fname*.

•

The following validity constraint expresses this rule:

--- EXPLAIN CLAUSE 2 ---

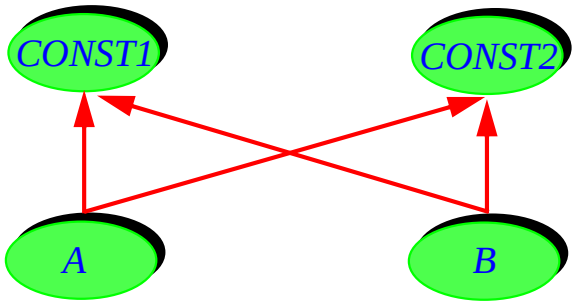
Multiple Constraints rule *VTMC*

A feature of name *fname* is applicable in a class *C* to a target *x* whose type is a Formal_generic_name *G* constrained by two or more types *CONST1*, *CONST2*,…, if and only if it satisfies the following conditions:

1 • At least one of the $CONST_i$ has a feature available to *C* whose generically constrained name for *G* in *C* is *fname*.

2 • If this is the case for two or more of the $CONST_i$, all the corresponding features names, after possible renaming through Renaming clauses in the constraints, are the same.

Can we stop here? Not quite. We do need a precise notion of base type reflecting the Multiple Constraints rule. Although there is nothing inherently difficult in the rule, turning it into a definition of the base type for a multiply constrained Formal_generic_name requires some care. Intuitively this base type should be the "lowest common ancestor" of the constraints *CONST1, CONST2*,…, but there is no such notion: the set of common ancestors — a non-empty set since it contains at least *ANY* — doesn't necessarily include one that inherits from all the others, as in this situation if there are no other non-kernel classes involved:



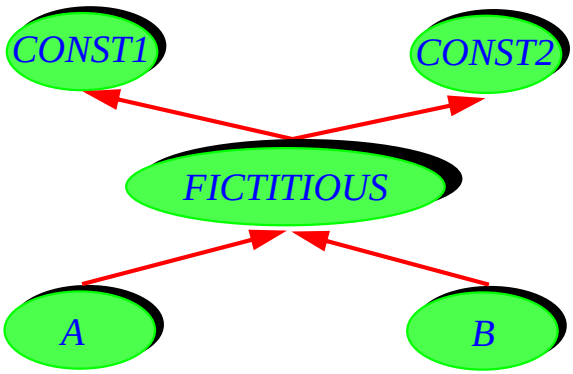
No lowest common ancestor

To obtain a theoretical answer (the practical answer being given by the informal rule above), we simply build a fictitious "lowest common ancestor" with all conflicts removed. Hence the definition:

Base type of a multi-constraint formal generic type

The base type of a multiply constrained Formal_generic_name type is a type generically derived, with the same actual parameters as the current class, from a fictitious class with none of the optional parts except for Formal_generics and an Inheritance clause that lists all the constraining types as parents and resolves any conflicts between potentially ambiguous features by renaming them to new names not available to developers.

This definition is a little as if we decided to replace the above inheritance structure by the following one, preventing *A* and *B* from resolving, each in its own desired manner, any name clashes that might arise between features of *CONST1* and *CONST2*.



Artificial lowest common ancestor

FICTITIOUS represents the constraint we would be using if we were limited to a single constraint for *G*: features applicable to *G*, assuming the declaration **class** *D* [*G* –> {*CONST1, CONST2*}], are those you could use if the declaration were **class** *D* [*G* –> {*FICTITIOUS*}].

Even though the basic idea of this definition is simple (you may apply to a multiply constrained Formal_generic any of the constraining types' features that are not ambiguous), the recourse to a fictitious type is not too pleasant conceptually, and explains the doubt, expressed earlier, whether multiple constraints are really worth the trouble.

Note that the last validity rule, the Multiple Constraints rule, is conceptually redundant. The general rule that governs the applicability of a feature to a target is the Single-Level Call rule which (combined with the Export rule, both in the chapter on calls) essentially states that *f* is a valid feature for the call *x*.*f* (...) if it is a feature of the base type of *x* (and is exported as appropriate). This is in fact the reason why take the trouble to define the base type — always a Class_or_tuple_type, with clearly identifiable features — for every kind of type. All that the Multiple Constraints rule states is the application of the Single Call rule to the case of a multiply constrained Formal_generic_name, using the just given definition of the base type in this case.

→ *"Class-Level Call rule", page 628*; *"Export rule", page 624*.

But the Single Call rule in this case is so indirect, relying through the base type on a fictitious class, that compiler writers will likely prefer, for the error message they display in case of a wrong *f*, to cite the Multiple Constraints rule.