Attaching values to entities

22.1 OVERVIEW

← *Chapter 20.*

At any instant of a system's execution, every entity of the system has a certain attachment status: it is either attached to a certain object, or void (attached to no object). Initially, all entities of reference types are void; one of the effects of a Creation instruction is to attach its target to an object.

The attachment status of an entity may change one or more times during system execution through a **attachment** operations, in particular:

- The association of an actual argument of a routine to the corresponding formal argument at the time of a call.
- The Assignment instruction, which may attach an entity to a new object, or remove the attachment.

The validity and semantic properties of these two mechanisms are essentially the same; we study them jointly here.

------- REWRITE ---- You already know everything about the last case. This chapter explores the other three. It will also examine a closely related problem, for which the last chapter did the advance work: how to determine that two entities have the same attachment, or are **equal**, in any of the possible interpretations of this general notion.

22.2 ROLE OF REATTACHMENT OPERATIONS

Every reattachment operation has a **source** (an expression) and a **target** (a Variable entity). When the reattachment is valid, its effect will be ----

Reattachment, source, target

A **reattachment** operation is one of:

1 • An Assignment $x := y$; then y is the attachment's source and x its target.

2 • The run-time association, during the execution of a routine call, of an actual argument (the source) to the corresponding formal argument (the target).

We group assignment and argument passing into the same category, reattachment, because their validity and semantics are essentially the same:

- Validity in both cases is governed by the type system: the source must *conform* to the target's type, or at least *convert* to it. The Conversion principle guarantees that these two cases are exclusive.
- The semantics in both cases is to attach the target to the value of the source or a copy of that value.

← *Chapter 14 presented both conformance and convertibility. See "Conversion principle", page 400.*

This chapter explores reattachment operations: their constraints, semantics, and syntactic forms.

22.3 FORMS OF UNCONDITIONAL REATTACHMENT

As noted, the two forms of unconditional reattachment, Assignment instructions and actual-formal association, have similar constraints and essentially identical semantics, studied in the following sections.

The syntax is different, of course. An assignment appears as

```
x := y
```

where x, the target, is a Variable entity and y, the source, is an expression.

Very informally, the semantics of this instruction is to replace the value of x by the current value of y; x will keep its new value until the next execution, if any, of a reattachment (unconditional, conditional, or new Creation) of which it is the target.

Actual-formal association arises as a byproduct of routine calls. A Call to a non-external routine r with one or more arguments induces an unconditional reattachment for each of the argument positions.

Consider any one of these positions, where the routine declaration (appearing in a class *C*) gives a formal argument *x*:

For an external routine, written in another language, the exact semantics depends on the other language's rules.

```
r (…, x: T, …) is …
```

Then consider a call to *r*, where the actual argument at the given position is *y*, again an expression. The call must be of one of the following two forms, known as unqualified and qualified:

See chapter 23 for the details of Call instructions and expressions.

```
r (…, y, …)
t.r (…, y, …)
        -- In this second form, t must conform to a type based on C.
```

Qualified or not, the call causes an unconditional reattachment of target *x* and source *y* for the position shown, and similarly for all other positions.

A qualified Call also has a "target", appearing to the left of the period, *t* in the second example. Do not confuse this with the target of the actual-formal attachment induced by the call, *x* in this discussion.

Informally again, the semantics of this unconditional reattachment is to set the value of *x*, for the whole duration of the routine's execution caused by this particular call, to the value of *y* at the time of call. No further reattachment may occur during that execution of the routine. Any new call executed later will start by setting the value of *x* to the value of the new actual argument.

22.4 SYNTAX AND VALIDITY OF ASSIGNMENT

Here is the syntax of an Assignment instruction:

Assignments

Assignment ≜ Variable ":=" Expression

Actual-formal association does not have a syntax of its own; it is part of the Call construct.

→See chapter 23 about Call. Syntax page 618.

The syntax of Assignment requires the target to be a Variable. Recall that a Variable entity is either an attribute of the enclosing class or a local variable of the immediately enclosing routine or agent. The latter case includes, in a function, the predefined entity *Result*. A formal routine argument is *not* a Variable; this property is discussed further in the next section.

← 19.8 introduced Variable entities, with syntax on page 504 and the associated Variable rule on page 506.

The principal validity constraint in both cases is that the source must conform or convert to the target. For Assignment this is covered by the following rule:

Assignment rule *VBAR*

An Assignment is valid if and only if its source expression is compatible with its target entity.

To be "compatible" means to conform or convert.

← *"Compatibility between types", page 376.*

This also applies to actual-formal association: the actual argument in a call must conform or convert to the formal argument. The applicable rule is **argument validity**, part of the general discussion of call validity.

The two cases, conformance and convertibility, are complementary:

→ *"THE CALL VALIDITY RULE", 25.10, page 673.*

- **Conformance** is the more common situation. As you will remember, type *U* conforms to type *T* — and, as a consequence, an expression of the first type to an entity of the second one — if the base class of *U* is a descendant of the base class of *T* and, if generic parameters are present, they also conform; the conformance chapter gave the details.

← *Chapter 14.*

- **Convertibility** allows reattachments that also perform a conversion, as when you are assigning an integer value to a real target.

← *Chapter 15.*

22.5 THE STATUS OF FORMAL ROUTINE ARGUMENTS

The syntax of Assignment requires the target to be a Variable. This includes, as noted, attributes and local variables, but not formal arguments of the enclosing routine. So in the body of a routine

```
r (x : SOME_TYPE)
        …
    do
            …
    end
```

an assignment *x* := *y*, for some expression *y*, would not be valid. The only reattachments to a formal argument occur at call time, through the actual-formal association mechanism.

It is indeed a general rule of Eiffel that routines may not change the values of their arguments. A routine is an operation to be performed on certain operands; arguments enable callers to specify what these operands should be in a particular application of the operation. Letting the operation change the operands would be confusing and error-prone.

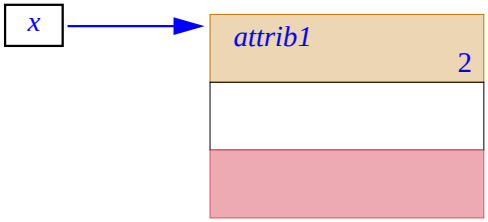
Although some programming languages offer "out" and "in-out" modes for arguments, they are a notorious source of trouble for programmers, and complicate the language; for example:

- You must have special rules for the corresponding actual arguments (they must be variable).
- You must prohibit using the same actual argument twice, as in *r* (*e*, *e*), but only if both of the affected argument positions are "out" or "in out".

The Eiffel rule does not prohibit a routine *r* from modifying the **objects** that it is passed: if a formal argument *x* is a non-void reference, *r* has access to the attached object and can perform any valid feature call on it. In the situation pictured below the body of *r* may include a procedure call

```
x.set_attrib1 (2)
```

where *set_attrib1* will update the value of the integer field *attrib1*. What is **not** permitted is an Assignment of target *x*, which would affect the reference rather than the object.



Object may change, reference not

22.6 CONVERSIONS

All that beginning Eiffel programmers really need to know about convertibility is that commonly accepted mixed-type arithmetic assignments with no loss of information, such as *your_real* := *your_integer* (but not the other way around, which requires using a truncation or rounding function) are OK and will cause the proper conversions. So on first reading you should skip this section.

Skip to "SEMANTICS OF REATTACHMENT", 22.7, page 585.

Conformance and convertibility are, as noted, mutually exclusive cases. Let us start our study of reattachment semantics by the second one — even though conformance is by far the more common case — because the discussion of convertibility already told us most of what we need to know.

← *Chapter 15. See also "The Target Conversion mechanism deserves some justification…", page 762.*

In that discussion we saw that it is possible for a class to declare, through its creation procedures, one or more **creation types**, as in:

```
class DATE create
    from_tuple convert {TUPLE [INTEGER, INTEGER, INTEGER]}
…
```

This is intended to permit attachments from any of the conversion types (here only one) to the current type, so that you may write

```
compute_revenue ([1, "January", 2000], [1, "January", 2001])
```

where *compute_revenue* expects two date arguments. Argument passing in this case will cause, prior to actual attachment, the creation of a new object of type *DATE* and its initialization through the given creation procedure *from_tuple*. As was noted in the earlier discussion, this means that the call is equivalent to

```
compute_revenue (create {DATE}.from_tuple ([1, "January", 2000]),
                 create {DATE}.from_tuple ([1, "January", 2001]))
```

Similarly, a call *your_date* := [*1*, "*January*", *2000*] is equivalent to **create** *your_date*.*from_tuple* ([*1*, "*January*", *2000*].

It is also possible to specify conversion through a function in the source type, rather than a procedure in the target type. Between any two given types, at most one of these possibilities may apply. If it is possible to convert an expression *exp* to an entity *e*, we say that *exp converts to x*, through a conversion routine (procedure or function).

← *"EXPRESSION CONVERTIBILITY: THE ROLE OF PRE-CONDITIONS", 15.10, page 412*.

This semantic specification and the supporting definition rely on the properties of the conversion mechanism, expressed by the Conversion Procedure rule and the associated definitions (convertible types of a class), which guarantee that everything is unambiguous:

← *"Conversion Procedure rule", page 403*; *"Converting to and from a type", page 406*;

- The definition of "convertible types" tells us that *SOURCE* must appear among the Conversion_types of a creation procedure of the base class of *TARGET*.
- Clause 4 of the Conversion Procedure rule, requiring all the convertible types of a class to be different, guarantees that there is only one such procedure, making the definition of "applicable conversion procedure" legitimate.
- Clauses 6 and 7 of the rule guarantee that this procedure has exactly one formal argument, of a type *ARG* to which *SOURCE* must conform or convert.

If *SOURCE* converts (rather than conforms) to *ARG*, then the attachment will, as was noted in the earlier discussion, cause two conversions rather than one, since to the conversion procedure must convert its argument to type *ARG*. As was also noted, things stop here: a conversion reattachment may cause one conversion (the usual case), or two (if the *SOURCE* type converts to the *ARG* type), but no more.

← *See discussion of clause 6 of the Conversion Procedure rule on page 403.*

This discussion completes the specification of reattachment in the convertibility case. Since the Conversion principle tells us that a type may not both convert and conform to another, we may limit our attention, for the rest of this chapter, to the more common case: reattachments in which the source of an assignment or argument passing *conforms* to the target.

← *"Conversion principle", page 400*.

--- TEXT BELOW MAY HAVE TO BE TRANSFERRED ELSEWHERE ---

22.7 SEMANTICS OF REATTACHMENT

Let us examine the precise effect of executing an unconditional reattachment of either of the two forms, for a source conforming to the target.

Because that effect is the same in both cases — an Assignment *x* := *y* and a call that uses *y* as actual argument for the formal argument *x* of a routine — we can use the first as our working example: the assignment

```
x := y
```

where *x* is of type *TX* and *y* of type *TY*, which must conform to *TX*.

The effect depends on the nature of *TX* and *TY*: reference or expanded? Here is the basic rule, covering the vast majority of practical cases:

- .If both *TX* and *TY* are expanded, the assignment copies the value of the object attached to the source onto the object attached to the target.
- If both are reference types, the operation attaches *x* to the object attached to *y*, or makes it void if *y* is void.

As an example of the first case, in

```
x, y: INTEGER
…
y := 4
x := y
```

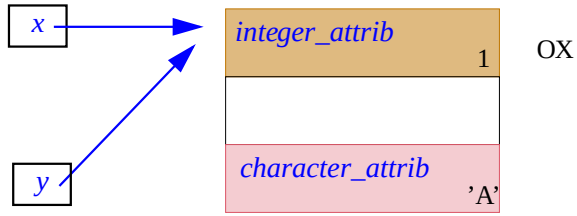
the resulting value of *x* will be 4, but the last Assignment does not introduce any long-lasting association between *x* and *y*; this is because *INTEGER* is an expanded type.

As an example of the second case, if *TC* is a reference type, then

```
x, y: TC
…
create y …
x := y
```

Effect of reference reattachment

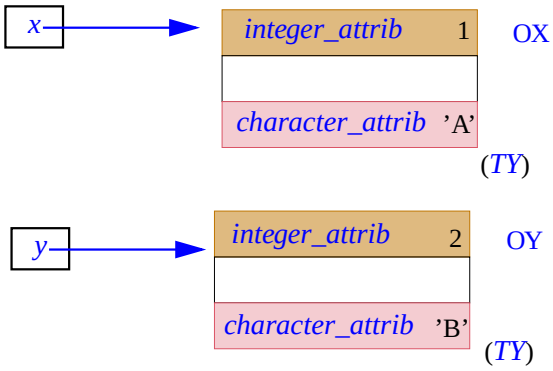
will result in *x* and *y* becoming attached to the same object:



This rule addresses the needs of most applications. There remains, of course, to see what happens when one of *TX* and *TY* is expanded and the other reference. But it is more important first to understand the reasons for the rule by exploring what potential interpretations make sense in each case.

Consider first the case of references. We start from the run-time situation pictured below, with two objects labeled OX and OY, assumed for simplicity to be of the same type *TY*, and accessible through two references *x* and *y*. Of course, since the Eiffel dynamic model is fully based on objects, *x* and *y* themselves will often be reference fields of some other objects, or of the same object; these objects, however, are of no interest for the present discussion and so they will not appear explicitly.

Before a reattachment

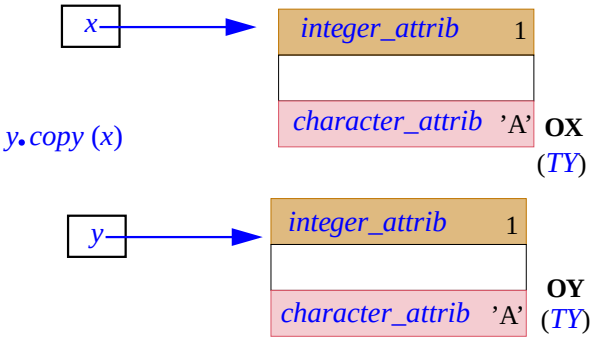


Three possible kinds of operation may update *x* from *y*: copying, cloning and reference reattachment.

The first, copying, makes sense only if both *x* and *y* are attached (non-void). Its semantics, seen in the last chapter, is to copy every field of the source object onto the corresponding field of the target object. It does not create a new object, but only updates an existing one. We know how to achieve it: through procedure *copy* of the universal class *ANY* or, more precisely, its frozen version *identical_copy*, ensuring fixed semantics for all types (whereas *copy* may be redefined). The next figure illustrates the effect of a call *y.identical_copy (x)* starting in the above situation.

← *See 21.2, page 557 on copy and its frozen version identical_copy.*

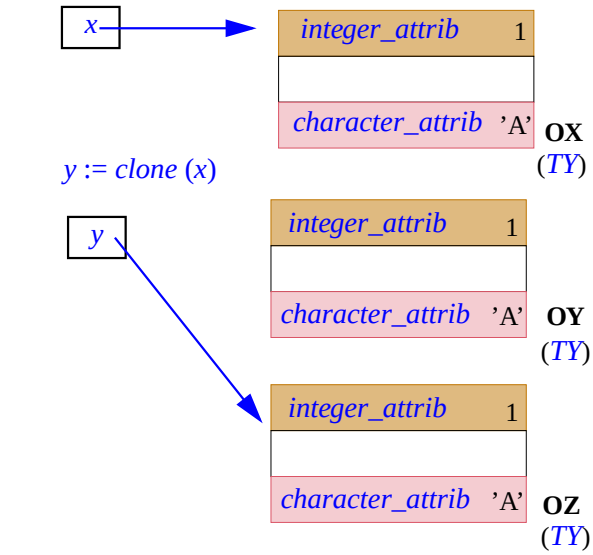
Effect of standard copy



The second operation is a close variant of the first: cloning also has the semantics of field-by-field copy, but applied to a newly created object. No existing object is affected. Here too a general mechanism is available to achieve this: a call to function *clone* which (anticipating on this section) we have learned to use in an assignment *x := clone (y)*. To guard against redefinition we may use the frozen version *identical_clone*. The result is shown below; the cloning creates a new object, OZ, a carbon copy of OX.

← *See 21.4, page 567,about clone and identical_clone.*

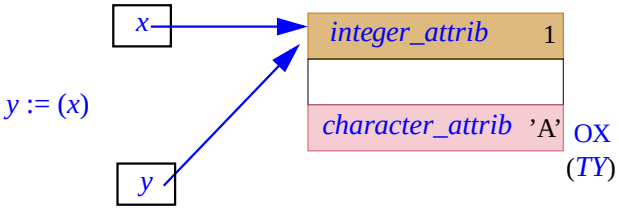
Effect of standard clone



Assuming *y* was previously attached to OY as a result of the preceding operation, it is natural to ask: "What happens to the object OY?". This will be discussed in a later section.

→ "MEMORY MANAGEMENT", 22.15, page 608.

The third possible operation is reference reattachment. This does not affect any object, but simply reattaches the target reference to a different object. The result (already visible in the last figure) may be represented as follows:



Effect of reference reattachment

To devise the proper rule for semantics, we must study which of these operations make sense in every possible case. Since the source and target types may each be either expanded or reference, there will be four cases:

SOURCE TYPE → / *TARGET TYPE* ↓	**Reference**	**Expanded**
Reference	**[1]** • Copy (if neither source nor target void) • Clone • Reference reattachment	**[2]** • Copy (if target not void) • Clone
Expanded	**[3]** • Copy (will fail if source is void)	**[4]** • Copy

Meaningful possibilities for the semantics of reference reattachment

This list only takes into account shallow operations. Deep variants were discussed in 21.5, page 571.

If all we were interested in was copying and cloning, we would not need any new mechanism: routines *identical_copy* and *identical_clone*, from *ANY*, are available for these purposes. The only operation we would miss is reference reattachment, corresponding to the last figure. This only makes sense for case 1, when both target and source are of reference types: if the target is expanded, as in cases 3 and 4, there is no reference to reattach; and if the source is expanded, as in cases 2 and 4, a reattachment would introduce a **reference to a sub-object**, a case discussed and rejected in the discussion of the dynamic model.

← *"REFERENCE ATOMICITY", 19.7, page 502; the excluded case is illustrated by the figure on page 502.*

In case 1, however, we do need the ability to specify reference reattachment, not covered by *copy*, *clone* or their frozen variants. This will be the semantics of the Assignment $x := y$ and of the corresponding actual-formal association when both x and y are of reference types.

We now have notations for expressing meaningful operations in every possible case: reference assignment in case 1, routines *identical_copy* and *identical_clone* in the other cases. At least two reasons, however, indicate that in addition to these case-specific operations we also need a single notation applicable to all four cases:

- In a generic class, *TX* and *TY* may be a Formal_generic_name; then the class text does not reveal whether x and y denote objects or references, since this depends on the actual parameter used in each generic derivation of the class. But it must be possible for this class text to include an Assignment $x := y$, or a call $r\,(\ldots, y, \ldots)$, with a clearly defined meaning in all possible cases.
- The availability of general-purpose copying and cloning mechanisms does not relieve us from the need to define a clear, universal semantics for actual-formal association.

← *If the formal generic is TX, conformance requires TY to be identical to TX. If the formal is TY, TX is either TY or an ancestor of TY's constraint (ANY if TY is unconstrained). See "Direct conformance: formal generic", page 385.*

Examination of the above table suggests a uniform notation addressing these requirements. What default semantics is most useful in each case?

- In case 1, where both x and y denote references, the semantics should be reference reattachment, if only (as discussed above) because no other notation is available for that operation.
- In case 4, with both x and y denoting objects, only one semantics makes sense for a reattachment operation: copying the fields of the source onto those of the target.
- In case 2, with x denoting a reference and y an object, both copying and cloning are possible. But copying only works if x is not void (since there must be an object on which to copy the source's fields). If x is void, copying will fail, triggering an exception. It would be unpleasant to force class designers to test for void references before any such assignment. Cloning, much less likely to fail, is the preferable default semantics in this case.

Cloning may also fail, triggering an exception, if there is no more memory available (21.2). But this is a much less frequent situation than the target being a void reference.

- In case 3, as in case 1, the target *x* is an object, so copying is again the only possible operation. In this case it will fail if *y* is void (since there is no object to copy), but then no operation exists that would always work.

This analysis leads to the following definition of the semantics of unconditional reattachment in the case of a source conforming to its target.

← *Remember that the* **convertibility** *case is distinct (“CONVERSIONS”, 22.6, page 583)*

SOURCE TYPE → / *TARGET TYPE* ↓	**Reference**	**Expanded**
Reference	**[1] Reference reattachment**	**[2] Clone**
Expanded	**[3] Copy** (Fails if source void)	**[4] Copy**

The semantics of conformance reattachment

NOT a semantic specification but only a list of available possibilities for such a specification. The actual semantics appears next.

→ *The table giving equality semantics on page611 will be organized along similar lines.*

In this semantic specification, “**Copy**” and “**Clone**” refer to the frozen features *identical_copy* and *identical_clone* that every class inherits from the universal class *ANY*.

Arguments could be found for using instead the redefinable version *copy*, and *clone* which is defined in terms of *copy*: after all, if the author of a class redefined these routines, there must have been a reason. But it is more prudent to stick to the frozen versions, so that the language defines a simple and uniform semantics for assignment and argument passing on entities of all types. If you do want to take advantage of redefinition, you can always use the call. *copy* .(*y*) instead of the assignment *x* := *y*, or pass *clone* (*y*) instead of *y* as an actual argument to a call. These alternatives to unconditional reattachment apply of course to reference types as well as expanded ones.

For the exception raised in case 3 if the value of *y* is void, the Kernel Library class *EXCEPTIONS* introduces the integer code *Void_assigned_to_expanded*.

See chapters 26 on exceptions and 37 on class EXCEPTIONS.

This semantic definition yields the most commonly needed effect in each case. This applies in particular to cases 1 and 4, which account for the vast majority of reattachments occurring in practice: for an integer variable (case 4), it is pleasant to be able to write

```
n := 3
```

to produce the effect of

```
n.copy (3)
```

Here copy and identical_copy are the same.

but uses a commonly accepted notation and has the expected result. For a reference variable *y*, it is normal to expect the call

```
some_routine (y)
```

simply to pass to *some_routine* a reference to the object attached to *y*, if any, rather than to duplicate that object for the purposes of the call. If you do wish duplication – shallow or deep – to occur, you may make your exact intentions clear by using one of the calls

```
some_routine (clone (y))
some_routine (identical_clone (y))
some_routine (deep_clone(y))
```

An interesting application is the case of generic parameters and generically derived types. If the type of *x* and *y* is a formal generic parameter of the enclosing class, as in

```
class GENERIC_EXAMPLE [G] feature
    example_routine
        local
            x, y: G
        do
            x := y
        end
end
```

the effect of the highlighted assignment may be reference reattachment or copying depending on the actual generic parameter used for *G* in the current generic derivation. (Cloning, which only occurs for reference target and expanded source, does not apply to this case since, by construction, *x* an *y* are of the same type.) We will shortly come back to the effect of reattachment semantics on generic programming.

→ *"EFFECT ON GENERIC PROGRAMMING", 22.10, page 596*.

A consequence of the validity and semantics rules is the following semantic principle, which will be important to understand the run-time behavior of our systems:

Reattachment principle

After a reattachement to a target entity *t* of type *TT*, the object attached to *t*, if any, is of a type conforming to *TT*.

"If any" because the source of the attachment might have been void. If not, its value *v* is of a type *VT* that either conforms or converts to *TT* (but not both). If it conforms, the operation simply reattaches *t* to *v*, satisying the principle. If it converts, the operation produces a new object of type *TT*; this satisfies the principle too since *TT* conforms to itself.

Attaching an entity, attached entity

Attaching an entity *e* to an object O is the operation ensuring that the value of *e* becomes **attached to O**.

Although it may seem tautological at first, this definition simply relates the two terms "attach", denoting an operation that can change an entity, and "attached to an object", denoting the state of such an entity — as determined by such operations. These are key concepts of the language since:

- A reattachment operation (see next) may "*attach*" its target to a certain object as defined by the semantic rule; a creation operation creates an object and similarly "*attaches*" its creation target to that object.
- Evaluation of an entity, per the Entity Semantics rule, uses (partly directly, partly by depending on the Variable Semantics rule and through it on the definition of "value of a variable setting") the object *attached* to that entity. This is only possible by ensuring, through other rules, that prior to any such attempt on a specific entity there will have been operations to "attach" the entity or make it void.

Reattachment Semantics

The effect of a reattachment of source expression *source* and target entity *target* is the effect of the first of the following steps whose condition applies:

1 • If *source* converts to *target*: perform a conversion attachment from *source* to *target*.

2 • If the value of *source* is a void reference: make *target*'s value void as well.

3 • If the value of *source* is attached to an object with copy semantics: create a clone of that object, if possible, and attach *target* to it.

4 • If the value of *source* is attached to an object with reference semantics: attach *target* to that object.

As with other semantic rules describing the "effect" of a sequence of steps, only that effect counts, not the exact means employed to achieve it. In particular, the creation of a clone in step 3 is — as also noted in the discussion of creation — often avoidable in practice if the target is expanded and already initialized, so that the instruction can reuse the memory of the previous object.

Case 1 indicates that a conversion, if applicable, overrides all other possibilities. In those other cases, if follows from the Assignment rule that *source* must **conform** to *target*. →.

Case 2 is, from the validity rules, possible only if both *target* and *source* are declared of *detachable* types.

In case 3, a "clone" of an object is obtained by application of the function *cloned* from *ANY*; expression conformance ensures that *cloned* is available (exported) to the type of *target*; otherwise, cloning could produce an inconsistent object. →.

> The cloning might be impossible for lack of memory, in which case the semantics of the cloning operation specifies triggering an exception, of type *NO_MORE_MEMORY*. As usual with exceptions, the rest of case 3 does not then apply.

In case 4 we simply reattach a reference. Because of the validity rules (no reference type conforms to an expanded type), the target must indeed be of an reference type.

This rule defines the *effect* of a construct through a sequence of cases, looking for the first one that matches. As usual with semantic rules, this only specifies the result, but does not imply that the implementation must try all of them in order.

The semantics of assignment is just a special case of this rule:

Assignment Semantics

The effect of a reassignment *x* := *y* is determined by the Reattachment Semantics rule, with source *y* and target *x*.

The other cases where Reattachment Semantics applies is actual-formal association, per step 5 of the General Call rule.

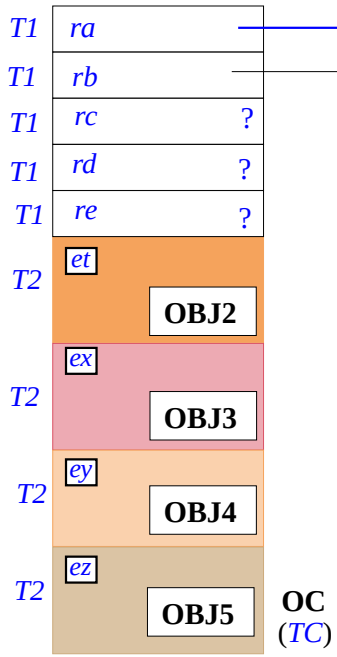
→ *"General Call Semantics", page 645*.

> On the other hand, the semantics of Object_test, a construct which also allows a Read_only entity to denote the same value as an expression, is simple enough that it does not need to refer to reattachment.

22.8 AN EXAMPLE

---- WRONG (OLD SEMANTICS), TO BE REMOVED

To see the effect of reattachment in various cases, consider the run-time situation pictured below.



A run-time system snapshot

All the entities considered are attributes of a class *C*. OC, the complex object on the left, is a direct instance of type *TC*, of base class *C*.

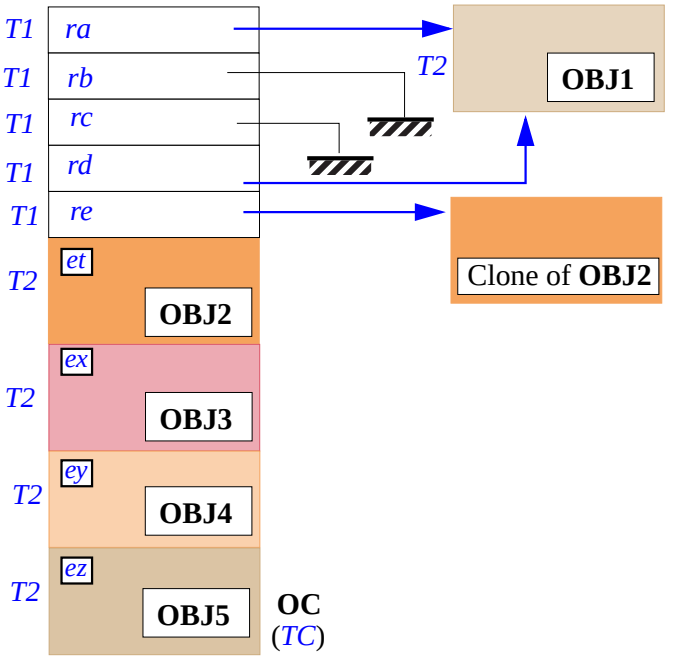
OC is not only complex but composite.

The first five attributes (*ra, rb, rc, rd, re*), whose names begin with *r*, are of a reference type *T1*. The corresponding fields of OC are references. The four others (*et, ex, ey, ez*), whose names begin with *e*, are expanded. The corresponding fields are sub-objects of OC, which have been given the names OBJ2 to OBJ5. The reference field *ra* is originally attached to another object OBJ1, also of type *T2*.

Assume that class *C* has the following routine, using Assignment instructions to perform a number of reattachments:

```
assignments is
            -- Change various fields.
      do
            rc := rb
            rd := ra
            re := et
            ex := ey
            ez := ra
      end
```

If applied to the above OC, this procedure will produce the following situation:



Snapshot after assignments

The assignment *re* := *et*, with reference target and expanded source, produces a duplicate of object OBJ2.

An attempt to execute *et* := *rb*, with an expanded target and a void source, would trigger an exception.

22.9 ABOUT REATTACHMENT

(This section brings no new Eiffel concept. It will only be of interest to readers who wish to relate the above concepts to the argument passing conventions of earlier programming languages.)

It may be useful to compare the semantics of unconditional reattachment to the mechanisms provided by other languages, in particular to traditional variants of argument passing semantics.

Consider a call of the form

```
r (…, y, …)
```

This causes an attachment as a result of actual-formal association between the expression *y*, of type *TY*, and the corresponding formal argument *x*, of type *TX*.

An examination of the semantics defined above in light of other argument passing conventions yields the following observations:

- If both *TX* and *TY* are reference types (case 1 of the table of reattachment semantics), the reattachment causes sharing of objects through references, also known as **aliasing**. For actual-formal association this achieves the effect of **call by reference**, with the target being protected against further reattachment for the duration of the call.
- If both *TX* and *TY* are expanded types (case 4), reattachment copies the content of *y*, an object, onto *x*. This achieves the effect of **call by value**.
- If *TX* is an expanded type and *TY* a reference type (case 3), the operation copies onto *x* the content of the object attached to *y* (*y* must be non-void). This achieves what is often called **dereferencing**.
- If *TX* is a reference type and *TY* an expanded type (case 2), the operation attaches to *x* a newly created copy of *y*. This case has no direct equivalent in traditional contexts; it may be viewed as a form of call by value combined with call by reference.

← *Page 588.*

22.10 EFFECT ON GENERIC PROGRAMMING

The semantics of unconditional reattachment has a direct effect on both the production and the use of generic classes — a cornerstone of reusable software production.

For a generic class such as *GENERIC_EXAMPLE* above, it may seem surprising to see a given syntactical notation, the assignment symbol :=, denote different operations depending on the context, and similarly for argument passing.

← *Page 591.*

This convention corresponds, however, to the most common needs of generic programming. The container classes of EiffelBase, such as *LINKED_LIST*, *TWO_WAY_LIST*, *HASH_TABLE* and many others, used to store and retrieve values of various types, provide numerous examples. These classes are all generic and, depending on their generic derivations, the values they store may be references or objects.

The notion of container data structure was presented in 10.21, page 286, and 12.3, page 343.

All of these classes have one or more procedures for adding an element to a data structure; for example, to insert an element to the left of the current cursor position in a linked list a client will execute

```
some_list.put_left (s)
```

Almost all of these procedures use assignment for fulfilling their task. Many do this not directly but through a call of the form

```
some_cell.put (x)
```

where *some_cell*, representing some individual entry of the data structure, is of a type based on some effective descendant of the deferred generic class *CELL*; for example, *LINKED_LIST* uses the descendant *LINKABLE*, describing cells of linked lists. Procedure *put* comes from *CELL*, where it appears (in effective form) as [

See page ====for an illustration of a LINKABLE list cell.

```
class CELL [G] feature
        item: G;
        put (new: G)
                -- Replace the cell value by new
            do
                item := new
            ensure
                item = new
            end
… Other features …
```

This is a slight simplification; the type of the argument 'new' is actually ***like*** *item, which has the same immediate effect since item is of type G.*

Because the addition of an element *x* by *put* uses assignment, what will be added to the data structure is an object value if *x* is of expanded type, and otherwise a reference to an object.

This policy means that if you are a "generic programmer" (a developer or user of generic classes) you must exercise some care, when dealing with data structures having diverse possible generic derivations, to make sure you know what is involved in each case: objects or references to objects. But it provides the most commonly defaults: a call

```
some_list_of_integers.put_left (25)
```

inserts the value 25, whereas

```
some_list_of_integers.put_lift (her_bank_account)
```

does not duplicate the object representing the bank account. Storing a reference in this case is the most conservative default policy. As in earlier examples, you can always obtain a different policy by using such calls as

```
some_list_of_integers.put_left (clone (her_bank_account))
some_list_of_integers.put_left (deep_clone (her_bank_account))
```

which guarantee uniform semantics (duplication, shallow in the first case and deep in the second) across the spectrum of possible types.

The discussion also applies to the problem of **searching** a data structure, discussed below.

→ *End of "SEMANTICS OF EQUALITY", 22.16, page 610.*

22.11 POLYMORPHISM

The only type constraint on unconditional reattachment is that (aside from the convertibility case) the type of the source must conform to the type of the target.

← "CONVERSIONS", 22.6, page 583.

If the target is expanded, this means that the types must essentially be the same; the only permitted flexibility is that one may describe objects of a certain form and the other references to objects of exactly the same form. This follows directly from the rule defining conformance when an expanded type is involved.

← "General conformance", page 380 and "Direct conformance: expanded types", page 388.

If the target is a reference, however (cases 1 and 2 of the reattachment semantics table), the situation is more interesting. If the target's base type is based on a class *C*, the validity rules mean that the base class of the source may be not just *C* but any proper descendant of *C*. This gives a remarkable flexibility to the type system, while preserving safety thanks to the conformance restrictions.

← Page 588.

As a consequence, an expression declared of type *TC* may at run time denote objects not just of type *TC* but of many other types, all based on descendants of the base class of *TC*.

So to study the run-time semantics of Eiffel systems we need to consider, along with the *type* of an expression (its type as deduced from declarations in the software text), its possible *dynamic types*:

Dynamic type

The **dynamic type** of an expression *x*, at some instant of execution, is the type of the object to which *x* is attached, or *NONE* if *x* is void.

This should not be confused with the **type** of *x* (called its **static type** if there is any ambiguity), which for an entity is the type with which it is declared, and for an expression is the type deduced from the types of its constituents.

→ "Type of an expression", page 774.

An expression has, of course, only one (static) type. But, as a key property of Eiffel's object-oriented style of computation, it may have more than one dynamic type. This is known as *polymorphism*.

Polymorphic expression; dynamic type and class sets

An expression that has two or more possible dynamic types is said to be **polymorphic**.

The set of possible dynamic types for an expression *x* is called the **dynamic type set** of *x*. The set of base classes of these types is called the **dynamic class set** of *x*.

Eiffel has a strongly typed form of polymorphism: the dynamic type set of an expression is not arbitrary. The type rules are organized to guarantee that the possible dynamic types for *x* all conform to the (static) type of *x*. This is how the type system keeps polymorphism is under control.

It is possible to determine the dynamic type set of *x* through analysis of the classes in the system to which *x* belongs, by considering all the attachment and reattachment instructions involving *x* or its entities.

22.12 ASSIGNER CALL

You may have noted that the syntax for assignment

```
some_variable := some_expression
```

only supports assignment to a Variable entity; it does not allow assignment to a field of an object, as in

```
x.a := b                                                    [1]
```

Warning: invalid except as abbreviation for procedure call. See below.

Some programming languages permit such assignments, but — if viewed just as assignments — they violate fundamental rules of methodology (information hiding, data abstraction): clients of a class should not have the ability to modify class instances directly; they should only do so through the exported procedures of the class. A typical client call may be

```
x.set_a (b)                                                 [17]
```

assuming the author of the class — who is solely responsible for deciding what clients may and may not do — has provided a procedure *set_a* that sets the value of the *a* field. The procedure might have other properties, such as imposing requirements on the new values, or triggering a database update:

```
set_a (x: T)
            -- Update a to value x.
        require
            "Some condition on x, for example to ensure compliance
            with an invariant clause involving a"
        do
            a := x
            "Possibly some other action, for example updating a log
            or database to record that a has been updated"
        ensure
            set: a = x
        end
```

While [1] is not acceptable as a way to let clients modify fields directly, some programmers may find it more directly meaningful than [17] as a notation to represent the procedure call to *set_a*.

Assigner commands provide this syntactic simplification. When you declare a query in a class, you may associate with it an assigner command; in the example this means that the author of the supplier class must have declared *a* accordingly, as

```
a: SOME_TYPE assign set_a
```

which specifies *set_a* as the assigner command associated with the query *a*. The consequence of this declaration is to make form [17], *x.a := b*, valid, with the same semantics as form [1], *x.set_a (b)*.

Form [17] is known as an Assigner_call.

Remember that it is only a syntactical convenience: Eiffel doesn't permit violating principles of information hiding and data abstraction, as would be the case if clients could directly modify fields of objects. You have no choice but to go through the official interface as defined by the supplier class author. Assigner call— available only if that author has decided to provide it, by specifying an assigner command for the query — simply lets you call the procedure through assignment-like syntax. But the instruction is still a procedure call, not an assignment.

The instruction is in fact more general than a plain assignment since it allows you to use arguments. The target query may have any number of arguments; this is what allows you to write

```
your_array.item (i) := new_value                    [18]
```

as a shorthand for the procedure call

```
your_array.put (new_value, i)                       [19]
```

This shorthand is made possible by the declaration of *item* in class *ARRAY*, which specifies put as an assigner command:

```
item (i: INTEGER): G alias "[ ]" assign put …
```

In this case the **alias** "[ ]" specification makes bracket syntax also possible, allowing the following form as a synonym for [19] and hence for [18]:

```
your_array [i] := new_value                         [20]
```

which makes traditional array assignment syntax available in a fully object-oriented context.

More generally, if *q* is a query with *n* arguments and has an associated assigner command *p*, which must have *n* + 1 arguments, you may use

$$x.\ q\ (a_1,\ a_2,\ \ldots,\ a_n) := e$$

as an abbreviation for

$$x.\ p\ (e,\ a_1,\ a_2,\ \ldots,\ a_n)$$

The syntax is straightforward:

Assigner calls

Assigner_call ≜ Expression ":=" Expression

The left-hand side is surprisingly general: any expression. The validity rule will constrain it to be of a form that can be interpreted as a qualified call to a query, such as *x.a*, or *x.f (i, j)*; but the syntactic form can be different, using for example bracket syntax as in *a [i, j] := x*.

You could even use operator syntax, as in

$$a + b := c$$

assuming that, in the type of *a*, the function *plus* **alias** "+" has been defined with an assigner command, maybe a procedure *subtract*. Then the left side *a + b* is just an abbreviation for the query call

$$a.plus\ (b)$$

and the Assigner_call is just an abbreviation for the procedure call

$$a.subtract\ (c,\ b)$$

A Call_chain — the syntax appears in the study of calls — is a dot-separated sequence of two or more features, each possibly with arguments; examples of Call_chain are

→ *Page 618.*

$$x.a$$
$$your\_array.item\ (i)$$
$$x.f\ (b).\ g\ (c,\ d)$$

Both the validity and the semantics of an Assigner_call follow from this construct's role as a syntactic simplification for a call.

As implied by the rules on assigner commands, *p* must have one more argument than the associated query *q*. Here are a few examples of assigner calls and their unfolded forms:

Assigner_call	**Unfolded form** (assuming *q* has an assigner command *p*)
x.*q* := *e*	*x*.*p* (*e*)
x.*q* (*a*) := *e*	*x*.*p* (e, *a*)
x.*f* (*a*, *b*).*q* (*c*, *d*) := *e*	*x*.*f* (*a*, *b*).*p* (*e*, *c*, *d*)

From this notion we derive the validity rule for assigner calls:

Assigner Call rule *VBAC*

An Assigner_call of the form *target* := *source*, where *target* and *source* are expressions, is valid if and only if it satisfies the following conditions:

1 • *source* is compatible with *target*.

2 • The Equivalent Dot Form of *target* is a qualified Object_call whose feature has an assigner command.

The first two clauses ensures the conditions of the definition of "unfolded form" above, so it's indeed legitimate for the third clause to to rely on the unfolded form of the instruction.

The unfolded form also gives us the semantics:

Assigner Call semantics

The effect of an Assigner_call *target* := *source*, where the Equivalent Dot Form of *target* is *x*.*f* or *x*.*f* (*args*) and *f* has an assigner command *p*, is, respectively, *x*.*p* (*source*) or *x*.*p* (*source*, *args*).

This confirms that the construct is just an abbreviation for a procedure call.

22.13 SEMI-STRICT OPERATORS

(This section is only for the benefit of readers with a taste for theory, and may be skipped. They bring new light on earlier concepts, but introduce no new language rules.)

If skipping go to "CONDITIONAL REATTACHMENT", 22.14, page 607.

The application of reattachment semantics to argument passing has the interesting consequence of making *semi-strict* implementations possible. Let us see what this means.

The notion of strictness

We may use a definition from programming theory:

For a full discussion see the book "Introduction to the Theory of Programming Languages".

Strict, non-strict

An operation is **strict** on one of its operands if it is always necessary to know the value of the operand to perform the operation.

It is **non-strict** on that operand if it may in some cases yield a result without having to evaluate the operand.

Many common operations are strict on all arguments: for example you cannot compute the sum of two integers *m* and *n* unless you know their values, so this operation is strict on both arguments.

Not all operations are strict on all arguments, however. Consider a conditional operation

***WARNING**: this is a mathematical notation, not Eiffel syntax.*

test *c* **yes** *m* **no** *n* **end**

which yields *m* if the value of *c* (a boolean) is true, *n* otherwise. This is strict on *c*, but not on the other two arguments, since it does not need to evaluate *m* when it finds that *c* is false, or to evaluate *n* when *c* is true.

Detecting that an operation is non-strict on an argument may be interesting for performance reasons (since it may avoid unnecessary computations); more importantly, however, non-strict operations may be more broadly applicable than their strict counterparts. This is immediately visible on the previous example: a fully strict version of the **test** operation would always start by evaluating *c*, *m* and *n*; but then it would fail to yield a result when *c* is true and *n* not defined, and when *c* is false and *m* not defined. A "semi-strict" version (strict on *c* but not on *m* and *n*) may, however, yield results in these cases, provided *m* is defined in the first and *n* in the second.

The need for semi-strict operators

How does this apply to Eiffel programming? Here the operations of interest are calls, of the general form

t.*r* (…, *y*, …)

and the operands are the target *t* and the actual arguments such as *y*, if any. Such a call is always strict on its target (which must be attached to an object). In a literal sense, it is also strict on its actual arguments, since it will need to pass their values to the routine *r*.

When considering an actual argument such as *y*, however, it is more interesting to analyze strictness not for the value of *y* but for the attached object, if any. Then the specification of unconditional reattachment semantics yields two cases, depending on the types of *y* and of the corresponding formal argument in *r*:

A •If both are reference types, the call passes to *r* a reference, not the attached object (which does not exist if the value of *y* is void).

B •If either type is expanded, the call passes the attached object. (The value of *y* may not be void in this case.)

Case A corresponds to case 1 of reattachment semantics, page 588, and case B to 2, 3 and 4.

In other words, taking the object to be the operand, actual-formal association is non-strict on *y* in case A, and is strict in case B.

If the target is a reference and the source is expanded (case 2 of the table), actual-formal association results in reference reattachment, but the source must first be cloned, so that the operation is indeed strict on y.

The call as a whole will be said to be strict if it is strict on all arguments, and *semi-strict* otherwise:

Semi-strict

A call is **semi-strict** if it is non-strict on one or more arguments.

This case is called "semi-strict" rather than non-strict because an Eiffel call is always strict on at least one of its operands: the call's target.

If a call may be semi-strict and you want to guarantee strictness on a particular argument without changing anything in the routine's text, this is easy: just use cloning on the actual argument, passing *clone* (*y*) rather than *y*. Function *clone* is clearly strict. The reverse change is not always possible: if the routine has a formal argument of expanded type, it will always be strict on the corresponding actuals.

What does semi-strictness mean in practice? Essentially that if both an actual argument *y* and the corresponding formal argument are of reference types the implementation **may** choose a non-strict argument passing mechanism, which evaluates *y* when and only when the routine actually needs *y*'s value.

The exception is semi-strict boolean operators, as explained below.

Such a semi-strict implementation is possible, but, except in one case, it is **not guaranteed**. Implementations are not required to use a non-strict argument passing mechanism even if the formal and actual arguments are both references. This means that when you write a call of the form

```
t.r (..., y, ...)
```

you must make sure that the value of *y*, which may be a complex expression, is always defined at the time of call execution — even in cases for which *r* does not actually need that value. The call may evaluate *y* anyway.

Consider for example a routine

```
too_strict_for_me
        (i: INTEGER; arr: ARRAY [REAL]; val: REAL): REAL
    do
        if  i > = arr.lower and i < = arr.upper  then
            Result := val
        end
    end
```

which returns the value of its last argument if its first argument, *i*, is within the bounds of the middle argument, an array, and returns 0.0 (the default value for *REAL*) otherwise. Then consider a call in the same class:

```
your_array: ARRAY [REAL]; a: REAL; n: INTEGER
…
a := too_strict_for_me (n, your_array @ n) )
```

WARNING: *potentially incorrect*!

If the value of *n* may be outside of the bounds of *your_array*, then this call is not correct since *your_array @ n*, denoting the *n*-th element of *your_array*, is not defined in this case. Semi-strict implementation (non-strict on the last argument) would avoid evaluation of *some.array @ n* and hence ensure proper execution of the call, returning zero; but you may **not** assume that the implementation uses this policy.

There is, however, one exception. As will be seen in detail in the discussion of operator expressions, three functions of the Kernel Library class *BOOLEAN*, are required to be semi-strict (that is to say, non-strict on their single argument). These are functions representing a variant of the common boolean operations: and, or, implies. Their declarations in class *BOOLEAN* are

→ *"SEMISTRICT BOOLEAN OPERATORS", 28.6, page 765*.

```
conjunction_semistrict alias "and then"
        (other: BOOLEAN): BOOLEAN is do … end;
disjunction_semistrict alias "or else"
        (other: BOOLEAN): BOOLEAN is do … end;
implication alias "implies"
        (other: BOOLEAN): BOOLEAN is do … end;
```

The semantics of these functions readily admits a semi-strict interpretation: *a* **and then** *b* should yield false whenever *a* is false, regardless of the value of *b*, and similarly for the others. To state this property concisely for all three operations, it is useful to express the value of each, as applied to arguments *a* and *b*, in terms of the above *ad hoc* **test** notation:

```
test not a yes false no b end
test a yes true no b end
test not a yes true no b end
```

Remember that an operator expression such as a **and then** *b stands for a call of target a and actual argument b. This explains why all the expressions considered here are strict on a, since a call is always strict on its target. See "THE EQUIVALENT DOT FORM", 28.8, page 771.*

This semi-strictness of these boolean operators is important in practice because it makes it possible to use them as conditional operators. As a typical example, again using arrays, it is often convenient to write instructions of the form

```
if
        i >= your_array.lower and then
        i <= your_array.upper and then
        (arr @ n).your_property
then
…
```

where the last condition is not defined unless the first two are true (because *i* would then be outside of the bounds of *arr*). In the absence of a semi-strict version of "and", it would be much more cumbersome (as Pascal programmers know) to express such examples.

The discussion of boolean operators will show further uses of this semi-strict policy, especially for writing iterators on data structures, with examples from the EiffelBase library.

→ *See for example continue_until from LINEAR_ITERATION on page ====*

More on strictness

(This more theoretical section may be skipped on first reading.)

What about the ordinary boolean operators **and** and **or**? You may expect them to have a strict semantics, but this is not the case — at least not necessarily. Here the language definition is simply less tolerant: it makes it incorrect to evaluate expressions *a* **and** *b* and *a* **or** *b* when *b* is not defined, even if *a* has value false in the first case and if *b* has value true in the second case. There is nothing surprising in this convention, which has its counterpart in all other forms of expression except those involving semi-strict operators: no rule in this book will tell you how to compute the value of *m* + *n* if the value of the integer expression *n* is not defined.

Because the language definition does not cover cases in which the second operand of **or** or **and** has no value, an implementation that uses **and then** to compute **and**, and **or else** to compute **or**, is legitimate; it may produce results in cases for which a strict implementation would not, but these cases are incorrect anyway.

The reverse is not true: a correct implementation of **and** and **or** does not necessarily provide a correct implementation of **and then** and **or else** since it may be strict. In other words: non-semi-strict does not necessarily mean strict! If you want to guarantee strictness, it does not suffice to rely on the operator **and** and the operator **or**; you should use cloning as suggested above. (For **implies**, which is semi-strict, there is no equivalent non-semi-strict operator, but you can use **not** *a* **or** *b*.)

It is legitimate to ask why the semi-strict property of three boolean operators — **and then**, **or else**, **implies** — is not expressed as part of the language syntax. One could indeed envision a special optional qualifier **nonstrict** applicable to formal arguments of reference type:

```
implication alias "and then"
        (nonstrict other: BOOLEAN): BOOLEAN
```

WARNING: *not legal Eiffel*!

Such a facility was not, however, deemed worth the trouble, since the common practice of software development seldom requires semi-strictness outside of two special cases: the three boolean operators just studied; and, as we will see in the relevant chapter, concurrent computation.

→ *Chapter 33*.

22.14 CONDITIONAL REATTACHMENT

To complete the study of reattachment, there remains to see one mechanism which, like the operations examined so far, may reattach a reference to a different object. The semantics will in fact be reference reattachment; what differs is the validity constraint under which you may apply this mechanism, and also the conditional nature of its effect.

--- REPLACE WITH A SHORT PREVIEW OF Object_test

Limitations of unconditional reattachment

The need for a conditional form of reattachment arises when you must access an object of a certain type *TX*, but the only name you have to denote that object is an expression of type *TY*, for two different types with the "wrong" conformance (*TX* conforms to *TY* rather than the reverse), or even no conformance at all. Normally, you would use the assignment

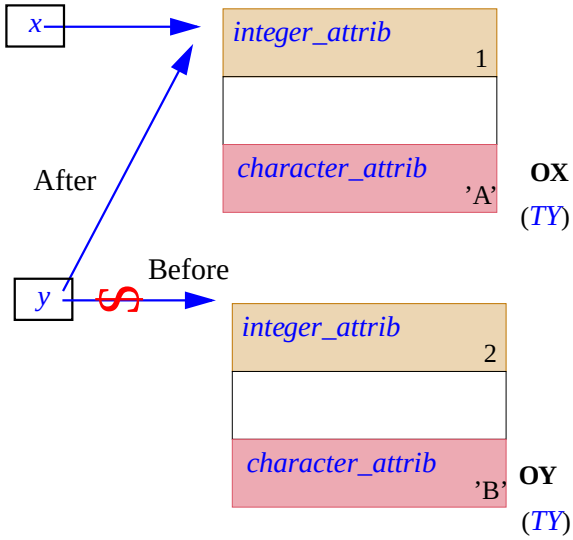
```
x := y
```

with *x* of type *TX*; but this will not work because the fundamental constraint of unconditional reattachment, expressed in the Assignment rule, assumes conformance from *y* to *x*. Calling a routine with *y* as actual argument corresponding to a formal argument *x* of type *TX* would also be invalid for the same reason. This conformance property is essential to the soundness of the type system.

22.15 MEMORY MANAGEMENT

A practical consequence of the reference reattachment mechanism, both in the unconditional form (assignment, argument passing) and in the conditional form (assignment attempt), is that some objects may become useless. This raises the question of how, if in any way, the memory space they used may be reclaimed for later use by newly created objects.

For example, the reference reattachment illustrated by the figure below may make the object labeled OY unreachable from any useful object.



Effect of reference reattachment

This is the same as the second figure of page 588.

In a similar way, the result of a cloning operation may make an object unreachable. This may be the case with the middle object (also labeled OY) in the earlier illustration of cloning.

← *First figure on page 587.*

What does it mean for an object to be ‘"useful"? Remember that the execution of a system is the execution of a creation procedure (the root creation procedure) on an object (the root object, an instance of the system’s root class). The root object will remain in place for the entire duration of the system’s execution. An object is useful if it may be reached directly or indirectly, following references, from the either root object or any of the local variables of a currently executing routine. Because a non-useful object can have no effect on the remainder of the system’s execution, it is permissible to reclaim the memory space it uses.

← *“System execution”, page 114.*

Should a reattachment as illustrated above (or its clone variant) automatically result in freeing the associated storage? Of course not. The object labeled OY may still be reachable from the root through other reference paths.

It would indeed be both dangerous and unacceptably tedious to lay the burden of object memory reclamation on developers. Dangerous because it is easy for a developer to forget a reference, and to recycle an object’s storage space wrongly while the object is still reachable, resulting in disaster when a client later tries to access it; and unacceptably tedious because, even if you know for sure that an object is unreachable, you should not just recycle its own storage but also analyze all its references to other objects, to determine recursively whether other objects have also become unreachable as a result. This makes the prospect of manual reclamation formidable.

Authors of Eiffel implementation are encouraged to provide a **garbage collection** mechanism which will take care of detecting unreachable objects. Although many policies are possible for garbage collection, the following properties are often deemed desirable:

- Efficiency: the overhead on system execution should be low.
- Incrementality: it is desirable to have a collector which works in small bursts of activity, being triggered at specified intervals, rather than one which waits for memory to fill up and then takes over for a possibly long full collection cycle. Interactive applications require bursts to be (at least on average) of a short enough duration to make them undetectable at the human scale.
- Tunability: library facilities should allow systems to turn collection off (for example during a critical section of a real-time application) and on again, to request a full collection cycle, and to control the duration of the bursts if the collector is incremental.

The Kernel Library class “MEMORY”, A.6.25 CLASS, page 996, provides such facilities.

22.16 SEMANTICS OF EQUALITY

The previous discussions have shown how to reattach values. A closely related problem, whose study will conclude this chapter, is to **compare** values, for example to see if they are attached to the same object. This raises the question of the semantics of the equality operator = and its alter ego the inequality operator /=.

If you remember how the study of object duplication (*copy*, *clone* and variants) led us to object comparison (*equal* and its variants), you will probably have anticipated the current section: just as the assignment operator := has the semantics of reference attachment, copy or clone depending on the expansion status of its operands, so will the equality operator = have the semantics of reference or object equality.

← "OBJECT EQUALITY", 21.6, page 572.

We can devote all our attention to equality since inequality follows: the effect of *x* /= *y* is defined in all cases to be that of

```
not (x = y)
```

Two meanings of equality are *a priori* possible: reference equality, true if and only if two references are either attached to the same object or both void; and object equality.

The previous chapter introduced a function to test object equality: *equal* from the universal class *ANY*, which in its original version will return true if and only if two objects are field-by-field equal. As with copying and cloning operations, it is more prudent to rely on the frozen version *identical*, guaranteeing uniform semantics. (By redefining *is_equal*, you may provide another version of *equal* for a specific class.) For convenience, *identical* (like *equal*) also applies to void values. In the present discussion, "object equality" denotes an operation that can only compare two objects, and so must be applied to non-void references.

← "OBJECT EQUALITY", 21.6, page 572.

Here is the table of possibilities, which closely parallels the corresponding table for unconditional reattachment:

← Page 588.

Possible semantics for shallow equality

NOT a semantic specification but only a list of available possibilities for such a specification. The actual semantics appears next.

TYPE OF FIRST→ *TYPE OF SECOND* ↓	**Reference**	**Expanded**
Reference	**[1]** • Reference equality • Object equality (if neither void)	**[2]** • Object equality

Expanded	**[3]** • Object equality (if first not void)	**[4]** • Object equality

For each of the four cases, we must give a reasonable meaning to the equality operator =. The line of reasoning applied earlier to unconditional reattachment yields the following semantics, which again parallels the table for unconditional reattachment.

← *Page 590.*

TYPE OF FIRST→ *TYPE OF SECOND* ↓	**Reference**	**Expanded**
Reference	**[1]**Reference equality	**[2]***identical*
Expanded	**[3]***identical*	*identical*

So if *x* and *y* are references the result of a test

```
x = y
```

is true if and only if *x* and *y* are either both void or both attached to the same object; if either or both of *x* and *y* are objects, then the test yields true if and only if they are attached to field-by-field equal objects, as indicated by function *identical_equal* from class *ANY*.

As with unconditional reattachment, the semantics given is the most frequently needed one for each case, and in particular is usually appropriate for operations on arguments of a Formal_generic_name type. For more specific semantics, you may use one of the calls

```
equal (x, y)
deep_equal (x, y)
identical_equal (x, y)
identical_deep_equal (x, y)
```

Many container classes of EiffelBase have routines that query a data structure such as a list, set, tree or hash table for occurrences of an object (or more generally a value). This may mean either of two things: does the structure contain a reference to the object of interest? Does it contain a reference to an object equal to it? You can switch between these two interpretations by applying the procedures *compare_objects* and *compare_references* to a certain container, as in *my_list*.*compare_objects*. This governs not only searching operations, such as the function *has*, but also certain insertion and replacement operations that will only add an element to a structure if it is not already present.

See "Reusable Software". The notion of container data structure was presented in 10.21, page 286, and 12.2, page 341.

For basic arithmetic types, which are expanded, the = and /= operators will always call *identical*. Thanks to the conversion mechanism studied earlier in this chapter, you may use mixed-type equality expressions within the limits of the conversions specified in the corresponding classes. For example the expression *1.0 = 1* is valid (and will return true) even though it has a *REAL* operand and the other is an *INTEGER*. This is because according to the above semantics the expression means *1.0*.*identical* (*1*), and *INTEGER* converts to *REAL*. Thanks to the target conversion mechanism, you may also write *1 = 1.0*, with the same result.

← *"CONVERSIONS", 22.6, page 583*.

← *"Accounting for target conversion", , page 762*