23

Feature call

23.1 OVERVIEW

How does a software system perform its job — its computations?

It must first set the stage: create the needed objects and attach them to the appropriate entities. The preceding chapters discussed how to do this. But once it has the objects in place and knows how to access them, the system should do something useful with them.

In Eiffel's model of computation, the fundamental way to do something with an object is to apply to it an operation which — because the model is class-based, and behind every run-time object lurks some class of the system's text — must be a feature of the appropriate class.

This is feature call, one of the most important constructs in Eiffel's object-oriented approach, and the topic of the following discussions.

One of the risks with calls in object-oriented languages is the *void call*: a run-time attempt to apply a feature to an object that doesn't exist because a reference is void (or, in other terminology, a pointer is null). Eiffel distinguishes itself by making such a failure impossible thanks to the notion of *attached type* and associated constructs studied in previous chapters. Here we will reap the benefits of these mechanisms, which ensure statically — at compile time — that no Eiffel call can apply to a void target. This removes the principal source of run-time failure in object-oriented programming.

Three topics related to calls merit their own discussions in other chapters:

- The validity of calls raises the general question of **type checking**: how to make sure that the target of every call will be an object equipped with the appropriate feature. → *Chapter 25*.
- A call has a **target**, which must be an object. If the target is known through a reference, we must be sure that the reference will never be void upon execution of the call. → *Chapter 24*.
- **Operator expressions** are conceptually calls, but use traditional mathematical syntax. We'll see them as part of the chapter on expressions, although there will be little new to learn about their validity and semantic properties, which are those of calls. → *Chapter 28*.

23.2 PARTS OF A CALL

A call is the application of a certain feature to a certain object, possibly with arguments. As a consequence, it has three potential components:

- The *target* of the call, an expression whose value is attached to the object.
- The *feature* of the call, which must be a feature of the object's type.
- An *actual argument list*.

The target and argument list are optional; the feature is required.

Here is a typical example showing all three components:

```
remote_bank.transfer_by_wire (20000, Today)
```

This call uses **dot notation**. The target of the call is *remote_bank*; the feature of the call is *transfer_by_wire*; and the actual argument list contains the two elements 20000 and *Today*.

The target is separated from the feature of the call by a period, or *dot*, hence "dot notation".

If the target is the predefined entity **Current**, representing the "current object" of system execution, as explained below, you may use, instead of the fully qualified form

→ "Current object, current routine", page 641.

```
Current.print (message)                                    [1]
```

a form which leaves the target implicit:

```
print (message)                                            [2]
```

This is still considered to be a case of dot notation even though the dot is implicit. If the call does include an explicit target and dot, it is **qualified**; otherwise, as in the last example, it is **unqualified**.

In the presence of run-time assertion monitoring, there is a slight semantic difference between [1] and [2]: a qualified call causes invariant checking, an unqualified call doesn't.

A qualified call may have more than one level of qualification and is then said to be a **multidot** call, as in

```
paragraphs (2).line (3).second_word.set_font (Bold)
```

For a feature without arguments, the actual argument list will be absent, as in the source expression of the Assignment

```
code := remote_bank.authorization
```

where *authorization* is a query (attribute or function) without arguments.

In some cases we don't need a target object (as in a qualified call) but we still need a target type. If *T* is a type, the notation

{T}.constant_or_external

denotes a call to a feature *constant_or_external* from *T*. This only makes sense if the feature is either a constant attribute or an external (non-Eiffel) feature; anything else would require a target object.

Non_object_call is a shorthand for "non-object-*oriented* call", as in *{T}.f (args)* where *T* is a type. The usual object-oriented style of computation, *x.f (args)*, requires a target object denoted by *x*.

Calls may appear in syntactic forms other than dot notation:

- **Operator expressions**, have the semantics of calls: *a – b* is, with a feature *minus* **alias** "–", equivalent to the dot-notation call *a.minus (b)*. Similarly, with *item* **alias** "[]", the expression *x [i]* has the same semantics as the dot-notation call *x.item [i]*.
- You may also write a **non-object call** of the form *{T}.f* where *T* is a type and *f* is either a constant attribute or an external feature of *T*. This is like a call in dot notation that would not need a target, but only a target type (to determine which *f* to use).

23.3 USES OF CALLS

A call may play either of two syntactic roles: instruction and expression.

A call is a specimen of construct Call, covering dot notation, qualified or unqualified, and non-object calls.

Operator_expression (in prefix or infix notation) and Bracket_expression are always used as expressions, but a Call in dot notation may be either an instruction or an expression. The syntax productions for both the Instruction and Expression constructs indeed include Call as one of the choices. To know which one applies, it suffices to look at the feature of the call:

Instruction: *page 224*; Expression: *page 753*.

Call Use rule *VUCN*

A Call of feature *f* denotes:

1 • If *f* is a query (attribute or a function): an expression.

2 • If *f* is a procedure: an instruction.

This rule has a validity code, so that compilers and other language processing tools may refer to it when detecting an error such as the use of a procedure call in an expression.

The above examples used calls to *transfer_by_wire*, *print* and *set_font* as instructions, and a call to *authorization* as an expression. The calls to *minus* **alias** "–" and *item* **alias** "[]" are also expressions. The non-object call *{T}.f* is an instruction if *f* is a procedure of *T* and an expression otherwise.

23.4 UNIFORM ACCESS

An important property applies to dot-notation calls used as expressions: the notation is exactly the same whether the feature of a Call is a function with no arguments or an attribute. The expression

```
pl.age
```

where *p1* is of type *PERSON* is applicable both if the feature *age* of class *PERSON* is a feature of either kind.

If *age* is an attribute, every instance of *PERSON* has a field which gives the value of *age* for the instance. If *age* is a function, that value is obtained, when requested, through some computation, presumably of the difference between the current date and a "‘birth date" field. For a client containing the above call, however, this makes no difference.

This principle of **uniform access** facilitates smooth evolution of software projects by protecting classes from internal implementation changes in their suppliers.

← *First discussed in "UNIFORMACCESS", 23.4, page 616.*

23.5 OPERATOR AND BRACKET FORMS

A call serving as an expression may use, instead of dot notation, the Operator_expression form based on unary or binary operators. Both of the two operator expressions, respectively unary (prefix) and binary (infix)

```
– 1
4 – 3
```

are calls to functions of the Kernel Library class *INTEGER*: the first, to the function *negated* **alias** "–"; the second, to *minus* **alias** "–". The Feature Declaration rule requires a feature associated with a unary or binary operator to be an attribute or function without argument, like *negated*, or a function with one argument, like *minus*. Note that here although both are associated with the same operator – there is no ambiguity since the same rule guarantees that there is at most one feature for each of these signatures.

← *Page 160, clause 7; see clause 1.of "Alias Validity rule", page 162.*

The difference between such an operator expression and a Call is only syntactical. You may also write the above two expressions as:

```
(|1|).negated
(|4|).minus (3)
```

with exactly the same effect.

The syntax of Call requires putting in " target parentheses" **(|** . . . **|)** around a Manifest_constant, such as 1 or 4, to use it as target of a call.

→ *"COMPLEX TARGETS", 23.6, page 617 below.*

Similarly, a bracket expression such as

```
your_array [some_index]
```

based on the feature *item alias* "[ ]" in class *ARRAY*, has the exact same semantics as

```
your_array.item (some_index)
```

The discussion of expressions will formalize the correspondence between the two syntactic forms by defining an **Equivalent Dot Form** for any operator expression.

→ *"THE EQUIVALENT DOT FORM", 28.8, page 771*.

23.6 COMPLEX TARGETS

In most cases the target *x* of a call *x.f (...)* is just an entity: a local variable, an attribute, a formal argument. Sometimes you may want to use a non-elementary expression, such as *a + b* (where *a* and *b* could be not just numbers but, for example, of some type *MATRIX*). Writing *a + b.f (c)* would, according to precedence rules, denote a sum of two elements, *a* and the application of *f* to *b*. If that's not what you want, you may use a local variable to specify applying *f* instead to the sum of *a* and *b*:

```
local
    sum
do
    ... sum := a + b
    x := sum.f (c) ...
end
```

This technique works but forces the introduction of extra local variables. To avoid them you may use the **parenthesized target** notation **(|** *Expression* **|)**:

```
x := (| a + b|).f (c)
```

The symbols use parentheses and a vertical bar. They remove any ambiguity by making clear that the feature, *f* in this example, is being applied to the whole expression.

You may also use a parenthesized target in connection with bracket notation, as in **(|** *a + b* **|) [***i***]**, assuming the type of *a + b* has a bracket feature.

→ *"BRACKET EXPRESSIONS", 28.7, page 769 ; see the syntax on page 769.*

Note that just using parentheses, as in *(a + b). f (x)*, would not be legal syntactically.

Why indeed not just use plain parentheses? The reason is syntactical. Eiffel **always** treats the semicolon separator as redundant, without making any difference between spaces, new lines and other break characters. If a parenthesized expression were permitted as target of a call, the assertion

```
require
    h
    (a + b).g
```

0would include two clauses,. But syntactically the beginning could be parsed as *h (a + b)*, denoting the application of a function *h* to an argument *a + b*, even though the remainder, **.***g*, doesn't have a proper syntactical interpretation.

This syntactical problem is typical of the confusion engendered by the dual use of parentheses, coming from mathematical conventions: as a *grouping* mechanism, as in *(a + b)*; and as a notation for *function application*, as in *f (c)*. The special symbols **(|** . . . **|)** avoid any such ambiguity.

23.7 CALL SYNTAX

We'll now examine the syntax of the construct Call, describing calls in dot notation, qualified or not, and non-object calls.

Prefix

, infix and bracket forms are specimens of Expression; we'll see their syntax in the corresponding chapter, which also defines their semantics in terms of the semantics of calls.

→ "GENERAL FORM OF EXPRESSIONS", 28.2, page 753 and rest s of chapter 28.

Feature calls

Call ≜ Object_call | Non_object_call

Object_call ≜ [Target "**.**"] Unqualified_call

Unqualified_call ≜ Feature_name [Actuals]

Target ≜ Local | Read_only | Call | Parenthesized_target

Parenthesized_target ≜ "**(|**" Expression "**|)**"

Non_object_call ≜ "**{**" Type "**}**" "**.**" Unqualified_call

A call is most commonly of the form *a***.***b***.**. . . where *a*, *b* . . . are features, possibly with arguments. Target allows a Call to apply to an explicit target object (rather then the current object); it can itself be a Call, allowing multidot calls. Other possible targets are a local variable, a Read_only (including formal arguments and **Current**) a "non-object call" (studied below), or a complex expression written as a Parenthesized_target (|. . .|).

When present, the optional Actuals part gives the list of actual arguments:

Actual arguments

Actuals ≜ "**(**" Actual_list "**)**"

Actual_list ≜ {Expression "**,**" . . .}$^+$

As the specification of Actual_list indicates, an Actuals argument list may not be empty: if *f* has no formal arguments, you must call it as *f* or *x*.*f*, not *f* () or *x*.*f* (). This is for simplicity and clarity.

An object-oriented call is either *qualified* or not. It's qualified if it involves at least one dot:

> **Unqualified, qualified call**
>
> An Object_call is **qualified** if it has a Target, **unqualified** otherwise.

The Address *form for* Actual *serves to pass the address of an Eiffel feature to a foreign (non-Eiffel) routine. See 31.8, page 823.*

In equivalent terms, a call is "unqualified" if and only if it consists of just an Unqualified_call component.

The call *f* (*a*) is unqualified, *x*.*f* (a) is qualified.

Another equivalent definition, which does not explicitly refer to the syntax, is that a call is qualified if it contains one or more dots, unqualified if it has no dots — counting only dots at the dot level, not those that might appear in arguments; for example *f* (*a*.*b*) is unqualified.

Of our earlier examples

```
print (message)
paragraph (2).line (3).second_word.set_font (Bold)
```

the first is unqualified and the second qualified. Both are instructions if we assume that *print* and *set_font* are procedures in their respective classes. The intermediate components of the second example

```
paragraph (2)
paragraph (2).line (3)
paragraph (2).line (3).second_word
```

are all specimens of construct ----------- FIX . They may themselves be viewed as calls; any such intermediate call must be an expression (rather than an instruction) so that it may serve as the target of further calls.

The features of all examples so far have arguments. Here are two examples where the call has no argument:

```
paragraph (2).indent;
f := that_word.current_font
```

They assume that *indent* is a procedure with no arguments and that *current_font* is an attribute or function without arguments. As a result, the source the following assignment, in the last example, is a call expression.

For examples of calls using a Parenthesized_target, in addition to *(|1|)•negated* and *(|4|)•minus (3)* (more simply written as –4 and 4 – 3), assume a class *VECTOR* with features *norm* and *plus*:

```
class VECTOR [G –> X] feature
    norm: G is do … end;
    plus alias "+" (other: like Current): like Current
        do … end
    … Other features …
end
```

Then with *a* and *b* of type *VECTOR [T]* for some appropriate *T* you may use the expressions

```
(|u + v|)•norm
(|u + v|)•norm•f•h.
```

f must be a feature of class X, hence applicable to (u + v)•norm since the type G of this expression, a formal generic parameter of VECTOR, is constrained by X.

both of which apply function *norm* to the result of applying function *plus* to *u* with argument *v*. The syntax specification allows for at most one Parenthesized_target, at the beginning of the Call. In the second example the Parenthesized_target is followed by the Call *norm•f•h*.

Thanks to this mechanism, you may use any valid expression as qualifier by parenthesizing it. Without parentheses, the Call would be syntactically illegal, as in *3•negated*, or legal but with a different semantics, as with *u + v•norm* which applies *norm* to *v*, not to the sum.

→ The dot has the highest precedence of all operators except parentheses, so in the second case it applies to v, not u + v. *See "SUBEXPRESSIONS", 28.3, page 756*

23.8 COMPONENTS OF A CALL

It is convenient to talk about "the target", "the target type" and "the feature" of a call.

> **Target of a call**
>
> Any Object_call has a **target**, defined as follows:
>
> 1 • If it is qualified: its Target component.
>
> 2 • If it is unqualified: **Current**.

The target is an expression; in *a (b, c)•d* the target is *a (b, c)* and in *(| a (b, c) + x |)•d* the target (case 1) is *a (b, c) + x*. In a multidot case the target includes the Call deprived of its last part, for example *x•f (args)•g* in *x•f (args)•g•h (args1)*.

A Non_object_call does not have a target; this is what distinguishes it from an Object_call. In both cases, however, there is a target *type*:

> **Target type of a call**
>
> Any Call has a **target type**, defined as follows:
>
> 1 • For an Object_call: the type of its target. (In the case of an Unqualified_call this is the current type.)
>
> 2 • For a Non_object_call having a type *T* as its Type part: *T*.

A call of any kind also has a feature:

> **Feature of a call**
>
> For any Call the "**feature of the call**" is defined as follows:
>
> 1 • For an Unqualified_call: its Feature_name.
>
> 2 • For a qualified call or Non_object_call: (recursively) the feature of its Unqualified_call part.

Case 1 tells us that the feature of *f (args)* is *f* and the feature of *g*, an Unqualified_call to a feature without arguments, is *g*.

The term is a slight abuse of language, since *f* and *g* are feature names rather than features. The actual feature, deduced from the semantic rules given below and involving dynamic binding. is the **dynamic feature** of the call.

→ "Dynamic feature of a call", page 631

It follows from case 2 that the feature of a qualified call *x*.*f (args)* is *f*. The recursive phrasing addresses the multidot case: the feature of *x*.*f (args)*.*g*.*h (args1)* is *h*.

23.9 NON-OBJECT CALLS

The remaining sections of this chapter discuss the validity and semantics of calls. The most interesting cases are the object-oriented form of call, *x*.*f (args)*, involving dynamic binding, and its unqualified variant *f (args)*. They will occupy most of the discussion. Let us dispose first of a specific case, available mostly to facilitate interaction with non-object-oriented facilities: Non_object_call. In an example such as

```
{CHARACTER_CODES}.Underscore
```

we use a Non_object_call to access directly a constant attribute present in a "utility class", *CHARACTER_CODES*. Were this mechanism not available in the language, you could still obtain the desired effect by either:

- Making the enclosing class inherit from *CHARACTER_CODES*, so that it can directly access its features such as *Underscore*.

- Declaring an entity *codes*: *CHARACTER_CODES* and using *codes*.*Underscore*.

Using inheritance as in the first solution is a bit heavy-handed for such a simple purpose. With the second solution, you must declare an entity that you won't use for anything else; in addition, if *CHARACTER_CODES* is not an expanded class, you'll have to perform a creation instruction **create** *codes* to obtain the corresponding object. All this is a diversion. With the Non_object_call you state, with no fuss, exactly what you need: feature *Underscore* from class *CHARACTER_CODES*.

The mechanism is applicable only in limited cases: we only allow {*T*}.*f* ... if *f* is a constant, like *Underscore*, or an external (non-Eiffel) function, as in

```
{NETWORK_CONTROLLER}.open_channel (port_number, timeout)
```

The reason is that any feature other than a constant attribute or an external feature might need to work on the target, which a Non_object_call lacks. Even an external feature could be a problem through its assertions: consider a call

```
open_channel (pn: INTEGER; to: REAL)
		-- Open a channel on port number pn with timeout to.
	require
		valid_state
	external
		"C"
	end
```

Warning: makes above Non_object_call *invalid.*

where *open_channel*, in class *NETWORK_CONTROLLER*, is an external routine with two arguments. The precondition has an Unqualified_call to *valid_state*, a function that might use the current object. Or it might not; but this can be tricky to determine, so we should just ban such assertions.

To specify both the validity and the semantics it is convenient to treat a Non_object_call as a special case of an Object_call:

Imported form of a Non_object_call

The **imported form** of a Non_object_call of Type *T* and feature *f* appearing in a class *C* is the Unqualified_call built from the original Actuals if any and, as feature of the call, a fictitious new feature added to *C* and consisting of the following elements:

1 • A name different from those of other features of *C* .

2 • A Declaration_body obtained from the Declaration_body of *f* by replacing every type by its deanchored form, then applying the generic substitution of *T*.

This definition in "unfolded" style allows us to view *{T}.f (args)* appearing in a class *C* as if it were just *f (args)*, an Unqualified_call, but appearing in *C* itself, assuming we had moved *f* over — "imported" it — to *C*.

← *"TWO-TIER DEFINITION AND UNFOLDED FORMS", 2.11, page 99.*

In item 2 we use the "deanchored form" of the argument types and result, since a type **like** *a* that makes sense in *T* would be meaningless in *C*. As defined in the discussion of anchored types, the deanchored version precisely removes all such local dependencies, making the type understandable instead in any other context.

← *"Deanchored form of a type", page 337.*

This notion helps us express the validity rule:

Non-Object Call rule *VUNO*

A Non_object_call of Type *T* and feature *fname* in a class *C* is valid if and only if it satisfies the following conditions:

1 • *fname* is the final name of a feature *f* of *T*.

2 • *f* is available to *C*.

3 • *f* is either a constant attribute or an external feature whose assertions, if any, use neither **Current** nor any unqualified calls.

4 • The call's imported form is a valid Unqualified_call.

Condition 2 requires *f* to have a sufficient export status for use in *C*; there will be a similar requirement for Object_call. Condition 3 is the restriction to constants and externals. Condition 4 takes care of the rest by relying on the rules for Unqualified_call.

→ *Through the notion of export validity defined in the next section.*

We also use the imported form to define the semantics:

Non-Object Call Semantics

The effect of a Non_object_call is that of its imported form.

23.10 CLASS VALIDITY

The rest of this chapter considers the most common — but also more delicate — case: object calls, involving dynamic binding. First, validity.

The basic idea is straightforward: in *x.f (args)* appearing in a class *C*, the base class of *x* must have a feature *f*, that feature must be available (exported) to *C*, and the elements of *args* must conform to the corresponding formal arguments as declared for *f*; in addition, the type of *x* must be *strict* to avoid the possibility of calls on a void target. In the unqualified version *f (args)*, *r* must be a feature of the current class and the arguments must conform. For the overwhelming majority of cases this is all you need to remember.

The full story is more subtle; in fact the next two chapters are devoted to filling in the details. In the present discussion we will examine the **Class-Level validity** of a call, which it is convenient to define in four parts:

- **Export validity**, to ensure that *f* is exported to the client class.
- **Argument validity**, to ensure that the *args* are of the right number and type.
- **Target validity**, to ensure that *x* is not void.

Target validity is defined in the next chapter; the following one will tackle the remaining notion of *System-Level validity.*

Elsewhere in this book, validity rules are of the form: "A specimen of construct C is valid if and only if …". The rules of this section appear instead as: "A Call is *X*-valid if and only if …", where *X* is one of Export, Argument, Target and Class-Level. The following chapter will define a Call as "valid", without further qualification, if and only if it is System-Level-valid and Class-Level-valid. Since the three components of Class-Level validity address distinct aspects, it is convenient for compilers to produce error messages that refer to each of them; so you can view the rules below, as normal validity rules, except that they are "only if" but not "if".

Export validity

The first of the three components of Class-Level validity, export validity, ensures that the caller is entitled to use the "feature of the call":

Export rule *VUEX*

An Object_call appearing in a class *C*, with *fname* as the feature of the call, is **export-valid** for *C* if and only if it satisfies the following conditions.

1 • *fname* is the final name of a feature of the target type of the call.

2 • If the call is qualified, that feature is available to *C*.

This defines export validity "for" a certain class *C*. Usually we consider a call appearing in a given class text, so we say just "export valid" to mean export-valid for the current class. In the discussion of type checking, we'll need to consider the call, and its export validity, for an arbitrary descendant of the original class.

For an unqualified call *f* or *f* (*args*), only condition 1 is applicable, requiring simply (since the target type of an unqualified class is the current type) that *f* be a feature, immediate or inherited, of the current class.

For a qualified call *x*.*f* with *x* of type *T*, possibly with arguments, condition 2 requires that the base class of *T* make the feature *available* to *C*: export it either generally or selectively to *C* or one of its ancestors. (Through the Non-Object Call rule this also governs the validity of a Non_object_call {*T*}.*f*.)

← *As defined in "Available for call, available", page 206.*

As a consequence, *s* (…) might be permitted and *x*.*s* (…) invalid, even if *x* is **Current**. The semantics of qualified and unqualified calls is indeed slightly different; in particular, with invariant monitoring on, a qualified call will — even with **Current** as its target — check the class invariant, but an unqualified call won't.

Clause 2 only applies to qualified calls. Clearly, a routine *r* of a class *C* can call another routine *s* of *C* on the current object unqualified, regardless of the export status of *s*. But in a qualified call *x*.*s* (…) the routine *s* must always be exported to *C*, even if *x* is of type *C*.

Because this property sometimes surprises programmers accustomed to the conventions of other languages, it is useful to make it prominent:

Export Status principle

The export status of a feature *f*:

- Constrains all qualified calls *x*.*f* (…), including those in which the type of *x* is the current type, or is **Current** itself.
- Does not constrain unqualified calls.

This is a validity property, but it has no code since it is not a separate rule, just a restatement for emphasis of condition 2 of the Export rule.

That clause also takes care of the multi-dot case: in *a*.*b*.*c*, the target, *a*.*b*, must itself satisfy the same condition. (This use of recursion is justified since the target has one more level of dot notation than the original Call, so the recursion cannot go on forever.)

In such multi-dot calls, all that counts is availability to the class *C* where the call appears; availability to intermediate classes is irrelevant. For example, if *C* contains the call

```
next_paragraph.line (3).second_word.set_font (Bold)        [3]
```

where successive features are of types *PARAGRAPH*, *LINE* and *WORD*, export validity means that *PARAGRAPH* must make function *line* available to *C*, *LINE* must make *second_word* available to *C*, and *WORD* must make *set_font* available to *C*. It does not matter whether *second_word* is available to *PARAGRAPH*, or *set_font* is available to *LINE*. To understand why, note that any such call may be rephrased in single-dot form:

```
l: LINE; w: WORD
...
l := next_paragraph.line (3)
w := l.second_word
w.set_font (Bold)
```

This shows multi-dot notation as just a notational facility — although an important one, avoiding the need for intermediate variables such as *l* and *w*.

Argument validity

The second component of Class-Level validity ensures that the number and types of actual arguments match those of formals:

Argument rule *VUAR*

An export-valid call of target type *ST* and feature *fname* appearing in a class *C* where it denotes a feature *sf* is **argument-valid** if and only if it satisfies the following conditions:

1 • The number of actual arguments is the same as the number of formal arguments declared for *sf*.

2 • Every actual argument of the call is compatible with the corresponding formal argument of *sf*.

→ *The S in ST and sf is for "static". See "Descendant Argument rule", page 659.*

For simplicity, the definition assumes export validity, ensuring that *f* exists.

Condition 2 is the fundamental type rule on argument passing, which allowed the discussion of direct reattachment to treat Assignment and actual-formal association in the same way. An expression is *compatible* with an entity if its type either conforms or converts to the entity's type.

← *Page "ROLE OF REATTACHMENT OPERATIONS", 22.2, page 580.*

In a generic context, condition 2 relies on the Generic Type Adaptation rule: in a call *a.sf (y)* where *a* is of type *C [T]* and *C [G]* has the routine *sf (x: G)*, the type to which *y* must conform is *T* — not *G*, which makes no sense outside of the text of *C*.

← *Page 359.*

A call to a feature with no arguments trivially satisfies the Argument rule if it doesn't include any Actuals. As noted at the beginning of this chapter, it's syntactically illegal to write a call as *f* () or *x*•*f* (); either the feature has formal arguments and you must specify the corresponding Actuals in parentheses, or it doesn't and you just don't include any Actuals list.

A consequence of the Arguments rule is that Eiffel doesn't directly allow a routine to be called with a variable numbers of arguments. But there's an easy way to achieve this purpose: simply give the routine a formal argument of a tuple type. With

```
print_formated (values: TUPLE [STRING])
```

a corresponding call may have any number of arguments greater than one as long as the first is a *STRING* (representing a format). Clients may call it as

```
print_formatted (some_format, int, re, str)
```

whatever the types of *int*, *re*, *str*, as long as the routine body handles them properly.

Target validity and Void-Safe Eiffel

The last component of Class-Level validity guarantees that a call *x*•*f* (…) can never fail at run time because *x* turned out to be attached to a void reference:

> **Target rule** *VUTA*
>
> An Object_call is **target-valid** if and only if either:
>
> 1 • It is unqualified.
>
> 2 • Its target is an attached expression.

Unqualified calls (case 1) are always target-valid since they are applied to the current object, which by construction is not void.

Another way of expressing this observation is to note that an unqualified call *g* (…) is always the result of a qualified call *x*•*f* (…) (or of an original root call to *f*, starting a system), where *f*, directly or indirectly, calls *g* unqualified on the same target *x* that was used for *f*; that *x* cannot have been void since the call to *f* would then never have started in the first place. Put yet another way, the unqualified call is generally equivalent to **Current**•*g* (…) where **Current**, representing the current object, is never void.

← *"System execution", page 114*.

For the target expression *x* to be "*attached*", in case 2, means that the program text guarantees — statically, that is to say through rules enforced by compilers — that *x* will never be void at run time. This may be because *x* is an entity declared as attached (so that the validity rules ensure it can never be attached a void value) or because the context of the call precludes voidness, as in if *x* /= *Void* **then** *x*.*f* (…) **end** for a local variable *x*. The precise definition will cover all these cases.

Combining the rules

Class-Level validity is the combination of the previous three constraints, and is the basic validity rule for calls:

Class-Level Call rule *VUCC*

A call of target type *ST* is **class-valid** if and only if it is export-valid, argument-valid and target-valid.

The last requirement, target validity, may raise issues for older Eiffel systems not yet checked for this property. The Standard, for that reason, allows compilers to offer a special tolerance, with the associated risk of run-time failure, as a temporary measure to facilitate transition:

Void-Unsafe

A language processing tool may, as a temporary migration facility, provide an option that waives the target validity requirement in class validity. Systems processed under such an option are **void-unsafe**. Void-unsafe systems are not valid Eiffel systems.

23.11 INTRODUCTION TO CALL SEMANTICS

Let us now examine the semantics of calls. This section and the next few discuss the concepts; the formal rules are collected at the end.

→ *"PRECISE CALL SEMANTICS", 23.17, page 643*.

It will suffice to consider as working example a qualified *Call*

target.*fname* (y_1, …, y_n)

where *target* is an expression, *fname* is a feature name of the appropriate class, and the y_i are expressions. We may further assume that *target* is either a Parenthesized expression or a single Unqualified_call, in other words that the Call is not a multi-dot of the form *a*.*b*.*c* … .*fname* (…).

Concentrating on this example simplifies the discussion but doesn't lose any generality:

- By not considering multi-dot expressions we simply understand a multi-dot call as a succession of single-dot calls, as in the above call to *set_font*. The formal semantic definition will justify this equivalence.

← [3], page 625.

- We already noted that infix, prefix and bracket expressions always have an Equivalent Dot Form.

← "OPERATOR AND BRACKET FORMS", 23.5, page 616.

- If there are no arguments, we simply consider that *n* is zero.
- Lastly, what of unqualified calls *fname* (y_1 …, y_n)? We'll also be able to handle them as a special case of qualified calls thanks to the notion of *current object* as discussed below.

We will also assume, on the basis of the preceding discussion of Void-Safe Eiffel, that at the time of execution *target* will not be void: either it is expanded, directly denoting an object, or it is a reference attached to an object. This is a universal requirement on call targets; if you want a feature to work on a void value for one of its operands *x* — definitely a useful possibility in some cases — you must treat *x* as an argument, not the target. You can only use *x* as target if its static type is an attached. Remember that this is not necessarily the declared type *T* of *x*: if *T* is not attached you can use the Object_test

```
if x /= Void then
        x.f (args)
                -- The static type of this occurrence of x is attached T
else
        … No calls with target x permitted here …
end
```

This discussion leads to our first semantic definition for calls:

Target Object

The **target object** of an execution of an Object_call is:

1 • If the call is qualified: the object attached to its target.

2 • If it is unqualified: the current object.

"Current object" has only been defined informally so far and its precise definition is forthcoming. The definitions, however, avoid circularity.

→ "Current object, current routine", page 641.

The notion of target object is used in all the semantic specifications for calls in the rest of this chapter.

In the qualified case (case 1) you will use, to obtain the target's value, the rules of expression semantics. They yield the target object itself for an expanded type, and for a reference type a reference attached to that object.

The validity rules, as noted, prevent a void reference. For compilers that support an option that doesn't enforce void-safety requirements, we provide an exception type anyway:

Failed target evaluation of a void-unsafe system

In the execution of an (invalid) system compiled in void-unsafe mode through a language processing tool offering such a migration option, an attempt to execute a call triggers, if it evaluates the target to a void reference, an exception of type *VOID_TARGET*.

23.12 DYNAMIC BINDING

So we want to execute or evaluate *target*.*fname* (*args*) at a certain instant of system execution, on a non-void *target*.

"*Execute*" for an instruction, "*evaluate*" for an expression. The rest of the discussion uses the first of these terms for simplicity, except when the context implies an expression.

The D in the type and class names stands for "dynamic", the S for "static".

Assumed to be non-void, *target_value* is attached to a target object OD. OD is a direct instance of some type *DT*, of base class *D*. *D* (the generating class of OD) must be effective: otherwise *DT* could not have any direct instance.

"Generating class": 19.2, page 498.; "Dynamic type", page 598; "Type of an expression", page 774; "POLYMORPHISM", 22.11, page 598

The expression *target_value* has a certain type *ST*, of base class *S*. Recall that *ST* is also called — when we need more precision — the **static** type of *target*, and *DT* its **dynamic** type at the time the call is executed. The static type is obvious from the software text and is fixed for any occurrence of *target* in that text; polymorphism means that the dynamic type may change in successive executions of the call, as a result of reattachments.

← *"Reattachment principle", page 591*.

The typing constraints imply that *DT* will always conform to *ST*, and hence that *D* is a descendant of *S*. The validity rules just seen imply that the feature of the call, *fname*, must be the final name in *S* of a feature of *S*, available to the class which includes the call. Let *sf* be that feature.

Actually, as you guessed, we couldn't care less about *sf*. Using *sf* for the call would be committing the gravest possible crime in object technology: **static binding**. What matters is not the type of *target* (what was declared in the software text) but the type of the object attached to *target_value* (what is actually found at run time). Using that type, *DT*, to determine the appropriate feature, yields the appropriate policy: **dynamic binding**.

The feature to be used, *df*, is the version of *sf* that applies to *D* and hence to *DT*. The two features will be different if *DT* or some intermediate class has redefined *sf*. The purpose of such a redefinition is precisely to ensure that the feature performs for instances of *DT* in a way that differs from its default behavior for instances of *ST*. Not using the redefined version would mean renouncing the power of the inheritance mechanism.

The word "version", as used here, has a precise meaning, defined as part of inheritance. Every feature of a class has a single "dynamic binding version" in any descendant of that class; that version is the result of applying any redefinition, undefinition or effecting that may have occurred since the original introduction of the feature. The definition takes into account the case of repeated inheritance, for which the Select subclause removes any ambiguity that could be caused by conflicting redefinitions on different inheritance paths, or by the replication of an attribute.

← *"Dynamic binding version", page 460*.

The following semantic definition captures dynamic binding:

SEMANTICS

Dynamic feature of a call

Consider an execution of a call of feature *fname* and target object O. Let *ST* be its target type and *DT* the type of O. The **dynamic feature** of the call is the dynamic binding version in *DT* of the feature of name *fname* in *ST*.

Behind the soundness of this definition stands a significant part of the validity machinery of the language:

- The rules on reattachment imply that *DT* conforms to *ST*.
- The Export rule imply that *fname* is the name of a feature of *ST* (meaning a feature of the base class of *ST*).
- As a consequence, this feature has a version in *DT*; it might have several, but the definition of "dynamic binding version" removes any ambiguity.

← *"Reattachment principle", page 591*.

← *"Export rule", page 624*.

Combining the last two semantic definitions enables the rest of the semantic discussion to take for granted, for any execution of a qualified call, that we know both the target object and the feature to execute. In other words, we've taken care of the two key parts of Object_call semantics, although we still have to integrate a few details and special cases.

23.13 THE IMPORTANCE OF BEING DYNAMIC

PURPOSE

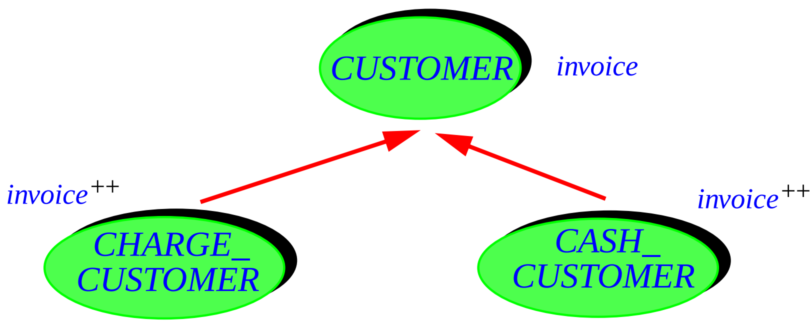
Dynamic binding is not just a useful convention but a condition of correctness. Every qualified call to an exported routine of a class must preserve its invariant, so as never to produce an inconsistent object — one that would not satisfy the invariant of its own generating class. This means that *sf* must preserve the invariant *SI* of *S*, and *df* the invariant *DI* of *D* (a possibly strengthened form of *SI*). But there is of course no requirement that *sf* preserve *DI*; in fact, the designer of *S* usually did not even know about class *D*, which may have be written much later by someone else. Static binding could then apply to an object, OD, a feature, *sf*, which does not preserve the invariant of the generating class — the ultimate disaster in the execution of a software system.

← *As required by the definition of "Class consistency", page 243*.

← *As implied by the definition of "Unfolded form of an assertion", page 281*.

Dynamic binding, then, is the only meaningful policy. In some cases, of course, *sf* and *df* are the same feature because no redefinition has occurred between *S* and *D*, or simply because *S* and *D* are the same class. Then static and dynamic binding trivially have the same semantics. A compiler or other language processing tool which is able to detect such situations through careful analysis of a system's source text use this insight to generate slightly more efficient object code. This is perfectly acceptable as long as the system's run-time behavior implements the semantics of dynamic binding.

Beyond its theoretical necessity, dynamic binding plays an essential role in the Eiffel approach to software structuring. It means that clients of a number of classes providing alternative implementations of a certain facility can let the mechanisms of Eiffel execution select the appropriate implementation automatically, based on the form of each polymorphic entity at the time of execution.



Kinds of customer

As a typical example, assume a class *CUSTOMER* with a procedure *invoice* used to bill customers. Heirs *CHARGE_CUSTOMER* and *CASH_CUSTOMER* may redefine this procedure in two different ways to account for different forms of invoicing. Then a Variable *c* of type *CUSTOMER* may be attached, at some run-time instant, to an instance of *CHARGE_CUSTOMER* or *CASH_CUSTOMER*. A call of the form

```
c.invoice
```

will, thanks to dynamic binding, be treated appropriately in each case.

This is a great advantage for the authors of client classes containing such calls, since they do not need to test explicitly for every possible case (charge customer, cash customer), and may integrate the introduction of a new case — such as check customers — at minimal change in their classes.

23.14 ONCE ROUTINES

We know the target of the call is not void, and we know (through dynamic binding) what feature was really meant. So the next thing to do is to execute the associated routine body, right? Wrong. The routine might be a **once routine**, designed to be executed only once, or once in a while.

Once basics

As you will remember, a Routine_body may start (other than **deferred** and **external** cases) not only with **do** but also with the keyword **once**, possibly followed by one or more "once keys" in parentheses as in **once** ("*THREAD*"). ← *"ROUTINE BODY", 8.5, page 218*.

In the basic case without once keys, this means that you want the routine's body to be executed at most once in the entire system execution. The first time — if at all — someone calls the routine, its body will be executed, with the actual arguments given if any; if it's a function, it will return its result normally. Any subsequent call, however, will not cause any new execution of the routine body or initialization of local variables; it will return immediately to the caller, giving as result — if the routine is a function — the value recomputed by the first call, whether an object (if the result type is expanded) or an object reference.

A constraint on once functions was introduced as part of the Feature Declaration rule (condition 5): if the enclosing class is generic, the result type may not be one of the formal generic parameters. This is necessary for the function to provide a consistent result: since the first client that calls the function will determine the result of all later calls, the result type must be meaningful for all clients; but different clients may use different actual generic parameters for the class. The formal parameter, which stands for any possible actual generic parameter, would represent incompatible types. ← *Page 160*.

Once uses

The Once mechanism is a versatile tool allowing flexible initialization and access to shared information in an O-O environment. In particular:

- **Smart initialization**: to make sure that a library works on a properly initialized setup, write the initialization procedure as a Once and include a call to it at the beginning of every externally callable routine of the library.

The alternative would be to require clients to take care of the setup themselves by calling an initialization procedure;. Because this is error-prone, you'll want to check in the library itself that the initialization has been done; but then you might just as well take care of it silently and avoid bothering clients. In any case, you need a way to find out if initialization has indeed been done, typically through a flag — which must also have been initialized, only pushing the problem further. Once procedures provide a general solution.

- **Shared objects**: To let various components of a system share an object, represent it as a once function that creates the object. Clients will just "call" that function, although in all cases but the first such a call just returns a reference to the object created the first time around.

In this last case, the scheme is a common one in Eiffel programming:

```
shared_object: SOME_REFERENCE_TYPE
            -- A single object useful to several clients
      once
            … ; create Result
      end
```

This declaration may for example appear in a service class inherited by the affected clients.

Predefined once keys

What exactly does "once" mean? By default, the semantics is to execute the routine body once over every execution of a system. By using once keys, however, you may exert finer control, specifying an execution every once in a specific while. For example by declaring a routine as

```
r: SOME_TYPE
            -- A single object useful to several clients
      once ("OBJECT")
            …
      end
```

you specify that the body will be executed the first time it is called on **any specific instance** of the class. This provides welcome flexibility. Assume for example that some objects have associated information, much bigger than the object itself and needed only in certain cases. This could be (among many other examples) the list of all previous states of an object stored in a database. It's not something that you want to load by default into memory with every object that you retrieve from the database; but it should be easy to access when you need it. The following function does the job smoothly and (for the programmer) effortlessly:

```
history: ARRAY [like Current)
            -- A single object useful to several clients
      once ("OBJECT")
            create Result (…)
            … Retrieve previous values and fill Result with them …
      end
```

Traditional programming techniques — using flags to check whether the function has been called — would be quite cumbersome here, especially if you have a need for several such functions.

The following once keys have a preset meaning:

```
"OBJECT"            -- Once for each instance
"THREAD"            -- Once per execution of a thread
"PROCESS"           -- Once per execution of a process
```

"*PROCESS*" is the default, equivalent to not specifying a once key.

Further once tuning

For even more flexibility, you may define your own meaning of "while" in "once in a while". You'll do this by choosing as once key an arbitrary string, beyond the three possibilities listed above. You can take advantage of this possibility in two ways.

First, you can control the meaning from outside of the Eiffel text, by defining it in the **once** clause of the Ace file. The recommended convention in this case is to use a once key of the form *$KEYNAME*, using the dollar sign that serves in some scripting languages to denote the value of a variable. The Ace specification can set the key to mean, for example, *THREAD* in some executions and *PROCESS* in others, depending for example on the amount of multi-threading supported.

→ *"ONCE CONTROL", B.11, page 1023*.

In the Eiffel text itself, you can go further by deciding when once is enough and when you want more of it. More precisely you may **refresh** a once key; this means that the next call of any once routine that lists it as one of its once keys will execute its body. To refresh keys, class *ANY* has a feature *onces* of type *ONCE_MANAGER* (a Kernel Library class) which you can use for such calls as

→*ONCE_MANAGER*, *A.6.29, page 1000*.

```
onces.refresh ("SOME_KEY")
onces.refresh_some (["SOME_KEY", "OTHER_KEY"])
onces.refresh_all
onces.refresh_all_except (["SOME_KEY", "OTHER_KEY"])
```

You can also query *onces*.*nonfresh_keys*, returning an array of strings, to find out what keys have been exercised by at most one function.

A possible way to implement feature *onces* in class *ANY* is to make it a once function itself.

These are clearly advanced techniques, but they can help considerably in the building of sophisticated systems.

Once routine semantics

In defining the semantics of once routines we will rely on the following notion whose meaning follows directly from the preceding discussion:

Freshness of a once routine call

During execution, a call whose feature is a once routine *r* is **fresh** if and only if every feature call started so far satisfies any of the following conditions:

1 • It did not use *r* as dynamic feature.

2 • It was in a different thread, and *r* has the once key "*THREAD*".

3 • Its target was not the current object, and *r* has the once key "*OBJECT*".

4 • After it was started, a call was executed to one of the refreshing features of *onces* from *ANY*, including among the keys to be refreshed at least one of the once keys of *r*.

Note that *every* call started so far has to satisfy *any* of the conditions listed. So *r* is fresh for example if:

- It hasn't been called at all.
- It has been called on different objects, and is declared **once** ("*OBJECT*").
- It's declared **once** ("*SOME_KEY*") and there has been, since the last applicable execution of *r*, a call *onces*.*refresh* ("*SOME_KEY*").

An applicable call — for example, with the once key "*OBJECT*", a call on the same object — makes *r* unfresh again, since the rule's conditions have to apply to every call started so far.

The call *onces*.*refresh_all* is understood to refresh all once routines, including those without an explicit once key.

Also note that the condition applies to calls *started* so far; so if a once routine is directly or indirectly recursive, its self-calls will not execute the body (in the absence of an intervening explicit refresh) and, for a function, they will return the **Result** as computed so far.

Latest applicable target of a non-fresh call

The **latest applicable target** of a non-fresh call to a once function *df* to a target object O is last value to which it was attached in the call to *df* most recently started on:

1 • If *df* has the once key "*OBJECT*": O.

2 • Otherwise, if *df* has the once key "*OBJECT*": any target in the current thread.

3 • Otherwise: any target in any thread.

From these observations we may define the semantics of a call to a once routine. For fresh calls a once routine behaves like a non-once routine, and the rule correspondingly refers to the Non-Once Call Routine Execution rule appearing later in this chapter:

→ *“Non-Once Routine Execution rule”, page 644*.

Once Routine Execution Semantics

The effect of executing a once routine *df* on a target object O is:

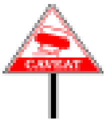
1 • If the call is fresh: that of a non-once call made of the same elements, as determined by the Non-once Routine Execution rule.

2 • If the call is not fresh and the last execution of *f* on the latest applicable target triggered an exception: to trigger again an identical exception. The remaining cases do not then apply.

3 • If the call is not fresh and *df* is a procedure: no further effect.

4 • If the call is not fresh and *df* is a function: to attach the local variable **Result** for *df* to the reused target of the call.

Case 2 is known as "*once an once exception, always a once exception*". If a call to a once routine yields an exception, then all subsequent calls for the same applicable target, which would normally yield no further effect (for a procedure, case 3) or return the same value (for a function, case 4) should follow the same general idea and, by re-triggering the exception, repeatedly tell the client — if the client is repeatedly asking — that the requested effect or value is impossible to provide.



There is a little subtlety in the definition of "latest applicable target" as used in case 4. For a once function that has already been evaluated (is not fresh), the specification does not state that subsequent calls return the result of the first, but that they yield the value of the predefined entity **Result**. Usually this is the same, since the first call returned its value through **Result**. But if the function is **recursive**, a new call may start before the first one has terminated, so the "result of the first call" would not be a meaningful notion. The specification states that in this case the recursive call will return whatever value the first call has obtained so far for **Result** (starting with the default initialization). A recursive once function is a bit bizarre, and of little apparent use, but no validity constraint disallows it, and the semantics must cover all valid cases.

Had you detected this case?

The Once Routine Execution rule describes the effect of executing a Call once we know its run-time feature *df*, its target object *O* and its arguments *arg_values*. For the full context, we need the general semantics rule for calls, which comes at the end of this chapter and, in the once case, relies on the above rule to specify the effect of the call once its components have been determined.

23.15 ATTRIBUTES AND EXTERNALS

We may now concentrate on the case of a qualified Object_call whose feature is not a once routine. From the discussion of features and routines, the dynamic feature of the call, if not a "once", may be one of:

← *Chapters 5 and 8.*

S1 •An attribute

S2 •An external routine (whose implementation is outside the system's direct reach, being written in another language).

S3 •A non-once, non-external routine.

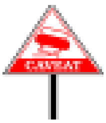
The syntax for Routine_body includes a fifth case: a routine with a **deferred** body. This case doesn't apply here, however, since as noted above *D* has a direct instance and hence must be effective.

← *Page 218.*

In case S1, *df* is an attribute; the object OD has a field corresponding to *df*. Then the call is an expression, whose value is that field. The sole effect of the call is to return that value.

In case S2, *df* is an external routine; execution of the call will mean passing the values of the actual arguments to that external routine, waiting for it to complete its execution, and obtaining its result if it is a function. The semantics of argument passing and of routine execution — which may depend on the conventions of the routine's native language — are examined in the chapter on interfaces with other languages.

→ *Chapter 31*.



Note that the target object is **not** passed by default to an external routine. If it's needed for the computation, you should pass it as actual argument to the routine, which should include a corresponding formal.

These two cases will be integrated in the final call semantics rule. For the moment we may concentrate on the remaining one.

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

23.16 THE MACHINERY OF EXECUTING CALLS

We'll investigate the effect of a non-once, non-external routine (S3) of actual arguments *args*, target object O and dynamic feature *df*. This will also lead us to the semantic notions of current object and current routine.

Scheme for a routine call

The semantic rule will specify the effect of the call as the result of applying a sequence of steps. This doesn't mean that the code must execute these exact steps, only that its effect must be the same as if it did. Somewhat informally and ignoring assertion monitoring, the steps are:

1 • Using the semantics of direct reattachment, attach every formal argument of *df* to the value of the corresponding actual from *args*.

← *"SEMANTICS OF REATTACHMENT", 22.7, page 585*.

2 • If *df* has any local variables, save their current values if any call to *df* has been started but not yet terminated; then initialize each local variable to the default value of its type.

3 • If *df* is a function, initialize the predefined entity *Result* to the default value for the function's return type.

4 • Execute the Compound of *df*'s Internal body, according to the conventions described next.

5 • If *df* is a function, the call is an expression. The value returned for that expression is the value of *Result* after the previous step.

6 • If the values of local variables have been saved under 2, restore the variables to these earlier values.

The Argument rule ensure that in step 1 the actual arguments (if any) match the formals in number, and that each actual is compatible with (conforms or converts to) the corresponding formal.

← *Page 626*.

In step 2, the default initialization values are the same as for the initialization of attributes in a Creation instruction.

The saving of local variables under 2, and their restoring under 6, are necessary because routines may be directly or indirectly recursive: the body of *df* may contain a call to another routine, and that routine may turn out to be *df*, or it may recursively call *df*. As a result, step 4 may start the whole process again on the same routine. The saving and restoring ensure that each incarnation of *df* recovers its local variables when it is resumed after a recursive call.

The local variables include **Result**: *"LOCAL VARIABLES AND RESULT", 8.6, page 221.*

Current object and routine

To interpret the Compound of a routine's Internal body in step 4, a little mystery remains. Assume the text of routine *df*, in class *D*, has the following simple form:

```
fname
    do
        some_proc
        x.other_proc
    end
```

The feature name might be something other than fname as a result of renaming.

where *x* is an attribute of *D*, *some_proc* a procedure of *D*, and *other_proc* is a procedure applicable to *x*. Step 4 — the core of the call's execution — consists of executing the two instructions of the Compound.

But what exactly do they mean? What does *x* represent? To what object should the computation apply *some_proc*?

To answer these questions we must put ourselves in the global context of system execution and remember how anything ever gets executed. Quoting from a very early part of this book:

> To execute (or ''run'') a system on a machine means to cause the machine to apply a creation instruction to the system's root class.

← "System execution", page 114.

In all but trivial cases, the root's creation procedure will create more objects and execute more calls. This extremely simple semantic definition of system execution has as its immediate consequence to yield a precise definition of the *current object* and *current routine*. At any time during execution, the current object is the object to which the latest non-completed routine call applies, and the current routine *cr* is the feature of that call:

Clause 4 addresses "*constructs whose semantics does not involve a call*" (rather than "constructs other than a call"). This is because the semantics of a construct that is not a calls may involve a call; this is the case with an Expression, whose semantics is defined through an Equivalent Dot Form denoting a call.

Current object, current routine

At any time during the execution of a system there is a **current object** CO and a **current routine** *cr* defined as follows:

1 • At the start of the execution: CO is the root object and *cr* is the root procedure.

2 • If *cr* executes a qualified call: the call's target object becomes the new current object, and its dynamic feature becomes the new current routine. When the qualified call terminates, the earlier current object and routine resume their roles.

3 • If *cr* executes an unqualified call: the current object remains the same, and the dynamic feature of the call becomes the current routine for the duration of the call as in case 2.

4 • If *cr* starts executing any construct whose semantics does not involve a call: the current object and current routine remain the same.

Note the implicit recursion in case 2: to know the target object of a call *target*.*fname* (*args*), we must evaluate *target*, which may itself be a call, whose evaluation requires using the above rule recursively.

There appears to be a cycle in the definitions since this definition of current object and current routine refers to "dynamic feature", defined in terms of "target object", itself defined in terms of "current object". You will note on closer examination, however, that this is not a real problem: the definition of target object only refers to the current object in the case of an Unqualified_call, for which the relevant clause in the definition of current object retains an object already known from the context.

← *Page 631*.
← *Page 629*.
← *Clause 2, page 629*.
← *Clause 3, page 641*.

Naming the current object

Even though the current object is at the heart of the execution machinery, most calls in dot notation do not refer explicitly to the current object: if you need a Call with the current object as target, you may just write it as an Unqualified_call, which does not name its target.

For some other kinds of operation, however, you may need an explicit notation to refer to the current object. An example is equality comparison. Assume a function computing the distance between two points, which might be written in a class *POINT* as

```
distance alias "|–|" (other: POINT): REAL
            -- Distance of current point to other.
      do
            …
      end
```

The routine's implementation may need to determine whether the *other* point is in fact the same point as the current object:

```
if "other is not the same as the current point" then
      Result := "… Normal distance computation …"
end
            -- Otherwise Result will be zero
```

To express the condition after **if** you may use the predefined entity **Current**:

```
if Current /= other then …
```

As noted above, an Unqualified_call such as *some_proc* or *x* does not need to use **Current** explicitly as its target, although you may if you want to:

```
Current.some_proc
Current.x
```

with the only difference that, under assertion monitoring, qualified calls such as these cause evaluation of the invariant; unqualified calls don't.

It may also be convenient to use **Current** in connection with binary features. Thanks to the infix alias "|–|", you may use the above *distance* function to express the distance of two points *p1* and *p2* as *p1* |–| *p2*. To express in a similar form the distance to *p2* of the current point, you may write

```
Current |–| p2
```

but even this use of **Current** is not strictly necessary, since there's always an identifier name, here *distance*, for such a feature, so that you may also use the plain Unqualified_call

```
distance (p2)
```

Similarly, if a class contains a unary function *negated* **alias** "–", you may express the negation of the current object as – **Current** as well as just *negated*.

Current, as indicated by its place in the syntax as one of the choices for the construct Read_only, is a **read-only** entity: you can't assign to it, or use it at the target of a creation instruction. A notation such as

← "EXPRESSIONS AND ENTITIES", 19.8, page 504.

Current.*q* := *v*	**[4]**

is permitted only if *q* is a query of the enclosing class and it has an associated **assigner procedure**, say *p*. Then [1] is simply a shorthand for an unqualified call

← "ASSIGNER CALL", 22.12, page 599.

p (*v*)	**[5]**

If *q* has arguments, **Current**.*q* (*a1*, *a2*) := *v* is an abbreviation for *p* (*a1*, *a2*, *v*). In either case, the instruction can't change **Current**.

The following rule gives the precise meaning of **Current**, distinguishing in particular between reference and expanded cases:

SEMANTICS

Current Semantics

The value of the predefined entity **Current** at any time during execution is the current object if the current routine belongs to an expanded class, and a reference to the current object otherwise.

23.17 PRECISE CALL SEMANTICS

We can now collect into precise rules the understanding of call semantics developed over the preceding sections. The rule for a Non_object_call appeared at the beginning of this chapter, so we only need to consider the case of an Object_call. For once routines we may refer to the earlier rule.

← "Non-Object Call Semantics", page 623.; "Once Routine Execution Semantics", page 637

Rule for non-once routines

Assume we have an Object_call and, at a particular stage of execution, we know the target object, the dynamic feature — which is not a "once" — and the argument values. Here then is the effect:

General call semantics

We have semantics for executing routines, both once (the earlier rule) and non-once (the last rule). To have the full semantics of calls we need a more general rule, since:

← "Once Routine Execution Semantics", page 637.

- Both of the previous rules assumed that we know the target object, the dynamic feature, and argument values. But the form of a qualified call, *target*.*fname* (*args*), doesn't give us that information; the execution must obtain the target object from *target*, the dynamic feature from that object and *fname*, and the argument values from *args*. We've actually

Non-Once Routine Execution rule

The effect of executing a non-once routine *df* on a target object O is the effect of the following sequence of steps:

1 • If *df* has any local variables, including **Result** if *df* is a function, save their current values if any call to *df* has been started but not yet terminated.

2 • Execute the body of *df*.

3 • If the values of any local variables have been saved in step 1, restore the variables to their earlier values.

given ourselves the rules to do this; but to make the semantics precise we need to specify the **order** in which to apply these rules. We'll require that the target be evaluated first, giving us the dynamic feature as a consequence, and then the arguments in the order listed.

← *“Target Object”, page 629*; *“Dynamic feature of a call”, page 631*.

- The rules covered non-external routines only; we must include the attributes and external routines, two cases discussed informally so far.

← *“ATTRIBUTES AND EXTERNALS”, 23.15, page 638*

- Execution of the feature body (step 2 of the last rule) may use the formal arguments. We need to specify how to attach them to the actuals' values.
- Finally, the scheme does not yet include assertion monitoring.

The following rule fills these gaps:

SEMANTICS

General Call Semantics

The effect of an Object_call of feature *sf* is, in the absence of any exception, the effect of the following sequence of steps:

1 • Determine the target object O through the applicable definition.

2 • Attach **Current** to O.

← *Page 629*.

3 • Determine the dynamic feature *df* of the call through the applicable definition.

4 • For every actual argument *a*, if any, in the order listed: obtain the value *v* of *a*; then if the type of *a* converts to the type of the corresponding formal in *sf*, replace *v* by the result of the applicable conversion. Let *arg_values* be the resulting sequence of all such *v*.

5 • Attach every formal argument of *df* to the corresponding element of *arg_values* by applying the Reattachment Semantics rule.

← *"Reattachment Semantics", page 592*.

6 • If the call is qualified and class invariant monitoring is on, evaluate the class invariant of O's base type on O.

7 • If precondition monitoring is on, evaluate the precondition of *df*.

8 • If *df* is not an attribute, not a once routine and not external, apply the Non-Once Routine Execution rule to O and *df*.

9 • If *df* is a once routine, apply the Once Routine Execution rule to O and *df*.

10 •If *df* is an external routine, execute that routine on the actual arguments given, if any, according to the rules of the language in which it is written.

11 •If *df* is a self-initializing attribute and has not yet been initialized, initialize it through the Default Initialization rule.

12 •If the call is qualified and class invariant monitoring is on, evaluate the class invariant of O's base type on O.

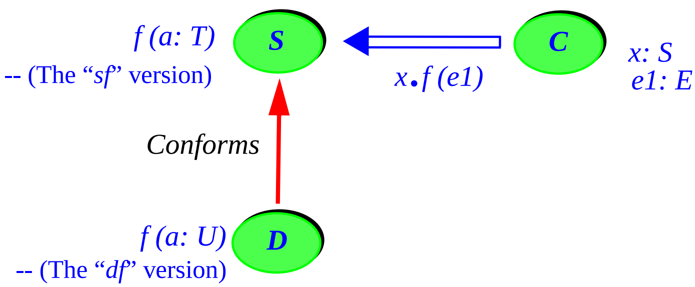
13 •If postcondition monitoring is on, evaluate the postcondition of *df*.

An exception occurring during any of these steps causes the execution to skip the remaining parts of this process and instead handle the exception according to the Exception Semantics rule.

For steps 1 and 3, the "applicable definitions" are those of Target Object and Dynamic Feature, as recalled above.

There is considerable implicit recursion in this definition: the target and the argument are expressions, and in many cases they will be calls, or operator expressions whose semantics is also defined as call semantics. So in steps 1, 3 and 4 we are potentially relying on the semantic rules of this chapter, including the above rule itself. The rule for once routines relies, for fresh calls, on the rule for non-once routines, so step 9 again causes recursion.

Step 4 specifies a somewhat subtle but important property: the precedence, statically, of convertibility over conformance. We know that every actual argument must be *compatible with* the corresponding formal: conform or convert to it. System validity will ensure that this requirement applies both to the "static" version of the feature *df* and to the "dynamic" version *sf*. Remember that *sf* is the feature named known from the text of the call: with *x*.*f* (*e1*), if *x* is of type *S*, *sf* is the feature of name *f* in *S*; as a result of dynamic binding, if *x* at execution time is attached to an object of a descendant type *D*, then *df* is the version in *D*.



Effect of redefinition on a client call

But while we want the type *E of e1* to be compatible with the formal arguments to both the *sf* and *df*, we want it, for every one of them, **in the same variant**: either conformance in both cases, or convertibility in both cases. Assume *E* conforms to *T*; then it cannot also convert to it. Now assume that *E* does not conform to *U*, the new formal argument type in *D*, but by some twist of fate *E* actually convert to *U*. Do we want to accept the call as descendant-argument-valid for *D*? System validity tells us "no". Accepting this would be confusing for the author of *C*, who does not realize that a conversion might be going on (since there's none in the case of the original *f*).

← *"Conversion principle", page 400*.

In addition, although this is not the main concern, the compiler writer would face the similar problem of not knowing whether to generate conversion code or not for the call.

So step 4 requires that we take care of any conversion on the basis of the argument types for the **static** feature *sf*; only then, in step 5, do we attach the values of actuals to formals. Note that the types in these attachments may still be different, but no further conversion will be involved, only conformance.

23.18 CALLS AS EXPRESSIONS

The two uses of a Call are, as we know, as an Instruction or as an Expression, specifically the Basic expression variant. If *f* is a query (attribute or routine), a valid call

← "Call Use rule", page 615. Instruction: page 224; Expression: page 753.

x.f (args)

or any of the other applicable variants — unqualified, non-object, multi-dot — is an expression, and can be included in a larger expression, such as *a + x.f (args) + b*.

For the instruction case we've seen all we need about calls. But to understand an expression we must also know its **type** and its **value**; these are defined for every kind of expression and we must now — as the final part of specifying calls — say what they are for a call used as expression.

First, the type. To make this concept useful in practice we must carry type analysis across class boundaries by defining the type of a call **with respect to** a certain type. Assume that *x*, in a class *C*, is of type *D* [*U*], where *D* [*G*] is a generic class with a query *f* of type *G*. The Call Expression Type definition given below will tell us that the type of *x.f* is the type of *f* *with respect to* the type of *x*, that is to say with respect to *D* [*U*]. Now *f*, a query of *D*, is also a query of *D* [*U*] thanks to the definition of "feature of a type" in the discussion of genericity. Its type as defined in *D* is *G*, which in the context of *D* [*U*] we must understand, through the Generic Type Adaptation rule, as representing the associated actual generic parameter, *U*.

The following rule determines the type of a call:

DEFINITIONS

Type of a Call used as expression

Consider a call denoting an expression. Its **type** with respect to a type *CT* of base class *C* is:

1 • For an unqualified call, its feature *f* being a query of *CT*: the result type of the version of *f* in *C*, adapted through the generic substitution of *CT*.

2 • For a qualified call *a.e* of Target *a*: (recursively) the type of *e* with respect to the type of *a*.

3 • For a Non_object_call: (recursively) the type of its imported form.

In case 2, the recursion applies to *a*; the type of the part after the dot, *e*, is determined through the general Expression Type definition — itself of course dependent, in several of its clauses, on the type of call expressions, causing more recursion.

→ "Type of an expression", page 774 (see among others its clauses 6 and 11).

Finally the semantics. If a call is used as an expression its execution will, in addition to any other actions, return a result:

Call Result

Consider a Call *c* whose feature is a query. An execution of *c* according to the General Call Semantics yields a **call result** defined as follows, where *O* is the target object determined at step 1 of the rule and *df* the dynamic feature determined at step 3:

1 • If *df* is a non-external, non-once function: the value attached to the local variable **Result** of *df* at the end of step 2 of the Non-Once Execution rule.

2 • If *df* is a once function: the value attached to **Result** as a result of the application of the Once Execution rule.

3 • If *df* is an attribute: the corresponding field in *O*.

4 • If *df* is an external function: the result returned by the function according to the external language's rule.

For a Non_object_call, whose semantics is defined in terms of the imported form, this definition also applies, as a consequence, to the execution of the imported form.

← *"Non-Object Call Semantics", page 623*

Functions should not produce any durable change to their environment; their sole role should be to return their result, and any computation they perform should be auxiliary to that goal. You may use **only** postcondition clauses to turn this methodological advice into an enforceable rule.

This book often refers, especially in the discussion of expressions, to the **value** of a call used as an expression. Here is what this precisely means:

→ *Chapter 28.*

Value of a call expression

The **value** of a Call *c* used as an expression is, at any run-time moment, the result of executing *c*.