This draft: Version 4.82-00-00, 21 September 02 12:15 (Zürich time)

Type safety and catcalls *(not done)*

24.1 OVERVIEW

In discussing calls, the previous chapter covered syntax and semantics, but set aside any consideration of validity – even though its semantic definitions only apply to valid constructs. It is time now to come back to the second horse of our troika and examine what it takes to make a call meaningful.

Calling features, it was already noted, is the principal means of performing computations in Eiffel. This is why the title of this chapter does not just read "validity of calls", but "type checking", since the type safety of a system is essentially defined by the validity of its calls. This is also why, in an approach that places so much emphasis on helping developers produce correct and robust software, it is crucial to ask what could go wrong at run time with a call – and see what we can do *before* run time to prevent it from going wrong.

Consider the basic form of a call in dot notation:

target.*fname* (*y1*, ..., *yn*)

For this to be properly executed, *target* must be attached to an object DO, and DO must be equipped with a feature corresponding to *fname*; that feature must have a signature (types of arguments and result, if any) and a specification (precondition and postcondition) compatible with what the caller expects.

Not all of these requirements may be handled statically by mere analysis of the software text. To ascertain statically that DO will always exist (that is to say, that *target* will never have a void value) and that the assertions will always be satisfied, we would need theorem and program provers beyond the reach of current software technology. For these properties, the presentation has reluctantly settled for run-time checks which, if not satisfied, may trigger exceptions.

For the remaining properties, however, the picture is brighter. Assuming the object DO exists, it is a direct instance of a certain class *D*, and if we have enough information about the possible *D* we will know statically what features they have. Determining the possible classes and checking that their features match the corresponding calls will enable us to perform static type checking. This chapter explains how to achieve this goal.

As usual, we will take an informal look first, then examine the precise rules.

24.2 SYNTAX VARIANTS

As noted in the previous chapter, dot notation is only one possible form of call. Operator expressions provide the other variant, meant for compatibility with traditional mathematical and programming language notation. The difference is just syntactical, however; an Operator_expression of the form

```
a + b
```

is semantically a call having *a* as its target, **infix** "+" as its feature, and *b* as its single argument.

The notion of equivalent dot form will serve to formalize the correspondence between dot notation and operator expressions; see 26.7, page 598. The ability to treat mixed arithmetic expressions according to mathematical tradition will require a specific rule, balancing; see 26.10, page 602.

For simplicity, this chapter will (like the discussion of call semantics in the previous one) rely on dot notation calls of the basic form shown in the previous section. Its results immediately carry over to operator expressions.

24.3 CLASS-LEVEL VALIDITY

At first sight, the conditions which will make a call valid appear straightforward. After all, *target*, like every expression, has a type; because all entities must be explicitly declared, the type of any expression is immediately obvious from the class text. Like any other type, it is based on a class; the properties of that class should tell us whether a certain feature is applicable to the expression.

Consider a typical context for the above call:

```
class C feature
    ...
    target: S;
    y: SOME_jTYPE
    routine is
    do
    ...
    target.fname (y);
    ...
    end; -- routine
    ...
end -- class C
```

Here the type of *target* is a non-generic class *S*, and there is a single argument *y*. The elementary type rule seems clear: *S* must have a feature of final name *fname*; this feature must be available to *C* (in other words, *S* must export it either generally or to a set of classes that includes an ancestor of *C*); and it must have the requested signature, which means that it must be a procedure with a single formal argument, to which *y* conforms.

"Available" was defined on page ====.

For example if class *C* uses the following call as expression:

```
next_paragraph.line (3)
```

then *C* must have a feature (attribute or function without argument) of final name *next_paragraph*, available to class *C*, and if *next_paragraph* is of type *PARAGRAPH*, class *PARAGRAPH* must have a function of final name *line*, with one formal argument of type *INTEGER* (the type of *3*, the actual argument), or perhaps of another type to which *INTEGER* conforms.

These conditions at first sight appear not only necessary but also sufficient to guarantee that the call will always make sense at run time.

The rule as sketched is still partial: we will need to extend it to account for zero-dot (Unqualified_call) and multi-dot calls, for targets whose type is anchored or generically derived, and for features with no arguments or more than one argument. But these extensions do not raise any particular difficulty.

Properly formalized, this is indeed a fundamental type rule, which will be defined below as **class-level validity**. Calls which satisfy this condition will be said to be **class-valid**.

The precise definition is part of the Single-dot Call rule on page 537.

24.4 SYSTEM-LEVEL VALIDITY

Although class-level validity may at first appear sufficient, the typing problem is in fact less trivial than the above would suggest. The reason is polymorphism and dynamic binding, which forces us to take into account not just the declared types of entities, but also their possible dynamic types (their dynamic type set).

Polymorphism means that the type used to declare the *target* (*S* above) is not the only possible type for the object DO to which the call will apply. To see this, let us extend the context introduced above:

```
class C feature
    target: S; other D; -- D must be a
descendant of S
    y: SOME_TYPE; ...
    routine is
    do
    if some_test then target := other
end;...
    target.fname (y);
    end; -- routine
    ...
end -- class C
```

where the Assignment rule requires *D* to be a descendant of *S*. Because of the possible polymorphism resulting from the assignment of *other* to *target*, the type of the object DO may now be not just *S* but *T* as well.

In other words, we need to consider not only the type of *target* as it results from the declarations, called the **static type** if there is any ambiguity, but also the set of all the types that *target* may assume at run-time as a result of polymorphic attachments. This set was defined in the discussion of polymorphism as the **dynamic type set** of *target*; a member of that set is said to be a possible **dynamic type** for *target*.

The Assignment rule is on page ====. Since the classes are non-generic, conformance is the same relation as "descendant of".

The base classes of the possible dynamic types constitute the **dynamic class set** of *target*.

In this example all types are classes, so that the dynamic type set and dynamic class set are the same, but with generic derivation, expansion and anchored types we will need to reintroduce the distinction between types and classes.

Dynamic type sets and dynamic class sets were defined in 22.11, page 480, and are covered more extensively below: first in 24.8, page 533, and then in 24.9, page 536, for the precise rules.

What then is the actual type constraint? It still applies to a class *D* the conditions defined above for class-level validity:

- *D* must have a feature corresponding to *fname*, available to *C*.
- That feature must have the required signature.

But there is an important difference: whereas for class-level validity, the only *D* of interest was *S*, the static type of *target*, here, as a result of polymorphism, we must enforce these two conditions for any *D* in the dynamic class set of *target*. This requirement is called **system-level validity**.

System-valid, valid

A call of target *target* is **system-valid** if and only if it is class-valid for *target* assumed to be of any type of its dynamic type set.

A call is **valid** (without further qualifier) if it is both class-valid and system-valid.

In a simple world we would expect any class-valid call to be system-valid. Unfortunately this is not always the case because of two important properties of the inheritance mechanism:

On P1, see "adapting the export status of inherited features", page====. On P2, see the informal discussion on "typing and redeclaration", page ====, and the Redeclaration rule, page====, clause 2.

P1 •A class may override the export policies of its parents; it may for example make secret its version of a feature which the parent exported.

P2 •A routine redefinition may replace the type of a formal argument by a type conforming to the original. This is know as the **covariant** argument typing policy.

Although they may seem surprising at first, properties P1 and P2 are important aspects of the typing policy. The rationale is discussed in detail below. First, we must understand why they may have unpleasant consequences if we limit ourselves to class-level validity checking.

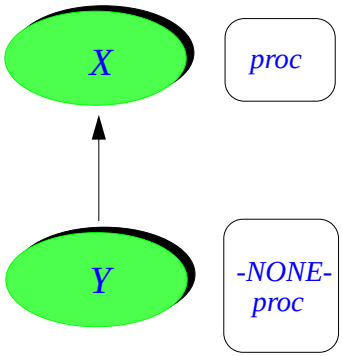
24.6, page 527 below, explains why these properties are essential for realistic uses of object-oriented concepts.

24.5 VIOLATING SYSTEM VALIDITY

(If this is your first reading, you may be content with the realization that type checking is less trivial than it appears at first, and that a systemwide type checker will detect the non-trivial errors. You may then want to skip the rest of this chapter.)

It is indeed not hard to put together an example where P1 prevents a class-valid system from working properly. Similar examples relying on P2 would be almost as immediate.

Hiding an inherited feature



Consider a class *X* which exports a procedure *proc* without arguments, and its heir *Y* which makes *proc* secret, as shown by the above figure. To hide *proc*, class *Y* will use the New_exports clause:

The New_exports *clause of a* Feature_adaptation *part enables a class to override the export policies of its parents. See 7.11, page ====.*

```
class Y inherit
    X
    export {NONE} proc end;
    ... Rest of class omitted ...
```

Then consider a class *C* which calls *proc* on a polymorphic entity which at run time may become attached to an object of type *Y*. This will be the case if *C* contains the following declarations and instructions, in some order:

```
a: X
b: Y            -- Y is an heir of X
!! b            -- Instruction β
a := b          -- Instruction δ
a.proc
```

WARNING: this is not a contiguous extract, just some lines which may appear anywhere in a class text in any order satisfying the constraints on declarations and instructions.

The call *a.proc* is class-valid since *X* exports *proc*. But it is not system-valid: the instruction labeled β may attach to *b* an object of type *Y* (the declared type of *b*); the instruction labeled δ may attach *a* to the same object as *b*; then the last instruction may call *proc* on that object, even though it is an instance of *Y*, and *proc* is secret in *Y*.

This example – or any similar one using P1 or P2 to violate system-level validity in the presence of polymorphism – immediately brings four important comments.

First, the example is not affected by the order of the offending instructions (β, δ and the call). As long as they appear in the same system and may all potentially be executed, the call is system-invalid. For obvious reasons of simplicity, system-level validity does **not** involve any flow analysis; even in the extreme case in which the polymorphic assignment δ would be replaced by

```
if False then a := b end
```

we would still consider the call to be invalid.

The second comment is that system-level invalidity in such an example is a serious problem, not just a matter of style. If the author of *Y* did not export *proc*, we must presume that this was for a good reason. Remember in particular that an exported routine must preserve the class invariant. So *proc* preserved the invariant of *X*, but perhaps the invariant of *Y* is stronger and *proc* does not preserve it any more. In this case, applying *proc* to an object of type *Y* may produce an inconsistent object – one which does not satisfy the fundamental consistency constraints expressed by the invariant. This is a potential disaster.

The invariant preservation requirement is part of class consistency, page ====.

Third, neither the call *a.proc* nor the polymorphic assignment *a* := *b* is wrong by itself. The call applies an exported procedure of class *X* to an entity *a* of type *X*; the assignment satisfies the type conformance rule. What is wrong is the possibility for these two individually legitimate constructs to be executed as part of an execution of the same system. To be more precise, even that combination would be harmless were it not for the presence of a third accomplice, the Creation instruction labeled β, which raises the possibility for *b*, and hence for *a* as well, to become attached to an object of type *Y*.

This brings the last comment, addressing a question that may well have been troubling you for some time now: isn't the type policy *wrong*? Why do we allow a class to hide some of its parent's exported features, or to replace an argument type by a more specific (conforming) one? Shouldn't we have a stricter policy, guaranteeing that class-level validity implies system-level validity?

As it turns out, however, the type policy, although perhaps surprising at first, is essential to support the practice of object-oriented software development. Let us take a closer look at the underlying issues.

24.6 NOTES ON THE TYPE POLICY

You may indeed have wondered what all the fuss was about. Shouldn't class-level validity imply system-level validity? Then type checking would be trivial, involving only local properties of classes.

The culprits were identified above: the two properties P1 and P2, which free heirs from some of the export and typing decisions made by their parents. We should really ask ourselves whether these properties are appropriate.

Here they are again:

- A class may override the export policies of its parents; it may for example make secret its version of a feature which the parent exported.
- A routine redefinition may replace the type of a formal argument by a type conforming to the original (covariant argument policy).

A third related property is that a creation procedure of a class may not enjoy the status of creation procedure any more in a proper descendant. See the Creation rule, page ====.

Then if *S* has an exported routine *sf* of name *fname* with a formal argument of type *SOME_TYPE* the call used earlier as example will be class-valid. Here it is again, with some of the enclosing class text omitted:

```
target: S; other: B; -- D must be a descendant of S
y: SOME_TYPE; ...
routine is
    do
    if some_test then target := other
end; ...
    target.fname (y); ... Rest of routine omitted ...
```

But that call is not necessarily system-valid. *D* may redefine *target* to be of some type *D*; or it may make its version of *sf* secret; or it may redefine this routine to take an argument of type *B*, a proper descendant of *SOME_TYPE*. Any of these cases makes the above call system-invalid since the dynamic class set of *target*, as a result of the polymorphic assignment *target* := *other*, includes *D*.

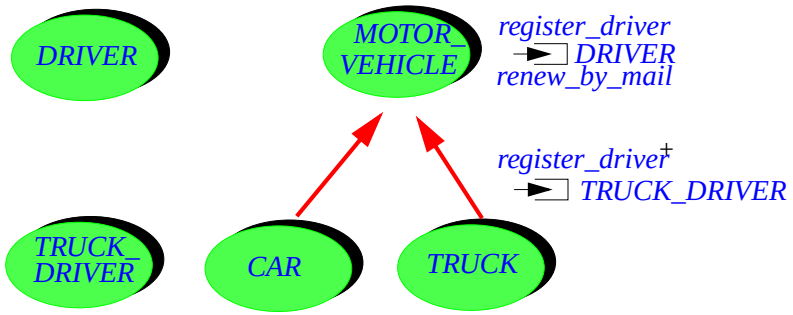
System-level checking will detect the problem and flag the system as invalid.

The above two properties (P1 and P2) often seem surprising at first. Why make type checking more difficult, and introduce the distinction between class-level and system-level validity by allowing classes to choose export and argument typing policies different from those of its parents?

The answer is that this flexibility is indispensable to the practice of object-oriented design. Without it, designers would constantly have to reshuffle inheritance hierarchies, and would have much difficulty observing the constraints of a typed object-oriented language. P1 and P2 serve to acknowledge the inescapable difficulty of reconciling the goals of orderly classification (as implemented through inheritance) and safety (as implemented through typing) with the irregularities and instability of the real-world situations which our software systems attempt to model through their inheritance hierarchies.

Although a full discussion of this question falls beyond the scope of this book, a simple example will serve to illustrate the need for properties P1 and P2.

Vehicles and drivers



Assume the two inheritance hierarchies represented above, with *MOTOR_VEHICLE* having heirs *CAR* and *TRUCK*, and *DRIVER* having *TRUCK_DRIVER* as heir. These classes could be part of the system used by a company to manage its fleet of vehicles, or by a Department of Motor Vehicles to keep track of driver registration.

To begin, this raises an obvious case of P2 (covariant argument type redefinition). Class *MOTOR_VEHICLE* has a procedure

```
register_driver (d: DRIVER) is...
```

which naturally takes an argument of type *DRIVER*. For trucks, however, the driver must be approved for truck driving; accordingly, class *TRUCK* redefines *register_driver* to take an argument of type *TRUCK_DRIVER*.

The type constraints in such a case permit the above inheritance structure and the redefinition of *register_driver* – a case of possibility P2. They even permit such polymorphic assignments as the one in

```
a: MOTOR_VIGICLE; t: TRUCK;
...
a := t
```

or Creation instructions such as ! *TRUCK* ! *a* ..., with the same declarations. What system-level validity will reject is the only case that could lead to an erroneous call at run time: the presence in the same system of a polymorphic reattachment such as the above and a Call such as

```
a.register_driver (dr1)
```

where *dr1* is of type *DRIVER* (not *TRUCK_DRIVER*). Clearly, the presence of this Call in a system that may also attach an instance of *TRUCK* to *a* is erroneous, and will be flagged as invalid. This, however, does not affect the need for P2-like covariant argument redefinition; in fact, the system-level validity rule is what makes P2 possible.

LINKED_LIST and its use of LINKABLE and 'put_linkable_left' are sketched in 5.5, page 74.

Examples of this kind, with two parallel inheritance hierarchies, are a constant occurrence in the development of systems and their class hierarchies. Many appear in the Data Structure Library. For example, to describe doubly linked lists, *TWO_WAY_LIST* inherits from *LINKED_LIST*; to describe two-way chained linked cells, *BI_LINKABLE* inherits from *LINKABLE*. The list classes have procedures manipulating list cells, such as *put_linkable_left*, which quite naturally take arguments of type *LINKABLE* in *LINKED_LIST* and *BI_LINKABLE* in *TWO_WAY_LIST*.

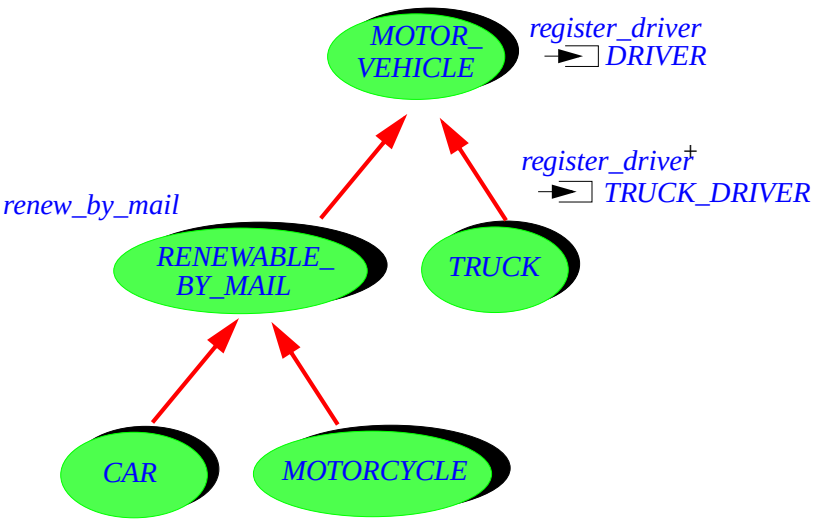
In this case, however, there is no explicit redefinition such as that of *register_driver* in *TRUCK*. The reason is the presence of the Anchored form of type declaration. Class *LINKED_LIST* contains the declarations

```
first_element: LINKABLE [like first]
put_linkable_left (new: like first_element) is..
```

so that *TWO_WAY_LIST* only needs to redefine *first_element* (to be of a type based on *BI_LINKABLE*); the argument *new* of *put_linkable_left*, and all the other entities declared **like** *first_element* in *LINKED_LIST*, follow automatically. As this example shows, the whole idea of anchored declarations is based on the principle of covariant argument redefinition.

The example of motor vehicles, trucks and drivers may also provide an example of the need for policy P1 (independence of heirs' and parents' export policies. Assume the permits for motor vehicles can normally be renewed by mail, hence the presence in *MOTOR_VEHICLE* of an exported procedure *renew_by_mail*. For trucks, however, this does not apply (the truck must be inspected for safety, every year at the time of re-registration). So *TRUCK* does not export *renew_by_mail* (which might violate an invariant of this class, although it preserves the invariant of *MOTOR_VEHICLE*).

In a case like this, one may always argue that the inheritance hierarchy was improperly designed, and should have separated renewable-by-mail vehicles from others, with *renew_by_mail* introduced not in class *MOTOR_VEHICLE* but one level down:



Not all registrations may be renewed by mail

But forcing this as the only acceptable choice would make the practice of object-oriented software development almost impossible.

In any practical problem, there will be many possible criteria for classification; what will happen if, after you have taken apart the original hierarchy because of the registration-by-mail problem, you must take into account other, independent criteria? For example, some vehicles will be for personal use and others for professional use; some will have two wheels and others will have more; some will pay a road tax and some will not; some will require smog inspections every three years; and so on. Since the original designers could not, without perfect foresight, have come up with the ideal inheritance hierarchy, the developers will find themselves constantly redoing the structure. The conflicting criteria may in fact make it impossible to obtain any acceptable inheritance structure at all.

The flexibility of policy P1 makes it possible to handle this problem by allowing a class to be selective about its inheritance – exporting or hiding inherited features to its own clients according to its own local properties. As before, this is an example of transferring part of the burden from developers (in the form of constant architecture redesign) to the supporting implementation (in the form of the more sophisticated form of type checking required by system-level validity).

It should be clear from this discussion that a well-designed inheritance hierarchy will include few occurrences of classes hiding some of their parent's features. If you find yourself constantly at odds with the parent designers' decisions, then you should probably consider improving the inheritance structure (assuming you are permitted to do so). This is why the default policy for inherited features is to retain the parents' export status; to override it, you must include an explicit New_exports clause. But the ability to do this in the minority of cases which call for it is a key component in the effort to make object-oriented software construction not just a pleasant theoretical idea but a practical way to produce real systems.

24.7 WHY DISTINGUISH?

If we accept that system-level validity is the appropriate notion of validity, it is fair to ask why one should bother at all with class-level validity. Why not have a single validity condition, as for the other constructs studied in this book?

The reason is pragmatic, and involves two complementary observations on possible violations of the validity constraints:

- First, it is easier (for a language processing tool as well as a human reader) to detect violations of class-level validity, since they only involve a local analysis of the features one class – *S*, the base class of *target*'s type. In contrast, checking system-level validity may involve systemwide analysis. The names "class-level" and "system-level" reflect this difference.
- Second, a study of errors as they occur in actual system development reveals that system-level validity violations which are not also class-level violations occur very rarely.

For these reasons, implementors may choose to design class-level and system-level checking as separate facilities. Class-level checking will detect a vast majority of errors; the remaining ones will be found by applying system-level checking.

Of course, to guarantee fully the type safety of a system, you must check both kinds.

Class-level checking is straightforward. The only non-trivial part of system-level validity is to determine the dynamic class set. Let us see concretely how this can be done.

24.8 A LOOK AT THE DYNAMIC CLASS SET

The dynamic class set of an entity is the set of base classes of all types that the entity may take on at run-time, as a result of polymorphic reattachements and creation instructions.

The call validity rule, appearing in the precise discussion at the end of this chapter, will give a full definition. It is important, however, to get first an intuitive view of what the base class represents. (Although "intuitive" this view is not incorrect; it simply misses some details and does not cover all cases.)

The idea will be to determine in a single process the dynamic class sets of *all* entities. The process is iterative; if you have a background in numerical mathematics, it will remind you of algorithms which compute the solution to a vector or matrix equation by successive iteration (for example over a grid); if you are familiar with the theory of programming languages, it will remind you of fixpoint methods for approximating the high-level functions and domains of denotational semantics.

Fixpoint methods for denotational semantics are covered in "Introduction to the Theory of Programming Languages". See bibliography.

An example will serve to illustrate the process. Consider a class extract containing the following four instructions, in some order:

```
! X ! a        -- α
! Y ! b        -- β
c := a         -- γ
a := b         -- δ
a.proc
```

This is a variant of the example on page ====.

Each instruction has been identified by a Greek letter. The context is missing, in particular the declarations; the Creation instructions have an explicit creation type for clarity, although α, for example, could appear as just !! *a* if *a* is of type *X*.

As before, the order of these instructions is irrelevant. If they appear in the same context, then *a*, as discussed above, may become attached to an object of type *Y*; this means that, for system validity, any routine *rout* appearing in a call *a.rout* using *a* must meet the appropriate conditions not just for *X* but also for *Y*.

System-level validity analysis will need to determine the base class sets of *a*, *b* and *c*. The result, obtained through a process explained below, will be the following:

The complete form of this result, given page ====, will include more information, in particular references to iteration steps.

a | *X* [α] ; *Y* [δ]

b | *Y* [β]

c | *X* [γ] ; *Y* [γ]

This shows a vector of dynamic class sets, one for each entity. Each class set contains a list of types. Any type which appears in one of these lists is there because of one of the instructions; to make this justification clear, the instruction's identifying greek letter appears in brackets next to the type.

In the rest of this section "class set" is an abbreviation for "dynamic class set".

For *b*, the class set includes just *Y*, resulting from the Creation instruction β. For a, it includes X, resulting from the Creation instruction α, and *Y*, resulting from the polymorphic assignment δ which adds all of *b*'s class set to the class set of *a*. For *c*, assignment γ means that the class set is the same as that of *a*.

How do we determine this result? An iterative process will provide the solution. Starting from the types given by Creation instructions, we may repeatedly extend the current sets by adding to the class set of every entity *e* the class set of any other entity *f* such that there is a reattachement of *f* to *e* somewhere. We stop when we have reached a "fixpoint", that is to say when our vector of class sets is stable.

Here is this process applied to the above example. To obtain the initial vector *v0* of class sets, just look at the creation instructions:

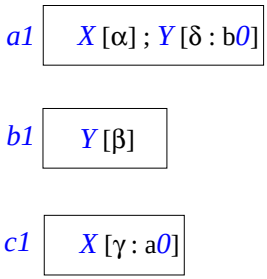
a0 | *X* [α]

b0 | *Y* [β]

c0 |

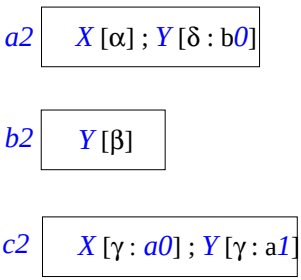
For each type appearing in a class set, a comment in brackets identifies the instruction which causes the class to be there: the Creation instruction α puts *X* in the class set of *a*, and β puts *Y* in the class set of *b*. Entity *c* is not the target of any Creation, so its class set is empty for the moment.

On each iteration, we will look at every reattachment and extend the target's class set with all the classes obtained so far in the source's class set. For the first iteration, this gives the new class set vector *v1*:



The class set of *a* now contains *Y*, again identified, for clarity, by its origin: the comment [δ : ~ *b0*] means that *Y* comes from *b0*, the earlier class set for *b*, and has be added to *a1*, the new class set of *a*, because of the polymorphic assignment a := b (δ). In the same way, *X* now appears in the class set of *c* because of its presence in *a0* and of the assignment *c* := *a* (γ). You may be tempted to add *Y*, which appears in *a1*, but this would be cheating: to update a vector at any step, we may only use vector elements from the previous step.

The next step, producing vector *v2*, will indeed use *a1* and γ to add *Y* to the class set of *c*:



If you apply the mechanism once more, you will find that it does not bring anything new: *v3* is the same as *v2*. We have put all the available type information to good use; *v2* gives the complete class sets for all entities involved. (In technical terms *v2* is a fixpoint.)

The process as illustrated on this example is not hard to generalize to the full language. The extension must integrate expressions which are function calls rather than simple entities; it must also account for the two other forms of reattachment beside Assignment: actual-formal association, which raises no particular problem, and Assignment_attempt. For this last case, the effect of

```
b: Y;
...
b ?= a
```

to extend *b*'s class set not with all elements of *a*'s class set (as with normal Assignment), but only with those which are descendants of *Y*.

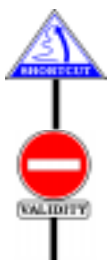
This discussion has outlined a way of obtaining the dynamic class sets of the entities in a system. Two words of warning will serve as its conclusion:

The precise specification of dynamic class sets appears as part of the call validity rule below. The iterative process that we have just discussed is only one concrete interpretation of that specification, although of course it satisfies that specification.

Although you may view the description of that process as an abstract algorithm for language processing tools that perform type checking, its purpose is explanatory only. Implementors of compilers and other type checking tools may well rely on totally different methods.

By now you should have a good understanding of the practical implications of type checking. All that remains is to give the rules in their full and precise form.

24.9 THE CALL VALIDITY RULE



(This last section formalizes the previous discussion of validity, but does not introduce any new concepts, so that you may safely skip to the next chapter. In fact this section will be mostly of interest to implementors of language processing tools.)

Since this general validity rule is very abstract, the successive definitions of this section, introducing the different aspects of call validity, have received separate validity codes for ease of reference.

General Call rule *CUGC*

A call is valid if and only if it is both class-valid and system-valid.

This is very general and means that we must now define class-level and system-level validity. Like the above validity constraint, the definition relies on further notions to be defined next: export validity, argument validity and the dynamic class set.

Single-level Call rule *CUSC*

Consider a single-dot call with target *x*, appearing in a class *C*. Let *S* be the base type of the type of *x*. Then:

1 • The call is **class-valid** if it is export-valid and argument-valid for *S*.

2 • The call is **system-valid** if for any element *D* of the dynamic class set of *x* it is export-valid and argument-valid for *D*.

Export validity will require suppliers to make the needed features available to *C*; argument validity will require every actual argument to conform to the corresponding formal argument.

The validity of a call at either level – class or system – will require both export and argument validity. The only difference is that for class validity you need only apply these criteria to *S*, the type declared for the target *x* of the call, whereas for system-level checking you will need to consider all possible dynamic classes of *x*.

First, export validity:

Export rule *CUEV*

A call appearing in a class *C*, having *fname* as the feature of the call, is **export-valid** for a class *D* if and only if it satisfies either of the following two conditions.

1 • The call is an Unqualified_call and *fname* is the final name of a feature of *C*.

2 • The call has at least one dot, *D* has a feature of name *fname* which is available to *C*, and the call's target is either a valid entity of *C* or (recursively) a call which is export-valid for *D*.

A call's target may be a Parenthesized expression, which is equivalent to a call. See 23.7, page ====.

Condition 1 takes care of unqualified calls. Condition 2 is the basic requirement: *D* must make the feature available to *C* – that is to say, export it generally, or selectively to *C* or one of its ancestors. Condition 2 also takes care of the multi-dot case: in *a.b.c*, the target, here *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.

For the export validity of multi-dot calls, be sure to note that 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)
```

where the 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*. If some of the exports are selective, it does not matter whether *second_word* is available to *PARAGRAPH*, or *set_font* is available to *LINE*. To understand why, it suffices to realize that the above call may be rephrased with just one level of dot notation as

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

This shows multi-dot notation as essentially a notational facility – although of course an important one in practice, avoiding the need for intermediate entities such as *l* and *w*.

Here now is argument validity. The definition will be simpler if we assume export validity as a prerequisite:

Argument rule *CUAR*

Consider an export-valid call of target *target* and feature name *fname* appearing in a class *C*. (For an Unqualified_call take *target* to be *Current*.) Let *ST* be the type of *target*, *S* the base class of *ST*, and *sf* the feature of final name *fname* in *S*. Let *D* be a descendant of *S*, and *df* the version of *sf* in *D*. The call is **argument-valid** for *D* if and only if it satisfies the following four conditions:

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

2 • Actual arguments, if any, each conform to the corresponding formal arguments of *df*.

3 • If *target* is itself a Call, it is (recursively) argument-valid for *D*.

4 • If any of the actual arguments is of the Address form *$ f* where *f* is a Feature_name *f* is the final name of a feature of *C*.

Condition 4 applies to external calls and will be examined in the discussion of the foreign language interface in chapter 28; see page ====. The Address form of argument, '$fn', is part of the syntax for Call, page ====. It serves to pass addresses of Eiffel features to foreign (non-Eiffel) routines.

Export validity ensures that *df* 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.

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

← *Page 268.*

Condition 3 handles, as before, the case of multi-dot calls.

In condition 2, remember from the discussion of expression conformance an actual argument *y* will conform to the corresponding formal argument *x* if the type of y conforms to the type of *x*, but also if *x* is of type **like** *z* and the type of *y* conforms to the type of *z*; also, if *x* is anchored to another final argument, as in

See 14.12, page 299, about expression conformance and the exact type rules which explain why a routine call as given is argument-valid.

```
r (z: T; x: like z) is ...
```

then *y* of type *YT* conforms to *x* in a call *r* (*u* , *y*) if *u* is of type *UT* (conforming to *T*) and *YT* conforms to *UT*.

The argument validity rule assumes that *D* is a descendant of *S*. This is always the case whenever you need this rule to ascertain argument validity as part of the requirements on call validity: for class-level checking, *D* will be *S*; for system-level checking, the validity constraints on reattachment indicate that all the dynamic types of an entity conform to its static type (and so are based on descendant classes). This is also clear from the definition of the dynamic type set below.

This gives the full validity rule on calls.

To remove any ambiguity, we must provide an equally precise definition of the **dynamic class set** of an expression. This is the set of base classes of all elements in the **dynamic type set** of the expression; the dynamic type set was itself defined as the set of all possible dynamic types of the expression, where a possible dynamic type for an expression is the type of any object to which it may become attached at run time. This definition is correct, but it does not enable us to determine easily the dynamic type set, or the dynamic class set, from the software text.

The original definitions are in 22.11; see page ====.

The above informal illustration constructed dynamic class sets through successive vector approximations, until it reached a fixpoint. Since it assumed all classes to be non-generic, its results were both dynamic classes and dynamic types. The full definition, which covers anchored and generically derived types, will yield the the dynamic *type* sets; to obtain the corresponding class sets, just replace every type by its base class.

The iterative process was described in 24.8, page 533.

The definition needs the following two notions to deal with genericity. Let *T* be a Class_type based on a class *C*. If *C* is a generic class C [*G1*, ...], *T* is *C* .[*A1*, ...] for some types *A1*, ...; if *C* is not generic, *T* is just *C*. Then:

- If *e* is an entity or expression appearing in a feature of *C*, the "dynamic type set of *e* for *T*" is the set of dynamic types of objects that may become attached to *e* as a result of calls to *C*'s features on direct instances of *T*. The "dynamic type set of *e*", with no further qualification, is the dynamic type set of *e* for *C* .[*G1*, ...], or just *C* if *C* is not generic.
- If *U* is a type, the notation *UT* will stand for the type obtained from *U* by substituting *Ai* for any occurrence of *Gi* – or just *U* if *C* is not generic. For example, if *U* is *X* .[*G* ,, *INTEGER*] appearing in a feature of class *D* .[*G*], and *T* is *D* .[*REAL*], then *UT* is *X* .[*REAL* ,, *INTEGER*].

Here is the full definition of dynamic type sets:

Dynamic type set

1 • The dynamic type sets of the expressions, entities and functions of a system result from performing all possible applications of the following rules to every Class_type *T*, of base class *C*, used in the system.

2 • If a routine of *C* contains a creation instruction, with target *x* and creation type *U*, the dynamic type set of *x* for *T* is $\{U_T\}$.

3 • The dynamic type set for *T* of an occurrence of *Current* in the text of a routine of *C* is $\{T\}$.

4 • For any entity or expression *e* of expanded type appearing in the text of *C*, if the type *ET* of *e* is expanded, the dynamic type set of *e* for *T* is $\{ET_T\}$. (Rules 4 to 7, when used to determine elements of the dynamic type set of some *e*, assume that *e*'s type is not expanded.)

5 • If a routine of *C* contains an Assignment of target *x* and source *e*, the dynamic type set of *x* for *T* includes (recursively) every member of the dynamic type set of *e* for *T*.

6 • If a routine of *C* contains an Assignment_attempt of target *x*, with type *U*, and source *e*, the dynamic type set of *x* for *T* includes (recursively) every type conforming to U_T which is also a member of the dynamic type set of *e* for *T*.

7 • If a routine of *C* contains a call *h* of target *ta*, *U* is (recursively) a member of the dynamic type set of *ta* for *T*, and *tf* is the version of the call's feature in the base class of *U*, then the dynamic type set for *U* of any formal argument of *tf* includes every member of the dynamic type set for U_T of the corresponding actual argument in *h*.

8 • If *h*, *tf* and *U* are as in case 6 and *tf* is an attribute or function, the dynamic type set of *h* for *T* includes (recursively) every member of the dynamic type set for U_T of the *Result* entity in *tf*.

Each of the seven cases of this definition, explained in detail below, is a rule which you may use to bring new elements to the possible dynamic type sets of the expressions, entities and functions of a system. More precisely:

- Rules 1, 2 and 3 are non-recursive: they yield elements of the dynamic type sets without further ado.

- Rules 3 to 7 are recursive: given known elements of the dynamic type sets of the expressions of a system, they may add new elements.

In other words, you may view the definition as describing an iterative process, generalized from the earlier discussion, for building the dynamic type sets: first apply rules 1 to 3 to every possible case, obtaining *v0*, the initial vector of dynamic type sets; then, at each successive step *i*, apply rules 3 to 7 to every possible case, obtaining new elements of the type sets in *vi* from elements of the type sets in *vi - 1*. The process terminates if the resulting *vi* is the same as *vi - 1* – that is to say, the last iteration has brought nothing new.

This process is finite since the set of types in the system is finite. To get an upper bound to the number of iterations, call DEPTH be the maximum depth of a call (number of dots in Call form, number of operators in Operator_expression form) and ATTACH the maximum length of a non-cyclic sequence *ei* such that there is a reattachment from *ei + 1* to *ei*; then the process will terminate in at most DEPTH + ATTACH steps.

Let us now make sure we understand the seven rules. Rule 1 addresses creation instructions. It considers that an instruction of the form

```
! U ! x
```

adds the creation type, here *U*, to the dynamic type set of *x*. (If *U* is absent, the creation type is *x*'s base type.) If *C* is generic, the rule pertains to the dynamic type sets relative to some generic derivation *T* of *C*; then we must perform the corresponding substitution of actual for formal generics, so the rule uses *UT* rather than just *U*.

Creation instructions were discussed in 20.10, with the definition of "creation type" appearing on page ====.

Rule 2 indicates that *Current*, when used in *C*, represents an object of type *T* – that is to say *C*, with the requested generic derivation if applicable.

Rules 1 and 2 reflect the pragmatism of system-level validity checking. Class-level checking considers the developer's intentions (the declarations); but system-level checking only considers deeds: the types of the objects that Creation and reattachement instructions may actually attach to entities.

Rule 3 takes care of expressions of expanded types, which are never polymorphic. In this case we just take the declarations at face value.

As you may remember, the conformance rule for expanded types allows for some tolerance in the case of basic arithmetic types. For example you may attach the integer value *3* to an entity *r* of type *REAL*. But this has no effect on the dynamic type set of *r*: such an assignment causes a conversion, and attaches to *r* a real value, here *3.0*. So there is no polymorphism in this case.

See 14.9, page ====, about conformance for basic types. Assignment semantics in this case ([4] on the table page 317) is copy, implying conversion to the "heavier" type (case 2 of copy semantics, page ====).

Rules 4, 5 and 6 covers the three forms of possibly polymorphic reattachment: Assignment, Assignment_attempt, and actual-formal association in a call. For a reattachment of a value *e* to an entity *x*, we must add all of *e*'s possible dynamic types to those of *x*. In addition:

- For an Assignment_attempt of the form x ?= e (rule 5), we must only consider those possible types for *e* which conform to the type of *x*: any other one would result, in accordance with the semantics of Assignment_attempt, in no object attachment for *ta*.

- For a call (rule 6), *e* is an actual argument and *x* is the corresponding formal argument in the appropriate version of the routine.

In rule 6, a member *U* of the dynamic type set of *ta*, the call's target, may be generically derived; then when need to perform the corresponding type substitutions in adding the members of the actual argument's dynamic type set to the dynamic type set of the formal argument. This is why the rule considers the dynamic type set of *e* (the actual argument) for U*T*.

A call whose feature is a function is itself an expression, with its own dynamic type set. Since the expression's value is the final value of *Result* as used in the function, rule 7 defines the dynamic type set of the expression as that of *Result*. As with rule 6, we must perform the appropriate substitution if the type of the call's target is generic, hence the use of *UT*.

We need one more convention to make the above rule fully applicable in practice: how to handle the dynamic type sets of array elements. The relevant features in the Kernel Library class *ARRAY* are *put* and *force*, which set an element's value, and *item*, which returns an element's value, as in

Arrays require a specific convention since the Kernel Library specification covers the interface of class ARRAY (A.6.16, page 817) but not its implementation.

'put' assumes that the index, 'i' in the example, is within bounds; 'force' resizes the array if necessary. Details in chapter 28.

```
x, some_entity: T; i, j: INTEGER; a: ARRAY [T];
...
        -- Assign the value of some_entity to the i-th element of a:
a.put (some_entity, i)
...
        -- Assign to x the value of the j-th element of a:
x := a.item (j)
```

To find out the dynamic type set of *a.item* (*j*) (and hence of *x*), note that the software text usually does not suffice to determine whether *i* and *j* will have the same value at execution time. So we must treat every *put* or *force* operation as affecting potentially every array element. Hence the rule:

Array type rule

To study the effect of array manipulations on dynamic type sets, assume that in class *ARRAY* feature *item* is an attribute, and that *put* (*v*, *i*) and *force* (*v*, *i*) are both implemented as

item := *v*

The rule also applies to manifest arrays. A manifest array is an expression of the form <<a*1* ,, ... ,, a*n*>>, denoting an array of *n* elements, containing the values given. For typing purposes, it will be treated as it had been initialized explicitly by *n* calls to *put*, each of the form

See 27.9, page 631, about manifest arrays.

```
a.put (ai, i)
```

24.10 CREATION VALIDITY (SYSTEM-LEVEL)

(This section explores a specialized type-checking issue and may be skipped at first reading. Even if you want all the details, you will probably have to come back after you have read the chapter on type checking, which is necessary for a full understanding of this discussion.)

Chapter 24 explains the type checking policy, with special emphasis on calls.

Although class-level validity generally suffices to determine the validity of a Creation, the complete definition will require system-level validity as well.

For our immediate purposes it suffices to note that some invalid cases may escape class-level validity checks. The reason is polymorphism. As a result of assignments of the form

Polymorphism is studied in 22.11, page 480.

```
x := y
```

a Writable entity *x* of type *T* may become attached to objects of *y*'s type (which the Assignment rule requires to be a descendant of *T*). These types, for all possible *y*, make up the set of all **possible dynamic types** of *x*, also called its **dynamic type set**. The dynamic type set may contain types other than *x*'s declared type, *T*.

The rigorous definition of the dynamic type set is on page 541.

But a Creation_instruction involving *x*

```
create x.make (...)
```

may be invalid even if it is class-valid. This will be the case, for example, if in the above assignment *y* is of a type *U* based on a class *D* (a proper descendant of *T*'s base class), and *D* fails to list its version of *make* as a creation procedure.

System-level validity avoids any such problem:

Creation System-Validity rule *CGCS*

A Creation_instruction is **system-valid** if and only if it satisfies one of the following two conditions:

C1 •The creation type is explicit (in other words, the instruction begins with **create** {*ET*}... for some type *T*).

C2 •The creation type is implicit (in other words, the instruction begins with **create**...) and every possible dynamic type *T* for *x*, with base class *C*, satisfies conditions 1 to 6 of the Creation Instruction rule (page \n(9g). In applying conditions 5 and 6, the feature of the call, *f*, must be replaced by its version in *C*.

CGCS

In other words, system-level validity is the same property as its class-level counterpart, but applied to all possible dynamic types of the target *x*. In interpreting conditions on the creation procedure *f*, we must take into account the dynamic binding version of that procedure in a descendant class, which may be different from the original because of redeclaration, and may have a different name because of renaming.

← *"Dynamic binding version", page 345*.

As condition 1 of the definition indicates, the problem of system-level validity only arises for Creation instructions with an implicit type. If the type is explicit, as in **create** {*T*} *x* ..., the possible dynamic types of *x* do not affect the validity of the instruction, which in this case is entirely covered by class-level validity.

System-level validity, as all other validity properties, is a **static** requirement, which a human reader or language processing tool may ascertain simply by looking at the software text. Checking it does not require any control flow analysis: whenever a given context contains both an assignment *x* := *y* and a Creation with target *x* which would be invalid for *y*'s type, the Creation will be system-invalid – even if clever control flow analysis would in fact show that no control flow path will ever execute the assignment and the Creation in sequence. Static validity checking doesn't need to be clever; it needs to be safe. This discussion will be generalized to calls in the discussion of type checking.

To be valid, a Creation must satisfy the requirements at both levels:

Creation Instruction rule *CGCI*

A Creation_instruction is valid if and only if it is both class-valid and system-valid.