Attached Types and Their Application to Three Open Problems of Object-Oriented Programming

Bertrand Meyer

ETH Zurich and Eiffel Software
http://se.inf.ethz.ch – http://www.eiffel.com

Abstract. The three problems of the title — the first two widely discussed in the literature, the third less well known but just as important for further development of object technology — are:

- Eradicating the risk of **void calls**: *x.f* with, at run time, the target *x* not denoting any object, leading to an exception and usually a crash.
- Eradicating the risk of "**catcalls**": erroneous run-time situations, almost inevitably leading to crashes, resulting from the use of covariant argument typing.
- Providing a simple way, in concurrent object-oriented programming, to **lock** an object handled by a remote processor or thread of control, or to access it *without* locking it, as needed by the context and in a safe way.

A language mechanism provides a combined solution to all three issues.

This mechanism also allows new solutions to two known problems: how to check that a certain object has a certain type, and then use it accordingly ("Run-Time Type Identification" or "downcasting"), for which it may provide a small improvement over previously proposed techniques; and how to provide a "once per object" facility, permitting just-in-time evaluation of certain object properties.

The solution relies on a small extension to the type system involving a single symbol, the question mark. The idea is to declare certain types as "attached" (not permitting void values), enforce some new validity rules that rule out void calls, and validate a number of common programming schemes as "Certified Attachment Patterns" guaranteed to rule out void calls. (In addition, the design replaced an existing type-querying construct by a simpler one.)

The mechanism is completely static: all checks can be performed by compilers as part of normal type system enforcement. It places no undue burden on these compilers — in particular, does not require dataflow analysis — and can be fairly quickly explained to programmers. Existing code, if reasonably well-written, will usually continue to work without change; for exceptions to this rule, often reflecting real risks of run-time crashes, backward-compatible options and a clear transition path are available.

The result is part of the draft ECMA (future ISO) standard for Eiffel.

There is one and only one kind of acceptable language extension: the one that dawns on you with the sudden self-evidence of morning mist. It must provide a complete solution to a real problem, but usually that is not enough: almost all good extensions solve several potential prob-

lems at once, through a simple addition. It must be simple, elegant, explainable to any competent user of the language in a minute or two. (If it takes three, forget it.) It must fit perfectly within the spirit and letter of the rest of the language. It must not have any dark sides or raise any unanswerable questions. And because software engineering is engineering, and unimplemented ideas are worth little more than the whiteboard marker with serves to sketch them, you must see the implementation technique. The implementors' group in the corner of the room is grumbling, of course — how good would a nongrumbling implementor be? — but you and they see that they can do it.

When this happens, then there is only one thing to do: go home and forget about it all until the next morning. For in most cases it will be a false alarm. If it still looks good after a whole night, then the current month may not altogether have been lost.

From "Notes on Language Design and Evolution" in **[6]**

1 Overview

The design of a programming language is largely, if the designer cares at all about reliability of the resulting programs, the design of a type system. Object-oriented programming as a whole rests on a certain view of typing, the theory of abstract data types; this makes it natural, when searching for solutions to remaining open problems, to turn for help to typing mechanisms.

One such problem is **void-safety**: how to guarantee that in the fundamental object-oriented operation, a feature call *x*. *f* (*args*), the target *x* will always, at execution time, denote an object. If it does not — because *x* is "void" — an exception will occur, often leading to a crash.

This article shows that by fine-tuning the type system we may remove this last significant source of run-time errors in object-oriented programs. The basic language extension is just one symbol (the question mark).

As language extensions should, the mechanism yields other benefits beyond its initial purpose. It provides a solution to the "catcall" issue arising from covariant argument redefinition; a better technique of run-time object type identification; a flexible approach to object locking in concurrent programming; and a simple way to perform lazy computation of attributes ("once per object").

1.1 Mechanism Summary

Here is a capsule description of the mechanism:

- A call *x*. *f* (*args*) is only valid (as enforced statically) if its target *x* is **attached**.
- The simplest way for a variable to be attached is to be declared of an "attached type", guaranteeing that no value can be void. Types are indeed attached by default. To get the "detachable" version of a type *T* (permitting void values) use **?** *T*.

- For variables of a detachable type, some simple and common program schemes guarantee a non-void value. For example, immediately after **create** *x*... or the test **if** *x* /= *Void* **then** ... , *x* is not void. The mechanism uses a small catalog of such "*Certified Attachment Patterns*" or CAPs, easy for programmers to understand and for compilers to implement. A variable used in such a CAP is attached (again, statically), even if its type is not. CAPs are particularly important in ensuring that reasonably-written existing software will run unmodified. Initial evaluation suggests that this will be the case with the vast majority of current code.
- Outside of these patterns, a call of target *x* requires *x* to be of an attached type. The remaining problem is to guarantee that such a variable will never be void. The basic rule concerns assignment and argument passing: if the target is attached, the source must be attached too. This leaves only the question of initialization: how to ensure that any attached variable is, on first access, not void.
- Some types guarantee non-void initialization by providing a default initialization procedure that will produce an object. We call them "self-initializing types".
- A variable that is not of such a type may provide its own specific initialization mechanism. We call it a "self-initializing variable".
- Generic classes can use a question mark to specify a self-initializing type parameter.
- This leaves only the case of a variable that could be accessed while void (because no CAP applies, the type is not self-initializing, and neither is the variable itself). The "Object Test" construct makes it possible to find out if the variable is attached to an object of a specific type, and then to use it safely.

1.2 The Void Safety Issue

The basic idea of typed object-oriented languages is to ensure, thanks to validity rules on program texts enforced statically (at compile-time), that the typical object-oriented operation, *x*. *f* (*args*), known as a "qualified call", will never find *x* attached to an object not equipped to execute the operation *f*. The validity rules essentially require the programmer to declare every variable, routine argument, routine result and other entity with an explicit type (based on a class, which must include the appropriate *f* with the appropriate arguments), and to restrict polymorphic assignments *x* := *y*, as well as actual-to-formal argument associations, to those in which the type of *y* **conforms** to the type of *x*; conformance is governed by inheritance between classes, so that if *f* is available for the type of *y* it will also be available, with a compatible signature, for the type of *x*.

This technique, pioneered by Eiffel and Trellis-Owl and since implemented in various ways in typed O-O languages, eliminates many potential run-time errors, and has succeeded in establishing static typing firmly. But — notice the double negation in the above phrasing of the "basic idea" — it only works if the target entity, *x*, is **attached** to an object at the time of execution. Rather than directly denoting an object, *x* is often a *reference* to a potential object; to support the description of flexible data structures, programming languages generally permit a reference to be *void*, or "null", that is to say attached to no object. If *x* is void at the time of the call, an exception will result, often leading to a crash.

The initial goal for the work reported here was once and for all to remove this sword of Damocles hanging over the execution of even a fully type-checked program.

1.3 General Description

The basis of the solution is to extend the type system by defining every type as "*attached*" or "*detachable*", where an attached type guarantees that the corresponding values are never void. Attached is the default. A qualified call, *x*.*f* (*args*), is now valid **only** if the type of *x* is attached. Another new validity rule now allows us to assign (or perform argument passing) from the attached version of a type to the detachable version, but not the other way around without a check of non-voidness. Such a check, applied to an expression *exp* of a detachable type, is a new kind of boolean expression: an "Object Test" of the form {*x*: *T*} *exp*, where *T* is the desired attached type and *x* is a fresh variable. In the Conditional instruction

```
if {x: T} exp then                                                        /1/
        ... Instructions, in particular calls of the form x.f (args)...
end
```

if the Object Test evaluates to true, meaning that *exp* is indeed attached to an object of type *T*, *x* is bound to that value of *exp* over the "scope" of the Object Test, here the whole **then** clause. Calls of target *x* are then guaranteed to apply to a non-void target over that scope. It is necessary to use such a locally bound variable, rather than directly working on *exp*, because if *exp* is a complex expression or even just an attribute of the class many kinds of operation occurring within the **then** clause, such as calls to other routines of the class, could perform assignments that make *exp* void a gain and hence hang the sword of Damocles back up again. The variable *x* is a "read-only", like a formal routine argument in Eiffel: it cannot figure as the target of an assignment, and hence will keep, over the scope of the Object Test, the original value of *exp*, guaranteed to be non-void.

The Object Test resembles mechanisms found in typed object-oriented languages under names such as "Run-Time Type Identification", "type narrowing", "downcasting", and the "with" instruction of Oberon; it addresses their common goal in a compact and general form and is intended to subsume them all. In particular, it replaces Eiffel's original "Assignment Attempt" instruction, one of the first such mechanisms, written *x* ?= *exp* with (in the absence of a specific provision for attached types) the semantics of assigning *exp* to *x* if *exp* happens to be attached to an object of the same type as *x* or conforming, and making *x* void otherwise. An assignment attempt is typically followed by an instruction that tests *x* against *Void*. The Object Test, thanks to its bound variable and its notion of scope, merges the assignment and the test.

Relying on the Object Test instruction alone would yield a complete solution of the Void Call Eradication problem, but would cause considerable changes to existing code. Sometimes it is indeed necessary to add an Object Test for safety, but in a huge number of practical cases it would do nothing but obscure the program text, as the context guarantees a non-void value. For example, immediately after a creation instruction **create Result**..., we know that **Result**, even if declared of a detachable type, has an attached value and hence can be used as the result of a function itself declared attached. We certainly do not want in such a case to be forced to protect **Result** through an Object Test, which would be just noise. An important part of the mechanism is the notion of *Certified Attachment Patterns*: a catalog of program schemes officially guaranteeing that a certain variable, even if declared of an attached type, will in certain contexts always have a certifiably attached value. The catalog is limited to cases that can be safely and universally guaranteed correct, both easily explainable to programmers and easily implementable by compilers; these cases cover a vast number of practical situations, ensuring that the Object Test, however fundamental to the soundness of the approach as a whole, remains — as it should be — a specialized technique to be used only rarely.

An immediate consequence of these techniques will be to remove preconditions, occurring widely in libraries of reusable classes as well as in production applications, of the form **require** *x* / = *Void* for a routine argument *x* (sometimes for an attribute as well). Informal surveys shows that in well-written Eiffel code up to 80% of routines contain such a precondition. With the new type system, it is no longer necessary if we declare *x* of an attached type. Going from preconditions to a static declaration, and hence a compile-time check, is a great boost to reliability and a significant simplification of the program text.

To go from these basic ideas to a full-fledged language mechanism that delivers on the promise of total, statically-enforced Void Call Eradication, the solution must address some delicate issues:

- In a language framework guaranteeing for reliability and security that all variables, in particular object fields, local variables of routines and results of functions, are automatically initialized (an idea also pioneered by Eiffel and widely adopted by recent languages), how to ensure that variables declared of an attached type are indeed initialized to attached values.
- How to handle attached type in the context of genericity. For example, the Eiffel library class *ARRAY* [*G*] is generic, describing arrays of an arbitrary type *G*. Sometimes the corresponding actual parameter will be attached, requiring — or not! — automatic initialization of array entries; sometimes it will be detachable, requiring automatic initialization of all entries to *Void*. It would be really unpleasant, for this and all other container classes, to have to provide two versions, one for detachable types and one for attached types, or even three depending on initialization requirements for attached types. The solution to this issue is remarkably simple (much shorter to explain than the details of the issue itself): if a generic class needs to rely on automatic initialization of variables of the formal generic type (here *G*), make this part of the declaration for the parameter, requiring clients to provide an initialization mechanism for actual parameters that are attached types.
- How to make the whole mechanism as invisible as possible to programmers using the language. We must not force them to use any complicated scheme to attain ordinary results; and we must guarantee an "effect of least surprise". In other words they should be able to write their application classes in a simple and intuitive way, the way they have always done, even if they do not understand all the subtleties of attachment, and it is then our responsibility to ensure that they get safely operating programs and the semantics corresponding to their intuition.

- How to ensure that the resulting type system achieves its goal of total Void Call Eradication. The authors of Spec#, a previous design which influenced this work, write that they expect "*fewer unexpected non-null reference exceptions*" **[3]**. We are more ambitious and expect to remove such exceptions entirely and forever. Here it must be mentioned that although we believe that the design described here reaches this goal we have not provided a mathematical proof or, for that matter, do not yet have a formal framework in which to present such a proof.
- In the case of Eiffel, a well-established language with millions of lines of production code, how to provide a smooth transition to the new framework. The designers of Spec# have the advantage of working on a new research language; Eiffel has commercial implementations with heavy customer investment in business-critical applications, and we must guarantee either backward compatibility or a clear migration path. This alone is a make-or-break requirement for any proposed Eiffel solution.

Our solutions to these issues will be described below.

In finalizing the mechanism we realized that it appears to help with two other pending issues, one widely discussed and the other more esoteric at first sight but important for the future of object technology:

- A *covariant* type system (where both arguments and results of functions can be redefined in descendant classes to types conforming to their originals) raises, in a framework supporting polymorphism and dynamic binding, the specter of run-time type mismatches, or "catcalls", another source of crashes. We suggest the following solution to remove this other threat to the reliability of our software: permit covariant redefinition of an argument (covariant result types are not a problem) *only* if the new type is detachable. Then the new version must perform an Object Test, and no catcall will result. This is a way of allowing the programmer to perform covariant redefinition but forcing him to recognize that polymorphism may yield at run time an actual argument of the old type, and to deal with that situation explicitly. The rule also applies to the case of "anchored types", which is a form of implicit covariance, and appears to resolve the issue.
- An analysis of what it takes to bring *concurrent programming* to the level of quality and trust achieved by sequential programming, and bring it up to a comparable level of abstraction, has led to the development of the SCOOP mechanism [11] based on the transposition to a concurrent context of the basic ideas of Design by Contract. One of the conclusions is to allow a call *x*.*f* (*args*) to use a target *x* representing a "separate" object — an object handled by a different processor — and hence to support asynchronous handling, one of the principal benefits of concurrency, *only* if *x* is one of the formal arguments of the enclosing routine. Then a call to that routine, using as actual argument for *x* a reference to such a separate object, will block until the object becomes available, and then will place an exclusive hold on it for the duration of the routine's execution. But it turns out that, conversely, a call using a separate actual argument should not always reserve the object; for example we might only want to pass to another routine a reference to that object, without performing any call on it. It would not be appropriate to decide on

the basis of the routine's code whether object reservation is needed or not, as a kind of compiler optimization: clients should not have to know that code, and in any case the body of a routine may be redefined along the inheritance hierarchy, so that the language would not guarantee a specific semantics for a routine under polymorphism. Instead, the rules will now specify that passing a separate object as actual argument causes the call to place a reservation on the object *if and only if* the corresponding formal argument is declared of an *attached* type. If not, the routine can assign the argument to another variable, or pass it on to another routine; the target of the assignment, or the corresponding formal argument, must themselves be of an unattached type in accordance with the basic rule stated above. To perform a *call* using such an argument as target, one must check its attachment status, relying as usual on an Object Test; the final new semantic rule is that an Object Test on a separate expression will (like its use as actual argument to a routine with a corresponding attached formal) cause reservation of the object. So a simple convention to define the effect of combining two type annotations, "separate" and "attached", appears to provide the flexible and general solution sought.

In passing, we will see that the mechanism additionally addresses two problems for which solutions were available before, but perhaps addresses them better. One of the problem is Run-Time Type Identification: the Object Test construct provides a simple and general approach to this issue. The other, for which Eiffel already provided a specific mechanism, is "once per object": how to equip a class with a feature that will be computed only once for a given object, and only if needed at execution time. For example a field in objects representing the stock of a company might denote the price history of the share over several years. If needed, this field, pointing to a large list of values, will have to be initialized from a database. If only because of the time and space cost, we want to retrieve these values only if needed, and then the first time it is needed.

The following sections detail the mechanism and these applications.

2 Previous Work, Context and Acknowledgments

The "non-null types" of Spec# are the obvious inspiration for the design presented here. It is a pleasure to acknowledge the influence of that work. Our goal has been to try for a simpler and more general mechanism. The reader who would like to compare the two designs should note that references to Spec# in this article are based on 2003-2004 publications **[3]** **[1]** and check more recent work since Spec# has been progressing rapidly.

Other work addressing some of the same issues has included the Self language's attempt to eliminate Void values altogether **[2]** and my own earlier (too complicated) attempt to provide void-avoidance analysis **[10]**. I also benefited from early exposure to the type system work of Erik Meijer and Wolfram Schulte **[5]**.

The design reported here resulted from the work of the ECMA standardization effort for Eiffel (ECMA TC39-TG4), intended to yield an ISO standard **[4]**. The basic ideas are due to Éric Bezault, Mark Howard, Emmanuel Stapf (TG4 convener and secretary) and Kim Waldén. Mark Howard first proposed, I believe, the idea of replacing Eiffel's Assignment Attempt by a construct also addressing void call eradication. The actual design of that construct, the Object Test, is due to Karine Arnout

and Éric Bezault. This article largely reports on the ideas developed by this group of people. As the editor of the standard I bear responsibility for any remaining mistakes in the mechanism and of course in this article.

Numerous discussions with Peter Müller from ETH have been particularly fruitful in shaping the ideas. The application of the mechanism to SCOOP (the last problem) is part of joint work with Piotr Nienaltowski of ETH. Also helpful have been comments on the Eiffel draft standard from David Hollenberg and Paul-Georges Crismer.

In addition I am grateful to Andrew Black and Richard van de Stadt for their tolerance and kind assistance (extending beyond the normal duties of editors) in getting this article to press.

3 Syntax Extension

In Eiffel's spirit of simplicity the advances reported here essentially rely on one single-letter symbol: it is now permitted to prefix a type by a question mark, as in

```
x: ? T
```

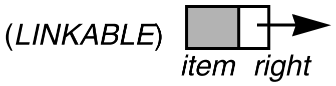
instead of the usual *x*: *T*. (The other syntactical novelty, Object Test, is not an addition but a replacement for the previous Assignment Attempt mechanism.) The question mark turns the type from attached to detachable. It is also possible to prefix a formal generic parameter with a question mark, as in

```
class ARRAY [? G] ...
```

with semantics explained in section 7.

The standards committee decided that in the absence of a question mark **types are attached by default** and hence do not support *Void* as a possible value. This is based on the analysis that void values are of interest to authors of fundamental data structure libraries such as EiffelBase **[7]**, which include classes representing linked data structures such as void-terminated linked lists, but much less to authors of application programs; classes *COMPANY_STOCK* in a financial application or *LANDING_ROUTE* in an aeronautic application are unlikely to require support for void values. So we ask professional library developers working on the basic "plumbing" to specify the possibility of void values when they need it, by using detachable types for example in the declaration of the neighboring item in class *LINKABLE* [*G*] describing linked list items:

```
right: ? LINKABLE [G]
```



but leave application programmers in peace when, as should usually be the case, they don't care about void values and, more importantly, don't want to worry about the resulting possibility of void calls.

This choice of default semantics raises a backward compatibility problem in the context, mentioned above, of preserving the huge commercial investment of Eiffel users; in the previous versions of the language, reference types support void by default, and some programs take advantage of that convention. To address this issue, we provide the symbol **!** as a transition facility. **!** *T* means the attached version of type *T*. In standard Eiffel this will mean the same as *T*, so the exclamation mark symbol is redundant. But offering an explicit symbol enables compilers to provide a migration option whereby the default semantics is reversed (*T* means **?** *T*), compatible with the previous convention. Programmers can then continue to use their existing classes with their original semantics, while starting to take advantage of void-call avoidance guarantees by declaring attached types with the explicit **!**. In the final state, the need for **!** will go away. In the rest of this article we stick with the Standard option: we don't need to use **!** at all, with the understanding that *T* means **!** *T*.

The **?** and **!** symbols are inspired by the conventions of Spec#. There has been criticism on the part of some Eiffel users that these are cryptic symbols ("*it looks like C++ !*") not in the Eiffel style; the symbol **!** in particular has bad karma since it was part of a short-lived syntax variant for the creation instruction now written in the normal Eiffel style as **create** *x*. Although the symbols have the benefit of brevity, they might similarly go away in favor of keywords, not affecting the validity rules, semantics and discussion of the present article.

To understand the rest of that discussion, note that Eiffel has two kinds of type: *reference* types, the default, whose values are reference to objects (or void in the case of detachable types); and *expanded* types, equipped with copy semantics. (The "value" types of C# are a slightly more restricted form of expanded types.) A type is expanded if it is based on a class declared as **expanded class** *C* ... rather than just **class** *C* ... Expanded types serve in particular to represent subobject fields of objects, as well as to model the basic types such as *INTEGER* and *REAL*, enabling Eiffel to have a consistent type system entirely based on the notion of class. Obviously expanded types do not support *Void* as one of their possible values. In the rest of this discussion the term "attached type" covers both non-detachable reference types (the most common case) and expanded types; that is to say, every type except a (reference) detachable type declared explicitly as **?** *T*.

4 Constraints on Calls and Attachment

The fundamental new constraint ensuring avoidance of void calls restricts the target of a qualified call:

Target Validity rule

A qualified call *a*.*f* or *a*.*f* (*args*) is valid only if the target expression *a* is attached.

An expression *a* is said to be attached, in the usual case, if its type is attached. This notion will be slightly generalized below.

A general note on the style of language description: "validity rules" in the specification of Eiffel **[6] [4] [12]** stand between syntax and semantics; they supplement the syntax by placing constraints (sometimes known as "static semantics") on acceptable language elements. Unlike in many other language descriptions, Eiffel's validity rules are always phrased in "if and only if" style: they don't just list individual permitted and prohibited cases, but give an exhaustive list of the necessary *and sufficient* conditions for a construct specimen to be valid, thus reinforcing programmer's confidence in the language. This property obviously does not apply to the rules as given in this article, since it is not a complete language description. The Target Validity rule, for example, appears above in "only if" style since it supplements other clauses on valid calls (such as *a* being of a type that has a feature *f* with the appropriate arguments, exported to the given client). The rules respect the spirit of the language definition, however, by essentially specifying all the supplementary clauses added to the existing rules.

The Target Validity rule will clearly ensure eradication of void calls if attached types live up to their name by not permitting void values at run time; the discussion will now focus on how to meet this requirement.

The other principal new constraint on an existing construct governs attachment. The term "attachment", for source *y* and target *x*, covers two operations: the assignment *x* := *y*, and argument passing *f* (..., *y*, ...) or *a*.*f* (..., *y*, ...) where the corresponding formal argument in *f* is *x*. The basic existing rule on attachment is *conformance* or *convertibility* of the source to the target; conformance, as mentioned, is based on inheritance (with provision for generic parameters), and convertibility is based on the Eiffel mechanism, generalizing ordinary conversions between basic types such as *INTEGER* and *REAL*, and allowing programmers to specify conversions as part of a class definition. Now we add a condition:

Attachment Consistency rule

An attachment of source *y* and target *x*, where the type of *x* is attached, is permitted only if the type of *y* is also attached.

This rule is trivially satisfied for expanded types (the only type that conforms to an expanded type *ET* is *ET* itself) but new for attached reference types.

A companion rule lets us, in the redefinition of a feature in a descendant of the original class, change a result type from detachable to attached, and an argument type from attached to detachable. The rationale is the same, understood in the context of polymorphism and dynamic binding.

This rule narrows down the risk of void call by guaranteeing that if a void value arises somewhere it will not be transmitted, through assignment or argument passing, to variables of attached types. There remains to guarantee that the values *initially* set for targets of attached type can never be void. This sometimes delicate initialization issue will indeed occupy most of the remaining discussion.

5 Initialization

5.1 Variables and Entities

Initialization affects not just variables but the more general notion of "entity". An entity is any name in the program that represents possible values at run time. This covers:

- *Variables*: local variables of routines, attributes of classes (each representing a field in the corresponding instances).
- "*Read-only*" entities: manifest constants, as in the declaration *Pi*: *REAL* = 3.141592, formal arguments of routines, **Current** representing the current objects (similar to this or self).

A variable *x* can be the target of an assignment, as in *x* := *y*. Read-only entities can't, as they are set once and for all. More precisely: a constant has a fixed value for the duration of the program; **Current** is set by the execution (for the duration of a call *x*.*f*, the new current object will be the object attached to *x*, as evaluated relative to the previous current object); formal arguments are attached to the value of the corresponding actuals at the time of each call, and cannot be changed during the execution of that call.

Local variables include a special case, the predefined local **Result** denoting the result to be returned by a function, as in the following scheme:

```
clicked_window (address: URL) : WINDOW                              /2/
                -- Window showing URL for address: depending on user
                -- request, either same as current display window or
                -- newly created one.
        do
                if must_open_in_new_window then
                        create Result.make (address)
                else            -- Keep current window, but display address
                        Result := display_window.displaying (address)
                end
        end
```

This example also illustrates the creation instruction, here using the creation procedure *make*. Unlike the constructors of C++, Java or C#, creation procedures in Eiffel are normal procedures of the class, which happen to be marked as available for creation (the class lists them in a clause labeled **create**).

The example also shows a typical context in which the initialization issue arises: *WINDOW* being an attached type, we must make sure that **Result** is attached (non-void) on exit. Clearly a creation instruction (first branch) produces an attached result. The second branch will work too if the function *displaying*, returning a *WINDOW* and hence required to produce an attached result, satisfies this requirement.

5.2 Self-initializing Types

In earlier versions of Eiffel, initialization has always been guaranteed for all variables, to avoid the kind of run-time situation, possible in some other languages, where the program suddenly finds a variable with an unpredictable value as left in memory by the execution of a previous program if any. This would be a reliability and security risk. Any solution to the initialization issue must continue to avoid that risk.

Since read-only entities are taken care of, it remains to ensure that every variable has a well-defined value before its *first use*, meaning more precisely:

- For local variables of a routine *r*, including **Result** for a function: the first use in any particular call to *r*.
- For attributes: the first use for any particular object. This doesn't just mean the first use in a routine call *x*. *r* (...) where *r* is a routine of the class: it can also be during a creation operation **create** *x*. *make* (...) at the time the object is being created, where *make* may try to access the attribute; or, if contract monitoring is on, in the evaluation of the class invariant, before or after the execution of a routine call.

Eiffel's earlier initialization rules were simple:

1. A variable of a reference type was initialized to *Void*. This policy will be retained for detachable types, but we need a different one for attached types; this is the crux of our problem.
2. The basic types *BOOLEAN*, *CHARACTER*, *INTEGER*, *REAL*, all of them expanded types, specify default initialization values, respectively **False**, null character, 0, 0.0.
3. Programmer-defined expanded types were required to include *default_create* among their creation procedures. *default_create* is a procedure defined in class *ANY* (the top-level class of which all other classes are descendants, similar to Object in other frameworks but in the context of multiple inheritance) where it does nothing; any class can redefine it to implement a specific initialization scheme. Although implicitly present in every class, *default_create* is not necessarily available as a creation procedure; this happens only if the class lists it explicitly in its **create** clause.

Case 2 is in fact an application of case 3, assuming proper versions of *default_create* in the basic types. Note that *default_create* only needs to create a new object in the case of reference types; for variables of expanded types, it can simply apply its algorithm to an existing object.

It is tempting to keep this *default_create* requirement for expanded types, extend it to attached types, and declare victory. This was, however, found too restrictive. First, it would break most existing code: as noted above, we would like to assume that most application classes do not need void values, and so can effortlessly be reinterpreted, under the new scheme, as attached; but we cannot assume that all or even a majority already support *default_create* as creation procedure. In fact this is not such a common case since most non-trivial class invariants require creation procedures with arguments. Even for new classes, the *default_create* requirement is not one we can easily impose on all application programmers.

Even if we can't use impose it universally, this requirement does address the initialization problem for variables of the corresponding types, so we may rely on it when applicable. We give such types a name:

Definition: Self-initializing type

A type is **self-initializing** if it is one of:

- A detachable type.
- A type (including the basic types) based on a class that makes *default_create* from *ANY* available for creation.

For variables of self-initializing types we adopt a policy of **lazy initialization**. The previous policy was systematically to initialize object fields (corresponding to attributes) on object creation, prior to the execution of any creation procedure such as *make* above, and local variables on routine entry, using in both cases the default value, language-set or provided by *default_create*. Instead, we can now afford a more flexible policy: no sweeping general initialization, but, on first access to a variable of a self-initializing type, check whether it has already been set; if not, call *default_create*. This actually implies a slight change of semantics for expanded types:

- Under the previous rules, the semantics for expanded types was that a variable directly denoted an object of that type, rather than a reference. For an attribute, this means a **subobject** of the current object; for a local variable, the compiler-generated code may allocate the object directly on the **stack** rather than on the heap. One of the disadvantages of this approach, apart from its too greedy approach to initialization with *default_create*, is that it requires a special rule prohibiting cycles in the client relation between expanded types: if both *A* and *B* are expanded classes, you can't have *A* declare an attribute of type *B* and conversely, since this would mean that every object of type *A* has a subobject of type *B* and conversely.
- The new semantics is simply that expanded types simply represent objects with **copy semantics** rather than the default *reference semantics*. Using such an object as source of an assignment will imply copying, rather than assign a reference.
- As a result, the clumsy prohibition of no client cycles between expanded classes goes away.
- We also removed the requirement that expanded types provide *default_create* for creation; in other words, they do not have to be self-initializing. When they are not, the same alternative initialization techniques as for attached reference types, discussed below, are available to them, and the same lazy initialization semantics.
- Compilers can now implement expanded types through references; this is purely a matter of implementation, as the only requirement is copy semantics.
- In the vast majority of cases, there are indeed no cycles in the client relation; compilers can then optimize the representation by using subobjects and stack-based allocation as before. In the general spirit of the language's evolution, the idea is to make things simpler and more easy to learn for programmers (just talk about copy semantics, don't worry about implementation), remove hard-to-justify restrictions, and expect a little more of the compiler writer.

- Previously, a creation instruction **create** *x.make* (...), where *make* can be *default_create*, would not (as noted) create an object for expanded *x*, but simply apply *make* to an existing stack object or subobject. Now it may have to create an object, in particular if the relation does have cycles. This is an implementation matter not affecting the semantics.
- Whether or not it actually creates an object, the creation instruction will be triggered the first time the execution needs a particular expanded variable. This change from a greedy policy (initialize everything on object creation or routine entry) to a lazy one can break some existing code if *make* or *default_create* performs some significant operations on the current object and others: this initialization can occur later, or not at all. The new policy seems better, but maintainers of existing software must be warned of the change and given a backward-compatibility option to keep the old semantics.

Except for copy semantics, the rest of this discussion applies to self-initializing reference types as well as to expanded types.

To summarize the results so far, we have narrowed down the initialization problem by taking care of one important case: self-initializing types, for which the policy will be to create the object (or possibly reinitialize an existing object in the expand case) if its first attempted use finds it uninitialized.

This leaves — apart from generic parameters — the case of non-self-initializing types.

5.3 Self-initializing Attributes

If the type is not self-initializing, we can make an individual *attribute* (instance variable) self-initializing. (The technique will not be available for local variables.)

Here, especially for readers steeped in C++ or its successors such as Java and C#, a little digression is necessary about what I believe to be a misunderstanding of object-oriented principles in a specific aspect of the design of these languages. They consider an attribute (also called *instance variable*, *member variable* or *field*) as fundamentally different from a function (or *method*); this is illustrated by the difference in call syntax, as in

```
    y := x.my_attribute                                          /3/
```

versus

```
    y := x.your_function ()          -- Note the parentheses     /4/
```

which makes it impossible to change your mind — go from a storage-based implementation to a computation-based one for a certain query returning information on objects of a certain type — without affecting *every single client* using the query in the above styles. The Principle of Uniform Access **[8]** requires instead that such a choice of implementation should not be relevant to clients. In Eiffel (as already in Simula 67) the syntax in both cases is simply

x.her_query

which could call either an attribute or a function; the term "query" covers both cases.

The problem goes further. Because a class in C++ etc., when it exports an attribute, exports the information that it is an attribute (rather than just a query), it exports it for both reading and writing, permitting remote assignments to object fields, such as

```
        x.my_attribute = new_value                                    /5/
```

This scheme is widely considered bad practice since it violates the principles of information hiding and data abstraction, which would require a procedure call

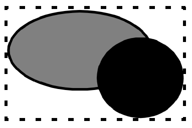
```
        x.set_my_attribute (new_value)                                /6/
```

with a proper set_my_attribute procedure. As a result, textbooks warn against exporting attributes — always a bad sign, since if a language design permits a construct officially considered bad the better solution would be to remove it from the language itself — and suggest writing instead an exported function that will return the value of the attribute, itself declared secret (private), so that instead of the plain attribute access **/3/** one will call, in style **/4/**, a function whose sole purpose is to access and return the secret attribute's value. But this leads to lots of noise in the program text, with secret attributes shadowed by little functions all of the same trivial form (one line to return the value). "Properties", as introduced by Delphi and also present in C#, handle such cases by letting the programmer associate with such a secret attribute a "getter" function and a "setter" procedure, which will respectively return the value and set it. The advantage is to permit the assignment syntax /5/ with the semantics of a procedure call /6/ (as also now possible in Eiffel, with examples below). But the price is even more noise: in C#, altogether three keywords (value, set, get) in the language, and still two separate features in the class — the attribute and the property — for a single query.

The Eiffel policy is different. The Uniform Access Principle suggests that we should make as little difference as possible between attributes and functions. Each is just a query; if exported, it is exported as a query, for access only. The interface of a class (as produced by automatic documentation tools) doesn't show the difference between an attribute and a function; nor, as we have seen above, does the call syntax (no useless empty parentheses).

Standard Eiffel goes further in the application of the principle. In particular, it was previously not possible, largely for fear of performance overhead, to redefine an attribute into a function in a descendant class (while the reverse was permitted). Partly as a result, attributes could not have contracts — preconditions and postconditions — as functions do; postcondition properties can be taken care of in the class invariant, but there is no substitute for preconditions. These restrictions are now all gone, in part because of the availability of better implementation techniques that avoid penalizing programs that don't need the extended facilities. With a new keyword **attribute**, one can equip an attribute with a contract:

```
bounding_rectangle: RECTANGLE                                         /7/
                -- Smallest rectangle including whole of current figure
        require
                bounded
        attribute
        ensure
                Result.height = height
                Result.width = width
                Result.lower_left = lower_left
                Result.contains (Current)
        end
```



With this convention the attribute can freely be redefined into a function and conversely. Note that **Result**, previously meaningful for functions only, is now available for attributes too; the example uses it for its postcondition. This further enhances the symmetry between the two concepts. The previous syntax for declaring an attribute, *x*: *SOME_TYPE*, remains available as an abbreviation for

```
x: SOME_TYPE
        attribute
```

End of digression. This new generality of the concept of attribute suggests another simple mechanism taking care of explicit attribute initialization, and making attributes even more similar to functions: give them an optional algorithm by allowing instructions after **attribute**, the same way a function has instructions after **do** (see e.g. /2/). So we can for example provide *shadow* with an explicit initialization:

```
bounding_rectangle: FIGURE                                          /8/
                -- Smallest rectangle including whole of current figure
                -- (Computed only if needed)
        require
                bounded
        attribute
                create Result.set (lower_left, width, height)
        ensure
                -- As above
        end
```

The semantics is to call this code if — and only if — execution finds, for a particular object, the attribute uninitialized on first use of that object.

An interesting benefit of this technique is to provide a "**once per object**" mechanism, letting us performing a certain operation at most one time on any object, and only when needed, in a lazy style. That's what the algorithm for *bounding_rectangle* does. Here is another example, from a class *COMPANY_STOCK*:

```
stock_history: LIST [VALUATION]                                     /9/
                -- Previous valuations over remembered period
        attribute
                if {l: LIST [VALUATION]}
                                database.retrieved (ticker_symbol) then
                        Result := l   -- Yields list retrieved from database
                else
                        create Result       -- Produces empty list
                end
        ensure
                -- ...
        end
```

The stock history list might be huge, so we only want to retrieve it into memory from the database for a particular company if, and when, we need it. This could be done manually by keeping a boolean attribute that says whether the list has been retrieved, but the technique is tedious is there are many such “lazy” queries. Self-initializing attributes solve the problem in a simpler way. Note the use of an Object Test to check whether the object structure retrieved from the database is of the expected type.

The presence of self-initialization for a particular attribute will, in the semantics, take precedence over self-initialization at the class level if also present.

This concept of self-initializing attribute further narrows down the initialization issue. But it does not yet solve it completely:

- It does not apply to local variables. In fact we could devise a similar notion of “self-initializing local”, where the declaration includes an initialization algorithm. But this seems overkill for such a narrowly-scoped notion.
- For both attributes and local variables the requirement of self-initialization cannot be the only possibility. In some cases a human reader sees immediately that for every use of a variable at run time an assignment or creation will have happened before, giving it a well-defined attached value. Then the lazy initialization-on-demand of either self-initializing types or self-initializing attributes is not necessary, and would in fact be deceptive in the program text since the initialization code will be never be executed. We should simply let things go as originally written, after checking that there is no risk of undefined or void value.

5.4 Certified Attachment Patterns

The last observation leads to the third and last initialization technique: rely on compilers (or other static checking tools) to verify that explicit assignment or creation will have occurred before every use. The authors of Spec# have reached a similar conclusion, taking advantage of modern compiler technology; they write **[1]**:

> *Spec# stipulates the inference of non-nullity for local variables. This inference is performed as a dataflow analysis by the Spec# compiler.*

We differ from this assessment in only one respect: it is not possible in Eiffel to refer to “the compiler”. There are a number of Eiffel compilers, and one of the principal purposes of the ECMA standard is precisely to keep maintaining their specific personalities while guaranteeing full syntactical, validity and semantic interoperability for the benefit of users. Even if there were only one compiler as currently with Spec#, we do not wish to let programmers depend on the smartness of the particular implementation to find out — by trying a compilation and waiting for possible rejection — if a particular scheme will work or not. There should be precise rules stating what is permissible and what is not. These rules should be available in a descriptive style, like the rest of a good language specification, not in an operational style dependent on the functioning of a compiler. They should be valid for any implementation; after all, much of the progress in modern programming language description has followed from the decision to abstract from the properties of a particular compiler and provide high-level semantic specifications instead.

Apart from this difference of view, the Eiffel rules result from the same decision of relying — for cases not covered by self-initializing types or attributes — on statically enforceable rules of good conduct. We call them Certified Attachment Patterns:

Definition: Certified Attachment Pattern (CAP)

A Certified Attachment Pattern for a non-self-initializing variable *x* is a general program context in which *x* is guaranteed to be non-void.

Here is a typical Certified Attachment Pattern, for an arbitrary attribute or local variable *x*. If the body of the routine starts with a creation instruction or assignment of target *x*, then the immediately following instruction position is a CAP for *x*. This is a very important pattern; in fact (as the reader may have noted) neither of the last two examples /8/ /9/ would be valid without it, because they rely on a **create Result** ... instruction to ensure that **Result** is non-void on return from the attribute evaluation. This property is trivial — since the **create** instruction is the last in the routine — but without the CAP there would be no way to rely on it.

The stock history example /9/ also relies on another CAP: if *cap1* and *cap2* are two Certified Assignment Patterns for *x*, then so is **if** *c* **then** *cap1* **else** *cap2* **end** for any condition *c*.

Here is a third CAP, assuming that *x* is a local variable or formal routine argument:

```
if x /= Void then                                                  /10/
        ... Any Instructions here, except for assignments of target x.
end
```

The **then** branch is a CAP for *x*. It would **not** be a valid CAP if *x* were an attribute, as the "Instructions" could include procedure calls that perform an assignment (of a possible void value) to *x*. But for a local variable we can ascertain just by looking locally at the **then** branch that there is no such assignment.

Certified Attachment Patterns, from the above definition, apply to "non-self-initializing variables". This includes variables of attached types that are not self-initializing, but also variables of **detachable** types, which we had not considered for a while. In fact, as the reader may have noted, /10/ is meaningful only for a detachable type; if the type of *x* is attached, and not self-initializing, then the attempt to evaluate it in the test *x* /= *Void* of /10/ would not work; and the test is meaningless anyway for *x* of an attached type. But for detachable *x* the CAP is useful, as it allows us to perform a call of target *x* as part of the Instructions.

Such calls are indeed valid. The Target Validity Rule, the basic constraint of the void-safe type system, stated that "A qualified call *a*.*f* is only valid if *a* is attached". As noted, this usually means that the type of *a* is attached, but we can generalize the definition to take advantage of CAPs:

Definition: Attached expression

An expression *a* is attached if and only if either:

- Its type is an attached type.
- It occurs as part of a Certified Attachment Pattern for *a*.

Without this CAP, we would have, for every use of a local variable *x* of a detachable types, to write an Object Test (with the need to shadow *x* with an explicitly declared Object-Test-Local *y*, as in **if** {*y*: *TYPE_OF_X*} *x* **then** ...) every time we want to use *x* as target of a call. Occasion ally this cannot be avoided, but often the routine's algorithm naturally includes **if** *x* /= *Void* **then** ..., which the CAP allows us to use as it stands, in the way we would normally do.

An associated CAP is for *x* in the else part of **if** *x* = *Void* **then** ... **else** ... end. Another one for *x*, particularly important for class invariants, is in *other_condition* in

```
x /= Void and then other_condition
```

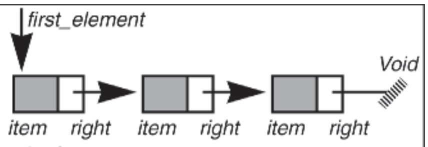
where **and then** is the nonstrict conjunction operator, guaranteeing that the second operand will not be evaluated if the first evaluates to false. This also works if we replace **and then** by **implies** (implication, nonstrict in Eiffel, i.e. *a* **implies** *b* is defined with value true if *a* has value false, even if *b* is not defined); it works for **or else** if we change the test to *x* = *Void*.

Another Certified Attachment Pattern, similar to the first, is particularly important for loops iterating on linked data structure. It is of the form

```
from
        ...
until
        x = Void
loop
        ... Any Instructions not assigning to x ...
end
```

If *x* is a local variable (again, not an attribute), it remains attached throughout the Instructions. This makes possible, without further ado — in particular, without any Object Test — a whole range of common traversal algorithms, such as this one for searching in a linked list:

```
from
        x := first_element
until
        x = Void or else Result
loop
        Result := (x.item = sought_value)
        x := x.right
end
```



(**Result** starts out false; the loop will set it to true if and only if the item in one of the list cells has an *item* field equal to *sought_value*. *x* is as before a local variable) The CAP enables us to write the loop exactly as we would write it anyway, with the guarantee that it will not produce any void call. A look the previous version of the EiffelBase library suggests that many existing loops will similarly compile and run "as is"; occasionally, application of the Target Validity rule will require a slight rewrite, at worst inclusion of some instructions in an Object Test. This is extremely encouraging (especially given the complexity of some of the intermediate suggestions, some involving changes to the loop construct, that were experimented before we arrived at the general solution reported here). More generally, we see as particularly attractive the prospect of replacing, in such a library, hundreds of occurrences of

```
some_routine (x: SOME_TYPE)
        require
                x /= Void
                x.some_property
```

by just

```
some_routine (x: SOME_TYPE)
        require
                x.some_property
```

with the non-void test turned into a compile-time guarantee (*SOME_TYPE* being an attached type) that *x* indeed represents an object, so that we can concentrate on the more meaningful contractual properties such as *x.some_property*.

A CAP, very useful in practice, applies to the instructions that immediately follow a series of creation instructions **create** *a* ..., for one or more *a*: these instructions are a CAP for such *a*. Beyond local variables, this also applies to attributes, somewhat neglected by the previous CAPs, and enables us to handle many simple cases such as guaranteeing that a just created **Result** of a function, as in /8/ and /9/, is attached as expected.

Finally, as a concession to programmers who prefer to run the risk of an exception in the case of a variable that shouldn't be void but is, we include as CAP the position immediately following

```
check
        x /= Void
end
```

taking advantage of Eiffel's **check** instruction. This instruction will raise an exception if *x* is void. This CAP is an escape valve, as we do not feel like preventing programmers from using an exception-based style if that's their choice (which we may disapprove).

Using CAPs to guarantee attachment is a pessimistic policy, erring, if at all, on the side of safety: if we cannot absolutely guarantee the impossibility of a void value, the Target Validity rule will (except, as noted, under backward-compatibility compiler options, to avoid breaking existing code) reject the code. The design rule for CAPs is not that they *support all correct cases*, but that they *reject any incorrect case*. We can afford to miss some correct cases if they do not occur too frequently; the only drawback will be that programmers may have, in some extreme and (we hope) rare situations, an Object Test that appears unnecessary. (Remember that one of the reasons those cases are so rare is that CAPs are only a technique of next-to-last resort, and Object Tests of the last one: in many practical cases the Eiffel programmer can rely on self-initializing types or variables.) As a result we can afford not to care too much about cases that worry the Spec# designers **[3]** **[1]**, such as a creation procedure that needs to access an attribute that one is not sure has already been initialized. In Eiffel, the attribute will often be of a self-initializing type, or itself be declared **attribute** ... so that it is self-initializing; if not, there might be a matching CAP; if not, the programmer can always get away with an Object Test or, if that's the preferred style, force a CAP with a **check** instruction as above. We don't have to turn our compilers into prodigies of dataflow analysis.

We do not, in fact, want CAPs to be too sophisticated. They should cover situations where it is immediately obvious to a human reader (and, besides, true!) that an expression cannot take on a void value even though it is neither of an attached type nor self-initializing. The argument should be simple and understandable. If it is convoluted, it may be just as well to force a slight rewrite of the immediate context to make the safety argument compelling. In other works, when it comes to establishing guaranteed attachment status, *we do not want Eiffel compilers to be too smart* about possible voidness. The argument should always remain clearly understandable to the reader of the program, in the Eiffel spirit of clarity and quality-focused software engineering. (There is still a great need for sophisticated dataflow analysis and more generally for very smart compiler writers: generate the fastest and most compact code possible.)

This approach rests under the assumption that a small number of simple CAPs capture the vast majority of practical situations. This seems to be the case with the set of CAPs sketched above, covering most of what has been included in the Eiffel standard, where they are of course specified much more precisely. On the organizational side, the existence of an international standards committee provides a good framework: even if the CAP catalog remains separate from the Eiffel standard proper, permitting more frequent additions, it should remain subject to strict quality control and approval by a group of experts after careful evaluation. Technically (beyond "proof by committee"), the goal should be, with the development of a proper mathematical framework, to *prove* — through machine-validated proofs — the validity of proposed CAPs. The three criteria that must remain in force throughout that process are:

- A guarantee of correctness beyond any doubt.
- Simple enforceability by any reasonable compiler, *without* dataflow analysis.
- Understandability of all CAPs by any reasonably qualified programmer.

6 Object Tests and Their Scopes

The Object Test form of boolean expression, {*x*: *T*} *exp*, was presented in the Overview, which gave the essentials. *T* is an attached type; *exp* is an expression; *x* is a fresh name not used for any entity in the enclosing context, and is known as the **Object-Test-Local** of the expression. Evaluation of the expression:

- Yields true if and only if the value of *exp* is attached to an object of type *T* (and so, as a particular consequence, not void).
- Has the extra effect of binding *x* to that value for the subsequent execution of the program extract making up the scope of the Object Test. *x* is a Read-Only entity and hence its value can never be changed over that scope.

The scope depends on where the Object Test appears. We saw that in **if** *ot* **then** ... **else** ... **end**, with *ot* an Object Test, the scope is the **then** part. Also, if a condition is of the form *ot* **and then** *boolexp* or *ot* **implies** *boolexp*, the scope includes *boolexp* as well. With a negated Object Test, **not** {*x*: *T*} *exp*, the scope, in a conditional instruction, is the **else** part; such negated variants are particularly important for loops, since in

```
from ... until not {x: T} exp loop ... end
```

the whole loop clause — the loop body — is part of the scope.

The notion of scope has been criticized by some experienced Eiffel programmers who in line with the Eiffel method's emphasis on command-query separation **[8]** do not like the idea of an expression evaluation causing initialization of an entity as a side effect. But apart from some unease with the style there seems to be nothing fundamentally wrong there, and the construct does provide a useful and general scheme.

In particular, it is easier to use than Eiffel's earlier Assignment Attempt mechanism *x* ?= *y*. Although an effective and widely used method of run-time type ascertainment, the Assignment Attempt treats the non-matching case by reintroducing a void value (for *x*), which in light of this entire discussion doesn't seem the smartest idea. An Assignment Attempt almost always requires declaring the target *x* specially as a local variable; with Object Test we integrate the declaration in the construct. It should almost always be followed by a test *x* /= *Void*, yet it is possible for programmers to omit that test if they think the object will always match; this is a source of potential unreliability. Here we essentially force such a test through the notion of scope.

In general, the Object Test seems an attractive alternative to the various run-time type identification and ascertainment (including downcasting) in various languages; it seems to subsume them all.

7 Generic Classes

Perhaps the most delicate part of the attachment problem is the connection with genericity. There turns out to be a remarkably simple solution. (This needs to be pointed out from the start, because the detailed analysis leading to that solution is somewhat longish. But the end result is a four-line rule that can be taught in a couple of minutes.)

Consider a container class such as *ARRAY* [*G*] (a Kernel Library class) or *LIST* [*G*]. *G* is the "**formal generic parameter**", representing an arbitrary type. To turn the class into a type, we need to provide an "**actual generic parameter**", itself a type, as in *ARRAY* [*INTEGER*], *LIST* [*EMPLOYEE*]. This process is called a "**generic derivation**". The actual generic parameter may itself be generically derived, as in *ARRAY* [*LIST* [*EMPLOYEE*]].

Genericity can be constrained, as in *HASH_TABLE* [*ELEMENT*, *KEY* –> *HASHABLE*] which will accept a generic derivation *HASH_TABLE* [*T*, *STRING*] only if *STRING* conforms to (inherits from) the library class *HASHABLE* (in the Eiffel Kernel Library it does). Unconstrained genericity, as in *ARRAY* [*G*], is formally an abbreviation for *ARRAY* [*G* –> *ANY*].

None of these class declarations places any requirement on the attachment status of a type. You can use — subject to restrictions discussed now — *ARRAY* [*T*] as well as *ARRAY* [**?** *T*]. The same holds even for constrained genericity: attachment status does not affect conformance of types. (So if *U* inherits from *T*, **?** *U* still conforms to *T*. It's only for entities and expressions that the rules are stricter: With *x*: *T* and *y*: **?** *U*, *y* does not conform to *x*, prohibiting the assignment *x* := *y*.) Without such rules, we would have to provide two versions of *ARRAY* and any other container class: once for attached types, one for detachable types. Not an attractive prospect.

Now consider a variable of type *G* in a generic class *C* [*G*]. What about its initialization **?** *G* stands for an arbitrary type: detachable or attached; if attached, self-initializing or not. Within the class we don't know. But a client class using a particular generic derivation needs to know! Perhaps the most vivid example is array access. Consider the declarations and instruction

```
x, y: T
i, j: INTEGER
arr: ARRAY [T]
...
arr.put (x, i)          -- Sets entry of index i to x; Can also be     /11/
                        -- written more conventionally as arr [i] := x
```

This sets a certain entry to a certain value. Now the client may want to access an array entry, the same or another:

```
y := arr.item (j)       -- Can also be written as y := arr [j]          /12/
```

T is an attached type. Instruction /11/ will indeed store an attached value into the *i*-th entry, assuming the array implementation does its job properly. Since the class *ARRAY* [*G*] will, as one may expect, give for function *item* the signature

```
item (i: INTEGER): G
```

and the actual generic parameter for *arr* is *T*, instruction /12/ correspondingly expects the call *arr*.*item* (*j*) to return a *T* result, for assignment to *y*. This should be the case for *j* = *i*, but what about other values of *j*, for which the entry hasn't been explicitly set by a *put* yet?

We expect default initialization for such items of container data structures, as for any other entities. But how is class *ARRAY* [*G*], or any other container class, to perform this initialization in a way that will work for all possible actual generic parameters:

detachable, as in *ARRAY* [**?** *T*], expanded, or attached as with *ARRAY* [*T*] but with *T* either self-initializing or not?

The tempting solution is to provide several versions of the class for these different cases, but, as already noted, we'd like to avoid that if at all possible. We must find a way to support actual generic parameters that are detachable, easy enough since we can always initialize a *G* variable to Void, or attached, the harder case since then we must be faithful to our clients and always return an attached result for queries such as *item* that yield a *G*.

The result of such a query will be set by normal instructions of the language, for example creations or assignments. For example the final instruction of a query such as *item* may be **Result** := *x* for some *x*. Then **Result** will be attached if an only if *x* is attached. Although *x* could be a general expression, the properties of expressions are deducible from those of their constituents, so in the end the problem reduces to guaranteeing that a certain entity *x* of the class, of type *G*, is attached whenever the corresponding actual parameter *T* is. Let's consider the possible kinds of occurrence of *x*:

G1 x may be a formal argument of a routine of C. From the conformance rules, which state that only *G* itself conforms to *G*, *x* will be of type *T* (the actual generic parameter of our example), detachable or attached exactly as we want it to be. Perfect! Other cases of read-only entities are just as straightforward. From then on we consider only variables.

G2 We may be using *x* as a target of a creation instruction **create** *x*.*make* (...) or just **create** *x*. That's the easiest case: by construction, *x* will always be attached, regardless of the status of *T*. (To make such creation instructions possible the formal generic parameter must satisfy some rules, part of the general Eiffel constraints: it must specify the creation procedures in the generic constraint, as in *C* [*G* –> *C* **create** *make* **end**], where *make* is a procedure of *C*, or similarly *C* [*G* –> *ANY* **create** *default_create* **end**]. Then the actual generic parameter *T* must provide the specified procedures available as creation procedures.)

G3 We may be using *x* as the target of an assignment *x* := *y*. Then the problem is just pushed recursively to an assessment of the attachment status of *y*.

G4 The last two cases generalize to that of an occurrence in a Certified Assignment Pattern resulting from the presence of such a creation instruction or assignment instruction guaranteed to yield an attached target, for example at the beginning of a routine.

G5 So the only case that remains in doubt is the use of *x* — for example in the source of an assignment — without any clear guarantee that it has been initialized. If *x*'s type were not a formal generic, we would then require *x* to be self-initializing: either by itself, through an **attribute** clause, or by being of a self-initializing type. But here — except if we get a self-initializing attribute *x* of type *G*, a possible but rare case — we expect the guarantee that *G* represents a self-initializing type.

We don't have that guarantee in the general case; *T*, as noted, may be of any kind. And yet if *T* is not self-initializing we won't be able to give the client what it expects. So what we need, to make the mechanism complete, is language support for specifying that a generic parameter must be self-initializing (that is to say, as defined earlier, either detachable or providing *default_create* as a creation procedure). The syntax to specify this is simply to declare the class, instead of just *C* [*G*], as

```
class C [? G] ...
```

This syntax is subject to criticism as it reuses a convention, the **?** of detachable types, with a slightly different meaning. But it seemed preferable to the invention of a new keyword; it might change if too many people find it repulsive, but what matters here is the semantic aspect, captured by the validity rule:

Generic Initialization rule

Consider a formal generic parameter *G* of a class C.

1. If any instruction or expression of C uses an entity of type *G* in a state in which it has not been provably initialized, the class declaration must specify **?** *G* rather than just *G*.
2. If the class declaration specifies **?** *G*, then any actual generic parameter for *G* must be self-initializing.

A formal generic parameter of the form is known as a **self-initializing formal**; clearly we must add this case to the list of possibilities in the definition of self-initializing types.

In the Standard these are two separate validity rules. There are both very easy to state and apply. The first is for the authors of generic classes — typically a relatively small group of programmers, mostly those who build libraries — and the second for authors of clients of such classes; they're a much larger crowd, typically including all application programmers, since it's hard to think of an application that doesn't rely on generic classes for arrays, lists and the like.

Class *ARRAY* will fall under clause 1, declared as *ARRAY* [**?** *G*]; this makes it possible to have arrays of *T* elements for an attached type *T*. The rule is very easy to explain to ordinary application programmers (the second group): *ARRAY* gives you a guarantee of initialization — you'll never get back a void entry from an *ARRAY* [*T*], through *arr*. *item* (*i*), or arr [*i*] which means the same thing —, so you must provide that default initialization yourself by equipping *T* with a *default_create*. Now if you can't, for example if *T* is really someone else's type, then don't worry, that's OK too: instead of an *ARRAY* [*T*] use an *ARRAY* [**?** *T*]; simply don't expect arr [*i*] to give you back a *T*, it will give you a **?** *T*, possibly void, which you'll have to run through an Object Test if you want to use it as attached, for example as the target of a call. Fair enough, don't you agree?

This **?** *G* declaration leading to a requirement of self-initializing actual generic parameters applies to class *ARRAY* because of the specific nature of arrays, where initialization has to sweep through all entries at once. It doesn't have to be carried through to data structures subject to finer programmer control. For example, in class *LIST* [*G*] and all its EiffelBase descendants representing various implementations of sequential lists, such as *LINKED_LIST* [*G*], *TWO_WAY_LIST* [*G*], *ARRAYED_LIST* [*G*] etc., the basic operation for inserting an item is *your_list*. *extend* (*x*), adding *x* at the end, with implementations such as

```
extend (x: G)
                -- Add x at end.
        local
                new_cell: LINKABLE [G]
        do
                create new_cell.make (x)
        end
```

Then, to get the items of a list, we access fields of list cells, of type *LINKABLE* [*G*] for the same *G*, through queries that return a *G*. This is case G1, the easiest one, in the above list, guaranteeing everything we need to serve our attached and detachable clients alike.

Most generic classes will be like this and will require no modification whatsoever, taking just a *G* rather than a **?** *G*. *ARRAY* and variants (two-dimensional arrays etc.) are an exception, very important in practice, and of course there will be a few other cases.

8 Getting Rid of Catcalls

Having completed Void Call Eradication, we come to the second major problem, whose discussion (to reassure the reader) will be significantly shorter; not that the problem is easier or less important, but simply because the solution will almost trivially follow from the buildup so far.

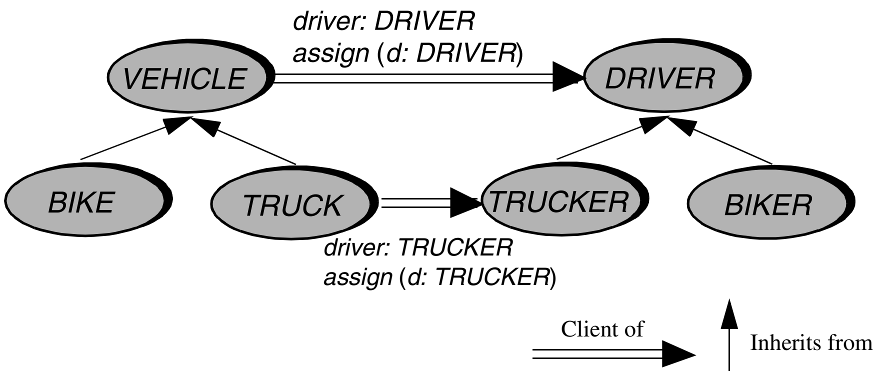
Typed object-oriented programming languages are almost all *novariant*: if you redefine a routine in a descendant of the class containing its original declaration, you cannot change its signature — the type of its arguments and results.

And yet... modeling the systems of the world seems to require such variance. As a typical example, consider (see the figure on the opposing page) a class *VEHICLE* with a query and command

```
driver: DRIVER
register (d: DRIVER) do driver := d end                                  /13/
```

A vehicle has a driver, of type *DRIVER* (a companion class) and a procedure *register* that assigns a driver. No we introduce descendant classes *TRUCK* and *BICYCLE* of *VEHICLE*, and *TRUCKER* and *BIKER* of *DRIVER*. Shouldn't *driver* change type, correspondingly, in *TRUCK* and *BICYCLE*, to *TRUCKER* and *BIKER* respectively? All signs are that it should. But novariance prevents this.

The policy that would allow such type redefinitions is called covariance (from terminology introduced by Luca Cardelli); "co" because the redefinition follows the direction of inheritance.



In fact there is no type risk associated with redefining **query results**, such as *driver*, covariantly. Still, most languages don't permit this, probably because then programmers wouldn't understand why they can also redefine **routine arguments** covariantly. If you redefine *driver*, you will also want to redefine register so that its signature reads

```
register (d: TRUCKER) do driver := d end        -- in TRUCK   /14/
register (d: BIKER)   do driver := d end        -- in BIKE    /15/
```

and so on. Eiffel allows you to do this and in fact provides an important abbreviation; if you know ahead of time (that is to say, in the ancestor class) that an entity will be covariant, you can avoid redefinitions altogether by declaring the entity from the start as "anchored" to another through the **like** keyword: here in *TRUCK* you can replace /13/ by

```
register (d: like driver) ... Rest as before ...   -- in VEHICLE /16/
```

where the **like** type declaration anchors *d* to driver, so that the redefinitions of /14/ and /15/ are no longer needed explicitly (but the effect is the same). **like**, avoiding explicit "redefinition avalanche", is the covariant mechanism par excellence.

With covariant arguments we have a problem **[9]** because of polymorphism and dynamic binding. The declarations and call

```
v: VEHICLE
d: DRIVER
...
v.register (d)
```

look reasonable enough; but what the call is preceded by the assignments

```
v := some_truck
d := some_biker
```

with the types of the assignment sources as implied by their names? We end up assigning to a truck a driver qualified only to ride a bike. Then when the execution attempts, on an object of type *TRUCKER*, to access a feature of the driver — legitimately assumed to be a truck driver, on the basis of the redefinition —, for example *driver.license_expiration_date*, we get a crash, known as a catcall (assuming truck licenses expire, but bike licenses don't). This is the reason novariance is the general rule: even though catcalls happen rarely in well-written programs, they are just as much of a risk as void calls.

The solution proposed here is simple: force the programmer who makes a covariant argument redeclaration to recognize the risk through the following rule:

Covariant argument redeclaration rule

The type of a covariant argument redeclaration, or of an anchored (like) argument declaration, must be detachable.

In our example an explicit redeclaration will have to be written, instead of /14/

```
register (d: ? TRUCKER) ...                    -- in TRUCK          /17/
```

and an anchored one, instead of /16/:

```
register (d: ? like driver) ...                -- in VEHICLE        /18/
```

This requires the body of the routine (in the redefined version for the first case, already in the original version for the second case) to perform an explicit Object Test if it wants to apply a call to the argument, ascertaining it to be of the covariantly redefined type. Catcalls clearly go away.

The semantics of **?** U in the covariant redefinition of an argument *x* originally of type *T* is slightly different from the usual one involving possible void values. It really means "from *T* down to *U*". It also requires a particular convention rule for the semantics of a new precondition clause of the form require else *x.some_U_property* (we interpret it as {*y*: *U*} *x* **and then** *y.some_U_property*). So there is a certain amount of kludginess on the theoretical side. But in practice the technique seems to allow us to keep covariance for expressiveness, while removing the dangers.

This technique is not so far from what programmers instinctively do in languages such as C++, Java and C# which enforce novariance. The modeled system, as noted, often cries for covariance. So in the descendant class the programmer will introduce a variable of the new type, the one really desired, and "downcast" (the equivalent of an Object Test) the novariant argument to it. One finds numerous examples of this pattern in practical code from the languages cited. The above rule leads us to a similar solution, but it is more explicit and, one may argue, better for modeling realism: the programmer specifies, in the redefinition, the "true" new type of the argument (like *TRUCKER*); the type system accepts his covariant behavior, but forces him to recognize the risk to of that behavior to others around him, specifically to "polymorphic perverts" (callers of the original routine which, through polymorphism, actually use the new argument type disguised under the old one), and to handle that risk by checking explicitly for the type of the actual objects received through the formal argument.

9 An Application to Concurrency

The third major problem to which the ideas discussed here provide a solution is lazy object reservation in concurrent object-oriented programming.

Concurrency is badly in need of techniques that will make concurrent programs (multithreaded, multi-processed, networked, web-serviced...) as clear and trustworthy as those we write for sequential applications. Common concurrent mechanisms, most notably thread libraries, still rely on 1960-era concepts, such as semaphores and locks. Deadlocks and data races are a constant concern and a not so infrequent practical occurrence.

An effort to bring concurrent programming to the same level of abstraction and quality that object technology has brought to the sequential world led to the definition of the SCOOP model of computation (Simple Concurrent Object-Oriented Programming **[11]**, with a first implementation available from ETH). This is not the place to go through the details of SCOOP, but one aspect is directly relevant. If an entity *x* denotes a "**separate**" object — one handled by a thread of control, or "processor", other than the processor handling calls to the current object — it appears essential to permit a call of the form *x*.*f* only if *x* is an argument of the enclosing routine *r*. Then a call to *r*, with the corresponding actual argument *a* representing a separate object, will proceed only when it has obtained exclusive access to that object, and then will retain that access for the duration of the call. Coupled with the use of preconditions as wait conditions, this is the principal synchronization mechanism, and leads to elegant algorithms with very little explicit synchronization code (see for example the Dining Philosophers in **[11]**).

The rule then is:

Validity and semantics of separate calls

A call on a separate object is permitted only if the object is known through a formal argument to the enclosing routine.

Passing the corresponding separate actual arguments to the routine will cause a wait until they are all available, and will reserve them for the duration of the call.

A specific consequence of this policy is of direct interest for this discussion:

Object Reservation rule

Passing a separate actual argument a to a routine *r* reserves the associated object.

The corresponding formal argument *x* in *r* must also be declared as **separate**, so that it is immediately clear, from the interface of *r*, that it will perform an object reservation.

So far this has implied the converse rule: if a is separate, *r* (*a*) will wait until the object is available, and then will reserve it.

But this second part is too restrictive. If *r* doesn't actually include any call *x*. *f* where *x* is the formal argument corresponding to *a*, we don't need to wait, and the policy could actually cause deadlock. For example in

```
keep_it: separate T                -- An attribute

r (x: separate T)
          -- Remember a for later use.
     do
          keep_it := x
     end
```

we just use *x* as source of an assignment. The actual calls will be done later using *keep_it*, which we will have to pass (according to the above validity rule) as actual argument to some other routine, which performs such calls. But in a call to this *r* we don't need to wait on *x*, and don't want to.

This could be treated as a *compiler optimization*: the body of *r* doesn't perform any call on *x*, so we can skip the object reservation. But this is not an acceptable solution, for two reasons:

- The semantics — including waiting or not — should be immediately clear to client programmers. They will see only the interface (signature and contracts), not the **do** clause.
- In a descendant class, we can redefine *r* so that it *now* performs a call of target *x*. Yet under dynamic binding a client could unwittingly be calling the redefined version!

We need a way to specify, as part of the official routine interface, that a formal argument such as *x* above is, although separate, non-blocking.

The solution proposed is: **use a detachable type**, here **? separate** *T*. With the declaration

```
r (x: ? separate T)                -- The rest as above
```

no reservation will occur.

With this policy, whether reservation occurs is part of the routine's specification as documented to client programmers. The rule is consistent with the general property of detachable and attached types: we need *x* to be attached only if we are going to perform a call on it.

In the example, we can only retain the assignment to *keep_it* if this attribute is itself detachable. If it is attached, declared as **separate** *T* rather than **? separate** *T*, we must rewrite the assignment as

```
if {y: separate T} x then
        keep_it := y
end
```

with the obvious semantic rule that **an object test of separate type causes reservation of the object**. Unlike in the case of an argument, this rule is acceptable since no information hiding is involved: we are looking at implementation, not a routine's interface.

The technique also fits well with inheritance. If *r* had an argument of type **? separate** *T*, we cannot of course redefine it as **separate** *T* in a descendant (the reverse is, as always, possible). But if the descendant version does need to perform a call on *x*, whereas the original didn't, it can achieve the result through an Object Test:

```
if {y: separate T} x then
        y. some_operation
end
```

which will cause a wait — not on entry to the routine (which could contradict the semantics advertised to the client) but as part of that routine's implementation.

It appears then that the distinction between attached and detachable types, and the general-purpose Object Test with its semantics adapted to the concurrent (separate) case, solve this particular problem of concurrent object-oriented programming too.

10 Conclusion

It was impossible to resist including the self-citation that opens this article, but hard to resist the temptation of removing the parts that don't quite fit, especially the bit about the two or three minutes. The ideas presented here didn't come with the self-evidence of morning mist; it was more like the icy rain of an endless Baltic winter. Yet the mechanism is indeed a minuscule syntax extension, the **?** symbol (even if used with two slightly different semantics), combined with the replacement of an existing instruction, the assignment attempt, by a simpler and more general mechanism, the Object Test. With a few validity rules that any reasonable program should meet without the programmer thinking much about them — even though the presentation in this article may have appeared long-winded since it took into account many details, special cases, compatibility issues and the rationale for every decision — the result does address several major problems in one sweep; one of these problems, the starting point for the whole effort, is the only remaining source of crashes in typed object-oriented programs, and hence of critical practical importance. The second one has also been the subject of a considerable literature. The third one is less well known, but of importance for concurrent applications. And in passing we have seen that for two issues that had been addressed by previous mechanisms — Run-Time Type Identification, possible in many languages, and "once per object", for which Eiffel already had a solution — the mechanisms allows new techniques that may offer at least an incremental improvement on those already known.

So while it is for the reader to judge whether the citation is arrogant, I do hope that the mechanisms presented above, as available in Standard Eiffel, will have a lasting effect on the quality of software that we can produce, using the best of object technology.

References

[1] Mike Barnett, Rustan Leino and Wolfram Schulte: *The Spec# Programming System*; CASSIS proceedings, 2004.

[2] Craig Chambers et al., papers on the Self language at http://research.sun.com/self/papers/papers.html.

[3] Manuel Fähndrich and Rustan Leino: *Declaring and Checking Non-null Types in an Object-Oriented Language*; in OOPSLA 2003, SIGPLAN Notices, vol. 38 no. 11, November 2003, ACM, pp. 302-312.

[4] ECMA Technical Committee 39 (Programming and Scripting Languages) Technical Group 4 (Eiffel): *Eiffel Analysis, Design and Programming Language*, Draft international standard, April 2005.

[5] Erik Meijer and Wolfram Schulte: *Unifying Tables, Objects, and Documents*; in Proc. DP-COOL 2003, also at http://research.microsoft.com/~emeijer/Papers/XS.pdf.

[6] Bertrand Meyer: *Eiffel: The Language*, Prentice Hall 1990 (revised printing 1991). See also **[12]**.

[7] Bertrand Meyer: *Reusable Software: The Base Object-Oriented Component Libraries*, Prentice Hall, 1994.

[8] Bertrand Meyer: *Object-Oriented Software Construction, 2nd edition*, Prentice Hall, 1997.

[9] Bertrand Meyer, reference **[8]**, chapter 17: "*Typing*".

[10] Bertrand Meyer, *Prelude to a Theory of Void*; in *Journal of Object-Oriented Programming*, vol. 11, no. 7, November-December 1998, pages 36-48.

[11] Bertrand Meyer, reference **[8]**, chapter 30: *Concurrency, distribution, client-server and the Internet*.

[12] Bertrand Meyer, *Standard Eiffel*, new edition of **[6]**, in progress.