Draft of a paper submitted to a planned volume in honor of Tony Hoare's 75-th birthday, 2009.

Avoid a Void: The eradication of null dereferencing

Bertrand Meyer*⁺, Alexander Kogtenkov⁺, Emmanuel Stapf⁺

⁺Eiffel Software, www.eiffel.com
***ETH Zurich, Chair of Software Engineering,** se.ethz.ch

1 Repairing the one-billion-dollar mistake

Tony Hoare recently spoke [6] about the issue of null dereferencing:

> *I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*

The present note (a modest attempt at a 75-th birthday present) describes how a statically typed object-oriented programming language has, four decades after Algol W and half a century after the appearance of NIL in Lisp, liberated its users from the scourge of null-dereferencing failures.

Hoare's statement seems to hint that just a little more care and discipline in the design of Algol W would have prevented the problem from even arising. Even in light of Hoare's record of finding brilliantly simple solution to problems that had eluded other researchers, our experience makes us doubt that null dereferencing could have vanished at the stroke of a pen. The problem indeed is not the *null reference*, a concept that appears as necessary to the type systems of usable programming languages as zero — another troublemaker, the tormentor of division — to the number system of mathematics. What threatens to make programs crash is the risk of *null dereferencing* (or "void call"): a run-time attempt to apply to a null (void) reference an operation that can only work if the reference denotes an object — in other words, if it is not void. In mathematics we want zero and we also want to divide numbers (by non-zero denominators); the difficulty is to avoid ever applying a division to a zero denominator. In programming, we want void references and we also want to apply operations to objects (through the associated non-void references); the difficulty is to avoid ever applying a call to a void reference, a property we shall call **void safety**. This need to reconcile two desirable but conflicting goals explains that a void safety policy requires not only a sound theoretical concept but also careful engineering. As often in language design, success involves a delicate balancing act between programmer expressiveness and run-time safety.

Devising, refining and documenting the concept behind the mechanism presented here took a few weeks. The engineering took four years.

This article describes the concept, the same in its essence as first presented in the original article [10] and ISO language standard [5] but with important simplifications, and discusses the engineering of the implemented solution. All the mechanisms presented are in use today as part of EiffelStudio 6.4 (commercial and open-source licenses); in particular, the libraries have been updated to void safety.

While the authors take responsibility for the present article and the implementation, any credit for the design of the mechanism must be shared with other members of the ECMA TC49-TG4 committee, in particular Éric Bezault, Karine Bezault and Mark Howard.

We are grateful to Erik Meijer, Rustan Leino, Manuel Fähndrich, Wolfram Schulte and other members of Microsoft Research for introducing us to the Spec# non-null type mechanism [1] [4], which provided the decisive influence on our work. From the many people whose comments helped improve the mechanism we should cite Peter Müller, Piotr Nienaltowski (for applications to concurrency not detailed in this article), Kim Waldén, David Hollenberg, Bernd Schoeller, Paul-Georges Crismer, Ian King and Jocelyn Fiat.

2 Overview

We start with a description of the problem and an outline of the solution

2.1 The void safety issue

Finding a practical solution to void safety may be delicate, but describing the problem is easy. In an object-oriented language, the basic computational mechanism is a feature call (also known as method call and as message passing), of the form

$x.f(a, b, \ldots)$ **/E1/**

where x is expected to denote an object and f is a feature (operation). The arguments a, b, … play no role here and the discussion will ignore them. If x is of a *reference* type, its possible run-time values are references; a reference is either *attached* to an object or *void*. If the value of x is attached to an object, /E1/ will apply f to that object. A *void call* arises if the value of x is void. Void safety is the avoidance of void calls.

In a language that does not guarantee void safety, the run-time effect of a void call is either to crash the program or, if the language supports exception handling, to trigger an exception. (The difference is not necessarily significant in practice, since programs seldom provide sophisticated exception handling for such cases: if a programmer detects the risk of a void call, he will generally find it just as simple to remove it than to let an exception happen and try to recover through an exception handler.)

The risk of void calls is already present in Pascal and C programs, where f takes no argument and denotes a field. It is even more acute in O-O languages since /E1/ is the basic form of object-oriented computation. In the absence of a void safety mechanism, the risk of void call is a Sword of Damocles potentially threatening the execution of every O-O program.

With the possible exception of some arithmetic overflows, it is the *last* such risk. By adopting static typing, most modern O-O languages have ruled out *type mismatch*, the other potential source of run-time failure. Type mismatch would arise in /E1/ if x were dynamically attached to an object on which no appropriate feature f is available, or the feature exists but cannot accept a, b, … as arguments. The type system of such languages as Eiffel, Java and C# excludes this through simple rules enforced at compile time: every variable and expression must have an explicit type; this type must be based on a class that includes a feature f; the types of the actual arguments a, b, … must conform to those of the formal arguments of f; and in any assignment $x := y$ (or actual-formal argument association) the type of y must conform to the type of x.

Static typing ensures type safety. The goal of the present discussion is to replicate that success story for void safety.

2.2 Sketch of the solution

The basic elements of the mechanism detailed below are the following. “Attached”, as defined so far, is a property of run-time values (references). To be more explicit we may call it “*dynamically* attached”. The void safety mechanism defines another property, “*statically* attached”, applicable to variables and expressions; compilers (and human readers) can determine static attachment through a simple analysis of the program text. The goal is to ensure the following property:

Attachment consistency: If x is statically attached, its possible run-time values are dynamically attached.

The language rule is then simply:

Void safety rule: A call $x.f(\ldots)$ is only permitted if x is statically attached.

This rule will only ensure void safety, and hence deserve its name, if the language mechanism satisfies attachment consistency. The mechanism appears to meet this requirement; at present, however, we have not performed a mathematical proof.

The rest of this discussion uses the word “attached” without specifying “statically” or “dynamically”, since the context removes any ambiguity: static attachment is a property of program elements (variables and expressions); dynamic attachment is a property of run-time values.

The void-safety mechanism provides three ways to ensure that a variable or expression *x*, of some type *T*, is (statically) attached:

- *T* is an *attached type*. An attached type is devised so that all its values will be dynamically attached; in other words, it does not admit **Void** as one of its values. Using an attached type is in principle the most effective way to guarantee the absence of void calls; the disadvantage is that any attached type must possess an initialization mechanism ensuring that the corresponding variables have a non-void value on first use. The remaining two cases assume that *T* is a *detachable* (not attached) type.
- The context of a call may guarantee that *x* has a non-void value. For example if *x* is a local variable the call in

  ```
  if x /= Void then x.f (...) end                                /E2/
  ```

  is void-safe. The language definition includes a small number of such schemes, known as **certified attachment patterns** or CAPs.
- More delicate cases may use the *object test* construct to guarantee safety. /E2/ is not necessarily void-safe if *x* is an arbitrary expression, because of the possibility of side effects and also in the presence of multithreading. The scheme becomes void-safe if rewritten as

  ```
  if attached x as l then l.f (...) end                          /E3/
  ```

  The boolean expression **attached** *x* **as** *l* is the object test; it has value True if an only if the value of *x* is dynamically attached to an object, and also has the effect of binding *l* (a fresh name) to that object over the *scope* of the object test, which in this case is the **then** clause.

In addition, special care must be taken when handling variables of generic types.

These rules are the core of the void safety mechanism and will be reviewed in detail below.

3 Background

To understand the language mechanism it is necessary to review the constraints that apply to any solution, and the precise role of void references in practical programming.

3.1 Constraints on the solution

The following constraints governed the design of the solution:

Requirements on the void-safety mechanism

1 *Static*: compile-time mechanism ensuring full void safety.

2 *General*: applicable to generic types and concurrent programming.

3 *Simple*: no mysterious rules; for programmers, easy to learn; for compiler writers, realistically implementable.

4 *Compatible*: minimal language extension; respects the spirit of the language; fits well with other constructs; does not limit programmer expressiveness; minimum change for existing code.

Constraint 1 prescribes an entirely static mechanism, like type safety. If a compiler accepts a program, it must guarantee the absence of void calls in any execution of the generated code.

Constraint 2 requires support for advanced language mechanisms:

- A class may be **generic**. In the class *LIST* [*G*], the name *G* represents an arbitrary type; a call of the form /E1/ is more delicate to handle if *x* of type *G* than if it is of a known type, since *G* represents many possible actual types, as in the "generic derivations" *LIST* [*INTEGER*], *LIST* [*SOME_REFERENCE_TYPE*], *LIST* [*LIST* [*INTEGER*]] and so on.
- The mention of **concurrency** illustrates how perversely language mechanisms can interact with each other. Void safety might at first seem independent from concurrency issues, but it is not. In particular, scheme /E2/ is not void-safe if *x* is an attribute (representing object fields) rather than a local variable as previously assumed: in the presence of multithreading, another thread can make *x* void between the time the current thread ascertains that *x* /= **Void** and the time it takes advantage of that finding to execute *x*.*f* (...). This problem is the major irritant in the practical use of the void safety mechanism. Its presence is particularly frustrating to us since the SCOOP concurrency mechanism [11] designed for Eiffel handles multithreading at a higher level of abstraction and removes such fine-grain interference; since SCOOP is not yet available as part of standard Eiffel implementations, programmers today must use traditional multithreading techniques, which may cause interference in schemes such as /E2/ and as a result complicate the void safety mechanism.

Constraint 3 states that void safety should not come at the expense of simplicity. Professional programmers should learn the new mechanisms easily, and the language should still be teachable to novices. (In a university setting we rely on object-oriented techniques for first-semester introductory programming and like to present the language as used in industry). This constraint also cautions against making the language too hard to implement; we can be tougher on compilers writers than on language users, but within reason.

Relevant to both language users and compiler writers is the avoidance of *mysterious rules*. Today's compilers may use sophisticated techniques to determine that certain schemes are void-safe; such an approach is only acceptable if it relies on clear criteria which can be explained in the form of simple language rules. Otherwise programmers have to rely blindly on their compiler, not understanding what is going on; and one compiler may reject a program that another accepts.

Constraint 4 covers compatibility. It was a critical requirement for the work described here, whose goal was not to design a language from scratch but to make an existing language void-safe. There are several aspects to compatibility:

- *Minimal language extension*: additions to the programming language — constructs, keywords — should be kept to a minimum.
- *Respecting the spirit of the existing language design*: new mechanisms should fit with existing ones. In the Eiffel case, the language design follows a number of explicit principles. such as "Provide *one* good way to do anything important"; they should be retained.
- *Not limiting programmers' power of expression*: the void safety mechanism should avoid bothering programmers unless their code demonstrably causes a risk of void call. This requirement is one of the hardest to satisfy.
- *Minimum change for existing code*: We need the best possible mechanism for future generations, but we also have to contend with millions of lines of existing production code.

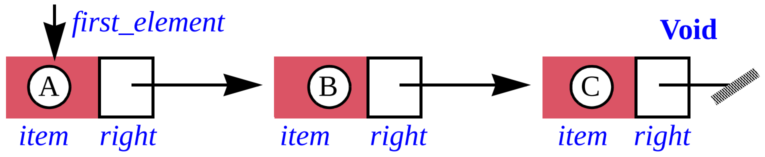
This final requirement causes the worst headaches. Some of the existing programs may contain sources of potential void calls (if programmers only ever wrote void-perfect code, there would be no need for any new mechanism); the problem is to avoid false alarms. As much as possible, we would like to accept existing code unchanged except for elements that are demonstrably void-unsafe. This goal has only been reached in part; we have succeeded in minimizing change to existing code, but not in eliminating it.

The adaptation of the standard Eiffel libraries themselves took up the better part of a release cycle (6 months). This measure is not representative, since the circumstances were peculiar: we were performing such an effort for the first time, learning along the way; we did not have a conversion guide (but as a by-product of the effort developed such a guide [3], facilitating the tasks of others who need to adapt their programs); and we were still refining the mechanism as we went, improving the end result but causing delays in our the process. Still, migration of existing code continues to require a significant effort.

3.2 Void and attached references

A literal reading of Hoare's comment might suggest that void references are dispensable. One may indeed wish for a language design that magically gets rid of them. Unfortunately, this is only a dream; attempts have been made, as in the Self language [2], but they do not remove the underlying problem.

This problem is, at its core, the need to terminate linked structures. In the same way that numbers need a zero to denote the number that can be added to another without changing it, linked representations of data structures such as lists and trees need **Void** to denote the reference that can be included in an object to refer to no other object. Linked lists are the archetypal example:



In the figure, the list is represented by three cells (of type *LINKABLE* [*CHARACTER*] in the EiffelBase library), each with a field *item* giving the value and a reference field *right* leading to the next cell. The *right* field in the last cell cannot lead to any object and hence must be void.

The presence of such possibly void references immediately introduces the risk of void calls. For example, if *first_element* is the reference to the first list cell (*first_element* is indeed the name of the corresponding attribute in the *LINKED_LIST* class), and we assume *first_element* itself has a non-void value, then *first_element*.*right* might be void, so an attempt to evaluate *first_element*.*right*.*right* may cause a void call. This result is not a nasty side-effect of introducing an ill-conceived notion (void references) into the programming language, but the expression of an incontrovertible property: the expression does not make sense if the list has fewer than two elements — just as $1 / (m - n)$, in ordinary mathematics, does not make sense if m happens to be equal to n.

Because void calls result from logical impossibilities, not artifacts of language design, simplistic attempts to remove void references disguise the problem rather than removing it. Two such ideas are:

- Prohibit void references and use some trick to represent structure termination; for example, in linked lists, the last *right* reference would point to the cell itself. This breaks the acyclic nature of linked lists, and requires checking every use of *right*, for example in a list traversal, to determine whether it is truly a reference to another element or an artificial self-reference marker.
- Use inheritance to distinguish between two kinds of *LINKABLE* cells: proper linkables and end markers. The last *right* would be of the second kind. This complicates the inheritance structure, and requires checking every use of *right* for its type. The effect on performance (in particular the extra load on the garbage collector) can also be noticeable.

Such solutions replace void checks (**if** *right* /= **Void then** ...) by checks of another kind, with no clear benefit. They cause the risk that a wrong dereferencing, such as *first_element*.*right*.*right*.*item* applied to a one-element list, will yield an incorrect result (in the first solution, the value of the first element) instead of an exception. In erroneous cases, it is generally preferable to crash than to continue and deliver wrong results.

This discussion dashes any hope of casting off the void safety problem by simply casting off void references. It also includes some good news. Terminating linked structures seems to be the *only* case that truly requires void references. That case occurs in system-oriented parts of programs, typically in libraries that implement fundamental data structures; in the more application-oriented parts of a program, void references are generally unnecessary. Consider a program manipulating bank accounts objects, each with an *owner* field denoting a *PERSON*. To represent the notion of a bank account by an unknown owner, a void reference is usually not the best solution; instead, the program can define a special object representing a *PERSON* with incomplete information.

This observation suggests a software design guideline: confine void references to specific parts of a system, largely preserving the application-oriented layers from having to worry about the issues discussed in this article. It also confirms our expectation that the conversion of typical user applications will require significantly less effort than our initial experiences, which involved system-level libraries.

4 Basic language mechanisms

We now review the three major techniques for ensuring void safety: attached types in the present section, CAP and object test in the next two.

4.1 Attached types

The void safety mechanism introduces into the type system an attachment qualifier for types. A reference type is either *attached* or *detachable*. The difference is that **Void** is a valid value for detachable types only.

Syntactically, a detachable type is indicated by the keyword **detachable**, as in

```
right: detachable LINKABLE [G]
```

Attached types can similarly use the keyword **attached**, but this is usually not needed as "attached" is the default status; a declaration

```
owner: PERSON            -- In class BANK_ACCOUNT
```

has the same meaning as *owner*: **attached** *PERSON*. This policy follows from the above observation that application-oriented types do not normally need a void value. Experience has shown it to be the right decision, even though it raises compatibility issues (see constraint 4) since previous code followed the inverse convention.

The explicit form **attached** *PERSON* is mostly useful during the transition period. The EiffelStudio compiler provides a compatibility option with three possible values, settable class-by-class to help programmers migrate existing code progressively:

- Enforce void safety. This is the default for the future.
- Do not enforce void safety; ignore **attached** and **detachable** type marks. This is the full-backward-compatibility option.
- Enforce void safety rules, but only for attached types; leave detachable types alone (void calls will still be possible in the corresponding cases). This is a transition option.

Defining a type *T* as attached — using any syntactic convention — is only a declaration; the important part of the language mechanism is the set of semantic rules that ensure the *attachment consistency* property introduced earlier: any run-time use of a variable *x* of type *T* must find *x* attached to an object. This requires the mechanism to enforce two properties:

A1 The value of *x* on first use, as defined by language initialization rules, must be non-void.

A2 Any assignment to *x* must leave the value of *x* non-void.

Objective A2 is the easier of the two, achieved through the following rule:

Attachment Preservation rule

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

An "attachment operation" is either an assignment *x* := *y* or an argument passing *f* (…, *y*, …) where the corresponding formal argument is *x*. (This meaning of the word "attachment" to denote an operation predates its use to describe, as in the rest of this article, the status of a reference.)

This leaves the issue of initialization (A1). Eiffel (followed in this respect by other modern object-oriented languages) includes initialization rules for all types; for example, all integer variables are initialized to zero and all booleans to False. For reference types, the earlier initialization value was **Void**: the most obvious one, and indeed the only one applicable to all (detachable) reference types — but also the worst possible choice for an attached type! The void safety mechanism requires a new policy.

The key notion is that of a variable being *properly set*, meaning that it has been given a non-void value. It figures in the rule for variables of attached types:

Attached Type Initialization rule

If a program uses the variable at a certain position, one of the following properties must hold:

- The variable is properly set at that position. This possibility applies to both kinds of variable: attributes of a class, and local variables of a routine.
- The variable is an attribute, and is properly set at the end position of every creation procedure of the class.

The *creation procedures* of a class, also known as its *constructors*, are the initialization operations associated with the class. A variable *x* of type *T* is properly set at a certain position if one of the instructions preceding that position is an assignment to *x* (which, thanks to the Attachment Preservation rule, gives it an attached value if *T* is an attached type) or a creation instruction of target *x* (of the form **create** *x*…).

As an example, the use of *x* in the last instruction of the following routine is valid:

```
r
        do
                create y
                …
                x := y
                print (x)
        end
```

This assumes that *x* and *y* are variables of the same type. In the last instruction, *x* is properly set since the preceding instruction is an assignment to *x*. In that assignment, the source *y* itself is properly set since it was earlier the target of a creation instruction; without this property, the assignment *x* := *y* would be invalid as it is uses the value of *y*.

This discussion and the rule do not apply to formal arguments of routines, which are not variables (they cannot be the target of assignments). In a routine *r* (*x*: *T*), the Attachment Preservation rule guarantees that the formal argument *x* will always be attached since in any call *r* (*y*) the actual argument *y* must be attached.

As stated, the condition guaranteeing that a variable is properly set is over-conservative; for example, it does not imply that *x* is properly set after

```
if c then create x else x := y end        -- With y properly set                /E4/
```

In moving library code to void safety we realized that scheme /E4/ is fairly common, and added it, along with a few others, to those accepted by EiffelStudio as void-safe. Since these cases fall outside of the current language definition, a compiler option is available to disable them for portability; we will submit them to the standards committee for inclusion in the initialization rule. The challenge — in line with constraint 3, "No mysterious rules" — is to replace operational, compiler-oriented descriptions of specific cases by general language rules, lending themselves to clear, abstract definitions.

The simplicity of the rules governing attached types suggests using these types as much as possible; as noted, the business-oriented parts of a program should limit themselves almost exclusively to attached types. This will spare them the difficulties of ensuring void safety for detachable types — the topic of the remaining sections.

4.2 Certified Attachment Patterns

In the interest of simplicity and compatibility (constraints 3 and 4), it is desirable to accept without change some program schemes that are demonstrably void-safe even though they involve detachable variables or expressions. Such a scheme is called a Certified Attachment Pattern or CAP. The language standard currently defines a single CAP, which covers a number of useful cases. It is phrased as follows:

Basic CAP

A call *x*.*f* (…), where *x* is a formal argument or local variable of a routine, is void-safe if this call both:

- Appears in the scope of a void test involving *x*.
- Is not preceded, in that scope, by a setter for *x*.

The arguments, variables and expressions considered in this rule and the rest of the discussion are all of detachable types, since no further rules are necessary for the attached case. A *void test* is one of the following, possibly involving a boolean expression *e*:

- Positive void test: the simple form *x* = **Void**, or the composite form *x* = **Void or else** *e*.
- Negative void test: the simple form *x* /= **Void**, or the composite form *x* /= **Void and then** *e*.

(The operators **and then** and **or else** are the non-commutative variants of **and** and **or**: if *a* is False, evaluating *a* **and then** *b* will not evaluate *b*; if *a* is True, evaluating *a* **or else** *b* will not evaluate *b*.) The scope of a void test *v* includes:

S1 If *v* is of a composite form: the rest of the expression, *e*.

S2 If *v* is a negative void test appearing in a conditional instruction **if** *v* **then** … **end** (where the instruction may include **elseif** clauses and an **else** clause): the **then** clause.

S3 If *v* is a positive void test appearing in a conditional instruction **if** *v* **then** … **else** … **end** (where the instruction may include **elseif** clauses): the **else** and **elseif** clauses.

S4 If *v* is a positive void test appearing in a loop instruction **from** … **until** *v* **loop** … **end**: the loop body (**loop** clause).

A "setter for *x*" is an assignment to *x* or a control structure that (recursively) includes a setter for *x*.

The CAP makes it possible to sanction a wide range of fundamental programming schemes. Examples, accepted as void-safe, include not only the simple conditional /E2/ but algorithms for traversing or searching linked structures, such as:

```
from
        l := first_element
until
        l = Void or else l.item ~ sought          -- ~ is object equality
loop
        l := l.right
end
```

Such schemes occur frequently in basic libraries of data structures and algorithms and reflect natural ways of expressing search and other traversal operations. It was critical, as part of the compatibility and simplicity of use requirements, to accept them without change.

Case S1 allows other frequent schemes, arising in boolean expressions:

```
x /= Void and then x.some_property                                   /E5/
x = Void or else x.some_property                                     /E6/
```

/E5/ is widely used in contracts: preconditions, postconditions and class invariants. It expresses a requirement that an object, *if it exists*, satisfies a certain property. For example the equality operation on two references *a* and *b* has the postcondition

```
Result =
    (a = Void and b = Void) or else ((a /= Void and b /= Void) and then a.is_equal (b))
```

defining them to be equal if they are either both void or attached to equal objects. The operation *is_equal* ascertains the equality of two objects, and hence can only be applied to a non-void target and a non-void argument. Case S1 of the CAP certifies this postcondition as void-safe.

These results extend to contracts consisting of several clauses; the language semantics treats them as if they were separated by **and then**, so that if a routine starts with

```
r (x: detachable T)                                                  /E7/
    require
        x /= Void
        x.some_property
```

its precondition is equivalent to /E5/; as a consequence, the second clause is void-safe.

The CAP is limited to formal arguments and local variables, excluding attributes. Cases such as the following are indeed not void-safe if *x* is an attribute:

```
if x /= Void then                                                    /E8/
    routine_call
    x.f (…)
end
```

One of the reasons why this example could lead to a void call was mentioned in connection with the simpler variant /E2/: in a multithreading execution, another thread could falsify the property *x* /= **Void** before the call. Another risk, even with single threading, is that a routine call may perform an assignment to the attribute *x*.

Similarly, the second precondition clause in the following variant of /E7/ is not void-safe if *x* is an attribute rather than a formal argument:

```
r                                                                    /E9/
    require
        x /= Void
        x.some_property
```

Achieving void safety in such cases may require a specific technique: object test.

4.3 Object test

The “object test” language construct provides, under a simple syntax, a general solution to the problem known as *run-time type identification* (or RTTI), with particular application to void safety.

In object-oriented languages, it is generally not necessary to query an object directly for its type; the preferred technique [8] is to use inheritance, polymorphism and dynamic binding. In a call *x*.*f* (…), the target *x* may be polymorphic, meaning that it may at run time become attached to objects of different types, all descending from a common ancestor. Any of these classes may provide a specific version of *f*; then dynamic binding ensures that each call will select the appropriate version, based on the type of the object actually attached to *x*. For the simplicity and ease of change of the software’s architecture these techniques are preferable to letting the program test explicitly for the type of the object and select the appropriate variant of *f* through a conditional instruction.

Techniques of run-time type identification, also known as “type narrowing” and “downcasting”, are necessary for more complex cases where these standard object-oriented techniques do not apply. The most obvious example is that of an object obtained from the outside world (a file, a network): the program has no choice but to posit a certain type and dynamically find out whether the object matches it.

Object test provides a general mechanism for run-time type identification, which we find preferable to existing approaches, including the construct previously available in Eiffel (assignment attempt, introduced in 1988). An object test checks that a reference is attached to an object of a specified type, and if so catches the object under a local name to avoid safe processing over a certain syntactic scope.

An object test on a given expression *exp* of a detachable type may appear as follows, here as part of a conditional instruction

```
if attached {T} exp as l then                                        /E10/
          … Operations on l, such as l.f (…) …
end
```

The object test is **attached** {*T*} *exp* **as** *l*. It is a boolean expression, evaluating to True if and only if the value of *exp* is a reference attached to an object of type *T* or conforming. Then the “object-test local” *l*, a fresh name (distinct from the names of all variables in the context), will denote a reference to that object throughout the *scope* of the object-test local, defined by the same rules as the scope of a void test.

Here, the scope is the **then** clause; the effect of the conditional instruction is to determine whether *exp* is attached to an object of type *T* and, if so, to apply the given operations to that object. Because the object is captured under the temporary name *l* at the time of the object test, no interference may arise from multithreading or from operations that could cause the value of *exp* to change.

In such uses of an object test for void safety rather than general run-time type identification, the type *T* is often just the (static) type of *exp*; for that reason the qualifier {*T*} is optional. We may indeed write the object test in /E10/ as just **attached** *exp* **as** *l*. (The “**as** *l*” part can also be omitted for RTTI uses that need the object only to test its type, but in void safety applications it is necessary.)

An object test will achieve void safety in the second variant of the precondition example /E9/, which involved an attribute and hence was not covered by the CAP. A void-safe version is

```
r                                  -- Void-safe version of /E9/              /E11/
          require
                    x /= Void
                    attached x as l and then l.some_property
```

The second clause is void-safe thanks to clause S1 of the CAP.

Although the first clause *x* /= **Void** is now redundant, we have retained it for the (frequent) occurrences of this pattern in our library conversions so far. The reason is to help testing and debugging. In the case of a contract violation, the run-time contract monitoring tools identify the violated clause; for a void *x*, it is clearer to see a violation of the first clause, since a violation of the second one does not distinguish between a void *x* (no object) and an object that exists but does not satisfy *some_property*.

5 Fine-tuning the mechanism

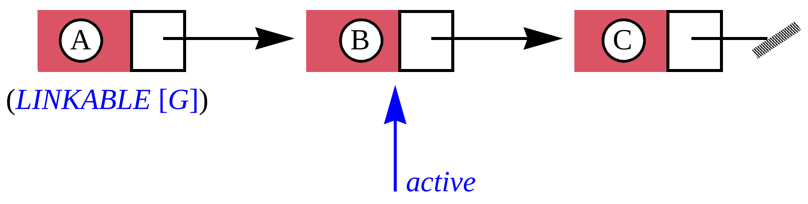
Object test would in principle provide, just by itself, a general solution to void safety: protect every feature call *x*.*f* (…) with an object test on *x*. This would be an extreme solution, and indeed the general design guideline, consistent with the object-oriented method's reluctance to use RTTI except when polymorphism and dynamic binding are not applicable, is to minimize the use of object test.

We have already seen two alternatives, superior to object test whenever they can be applied: attached types and CAPs. Two other facilities, the Check instruction and stable attributes, help limit the use of object test.

5.1 The Check instruction

The Check instruction — not a new construct, but an existing part of Eiffel's Design by Contract mechanism [8] — expresses that a certain property is assumed always to hold at a certain program position, whether or not a proof exists. It is particularly relevant for void safety, addressing cases where a detachable expression is known from the context to be non-void.

The following example is typical. Class *LINKED_LIST* describes lists in a linked implementation, with list cells implemented as instances of *LINKABLE* as discussed above. *LINKED_LIST* itself describes a list as a whole and includes a notion of "cursor", which can be moved to a list item, as well as to the positions immediately to the left of the first item if any and to the right of the last item if any:



The attribute *active* represents the cursor position. By convention, *active* is always attached to a list cell, except if the list is empty; this is reflect by a clause of the class invariant:

```
(active = Void) implies is_empty
```

Now consider a routine of the class with the precondition **not** *is_empty*. From the invariant, it also satisfies *active* /= **Void**. With the rules given so far, however, the following version will be rejected:

```
r
        require
                not is_empty
        do
                active.some_operation
        end
```

The attribute *active* is of type **detachable** *LINKABLE* [*G*] and it is not used as part of a void-safe CAP, so the value of *active* at the place of the call is not known to be attached. There is no easy way to address this problem through a solution based only on the type system. We can protect the call with an object test, through one of the forms

```
if attached active as l then l.some_operation else "Raise an exception" end      /E12/
if attached active as l then l.some_operation end                                 /E13/
```

In both cases we test for a property that we expect always to hold; the difference is that if it doesn't /E12/ raises an exception whereas /E13/ does nothing. Neither variant is satisfactory: the object test is in principle unnecessary; if there is a mistake, doing nothing is not an appropriate solution. The recommended scheme in such a case is

```
check attached active as l then l.some_operation end                /E14/
```

The semantics of **check** *e* **then** *Instructions* **end** (inspired, in its current form, by Spec#'s "assert" instruction) is that the construct requires any compiler to either:

- Prove that *e* will always hold. (This assumes a "verifying compiler" equipped with proof capabilities — another concept promoted by Tony Hoare [7].)
- Generate code with the effect of /E12/, triggering an exception if *e* is dynamically not satisfied.

The first solution is the desirable one, but places high demands on the compiler (see "realistically implementable" in constraint 3). The second solution, currently implemented in EiffelStudio, is open to criticism: one can argue that it simply replaces one kind of exception, void call, with another. To refute this criticism we note that there exists in practice a fundamental difference: a program written prior to the introduction of void safety is chock-full with potentially unsafe calls; adapted to obey the void safety rules, it may still contain a few **check** constructs in the style of /E14/, which may cause exceptions but are clearly identified; they will be the natural target of quality assurance efforts intended to demonstrate that the condition will never be True during execution.

5.2 Stable attributes

As described, the void safety mechanism does a reasonable job of not bothering programmers (especially, not forcing them to write object tests) when there is no real risk of void call; the annoying case that remain is scheme /E2/ applied to an attribute:

```
if x /= Void then x.f (...) end                                      /E2/
```

This is not void-safe and requires an object test. The culprit, as noted, is multithreading, and the annoyance will remain as long as SCOOP-based concurrency is not fully available. In one case, however, the annoyance can be avoided. The technique relies on the following notion:

Definition: stable attribute

A detachable attribute is *stable* if it does not appear as target of an assignment whose source is a detachable expression.

("Detachable attribute" and "detachable expression" are abbreviations for attribute and expression of a detachable type.) A stable attribute might have a void value on object creation, but once it becomes attached it will remain attached, since every value that gets assigned to it will be attached. The definition is restrictive: the source of an attachment cannot just be stable, it has to be attached. Even so, however, stable attributes do constitute a substantial share of detachable attributes; detecting them is useful since /E2/ is void-safe for a stable attribute, avoiding an object test.

A compiler can easily determine, when seeing a pattern such as /E2/, that *x* is a stable attribute (just check all the assignments in the class). In spite of this property, a recently adopted language extension introduces an explicit **stable** marker in the syntax for attribute declarations

```
stable x: detachable SOME_TYPE
```

The reason for this decision is that in the implicit approach, where the compiler silently blesses /E2/ when it determines that *x* is stable, a small change to a class, such as adding an assignment in a routine, may suddenly cause the program no longer to compile because of a void-safety violation in a seemingly unrelated part of the code.

It remains an open question in the standards committee whether — for the sake of minimizing programmer annoyance — /E2/ should be accepted anyway, possibly with a compiler warning, for an attribute that happens to be stable but has not been declared as such.

6 Handling genericity

All the types considered so far were explicit, and known directly from the program text. A typed object-oriented language will also provide a generic mechanism, with classes such as *LIST* [*G*] where *G* represents an arbitrary type. A particular use of the class uses a type obtained as a *generic derivation*, using a type as actual generic parameter; an example is *LIST* [*CHARACTER*]. The question of void safety arises here too: for *x* declared of type *G* in a class *C* [*G*], under what conditions is a call *x*.*f* (…) void-safe? The techniques that we have seen are different for attached and detachable types; but within the class we do not know whether the actual generic parameter *T* of any particular generic derivation *C* [*T*] will be of one kind or the other.

To obtain a suitable solution, we should first note the general convention regarding genericity. The basic form is *constrained* genericity, whereby a class is declared as

```
class C [G –> CT] …
```

for some type *CT* known as the *constraint* (or "constraining type"). This means that the only valid generic derivations are of the form *C* [*T*] where *T* conforms (according to inheritance-based rules) to the constraint *CT*. *Unconstrained* genericity, as in *LIST* [*G*], is simply an abbreviation for a special case of constrained genericity, *LIST* [*G* –> *ANY*], where *ANY* is the library class serving as ancestor to all programmer-defined classes (*ANY* is called Object in some object-oriented languages).

The introduction of void safety leads to a slight adaptation of this convention: unconstrained genericity *C* [*G*] is now an abbreviation for *C* [*G* –> **detachable** *ANY*]. The type system, as implied by the discussion in previous sections, ensures that *T* (synonym for **attached** *T*) conforms to **detachable** *T* but not the other way around. Within the class *C* [*G*], then, *G* represents a detachable type; with *x* declared of type *G*, the call *x*.*f* (…) is not void-safe and must be protected through a CAP, an object test, a Check instruction or a stable attribute status for *x*.

It is also possible to declare *x* of type **attached** *G*; then the call is void-safe, but *x* is subject to the initialization constraints on attributes of attached types.

As another possibility, the class declaration may read *C* [*G* –> *ANY*] or more generally *C* [*G* –> *T*] for some attached type *T* (remember again that these are synonyms for *C* [*G* –> **attached** *ANY*] and *C* [*G* –> **attached** *T*]). Then *G* represents an attached type and the call is void-safe.

These rules make it possible to write void-safe generic container classes — representing structures such as lists, stacks queues and many others — in a convenient fashion. One of the goals defined for the design of the void safety mechanism was indeed to enable a smooth and easy conversion of the EiffelBase library of fundamental data structures and algorithms, involving as little rewrite as possible.

One generic structure, however, remains beyond the scope of these techniques: arrays.

7 Arrays

The language definition is engineered in such a way that programmers get the advantages of two complementary views of arrays:

- The traditional view, with the performance benefits of direct-access, contiguous-memory representation, and usual array syntax such as.

```
a [i] := a [i] + p                                              /E15/
```

- The object-oriented view, where *ARRAY* [*G*] has all the prerogatives of a normal generic class: it is characterized by features (so that /E15/ is simply an abbreviation for the object-oriented form *a*.*put* (*a*.*item* (*i*) + *p*)), has preconditions, postconditions and a class invariant, and can be used by other classes, for example as ancestor in inheritance.

ARRAY, however, is a special class because its implementation relies on a contiguous memory area that cannot be entirely managed by the program. For void safety, this means that it is impossible without a further mechanism to accept a declaration

```
a: ARRAY [T]
```

where *T* is an attached type. Normally, an array is created through the *make* creation procedure, as

```
create a.make (l, h)
```

where the arguments *l* and *h* are the initial bounds, low and high, of the array. After that creation, any operation is possible on the array, including an access to *a* [*i*] as in /E15/ — assuming in this case that *T* has a "+" operation, so that the right side of /E15/ is an abbreviation for *a* [*i*].*plus* (*p*). But this call is not void-safe: the creation procedure *make* has no provision for initializing the array elements; there is no guarantee that *a* [*i*] will be non-void.

We see here a fundamental consequence of making the language void-safe: language-specified automatic initialization of variables, mentioned at the beginning of this article as a key property of modern object-oriented languages, is no longer possible, or at least not in a simple form. Previously every type had a default value, which was **Void** for all reference types; this property is still applicable to detachable types and to *expanded* types (non-reference types, including basic types such as *INTEGER*), but not to attached reference types.

If *T* is detachable, the items of an *ARRAY* [*T*] will be initialized to **Void**; for expanded types, they will be initialized to the default value such as zero for integers; but for attached *T* no such universal default is available.

This property does not cause any particular problem for ordinary generic classes such as *LINKED_LIST*, since they are entirely under the control of the programmer, who can also take care of initialization for any type, in accordance with rules of the previous section. But creating an array allocates a memory area that lies beyond the programmer's direct control. Another way to express this observation is that arrays remain a system-level island in a modern high-level language.

As a result of this situation, arrays require special treatment. The solution is simple: for attached reference types (as opposed to detachable types and expanded types), array creation may no longer use the *make* creation procedure, with its two arguments representing bounds; it must instead use a newly introduced creation procedure, *make_filled*, with an extra argument representing a default value. Its signature in *ARRAY* [*G*] is

```
make_filled (low, high: INTEGER; value: G)
```

allowing creation instructions of the form

```
create a.make (l, h, val)
```

where *val* is a value of the appropriate type; if that type is attached, the type rules guarantee that *val* must be attached as well — a reference to an actual object. Upon creation of the array, every one of its entries will be a reference to that object. More precisely, the abstract requirement is that for an entry that has not been explicitly initialized *a* [*i*] must yield such a reference; the implementation can achieve it in smarter ways than physically copying the reference. The implementation should also consider the needs of garbage collection (to make sure that the shared object can be freed if no longer useful).

> To avoid making up a specific default value for every array, a mechanism is under consideration that would allow defining a default value for any type, under the form {*T*}.*set_default* (*val*). (The notation {*T*}, a reflection mechanism, denotes an object representing the type *T*.) The creation procedure *make* would then be applicable to an *ARRAY* [*T*].

The current solution to the problem of making arrays void-safe is clearly sound, and is generally acceptable, at least for new code. In practice it causes some unpleasantness in only one case: when there is no obvious default *value*, and the program's makeup guarantees that no array entry will be accessed without having been set. The most important example we know is the *HASH_TABLE* class of EiffelBase: an implementation of hash tables using an array controls the array entirely, and can guarantee the set-before-accessing property; but it requires a default value to placate the void-safe type system.

Even though the solution works, it is an example of the worst possible nightmare for a language designer: having to change the usage instructions for a fundamental mechanism — arrays in this case — used by every single program in existence.

8 Measuring the conversion effort

The recently completed library conversion process provides a basis for assessing the amount of work needed to adapt code. Out of thousands of converted classes, the following measures apply to 215 fundamental classes of Free ELKS (the Eiffel Library Kernel Standard, a subset of EiffelBase common to all implementations):

- Introduction of object tests: before conversion, the code included 100 occurrences of the assignment attempt instruction, the predecessor to object test. After conversion, it contains 115 object tests. All the added object tests are in contract elements, specifically postconditions and class invariants.
- Introduction of check instructions: there were 71 Check instructions; the number is now 90.

It is encouraging to note that no object test had to be added outside of contract elements. In the case of contracts, many of the changes were of the kind illustrated with /E9/ and /E11/; for example the insertion routine *put_left* from *LINKED_LIST* now has the postcondition clauses

```
previous /= Void
attached previous as q and then q.item = v
```

with the redundancy noted in section 4.3. Another example of an added object test is the following clause in the postcondition of *put_right*, now reading:

```
(old before) implies (attached active as c and then c.item = v)
```

(**old**, in a postcondition, denotes the value of an expression on entry to the routine). In this case the object test causes no redundancy; indeed the previous version of the class had the clause as just

```
(old before) implies (active.item = v)
```

but no clause stating that *active* /= **Void**. In the absence of a proof that *active* cannot be void on routine exit, this expression was not void-safe.

Altogether, the percentage of lines changed in EiffelBase between versions 6.1 and 6.4 is 11% (9093 out of 82,459). This is probably not a reliable indicator of future conversion efforts as the changes span three release cycles over an eighteen-month period, during which many non-void-safety-related changes also took place (but are not separately identified in the change record). The reason an analysis of EiffelBase changes requires going back that far is that EiffelBase served as a testbed for the progressive implementation of void safety, starting with 6.2. Since the library covers fundamental data structures, it makes extensive use of void references and is probably a worst-case example.

The EiffelVision multi-platform graphical library was converted in a single shot between versions 6.3 and 6.4, with almost no other changes. The figures may be more generally relevant since EiffelVision is more similar to application libraries. It is also much larger than EiffelBase. In EiffelVision the number of changed lines was 10,909 out of 376,592 — less than 3%.

The conversion efforts that have occurred until now have mostly been applied to libraries. To maintain compatibility for existing client code, the converted versions must declare all externally visible reference types as *detachable*: using attached types would make existing client code invalid whenever it passes actual arguments — possibly void — to the corresponding routines. This policy goes against the general recommendation of using attached types as much as possible. No such precaution is necessary for converting non-library code and for writing *new* code; this is one of the reasons why we expect that the change effort in the future will be less than the above figures.

While there is not enough concrete experience to offer a definitive estimate, the experience so far is encouraging. It suggests that the effort of adapting existing software to void safety is tractable, and that requiring void safety for new code will not impose an undue burden on programmers.

9 Assessment and conclusion

Initial user reactions to the void safety mechanism have not all been positive. Predictably, the need to convert existing code has caused some concerns among users. At the time of writing, it seems that these objections are subsiding; we hope that users will come to view the benefits of void safety as justifying the costs of conversion.

The preceding sections have illustrated the challenges of language evolution, discussed in more general terms in a previous contribution to a Hoare anniversary volume [9]. Research on language design often focuses on inventing language constructs and trying them out separately. Such experimentation is obviously useful; but it may miss the difficulties faced by language designers who are in charge of an existing language, having existing users and an existing code base.

The development of the void safety mechanism and its insertion into a full-fledged, industrially-used language are a reminder of some of these difficulties:

- Interaction between language features. Two of the major obstacles to void safety turned out to be multithreading and arrays. It was not clear to us, at the beginning of this effort, that they would even figure in the discussion. The original article [10] does not mention them; their importance became clear as we started applying the mechanism to the mass conversion of existing programs and libraries.
- Engineering considerations. The most clever language design ideas will fail unless they can be properly engineered into compilers, explained to programmers and retrofitted into existing designs.
- Compatibility and migration. Whenever possible, new mechanisms should remain compatible with existing ones. Failing this goal, a migration path should be devised, enabling users to adapt their code progressively. We have repeatedly found, in the evolution of Eiffel, that users are not adverse to change; but the change must be justified, and a clear path charted. This starts with providing compatibility options in compilers, which will process existing code unchanged, and continues with providing migration guides (or, when possible, conversion tools).

The process is not always perfect; our own experience with void safety has involved many trials and many errors. We hope that the results will be useful to others: not just the lessons of that experience but, concretely, the availability for the first time of guaranteed void safety in a mainstream object-oriented language.

10 References

[1] Mike Barnett, Rustan Leino and Wolfram Schulte: *The Spec# Programming System*; CASSIS 2004, Lecture Notes in Computer Science 3362, Springer-Verlag, 2004.

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

[3] Eiffel community: Void safety migration guide, at dev.eiffel.com/Void-Safe_Library_Status.

[4] 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.

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

[6] C.A.R. Hoare: *Null References: The Billion Dollar Mistake*, abstract of talk at QCon London, 9-12 March 2009, at qconlondon.com/london-2009/presentation/Null+References:+The+Billion+Dollar+Mistake.

[7] C.A.R. Hoare and Jay Misra: *Verified Software: Theories, Tools, Experiments, Vision of a Grand Challenge Project*, in VSTTE 2005, eds. B. Meyer and J. Woodcock, Lecture Notes in Computer Science 4171, Springer Verlag, 2008, pages 1-18.

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

[9] Bertrand Meyer: *Principles of Language Design and Evolution*, in *Millenial Perspectives in Computer Science* (Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare), eds. Jim Davies, Bill Roscoe and Jim Woodcok, Cornerstones of Computing, Palgrave, Basingstoke-New York, 2000, pages 229-246.

[10] Bertrand Meyer, *Attached Types and their Application to Three Open Problems of Object-Oriented Programming*, in ECOOP 2005 (Proceedings of European Conference on Object-Oriented Programming, Edinburgh, 25-29 July 2005), ed. Andrew Black, Lecture Notes in Computer Science 3586, Springer Verlag, 2005, pages 1-32.

[11] SCOOP concurrency mechanism, see references at se.ethz.ch/research/scoop.