Prelude to a Theory of Void

Bertrand Meyer

Draft 1, June 1998

ABSTRACT

A set of rules to ascertain that in calls of the form *x*.*f* (...) there will always be an object attached to *x*.

1 PURPOSE

The theory developed in this article investigates void calls, a dangerous source of bugs in object-oriented computation, and ways to avoid them.

About the scope of this article

The article was initially written as a short note presenting the results of sections 3 to 5, but it turned out that a proper exposition requires some background elements (section 2). Since one may want to prove many properties of object-oriented software besides avoidance of void calls, there is room for a series of follow-up articles focusing on proofs of specific properties. Such eventual articles will not need to repeat the background elements, which define a general scheme for the theory of object-oriented computation.

It is indeed part of the aim of this exposition to set a framework for other theories of specific aspects of object-oriented computation, the accumulation of which might eventually yield a semantic definition and proof system for a full O-O language.

This is a pre-publication draft of a paper published in JOOP in 1998. Citation reference: Bertrand Meyer, *prelude to a Theory of Void*, in *Journal of Object-Oriented Programming*, vol. 11, no. 7, November 1998, pages 36-48.

Void calls

The specific theory developed here addresses a practical problem well known to object-oriented application developers.

In a strongly typed object-oriented language few run-time errors remain possible once the compiler has accepted a system. One of the most unpleasant is a run-time attempt to execute a call of the form *reference*. *feature* (*arguments*), intended to apply a *feature* to an object but unable to do so because the *reference* turns out to be "void" (or "null"), that is to say, not attached to any object. Such a *void call* will fail, often leading to abnormal termination of the entire execution.

This is a frustrating event and it would be a great benefit to developers if it were possible for compilers, in the same way that they detect type mismatches, to reject any programs that may cause a void call during execution. This article describes a set of rules, enforceable by compilers, which can help prevent void calls.

Trusted components

The work presented here is part of the Trusted Components Project, a general effort initiated by Monash University and Interactive Software Engineering with the aim of producing certifiably reliable library components to serve as basis for building high-quality appplications. The project is by nature collaborative; any institution or individual that shares its goals is welcome to join.

One part of this effort, although by no means the only one, is to investigate how much one can hope to prove, mathematically, the correctness of object-oriented software components.

Because the general problem of program proving is difficult, it is desirable to take smaller steps first. Any help in proving specific reliability properties of programs should be welcome to programmers, especially if it takes the form of rules that can be enforced by compilers. Preventing void calls is an attractive candidate, since the practical benefits are high, as any object-oriented software developer can testify, and the difficulty of implementing a solution should be, thanks to the theory developed here, much less than if the aim was to prove arbitrary correctness properties of object-oriented systems.

References and how they can lead to void calls

The basic operation performed at run-time by an object-oriented system is feature call. Also called "message passing" and "method call", this computational step has the general form

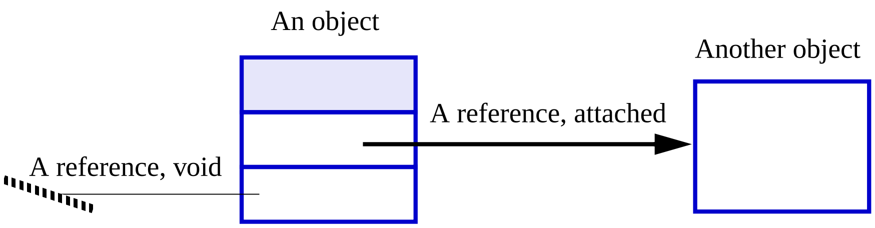
target. *feature* (*arguments*)

where *target* must represent an object and *feature* a feature of its generating class.

In a statically typed language and particularly in Eiffel, type checking rules will allow the compiler, before it accepts the text of a system containing such a call, to determine that for all possible run-time executions of this call *feature* will be compatible with the type of *target* and *arguments*. If not, the compiler will reject the submitted system.

Assuming the call passes the scrutiny of the type checker, its semantics simply is: apply *feature* to the object attached to *target*, passing to it the values of *arguments*.

In practice, the association between the name of *target*, as it appears statically in the program text, and the corresponding run-time objects, is often indirect: what *target* actually denotes is not an object but an object **reference**. A reference is either *attached*, in which case it gives access to an object, or *void*, as illustrated on the following figure.



Not all variables are references. In Eiffel one can declare a variable to be of an "expanded type", in which case its possible values are objects rather than references, and the problems discussed in this article do not arise. Expanded types are used in particular for basic values such as *INTEGER* for which references are generally not needed. Other object-oriented languages have their own mechanisms to give programmers a similar choice.

Regardless of the default convention, it would be hard to do without reference types, because of the flexibility they provide:

- References may be void. This convention is particularly useful for describing complex data structures: in a linked list, every element but the last one will have a reference to its right neighbor, and in the last object that reference will be void.
- This convention also provides an obvious default initialization value — *Void* — for all entities of non-basic types. This is the rule applied in Eiffel, enabling every object field and every variable to have a language-defined initial value so that there are no run-time "uninitialized value" error or implementation-dependent behavior.
- A reference is not tied forever to one object but may, during execution, be *reattached* to different successive objects. This is again necessary for implementing many data structures; for example one may want in an implementation of trees to re-parent a tree, making it a subtree of some other node; this is most easily done by ensuring that every node object has a reference to its parent, and reattaching that reference when needed.

For these reasons references are almost universally available in object-oriented languages, as they already were in pre-object-oriented languages such as Lisp, Algol W, Pascal and Ada that support the manipulation non-trivial data structures.

But the introduction of references also has some unfortunate consequences. If *target* may be void we cannot any more guarantee, even in in a strongly typed language, that the feature call makes sense, since it can only be executed when *target* denotes an object.

The effect of a void call

A void call cannot be carried out; in Eiffel it will cause an exception.

Although it is possible to recover from such an exception by writing an appropriate exception handler, this is only a solution of last resort, since if the programmer detects that the exception may occur a simpler and more direct remedy is to avoid the exception in the first place by modifying the software to ensure that *target* will never be void.

The possibility of handling a void call through an exception handler remains useful, but only as a "just in case" measure, when the programmer, although unable to pinpoint a specific source of void calls, fears that because of its complexity the software might still cause void calls in unknown cases that have escaped testing, and wants to provide a general-purpose rescue mechanism to limit the damage in such a case. Although preferable to ignoring the problem altogether, this approach is only a palliative.

Avoiding void calls

A better approach would be to let the compiler detect and reject any software element that can cause void calls. In simple cases this is possible. For example the GNU Eiffel compiler — which currently does not support Eiffel's exception handling, and hence needs, perhaps more than other compilers, to provide some help to users in this area — will flag a routine declaration of the form

```
bad_routine is
    local
        will_be_void: SOME_REFERENCE_TYPE
    do
        will_be_void.some_feature
            -- The preceding line will always cause a void call,
            -- since the language rules ensure that will_be_void
            -- is initialized to Void on every call.
    end
```

But other cases may escape detection. For example the following scheme, although not uncommon in practice, may in principle cause a void call:

```
suspicious_routine is
        local
                might_be_void: SOME_REFERENCE_TYPE
        do
                if some_condition then
                        create might_be_void
                                -- Creation instruction; ensures that might_be_void
                                -- is not void. Also written !! might_be_void.
                end
                ... Other instructions ...
                if some_other_condition then
                        might_be_void . some_feature
                end
        end
```

For some forms of *some_other_condition* void calls are indeed possible; in other cases, for example if *some_other_condition* is the boolean expression

yet_another_condition **and** *might_be_void /= Void*

then no void call will ever occur. But verifying this property, and distinguishing between safe and potentially unsafe cases, is beyond the abilities of current compilers.

The rest of this article presents a theory which compiler writers can use to include void-call prevention measures in their compilers, and application programmers to reason about void-related properties of their software.

2 THEORETICAL FRAMEWORK

The following paragraphs present the assumptions behind the theory.

Void-safe software elements

We say that a software element is **void-safe** if its execution will not produce any void call. The aim of the theory is to define conditions under which software elements will be provably void-safe.

Instructions and programs

The syntactic domain of interest is *Instruction*. The theory will define a boolean-valued function *precondition* applicable to elements of that domain, with the intention that if the function has value True for an instruction this indicates that the instruction is void-safe — its execution will not cause any void calls.

The definition of function *precondition* will rely on some auxiliary functions.

Instruction has a number of subsets, corresponding to the different kinds of instruction, such as *Call*, *Assignment*, *Creation*. The functions of interest, such as

Precondition and its auxiliaries, will be defined by case analysis: separate definitions will be applicable to arguments of each of the identified subsets.

We can consider an entire program (*system* in Eiffel terminology) to be a kind of instruction. The value of *precondition* applied to an element of the corresponding set *System* will determine whether the system as a whole is void-safe — whether it is guaranteed not to produce any void call at any time during its execution.

Mathematical basis and notation

The mathematics of the Theory of Void only relies on elementary set theory and logic. To express it, this article uses EFL (Expressive Formal Language), a simple notation for expressing mathematical descriptions. EFL will be presented in a forthcoming paper but should not raise any difficulty, since it is just elementary set theory couched in a form suitable for discussions of software engineering and programming language topics. This article uses only a handful of EFL constructs (the full notation does not have many more anyway); each will be explained when first introduced.

The syntax of EFL resembles that of programming languages rather than traditional mathematical notation; this facilitates exchange of specification elements by e-mail, and helps build large specifications incrementally from small pieces, supporting (like a good object-oriented programming language) the process of progressive extension, adaptation, modification and specialization. But one should not be misled by the external appearance of specifications: EFL describes purely *mathematical* objects — all of them sets — and their properties.

Accordingly, the results of this article are purely mathematical, and independent of the choice of notation. EFL is only a means of exposition.

Assertions

The principal semantic domain is the set *Assertion*. In a general program-proving context we would probably treat assertions as boolean-valued functions, but for the purposes of this discussion all that matters is our ability to guarantee that some references are not void. So an element of *Assertion* will simply be, for the moment, a set of variables, denoting the software property that all these variables' values are non-void.

```
Assertion
                -- Properties of computational states, limited for the moment
                -- to stating that certain references are not void
        has
                nonvoid: part Expression
        end
```

(A **has** clause describes the components of a mathematical object, as with a record or structure type in programming. This is like defining a set as a cartesian product, but with the advantage that the specification remains open since one can add new **has** components later as one discovers new properties. **part** *A* is the set of subsets of *A*, also known as the powerset of *A*. So here the specification states that an element of *Assertion* contains a set of *Assertions*. As in Ada and Eiffel, EFL comments are signaled by two hyphens: --.)

Expression is a syntactic domain, representing the set of expressions in the object-oriented programming language. The specification of *Assertion* uses *Expression* rather than *Variable* to allow expressing not just properties of the form

"*x* is not void"

but also (in terms of the notation of an object-oriented programming language):

"*a.b* is not void"

The latter case is necessary because if we consider — referring again to a programming language notation — the call

x.f

where the postcondition of routine *r* includes

a /= Void

for some attribute *a* of the corresponding class, then the interesting inference for the caller is

x.a /= Void

which we can formally express by including the expression *x.a* in the assertion obtained as postcondition to the call.

The functions of interest

For any Instruction we are interested in two principal functions: *precondition*, already mentioned, and *postcondition*. Here are their signatures:

precondition: *Instruction —> Assertion —> Boolean*

postcondition: Instruction —|> Assertion —> Assertion

—> indicates a total function; —|> indicates a possibly partial function, of which we must specify the domain. Arrows associate from the right, so that the first of these signatures means *Instruction —>* (*Assertion —> Boolean*). As a general rule it appears more convenient to curry the functions than to use cartesian products for arguments.

The intended meaning of *postcondition* is that if instruction *i* executes successfully with *a* as its input assertion then *postcondition* (*i*) (*a*) is the resulting assertion. In other words, if *a* is the set of references known to be non-void upon starting *i*, then *postcondition* (*i*) (*a*) is the set of references known to be non-void upon finishing.

The intended meaning of *precondition* is simply to define the domain of the partial function *postcondition*. In other words, to state that *precondition* (*i*) (*a*) is to state that *i* will execute successfully if started while no member of *a* is a void reference. For the present Theory of Void the notion of "executing successfully" is simply the property of not producing any void calls. If this is the case we say — as already noted — that *i* is *void-safe* for *a*.

The aim of the game

The preceding definitions set the aim of the game in the rest of the discussion:

1 • If we take an entire program to be an instruction *p*, then the property to prove is that *p* is void-safe for the empty assertion, that is to say: *precondition* (*p*) (*Empty*) = *True*. This expresses that *p* started from scratch will not cause any void calls.

2 • Computing *precondition* (*p*) requires having a definition of the function. So sections 3 to 5 of this article define *precondition* by case analysis, considering each kind of instruction in turn.

3 • As part of these definitions, the value of *precondition* (*i*) (*a*), if *i* represents the object-oriented call *x*.*f*, will be the condition that *x* is part of *a*. The definition for compound instructions will ensure that if a program *p* is void-safe then, for any call *x*.*f* that any of its instruction executes, *x* will be non-void at the time of the call — the general property that we seek to establish in this article.

Point 1 indicates that in the end the only function of interest is *precondition*. But we also need to define *postcondition* for each kind of instruction, as the definitions for *precondition* will involve the value of *postcondition*; the two function definitions will proceed hand in hand.

Soundness and completeness

Ideally the Theory of Void should be sound and complete according to the following definitions:

The theory is **sound** if it is sufficient (powerful enough): whenever *precondition* (*i*) (*a*) holds, *i* is void-safe for *a*, that is to say, execution of *i* under the input condition *a* will never cause a void call.

The theory is **complete** if it is necessary (not too pessimistic): whenever *precondition* (*i*) (*a*) does not hold, *i* is not void-safe for *a*, that is to say, some executions of *i* under the input condition *a* can cause a void call.

In practice, however, it seems possible to do with a theory that is neither complete nor even (shocking at this might appear at first) sound:

• If the theory is not complete, some potential void-call-prone situations will escape the scrutiny of our tools. Although it is desirable to catch as many such situations as possible, catching *any* of them is preferable to the current situation in which compilers detect *no* void calls except possibly for trivial cases. Although not complete in the state described by the article, the theory already rules out many potentially damaging constructions.

- If the theory is not sound, it may raise some false alarms. For example it might produce an error message for a scheme such as the following, assuming that the function *abs* returns the absolute value of its argument.

```
if n >= 0 then
    create might_be_void
end
if n = abs (n) then
    might_be_void.some_feature
end
```

  Here *might_be_void* will never be void in the call, since a positive number is always equal to its absolute value. For a tool to know this, however, it would need to have extensive theorem-proving and program-proving capabilities beyond today's compiler technology, and rely on a complete formalization of the programming language's semantics, far more ambitious than the limited Theory of Void developed in this article. So a compiler relying on the present theory will signal a possible error even though the program fragment is in fact safe. It will be the programmer's responsibility to determine that this is a false alarm and to make the corresponding decision: either ignore the warning or (perhaps preferably) change the form of the extract in a way that will pacify the tool. (This does not necessarily mean that error reports should be mere warnings. As will be seen below, it is possible to let a compiler issue fatal errors even if the theory seems not fully sound. See "The Check instruction", page 22.)

Another argument for accepting an imperfect theory, with respect to both soundness and completeness, is that to guarantee either of these two properties would require that we provide a full formal semantics for the underlying programming language, and hence that we *choose* one language as the object of our theory. By avoiding such a choice the theory developed here potentially applies to any object-oriented language (and even to some non-O-O ones), although the model it uses most closely fits Eiffel.

The principal motive remains pragmatism and incrementality. By not initially aiming at full soundness and completeness, we can achieve our practical aim of helping compiler writers *now* to produce tools that will enable programmers to detect and correct potential void calls. As we improve the theory — by covering more and more instructions, and getting closer to full coverage of specific programming languages — this help will be more and more precious. But even partial help, such as provided by the theory in its current initial state, is invaluable: if it avoids just one void call in a mission-critical application, the theory will have been worth developing.

For the long term we should of course retain the goal of providing a sound and complete Theory of Void for the programming language of interest.

Dynamic aliasing

An assertion was defined above as a set of expressions known to be non-void. It may be useful to extend this notion by also including a set of expression pairs denoting references known to be attached to the same object:

```
Assertion
    has
        equal: part Expression_pair
    end
```

(to be added to the previous **has** specification for *Assertion*, which read, on page 6, *Assertion* **has** *nonvoid*: **part** *Expression* **end**), with

```
Expression_pair
    has
        first, second: Expression
    end
```

When specifying the effect of a reference assignment *x* := *y*, we will not only consider its effect on the *nonvoid* set (if *y* was in *nonvoid*, *x* will now be in it, and if *y* was not in it but *x* was, *x* must be removed): we will also add the pair <*x*, *y*> to the *equal* set of the assertion. See "The Procedure_call instruction", page 24. This will improve the completeness of the theory by enabling us to know more about our software; in particular if the expression pair <*x*, *y*> is in *equal* and the theory tells us, from the properties of a routine *r*, that after the call *x*.*r* the value of *x*.*comp* is not void, then we can infer that *x*.*comp* is also not void. Reference equality is of course an equivalence relation, but we don't necessarily have to ensure that whenever <*a*, *b*> is in an *equal* set the pair <*b*, *a*> is in it too; we can just keep one pair and make sure that all functions on pairs treat the two elements symmetrically. This is the mathematical equivalent of what software engineers call "just an implementation decision".

A notational aside about EFL conventions. The **has** clauses accumulate; so *Assertion* as defined so far may be viewed as equivalent to the cartesian product

(**part** *Expression*) x (**part** *Expression_pair*)

Without the **has** mechanism, however, we might initially have defined the set *Assertion* as being the set **part** *Expression*, and later realized that we also need the *equal* component, making that initial definition obsolete and forcing us to revisit and adapt every part of the specification that referred to *Assertion*. Instead, EFL supports the incremental construction of specifications: we never define a new set as some combination of known sets but specify it step by step by listing, through **has** clause, the various components making up its elements. For *Assertion* the components are *nonvoid* (a member of the set *Expression*) and *equal* (a member of the set *Expression_pair*, itself specified in the same style). We can always add **has** clauses later, as we discover new components of members of our sets. Until the last moment we do not assume that the list of components is exhaustiv; when we are ready to "release" a specification we implicitly identify each set to the cartesian product of its components. This incremental approach is crucial to the practical effort of writing formal specifications of large and complex phenomena. It should be pointed out again, however, that these conventions are just a way of arriving at the result, and that this result — the final specification — is fully equivalent to a traditional one expressed through cartesian product or a similar technique.

Affected targets

We will see in the specification of routine calls ("Retaining unaffected properties", page 25) that it is useful, for practical proofs, to add one more component to assertions: affected variables. The new component is

```
Assertion
    has
        affected: part Expression
    end
```

and stands for the property that the computation so far may only have changed the expressions in *affected*. For example if *a* . *affected* is the set {*x*, y, *z*}, this means that no variable other than *x*, y and *z* has been touched by the computation.

To avoid any confusion, here is the full definition of *Assertion* reconstructed from the three components introduced separately so far (EFL tools could produce such a reconstruction from elements originally entered piecewise as here):

```
Assertion
    has
        nonvoid: part Expression
        equal: part Expression_pair
        affected: part Expression
    end
```

Routines

One final general definition is required before we start the definition of functions *precondition* and *postcondition* for the various kinds of instruction. Since relevant programming languages will have a notion of routine call, we need to define the syntactic domain *Routine*. Selecting Eiffel routines as our model, we can use

```
Routine
        -- Subprograms
    has
        pre, post: Assertion
            -- Ignore arguments for the moment!
        body: Instruction
    end
```

It will be central to the proof technique of the Theory of Void to assume that each routine is equipped with two assertions, any of which can of course be empty. The following three considerations justify this decision:

- Without the *pre* and *post* components of a routine we would not be able to prove anything about calls except by "unfolding" the proof of every routine of a system. This is not only impractical — requiring huge proofs — but contrary to the abstraction principles of object technology.
- Without the ability to rely on explicitly stated preconditions and postconditions of routines, we could not prove anything useful about a system unless we have access to the source code of *every single routine* it uses. As soon as it relies on just one compiled library routine we would be helpless. This is of course unacceptable, as it excludes all realistic software systems. With explict preconditions and postconditions we can proceed: whenever we use a library routine we should do so on the basis of a contract specification — a **short form** as offered by Eiffel environments and used as basic documentation for Eiffel libraries — which includes the relevant precondition and postcondition, even for a routine whose source is not available to us.

By equipping routines with assertions we can apply to proofs the abstraction and reuse techniques that make object-oriented programming possible.

Eventually we should of course prove that the body of every routine is consistent with its *pre* and *post*. This task is indeed the "unfolding" noted above. It is a supplier-side task (the responsibility of the library authors), separate from the client-side responsibility of proving the conditional correctness of an application (conditional on the library's correctness). This separation preserves abstraction and reusability.

Note that one should not confuse *pre* and *post* with the functions *precondition* and *postcondition*. The former are syntactical components: every routine will include, as part of its text, two assertions expressing its intended precondition and postcondition. The latter are semantic functions, which enable us (or a compiler) to compute properties of a program.

As stated in a comment, the definition of *Routine* does not account for routine arguments. This is part of what make the current discussion a "prelude" to the Theory of Void. There seems to be, however, no particular difficulty in adding arguments (at least for a language such as Eiffel where argument passing follows a simple semantic model);

it will be a matter of adding some substitution functions to the function specifications of section 4.

Routines include procedures and functions:

```
Procedure
        -- Routines that do not return a result
    subset
        Routine
    end
        -- No further properties
Function
        -- Routines that returns a result
    subset
        Routine
    has
        function_result: Variable
    end
```

These definitions use the EFL notion of **subset**, which serves to introduce a new set in terms of another, in a manner similar to inheritance in object-oriented programming, although mathematically this is a simple concept of subsetting. With **subset** the new set retains the components of the reference set (here *Routine*), while adding, if necessary, further components, as done here in the case of *Function*. Similarly, each of the construct specifications of sections 3 to 5 defines an individual instruction as a **subset** of *Instruction*.

A mathematical note

(This section is a comment on EFL and not indispensable to an understanding of the article.) For the reader who may have doubts about the mathematical nature of the **subset** relation, in particular whether it truly defines a subset, the following clarifications of the EFL model will help. The most convenient model for a definition of a set through its components, as in

```
Person
        has
            name: String
            age: Integer
        end
```

is *not* the first idea that comes to mind — that *Person* is the cartesian product *String* x *Integer*. The reason, among others, is that this would prevent the kind of subsetting in which are interested: if we add a new component, say *height* of type *REAL*, either through a new **has** clause (in the incremental style praised earlier) or through a **subset** definition for a new set *Person_with_height*, we would not be introducing a mathematical subset, since *A* x *B* x *C* is not isomorphic to a subset of *A* x *B*. In fact, the subset relation goes the wrong way (we can map *A* x *B* to a subset of *A* x *B* x *C*).

A more appropriate model is **partial functions**: we will understand the definition of *Person* as denoting the set of partial functions from *Tag* to *Value* whose domain includes both *name* and *age*, with the value for *name* being in *String* and the value for *age* being in *Integer*. Here *Tag* is the set of all possible tags, such as *name* and *age*, and *Value* is the set of all possible values, including strings, integers and reals.

With this definition it is indeed true that *Person_with_height* is a subset of *Person*: any element of *Person_with_height* has the tags *name*, *age* and *height* in its domain, so it is indeed an element of *Person* as well. Note the importance of stating, in the definition, that the domain *includes* the appropriate tags — not that the domain *is* the set of the tags considered (such as *name* and *age*), which would prevent subsetting.

This model is part of EFL's concern for the "engineering" of specifications — similar to the engineering of software systems thanks to object technology — for a smooth refinement process, incremental and evolutionary.

Note that once we have finished this refinement process we can, if we like, start thinking again in terms of cartesian product: if the set of tags is fixed, for example *name*, *age* and *height*, there is a trivial one-to-one correspondence between the set of partial functions with values in *String* for *Name* and in *Integer* for *age* and *height*, and the cartesian product *String* x *Integer* x *Integer*. So once again the result can be recast in entirely traditional terms.

Implication

In a number of cases we will need the ability to express that an assertion implies another, in a sense compatible with the usual definition of implication in logic. The following function definition expresses implication for our model of assertions:

```
infix "=>" (strong, weak: Assertion): Boolean is
        -- An auxiliary function: does strong imply weak?
    do
        Result := ((strong . nonvoid superset weak . nonvoid)
            and (strong . equal superset weak . equal))
            and (strong . affected superset weak . affected))
            -- In other words: if weak states that a reference is not void,
            -- so must strong; if weak specifies that two references
            -- are equal, so must strong; if weak states that a reference
            -- have been modified, so must strong.
    end
```

superset is the set inclusion operator, "is a superset of".

A note for readers who might at first think that the **superset** relations, in this definition of "=>", go the wrong way. When modeling semantics through program states we are indeed used to thinking that "*strong* implies *weak*" means "*strong* is a subset of *weak*" in the sense that any state that satisfies *strong* satisfies *weak*. But here we model an assertion as three sets of objects known in each case to satisfy a certain property —

being non-void, being equal, or being affected by the computation so far. So the subset relations go the other way: to say that *strong* implies *weak* is to say that whenever *weak* tells us that a certain object satisfies one of these properties, the object must satisfy that property in *strong*; in other words, that the objects known to satisfy the property in *strong* must constitute a superset of the objects known to satisfy that property in *weak*.

3 PROVING VOID-SAFETY: CONTROL STRUCTURES

We can now proceed to the relevant specifications for the variants of *Instruction*. The section for each variant follows the same pattern:

- Define the syntax of the corresponding construct in the form of the definition of a certain set, such as *Assignment*, in the form of a **subset** of *Instruction*. The syntax of interest is of course abstract syntax, since we are not concerned about the concrete notational conventions of a particular programming language.
- Define the values of functions *precondition* and *postcondition* for elements of the given set.

The set *Instruction* is never defined explicitly. The principle here is the same as that through which, at "release" time, we equate a set defined through one or more **has** clauses with the combination of its components. Similarly, if nothing is known about a set except for one or more **subset** definitions, we equate it in the end with the union of all its defined subsets.

Correspondingly, there is no explicit definition for functions *precondition* and *postcondition* for a general *Instruction*; the general definition will simply be the union of all the specific definitions for the listed subsets of *Instruction*.

This section addresses control structures and the Null instruction. The next two sections cover reference-modifying instructions (creation and assignment) and assertion-equipped instructions (call, loop, **check**).

The Null instruction

The first construct of interest is the null instruction, with the following definition:

```
Null
          -- The empty instruction
     subset
          Instruction
     end
```

Here are the corresponding function values:

```
precondition (n: Null) (a: Assertion): Boolean is
        -- Always OK
    do
        Result := True
    end
postcondition (n: Null) (a: Assertion): Assertion is
        -- No change to result
    do
        Result := a
    end
```

Result denotes the result of a function. The symbol := means "is defined as". *Empty* is the empty set. Again, these EFL notations, although resembling those of programming languages, denote purely mathematical concepts.

The definition of *Precondition* states that the *Null* instruction is always possible, regardless of the input assertion *a*, since the precondition is *True*.

The definition of *Postcondition* states that the *Null* instruction lives up to its name: it yields an output assertion identical to its input assertion — changing nothing to known information about which references are non-void.

The Compound instruction

A *Compound* instruction is made of one or more instructions, to be executed in sequence. To avoid distracting ourselves with a notation for sequences (such as the one in EFL), and keep the notational apparatus to the bare minimum, we can simply consider that a Compound has a "first" and a "rest":

```
Compound
        -- Instructions made of one or more instructions
        -- to be executed in sequence
    subset
        Instruction
    has
        first, rest: Instruction
    end
```

(A notational observation: this definition sounds suspicious at first since it is implicitly recursive. *Instruction*, as noted above, will be defined as the union of its defined subsets; here one of the subset definitions uses *Instruction* — twice. The difficulty is easily resolved thanks to the following rule. In EFL, all objects of discourse are sets, and the only operators used in the definition of a set are those of set theory, such as intersection and union. If a definition is recursive, defining *A* as *op* (*A*) where *op* stands for some combination of set operators, it is simply understood as a shorthand for the fixpoint definition

```
Empty union op (Empty) union op (op (Empty)) union ...
```

where **union** is set union. This immediately generalizes to sets of mutually recursive definitions.)

The precondition of a compound is defined as follows:

```
precondition (c: Compound) (a: Assertion): Boolean is
          -- The first instruction must be OK, and so must be the rest
          -- of the compound when executed in the resulting state
     do
          Result :=
               precondition (c.first, s) and
               precondition (c.rest, postcondition (c.first, s))
     end
```

This definition shows why we need, in the general case, to explore not just function *precondition* but also *postcondition*: the former relies on the latter. Here is *postcondition* in this case:

```
postcondition (c: Compound) (a: Assertion): Assertion is
          -- Result of executing rest of compound from state
          -- resulting from executing first instruction
     do
          Result := postcondition (c . rest, postcondition (c.first, a))
     end
```

The conditional instruction

We consider a simple form of conditional instruction, with abstract syntax

```
Conditional
          -- Instructions made of one or instruction (the body)
          -- to be executed only if a certain condition is satisfied
     subset
          Instruction
     has
          body: Instruction
          condition: Assertion
     end
```

This represents instructions of the form

```
if condition then
     body
end
```

where, in *condition*, we are only interested in clauses corresponding to our assertions, such as *x /= Void*. Then we have:

```
precondition (c: Conditional) (a: Assertion): Boolean is
        -- The body must be OK if the condition is satisfied
        -- (what happens otherwise doesn't matter)
    do
        Result := (a => c.condition)
    end
```

"=>" is the implication function defined earlier ("Implication", page 14).

4 CREATION AND ASSIGNMENT

The two constructs that directly affect references and void-safety are object creation, which makes a reference non-void, and reference assignment, which changes the value of a reference.

The Creation instruction

Syntax:

```
Creation
        -- Object creation
    subset
        Instruction
    has
        target Variable
    end
```

This definition ignore creation procedures ("constructors"). Here are the three function definitions:

```
precondition (c: Creation) (a: Assertion): Boolean is
        -- Always OK
    do
        Result := True
    end
postcondition (c: Creation) (a: Assertion): Assertion is
        -- Add target of creation instruction to set of references known
        -- to be non-void; remove it from any equality set
    do
        Result.nonvoid := trim (a.nonvoid, c.target) union {c.target}
        Result.equal := remove (a.equal, c.target)
        Result.affected := a.affected union {c.target}
    end
```

This uses two auxiliary functions *trim* and *remove*:

- If *es* is a set of expression and *v* is a variable, *trim* (*es*, *v*) is *es* deprived of any element of the form *v.something*.

- If *ps* is a set of pairs then *remove* (*ps, v*) is another, obtained from *ps* by removing any pair of which one of the two elements. is either *v* or of the form *v*▪ *something*.

These functions have not been spelled out but are easy to write in EFL or any other appropriate mathematical notation.

The notation {*a*, *b*, *c*, ...} defines a set by enumeration, here with just one element, *c* ▪ *target*. Another notational convention: the *Result* of a function can be defined through the values of its components, here *equal* and *nonvoid*. (In the incremental spirit of EFL specifications, there may be, for the same function, more than one definition similar to the above, each including only some of the components in the **do** clause. We may posit support tools with the ability to piece together all such partial definitions for a given function, and to check that the result covers all the components. Again none of this affects the mathematical significance of the final definitions.)

The definition of *precondition* expresses that creation will not cause any void calls. The definition of *postcondition* states the effect of a creation on our knowledge of reference properties:

- The target of the creation will be added to the *nonvoid* set, since making the corresponding reference non-void is the immediate effect of creation.
- Any pair containing the target must be removed from the *equal* set, since the creation instruction obsoletes any information we may have had about the target being equal to something else.

The Assignment instruction

The assignment instruction may cause “guilt by association” — references becoming potentially void through the assignment of a void expression to a variable. We can model the syntax as:

```
Assignment
        -- Object creation
    subset
        Instruction
    has
        target: Variable
        source: Expression
    end
```

An assignment will not cause any void call:

```
precondition (s: Assignment) (a: Assertion): Boolean is
        -- Always OK
    do
        Result := True
    end
```

The postcondition is less trivial:

```
    postcondition (s: Assignment) (a: Assertion): Assertion is
            -- Adapt the assertion by adding the source's
            -- properties transposed to the target
        do
            Result.nonvoid := adapt (a.nonvoid, s.target, s.source)
            Result.equal := add_pair (a.equal, s.target, s.source)
            Result.affected := a.affected union {c.target}
        end
```

using two auxiliary functions, *adapt* and *add_pair*. The function *adapt* (*es*: **part** *Expression* , *v*: *Variable*, *e*: *Expression*): **part** *Expression* yields the set of expressions obtained from *es* by applying the following changes (in order):

- Remove *v* if present.

- Remove any element of the form *v.something*.

- If *es* contains *e*, add *v*.

- If *es* contains any element of the form *e.something*, add *v.something*.

Similarly, *add_pair* (*ps*: **part** *Expression_pair*, *v*: *Variable*, *e*: *Expression*): **part** *Expression_pair* yields a set of expression pairs obtained from *ps* by adding the pair <*v*, *e*>, as well as all pairs of the form <*v.something*, *e.something*> for which *v.something* is in *ps*. As before, defining these two functions is straightforward.

5 CALLS AND ASSERTION-EQUIPPED CONSTRUCTS

In the last set of instructions we will encounter constructs that may have assertions as part of their syntax. They include the *Check* instruction, loops (with their invariants), and routine calls.

Pragmatic considerations

In defining the properties of assertion-equipped constructs, we may divide proof rules into two categories. Theoretically both are equally important, but for the practice of object-oriented development, with its emphasis on abstraction and reusability, we may have to treat them separately.

Consider a routine call *target.routine* (...) where *routine* has a precondition and a postcondition (Eiffel syntax):

```
routine (...) is
    require
        pre
    do
            -- Any number of instructions:
        body
    ensure
        post
    end
```

The rules of our theory should tell us (as they indeed do, in the form given below) that the precondition for the call instruction is *target*.*pre*, guaranteeing on completion of the call the postcondition *target*.*post*. The rule enabling us to deduce this property of the call from the preconditions and postconditions declared for the routine, pre and post, belongs to the first category of proof rules, which we may call **external** rules.

External rules enable us to prove properties of client modules, such as a module containing the call *target*.*routine* (...).

Such proofs are of course only valid if the assertions given for the routine are consistent with what the routine does — *body*. This property will be expressed by a function called *consistent* and defined below. For every routine *r*, we must prove that *consistent* (*r*.*body*) (*r*.*pre*, *r*.*post*) holds. Such properties constitute the second category, which we may call **internal** properties.

The object-oriented method’s emphasis on abstraction and layered architectures suggest that it is usually a good idea to prove internal and external properties separately: if we are using a routine from another cluster of the software, or a library cluster, we assume its properties, as expressed by its *pre* and *post*, to prove our own client software; the author of the routine’s class is responsible for proving its internal properties.

Sometimes we may not even have the option of proving the internal properties, for example if the source code of a library we use is not available. Then we can only hope that the library authors have taken care of the internal proofs; but we should still do the external proofs of our calls, which become *conditional* proofs of our software (conditional on the correctness of its supplier classes).

A similar issue arises when we think we know for sure the truth of a certain property that our theory is not powerful enough to prove. An example was given earlier (“Soundness and completeness”, page 8):

```
    if n > 0 then
        create might_be_void
    end
    if n = abs (n) then
        might_be_void.some_feature
    end
```

Here the theory may not be able to prove that *might_be_void* can in fact never be void. This may break down an entire proof. In order to be able to proceed, a workaround is to add, just before the last call, an instruction (imitated from Eiffel) of the form

```
check
    might_be_void /= Void
end
```

whose postcondition, as seen next, is the given assertion, here *might_be_void /= Void*, enabling us to proceed with the proof of the enclosing software element. Theoretically there is no cheating, since we still have to verify the precondition of the **check** instruction, requiring us in fact to establish that *might_be_void /= Void* holds. We will consider that such preconditions are **internal** properties, to be proved separately. If the theory doesn't enable such proofs but we might simply rely on our intimate conviction that the property is holds — thus indeed cheating.

A good supporting environment might keep track of what internal properties we haven't proved, reminding us of how much we have cheated.

The distinction between internal and external properties is not theoretical but simply mercenary — it enables us to concentrate on the part of the job that we have the best chance of completing, even if we have to sacrifice some soundness and completeness.

In the following rules, the internal components appear in red, as a reminder that they represent properties that may have to be proved separately, or even (in the case of some **check** constructs and library routines) taken at face value rather than actually proved.

The Check instruction

Eiffel has an instruction of the form

```
check some_assertion end
```

similar to the **assert** instruction of Algol W and C. Such an instruction can be understood, for the purposes of present discussion, to produce no effect at all if *some_assertion* happens to hold at the time of execution, and otherwise to produce an outcome so horrendous that we dare not even think about specifying it. Here are the syntax, precondition and postcondition.

```
Check
        -- Assertion verification
    subset
        Instruction
    has
        assumption: Assertion
    end
```

```
precondition (c: Check) (a: Assertion): Boolean is
                -- Possible only if the assumption holds under a
        do
                Result := (a => c • assumption)
                        -- Reminder: function "=>" is assertion implication
        end
postcondition (c: Check) (a: Assertion): Assertion is
                -- In normal cases, the instruction has no effect.
        do
                Result := a
        end
```

The definition of the precondition is highlighted in red to indicate that, as noted, it may have to be ascertained separately or outside of the theory.

Instruction consistency

Consider an instruction *i*, an input assertion *p* and an output assertion *q*, The following auxiliary function expresses that *i*, started in an initial state satisfying *p*, will produce a state satisfying *q*:

```
consistent (i: Instruction) (p, q: Assertion): Boolean is
                -- Will i, started with p, yield q?
        do
                Result := precondition (i) (p) and ( postcondition (i) (p) => q)
        end
```

The definition states that: *p* must satisfy the conditions needed to execute *i*; and the postcondition of executing *i*, starting with *p* satisfied, implies *q*.

This function will be useful in discussing the semantics of routines.

Transposing an assertion to the context of the target

Consider a routine equipped with assertions applying to its attributes, such as the following in Eiffel notation:

```
my_procedure is
        require
                some_attribute /= Void
        do
                ... Whatever ...
        ensure
                other_attribute /= Void
        end
```

If we study the behavior of a call of the form *my_target* • *my_procedure* relative to a certain assertion that involves properties (non-void, equal) of the references involved, we must transpose the routine's own assertions to the context of *my_target*. For example the precondition means, in the context of the call, the property that *my_target*

. some_procedure is not void. If, as will be done next, we study the value of *precondition* (*pc*) (*a*) where *pc* is the given call, using *my_target* as target, and *a* is some assertion, we will want to make sure that the set *a . nonvoid* includes *my_target . some_attribute* (not just *some_attribute*, which makes sense in the context of the routine but not in the context of the caller).

This prompts us to define a function

infix "+++" (*a*: *Assertion*; *v*: *Variable*): *Assertion*

whose result is obtained from *a* by replacing, in each of *a . nonvoid*, *a . equal* and *a . affected*, every expression *e* by *v . e*. In other words we are transposing everything to the context of a call having *v* as its target. Again, the formal definition of "+++" is straightforward and not included here.

The Procedure_call instruction

It was mentioned at the beginning of this discussion that feature call, of the form *reference . feature* (...), is the fundamental operation of object-oriented computation. We start with procedure calls; the next section considers functions. Remember that in the syntactic domain *Routine*, and hence in its subsets *Procedure* and *Function*, we have assumed components *pre* and *post*, listing the assertions with which a routine has been equipped.

```
Procedure_call
            -- The basic object-oriented call
      subset
            Instruction
      has
            target: Variable
            feature: Procedure
                  -- For the moment we ignore arguments
      end
precondition (p: Procedure_call) (a: Assertion): Boolean is
            -- Is it correct to call p under the initial condition a?
      local
                  -- Auxiliary names for convenience:
            feature_target := p . target
            feature_pre := (p . feature . pre)
            feature_post := (p . feature . post)
            feature_body := (p . feature . body)
      do
            Result :=
                        feature_target member (a . nonvoid) and
                        (a => (feature_pre +++ feature_target)) and
                        consistent (feature_body) (feature_pre, feature_post)
      end
```

member is the set membership operator. This definition of *precondition* states that the assertion *a* must guarantee that the target of the call (that is to say *reference* in *reference*.*feature* (...) must be non void — after all, this is what we are discussing all along), that the initial assertion must imply the routine's precondition (transposed to the target, hence the +++ *feature_target*) and that the body of the routine must implement its specification as stated by the precondition and postcondition. This last property is highlighted in red because it may be preferable, as explained earlier, to prove it separately.

```
postcondition (p: Procedure_call) (a: Assertion): Assertion is
                -- Effect of executing p under the initial condition a
                -- (Initial version — will be improved next).
        local
                        -- Auxiliary names for convenience:
                feature_target :=p . target
                feature_post := (p . feature . post)
        do
                Result := (feature_post +++ feature_target)
        end
```

This definition of *postcondition* states that the call achieves the postcondition listed in the routine, transposed to the context of the target.

Retaining unaffected properties

The definition of function *postcondition* for procedure calls, although useful, is not powerful enough in practice because our reliance on a routine's explicit assertions (*pre* and *post*) prevents us from retaining any assertion clauses that are not affected by the routine. Consider a call executed in a state where the following assertion (in Eiffel notation) holds:

```
one /= Void
```

and assume that *p* has the postcondition

```
other /= Void
```

Then the rule allows us to deduce, as postcondition of the call, the property

```
target.other /= Void
```

This is correct, but not good enough. Assuming that there is no connection whatsoever between references *one* and *other*, the original assertion will still hold after the call, so that we should be able to deduce the postcondition

```
one /= Void and target.other /= Void
```

The definition of *postcondition* obtained so far does not give this, since it defines the postcondition of the call on the sole basis of the routine's own stated postcondition.

This is where it pays off to have included in assertions the *affected* component, whose usefulness may not have been obvious so far. We replace the definition of *postcondition* by:

```
postcondition (p: Procedure_call) (a: Assertion): Assertion is
                -- Effect of executing p under the initial condition a
                -- (Final version).
        local
                        -- Auxiliary names for convenience:
                feature_target := p . target
                feature_post := (p . feature . post)
                from_routine := (feature_post +++ feature_target)
        do
                Result := extend_unaffected (a, from_routine)
        end
```

using a function *extend_unaffected* (*a*, *b*: *Assertion*): *Assertion* whose result is obtained from *b* as follows: add to *b* . *nonvoid* every element of *a* . *nonvoid* which is not an extension of an element of *b* . *affected* (where the "extensions" of *v* are *v* itself and any expression of the form *v*. *something*); add to *b* . *equal* every pair of *a* . *equal* of which neither element is an extension of an element of *b* . *affected*; and add to *b* . *affected* every element of *a* . *affected*.

Function calls

Along with procedure calls we must consider function calls. We assume for simplicity (but with no loss of generality, since the discussion can be transposed to the conventions of other languages) the Eiffel convention that a special variable (*Result* in Eiffel) is used in function bodies to denote the function's result.

There is little extra work to do if we restrict our attention to assignments of the following form (in Eiffel syntax):

```
some_variable := target . some_function
```

Then it suffices to apply the definitions obtained for *Procedure_call*, adapting the *postcondition* definition to ensure that any property of *Result* yields, through substitution, a property of *some_variable*.

Loops

As with routines, we can prove properties of loops by making sure that loops are equipped with assertions, specifically a loop invariant as in Eiffel:

```
Loop
        -- Loops with their invariants
    subset
        Instruction
    has
        exit, invariant: Assertion
        body: Instruction
            -- For the moment we ignore arguments
    end
```

Then:

```
precondition (l: Loop) (a: Assertion): Boolean is
        -- Does a enable us to execute the loop properly?
        -- Yes iff it guarantees initial satisfaction of the invariant,
        -- and the loop body preserves the invariant
    do
        Result := (a => l.invariant) and
                consistent (body) (l.invariant, l.invariant)
            -- In a more complete version the preceding line would also
            -- take exit into account. See the discussion below.
    end
postcondition (l: Loop) (a: Assertion): Boolean is
        -- The environment resulting from executing the loop:
        -- both the invariant and the exit condition hold
    do
        Result := (assertion_and (l.invariant, l.exit))
    end
```

In this function definition, *assertion_and* denotes a function, again easy to write, which produces the boolean "and" of two asssertions: take the union of the *nonvoid* sets,the *equal* sets, and the *affected* sets, respectively. (As with the definition of implication — "Implication", page 14 — this may at first seem counter-intuitive, since we are used to think that the "and" operator of logic has "intersection" as its counterpart in set theory; but union is indeed the right operator here since our assertions denote sets of objects satisfying certain properties, so that anding two assertions means including the objects satisfying these properties on either side.)

The *consistent* property to prove in the second line of the body of the definition of *precondition* is (as the reader familiar with Hoare semantics will have noted) too demanding. It requires that *l.body* always preserve *l.invariant*, whereas in fact it only needs to preserve this invariant when *l.exit* initially holds. So it would be desirable to replace this property by something like

```
(assertion_and (assertion_not (l.exit), l.invariant),
                l.invariant)
```

where the function *assertion_not* yields the negation of an assertion. Unfortunately we cannot, in the model described here, express *assertion_not*. (This has nothing to do with the use of EFL.) The model for assertions enables us to state that the *nonvoid* set of references known to be non-void is {*a*, *b*, *c*}; it provides not way to state the negation of that property — that one or more of *a*, *b* and *c* is void. Although it may be possible to extend the assertion model to support negation, this does not seem worth the trouble, since it appears unlikely that proving the kind of properties relevant for this article — that some references are *not* void, or are equal, or have been affected by the computation — would require the assumption, in the body of a loop, that some references *are* void (or not equal, or not affected).

If this happens, remember that the property will be needed in a "red" proof, which we will have either to take for granted or to prove through techniques that fall outside of the present state of the Theory of Void.

Acknowledgements: this paper benefited from comments by John Potter, Heinz Schmidt, Richard Mitchell and Kim Waldén.