Avoid a void

Bertrand Meyer

With major contributions by Emmanuel Stapf & Alexander Kogtenkov (Eiffel Software)
Basic O-O operation

\[ x.f(\text{args}) \]

Semantics: apply the feature \( f \), with given \textit{args} if any, to the object to which \( x \) is attached

... and basic issue studied here:

How do we guarantee that \( x \) will always be “attached” to an object?

(If not, call produces an exception and usually termination)
I call it my billion-dollar mistake. It was the invention of the null reference in 1965. 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 safe, checked by the compiler.

But I couldn't resist the temptation to put in a null reference, 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.
Plan

1. Context

2. New language constructs

3. Achieving void safety

4. Current status
Context
44.4% of Eiffel preconditions clauses are of the form

\[ x \neq \text{Void} \]
The standards effort

ECMA standard for Eiffel


ISO standard, November 2006

Mechanism described here is in EiffelStudio 6.3, Nov 2008; libraries fully adapted in 6.4, May 2009
Requirements

- Minimal language extension
- Statically, completely void safe
- Simple for programmer, no mysterious rules
- Reasonably simple for compiler
- Handles genericity
- Doesn't limit expressiveness
- Compatibility or minimum change for existing code
- 1st-semester teachability
"Spec# stipulates the inference of non-voidness for local variables. This inference is performed as a dataflow analysis by the Spec# compiler."

(Barnett, Leino, Schulte, Spec# paper)
Last night I had a dream. I was programming in Eiffel 5.7. The code was elegant. There was no need for defensive programming just by taking full advantage of design by contract. Thanks to these contracts the code was easy to reuse and to debug. I could hardly remember the last time I had a call-on-void-target. It was so pleasant to program with such a wonderful language.

This morning when I woke up I looked at the code that had been modified to comply with void-safety. This was a rude awakening. The code which was so elegant in my dream now looked convoluted, hard to follow. It looks like assertions are losing all their power and defensive programming is inviting itself again in the code. [...]
New language constructs
New constructs

1. Object test

   Replaces all “downcasting” (type narrowing) mechanisms

2. Type annotations: “attached” and “detachable”

New keywords: attached, detachable

(Plus: stable.)
The Object Test (full form)

Boolean expression:

\[
\text{attached } \{T\} \ exp \ as \ x
\]

Value:
True if value of \( exp \) is attached to an object of type \( T \) or conforming

Plus: binds \( x \) to that value over scope of object test
Object Test example

```plaintext
if `attached \{ T \} exp as x` then
  ... Arbitrary instructions ...
  x .operation
  ... Other instructions ...
end
```
Object Test variants

attached \{ T \} \text{exp} \text{ as } x

attached \text{exp} \text{ as } x

attached \{ T \} \text{exp}

attached \text{exp}

Same semantics as \text{exp} \neq \text{Void}
Another example of Object Test scope

```plaintext
from
...
until not attached exp as x loop

... Arbitrary instructions ...

x.some_operation

... Other instructions ...

end
```

Scope of x
Object test in contracts

my_routine

require

attached exp as x

and then x.some_property

do

...

end

Scope of x
Achieving void safety
A success story: static type checking

We allow

\[ x.f(\text{args}) \]

only if we can guarantee that at run time:

The object attached to \( x \), if it exists, has a feature for \( f \), able to handle the \( \text{args} \).

Basic ideas:

- Accept it only if type of \( x \) has a feature \( f \)
- Assignment \( x := y \) requires conformance (based on inheritance)

What if \( x \) is void?
The goal ("void safety"): at compile time, allow

\( x \cdot f \text{(args)} \)

only if we can guarantee that at run time:

\( x \text{ is not void.} \)
The basic rule

\[ x.f(args) \text{ is permitted only if } x \text{ is attached} \]

where “attached” is a static property

The rest of this discussion is about various ways to ensure that \( x \) is attached
Components of the solution

1. Some patterns guaranteed void-safe
   ("Certified Attachment Patterns" or CAPS)

2. Object test

3. Attached type
   (Issue here: initialization)

4. Rule for generic parameters
Components of the solution

1. Some patterns guaranteed void-safe ("Certified Attachment Patterns" or CAPS)
2. Object test
3. Attached type (Issue here: initialization)
4. Rule for generic parameters
Certified Attachment Patterns

CAP: a program scheme that guarantees the void-safety of a call
Consider a variable or expression $e$

Can this be guaranteed void-safe?

```plaintext
if $e \neq \text{Void}$ then
  other_instructions not assigning to $e$
  $e$.operation (args)
end
```

(Assume $e$ is a variable)

Answer: only if $e$ is a local variable! (or formal argument)
Not for an attribute, or a general expression.
CAP 2: loop

from
...
until
\[ x = \text{Void} \]
loop
... Any instructions not assigning to \( x \) ...
... (except possibly last instruction) ...
end

\( x \) must again be a local variable or a formal argument!
A typical loop, now safe

\[ x := \text{first\_element} \]

\[ \text{item right item right item right item right} \]

\[ \text{from} \]

\[ x := \text{first\_element} \]

\[ \text{until} \]

\[ x = \text{Void or else Result} \]

\[ \text{loop} \]

\[ \text{Result := (} x.\text{item = sought) } \]

\[ x := x.\text{right} \]

\[ \text{end} \]
Consider a class with the invariant

\[(after) \implies (active = Void)\]

and the routine

```plaintext
require not after
do
    check active /= Void end
active.some_operation
end
```

Semantics of `check α end`: compiler must either prove that `α` holds or generate code that triggers exception if it doesn't.
The CAP catalog

About 6 CAPs specified in the ECMA standard. Above them are the most important.

Another example: \( x \) in

\[ x /= Void \implies x\text{.some_property} \]

Criterion: simple; easy to understand; provably and obviously correct

Need to be approved by committee
Mathematical, machine-checked proofs desirable
CAP variant: stable attributes

if \( e \neq \text{Void} \) then

\[ \text{other\_instructions} \text{ not assigning to } e \]

\[ e.\text{operation} \]

end

OK if \( e \) is a “stable attribute”: no routine of the class assigns to it

\( e : \text{SOME\_TYPE} \)

\[ \text{note} \]

\[ \text{stable attribute} \]

end
Components of the solution

1. Some patterns guaranteed void-safe
   ("Certified Attachment Patterns" or CAPS)

2. Object test

3. Attached type
   (Issue here: initialization)

4. Rule for generic parameters
Using an object test for void safety

if attached \{T\} \text{exp} \text{ as } x \text{ then }

\begin{align*}
&\text{... Arbitrary instructions...} \\
&x \text{.} \text{operation} \\
&\text{... Other instructions ...}
\end{align*}

end
The remaining goal...

Minimize use of Object Test!

General but impractical solution: protect every qualified feature call by an Object Test!

Instead of

```
exp . operation
```

write

```
if attached \{ T \} exp as x then
    x . operation
end
```
Components of the solution

1. Some patterns guaranteed void-safe
   ("Certified Attachment Patterns" or CAPS)

2. Object test

3. Attached type
   (Issue here: initialization)

4. Rule for generic parameters
Attached and detachable types

For any type $T$, a variable declared as just

\[ x : T \quad -- \text{or:} \quad x : \text{attached } T \]

cannot be void!

Type $T$ is attached

To allow void values, declare

\[ x : \text{detachable } T \]

Type detachable $T$ is detachable
Default policy

Attached as a default because of “business objects”, e.g. bank accounts, persons, planes...

Void is useful for “Knuthware”, specifically linked structures
The remaining problem...

How to ensure that variables declared of an attached type

\[ x : T \]

meet that declaration, i.e. can never become void!
Conservation of attachment

In

\[ x := y \]

or

\[ r(y) \quad -- \text{Where formal argument is } x \]

if type of \( x \) is attached, \( y \) must be attached.
If assignment & argument passing are OK...

... there remains initialization!
The initialization problem

In previous Eiffel:

- Basic types initialized to standard values (zero, False, null character)

- Other expanded types must have `default_create` from `ANY`

- References initialized to `Void`
Initializing variables

Scheme 1: CAP

- An attribute is attached if every creation procedure sets it.

- A local variable is initialized if the beginning of the routine sets it.
Initializing variables

Scheme 1: CAP

Scheme 2: Self-initializing attributes
Digression: attributes

*bounding_rectangle*: RECTANGLE

-- Smallest rectangle including whole of current figure
Digression: attributes with contracts!

\hspace{1cm}

\textit{bounding\_rectangle: RECTANGLE}

\hspace{1cm}-- Smallest rectangle including whole of current figure

\hspace{1cm}

\textbf{require}

\hspace{1cm}\textit{bounded attribute}

\hspace{1cm}

\textbf{ensure}

\hspace{1cm}\textit{Result. height = height}

\hspace{1cm}\textit{Result. width = width}

\hspace{1cm}\textit{Result. lower\_left = lower\_left}

\hspace{1cm}\textit{Result. contains (Current)}

\hspace{1cm}

\textbf{end}
Self-initializing attributes

bounding_rectangle: FIGURE
-- Smallest rectangle including whole of current figure
-- (Computed only if needed)

require
  bounded
attribute

create Result.set (lower_left, width, height)

ensure

... As before ...
Initializing variables

Scheme 1: CAP

Scheme 2: Self-initializing attributes

Scheme 3: Properly set variables

What if an attribute is not self-initializing?
Properly set variables

At any place where it is accessed, a variable must be properly set
Components of the solution

1. Some patterns guaranteed void-safe
   ("Certified Attachment Patterns" or CAPS)

2. Object test

3. Attached type
   (Issue here: initialization)

4. Rule for generic parameters
In class $C[G]$, what about variables such as $x: G$?

Example *generic derivations:*

- $C[\text{INTEGER}]$
- $C[\text{EMPLOYEE}]$
Meaning of a formal generic parameter

**Constrained genericity:**

\[
\text{class } C[G \to T] \ldots
\]

Generic derivation must be \( C[U] \), where \( U \) conforms to \( T \)

**Unconstrained genericity:**

\[
\text{class } C[G] \\
\]

- In pre-void-safe Eiffel: abbreviation for
  \[
  \text{class } C[G \to \text{ANY}] \ldots
  \]

- Now: abbreviation for
  \[
  \text{class } C[G \to \text{detachable ANY}] \ldots
  \]

Can also use

\[
\text{class } C[G \to \text{attached T}] 
\]
The remaining issue: arrays

\[ a: ARRAY[T] \quad -- \text{Assume } T \text{ is an attached type} \]

Array creation (old):

\[ \text{create } a\cdot\text{make}(1, n) \]

Not void-safe!

Creation procedure now required for attached types:

\[ \text{create } a\cdot\text{make}(1, n, \text{default}) \]

Creation issue: classes inheriting from \text{ARRAY}. 
Current status
Full mechanism supported

Libraries entirely converted

Environment (EiffelStudio) partly converted

Compatibility option

Did not switch the defaults yet
Basic O-O operation

\[ x \cdot f(\text{args}) \]

Semantics: apply the feature \( f \), with given \( \text{args} \) if any, to the object to which \( x \) is attached

... and basic issue studied here:

How do we guarantee that \( x \) will always be “attached” to an object?

(If not, call produces an exception and usually termination)
Last night I had a dream. I was programming in Eiffel 5.7. The code was elegant. There was no need for defensive programming just by taking full advantage of design by contract. Thanks to these contracts the code was easy to reuse and to debug. I could hardly remember the last time I had a call-on-void-target. It was so pleasant to program with such a wonderful language.

This morning when I woke up I looked at the code that had been modified to comply with void-safety. This was a rude awakening. The code which was so elegant in my dream now looked convoluted, hard to follow. It looks like assertions are losing all their power and defensive programming is inviting itself again in the code. [...]

Requirements

- Minimal language extension
- Statically, completely void safe
- Simple for programmer, no mysterious rules
- Reasonably simple for compiler
- Handles genericity
- Doesn’t limit expressiveness
- Compatibility or minimum change for existing code
- 1st-semester teachability
I call it my billion-dollar mistake. It was the invention of the null reference in 1965. 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 safe, checked by the compiler.

But I couldn't resist the temptation to put in a null reference, 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.