Nature abhors a void

Bertrand Meyer
The basic O-O operation...

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

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

... and the basic issue studied here:

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

(If not, call produces an exception and usually program termination.)
Other problems solved by this work

Getting rid of “catcalls”, statically

A consistent approach to type casts

A better way to do “once per object”

A consistent approach to object locking in concurrent programming
There is one and only one kind of acceptable language extension: the one that dawns on you with the sudden self-evidence of morning mist.

It must provide a complete solution to a real problem, but usually that is not enough: almost all good extensions solve several potential problems at once...

It must fit perfectly within the spirit and letter of the rest of the language. It must not have any dark sides or raise any unanswerable questions.
The context

**ECMA standard for Eiffel**

Approved 21 June 2005

On its way ("Fast Track") to becoming an ISO standard
References

Useful to describe linked data structures

Can be void!
A precedent: static type checking

We allow

\[ x.f(args) \]

What if \( x \) is void?

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 \( args \).

Basic ideas:

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

Our goal ("void safety"): at compile time, allow

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

only if we can guarantee that at run time:

\[ x \text{ is not void.} \]
Requirements on an acceptable solution

- Statically, completely void safe: no exceptions
- Handles genericity
- Simple for programmer, no mysterious rules
- Reasonably simple for compiler
- Compatibility or minimum change for existing code
“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)
Components of the solution

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

2. Void value permitted only for types declared as
   "detachable". By default types are "attached"

3. Initialization rules ensure that any variable of an
   attached type has a non-void initialization value

4. Rule for generic parameters
Rule 1: Target Validity Rule

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

“Attached” is a static property. \( x \) is attached if either:

- Its type is attached
- Its type is not attached, but \( x \) itself is guaranteed attached in a specific context
An interesting pattern

Consider a variable or expression \( exp \)

Is this void-safe? (Assume \( exp \) is a variable)

\[
\begin{align*}
\text{if } \& exp \neq Void \text{ then} \\
\text{other_instructions} \\
\text{exp.some_routine(args)} \\
\text{end}
\end{align*}
\]

Answer: only if \( exp \) is a local variable! (or formal argument)
Not for an attribute, or a general expression.
Attached entity: Case #1

\( x \) is attached if used as part of a **Certified Attachment Pattern** (CAP).

Example CAP for a local variable or formal argument \( x \):

\[
\text{if } x \neq \text{Void} \text{ then}
\]

\[
\text{... Any instructions not assigning to } x \text{ ...}
\]

\[
\text{... (except possibly last instruction) ...}
\]

end

This is a CAP for \( x \)
Rule 1: Target Validity Rule

\[ x \cdot f(args) \] is permitted only if \( x \) is attached

Ways to ensure that \( x \) is attached:

1. Use it in a Certified Attachment Pattern
A loop CAP

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

```
first_element
```

```
from

x := first_element

until

x = Void or else Result

loop

Result := (x.item = sought)

x := x.right

end
```
The CAP catalog

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

Another example: \( x \) in

\[ x \neq \text{Void} \implies x.\text{some_property} \]

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

Need to be approved by committee
Mathematical, machine-checked proofs desirable
When no CAP applies: the Object Test

**Not** void-safe:

```plaintext
if exp /= Void then
  ... Various instructions ...
  exp.operation
  ... Other instructions ...
end
```
When no CAP applies: the Object Test

Previous scheme made void-safe!

\[
\text{if } \{ x : T \} \text{ exp then} \quad -- \ T \text{ is the type of } exp
\]

... Various instructions, anything OK! ...

\[
\text{x .operation}
\]

... Other instructions ...

end
The Object Test

\{ x \colon T \} \text{exp}

exp is an arbitrary expression
x is a fresh variable, read-only (like formal argument)

It’s a boolean expression: true if value of exp is attached to object of type T or conforming

Binds x to that value over scope of expression
Another example of Object Test scope

```plaintext
from

⋯

until not \{x: T\} exp then

  ... Various instructions, anything OK!
  ...

  \textcolor{green}{x.operation\_of\_T}

  ...

  Other instructions ...
```

end
Rule 1: Target Validity Rule

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

Ways to ensure that \( x \) is attached:

1. Use it in a Certified Attachment Pattern
2. Use it in the scope of an Object Test
Another example of Object Test

if \{e: EMPLOYEE\} database.retrieved then
    ... Various instructions, anything OK! ...

    e.print_paycheck

    ... Other instructions ...
end

Replaces “assignment attempt” of current Eiffel, and various “type narrowing”, “Run-Time Type Identification”, “downcasting” mechanisms
The remaining goal...

Minimize use of Object Test!
Attached and detachable types

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

\[ x : T \]

cannot be void!

Type $T$ is attached

To allow void values, declare

\[ x : ? T \]

Type $? T$ is detachable
Rule 1: Target Validity Rule

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

Ways to ensure that \( x \) is attached:

1. Use it in a Certified Attachment Pattern
2. Use it in the scope of an Object Test
3. Give it an attached type
The remaining problem...

How to ensure that variables declared of an attached type

\[
\mathbf{x} : T
\]

meet that declaration, i.e. can never become void!
Rule 2: 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.

(No “traitors”, see SCOOP.)
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`
Reminder: overall inheritance structure

\[ \text{ANY} \]

\[ \text{A} \]
\[ \text{B} \]
\[ \text{C} \]
\[ \text{D} \]
\[ \text{E} \]
\[ \text{F} \]
\[ \text{NONE} \]

Inherits from
Reminder: creation in Eiffel (1)

Basic creation instruction

```
create x.cp(...)  
```

where *cp* is one of the procedures of the class, marked as
creation procedure
Reminder: creation in Eiffel (2)

The form without an explicit creation procedure

```
create x                       -- With x of type T
```

is an abbreviation for

```
create x.default_create
```

and is valid if and only if $T$ specifies its version of `default_create` as a creation procedure.
Reminder: overall inheritance structure

ANY inherits from DEFAULT_CREATE

A inherits from B, C, D

B inherits from A, D, E

C inherits from A, B, F

D inherits from A, B, E

E inherits from B, F

F inherits from C, E

NONE inherits from A, B, C, D, E, F
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 types

A type $T$ is self-initializing if it specifies `default_create` as creation procedure.

Then initialization can be lazy:

- If execution accesses a variable $x$ of type $T$ uninitialized, $x$ will initialize itself with `default_create`.
Initializing variables

Scheme 1: CAP
Scheme 2: Self-initializing types
Scheme 3: Self-initializing attributes

What if an attribute is not of a self-initializing type, and is not set by creation procedures?
Digression: attributes (fields/data members)

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

**bounding_rectangle**: RECTANGLE

--- Smallest rectangle including whole of current figure

```require
bounded attribute

ensure
Result. height = height
Result. width = width
Result. lower_left = lower_left
Result. contains (Current)
```

end
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 ...
Another example: once per object

```plaintext
class STOCKS feature
  db: DATABASE

  history (ts: TICKER_SYMBOL): HISTORY
    -- List of previous valuations of ts
    attribute
      if {h: HISTORY} db.retrieved (ts) then
        Result := h
      else
        create Result  -- Produces empty list
      end
    end
end
end
```
Initializing variables

Scheme 1: CAP
Scheme 2: Self-initializing types
Scheme 3: Self-initializing attributes
Scheme 4: Variables of formal generic type

In class $C[G]$, what about the initialization of

$x : G$ ?
Genericity: the issue

class ARRAY[G] feature
  item alias "[]" (i: INTEGER): G
  put (x: G; i: INTEGER)
  ...
end

How can class \( \text{ARRAY} \) initialize the entries properly without knowing what \( G \) is?
Genericity: the solution

In a class $C[G]$, if the implementation accesses an attribute

$$x: G$$

that is not guaranteeably attached (through a CAP, or self-initialization), then the class **must** be declared as

$$C[? G] \quad \text{-- For example: ARRAY[? G]}$$

Then any actual parameter ($T$ in $C[T]$) **must** be either:

- Self-initializing
- Detachable!
Requirements on an acceptable solution

- **Statically, completely void safe: no exceptions**
- **Handles genericity**
- **Simple for programmer, no mysterious rules**
- **Reasonably simple for compiler**
- **Compatibility or minimum change for existing code**
Other problems solved by attached types

Getting rid of “catcalls”, statically

A better way to do “once per object”

A consistent approach to object locking in concurrent programming

(Through the Object Test) Consistent approach to type casts
The catcall issue

Under covariance, avoid bad calls due to polymorphism
The catcall solution

Accept a covariant redeclaration only if the new type is detachable!
The concurrency issue

\[ f(x: \text{separate } A) \text{ do } \ldots \text{ end} \]

Lock \( x \)?
There is one and only one kind of acceptable language extension: the one that dawns on you with the sudden self-evidence of morning mist.

It must provide a complete solution to a real problem, but usually that is not enough: almost all good extensions solve several potential problems at once...

It must fit perfectly within the spirit and letter of the rest of the language. It must not have any dark sides or raise any unanswered questions.
Acknowledgments

Mechanism design:

Mark Howard (AxaRosenberg)
Emmanuel Stapf (Eiffel Software)
Eric Bezault (AxaRosenberg)
Karine Arnout (ETH)
Alexander Kogtenkov (Eiffel Software)

Concurrency applications:

Piotr Nienaltowski (ETH)

Special thanks to:

Peter Mueller (ETH)
Rustan Leino, Erik Meijer (Microsoft)