Avoid a void

Bertrand Meyer

©Bertrand Meyer, 2008
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)
Other problems solved by this work

A consistent approach to type casts

A better way to do “once per object”

A consistent approach to object locking in concurrent programming
The context

ECMA standard for Eiffel


ISO standard, November 2006

Mechanism described here is in EiffelStudio 6.1, Nov 2007
8.27.8  Validity: Call Agent rule

A Call_agent involving a Feature_name $fn$, appearing in a class $C$, with target type $T0$, is valid if and only if it satisfies the following conditions:

1. $fn$ is the name of a feature $f$ of $T0$.
2. If there is an Agent_target, $f$ is export-valid for $T0$ in $C$.
3. If the Agent_actuals part is present, the number of elements in its Agent_actual_list is equal to the number of formals of $f$.
4. Any Agent_actual of the Expression kind is of a type compatible with the type of the corresponding formal in $f$. 

Validity code: VPCA
References

Useful to describe linked data structures

Can be void!
References are a conceptual modeling tool

Difference between

- Each car has an engine
- Each car has a manufacturer (e.g. Fiat, Renault)
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 \( \text{args} \).

What if \( x \) is void?

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

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

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

(Plus for me: 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)
44.4% of Eiffel preconditions clauses are of the form

$x \neq \text{Void}$
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

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

Can this be guaranteed void-safe?

\[
\text{if } \ exp \neq \text{Void} \text{ then } \left( \text{Assume } exp \text{ is a variable} \right)
\]

\[
\text{other_instructions not assigning to } exp \\
\text{exp.operation (args)}
\]

end

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 \( x \):

\[
\text{if } x \neq \text{Void} \text{ then} \\
\quad \text{... Any instructions not assigning to } x \ldots \\
\quad \text{... (except possibly last instruction) ...} \\
\text{end}
\]

This is a CAP for \( x \).
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
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

\[ \text{from} \]
\[ x := \text{first\_element} \]
\[ \text{until} \]
\[ x = \text{Void} \text{ or else Result} \]
\[ \text{loop} \]
\[ \text{Result} := (x.\text{item} = \text{sought}) \]
\[ x := x.\text{right} \]
\[ \text{end} \]
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:

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

... Various instructions ...

\[
\text{exp.operation}
\]

... Other instructions ...

\[
\text{end}
\]
When no CAP applies: the Object Test

Previous scheme made void-safe!

```
if {x: T} exp then
  -- T is the type of exp
  ...
  Various instructions, anything OK!
  ...
  x.operation
  ...
  Other instructions ...
end
```

Karine Arnout
Éric Bezault
The Object Test

\{ x: T \} 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
Object Test example

if \{x : T\} exp then

-- T is the type of exp

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

x.operation

... Other instructions ...

end
Another example of Object Test scope

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

    ... Various instructions, anything OK!
    ...
    x.operation_of_T

    ... Other instructions ...

end
```
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
Another example of Object Test

\[
\text{if } \{ e: \text{EMPLOYEE} \} \text{ database.retrieved then}
\]

\[...	ext{Various instructions, anything OK! ...}\]

\[e\text{.print_paycheck}\]

\[...	ext{Other instructions ...}\]

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!

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

Instead of

\[ \text{exp} . \text{operation} \]

write

\[
\begin{align*}
\text{if } \{ x : T \} \text{ exp then} \\
& \quad x . \text{operation} \\
\text{end}
\end{align*}
\]
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

\[ x : T \]

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`
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.
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

Scheme 2: Properly set variables
Properly set variables

At any place where it is accessed, a variable must be properly set.
Definition: Properly set variable

At an evaluation position `ep` in a class `C`, a variable `x` is properly set if one of the following conditions holds:

1. `x` is self-initializing.
2. `ep` is an evaluation position of the Compound of a feature or Inline_agent of the Internal form, one of whose instructions precedes `ep` and is a setter for `x`.
3. `x` is a variable attribute, `x` is (recursively) properly set at the end position of every creation procedure of `C`.
4. `ep` is an evaluation position in a Compound that is part of an instruction `ep'`, itself belonging to a Compound, and `x` is (recursively) properly set at position `ep'`.
5. `ep` is in a Postcondition of a routine or Inline_agent of the Internal form, and `x` is (recursively) properly set at the end position of its Compound.
6. `ep` is an Assertion_clause containing `Result` in the Postcondition of a constant attribute.
Initializing variables

Scheme 1: CAP
Scheme 2: Properly set variables
Scheme 3: Self-initializing attributes

What if an attribute 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

\textit{bounding\_rectangle: FIGURE}
--- Smallest rectangle including whole of current figure
--- (Computed only if needed)

\textbf{require} \quad \textit{bounded}
\textbf{attribute}

\textbf{create Result.set} (lower\_left, width, height)

\textbf{ensure}

... As before ...
Another example: once per object

\[
\text{class } \textit{STOCKS} \text{ feature} \\
\quad \textit{db: DATABASE} \\
\quad \text{history}(\textit{ts: TICKER\_SYMBOL}): \textit{HISTORY} \\
\quad \quad \text{-- List of previous valuations of } \textit{ts} \\
\quad \quad \text{attribute} \\
\quad \quad \text{if } \{\textit{h: HISTORY} \} \text{ db.retrieved(\textit{ts})} \text{ then} \\
\quad \quad \quad \text{Result} := \textit{h} \\
\quad \quad \text{else} \\
\quad \quad \quad \text{create Result} \quad \text{-- Produces empty list} \\
\quad \text{end} \\
\text{end} \\
\text{end}
\]
Initializing variables

Scheme 1: CAP
Scheme 2: Properly set variables
Scheme 3: Self-initializing attributes

Scheme 4: Variables of formal generic type

In class $C \ [G]$, what about the initialization of $x : G$?

Example generic derivations:

$C \ [INTEGER]$

$C \ [EMPLOYEE]$
Genericity: the issue

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

How can class ARRAY initialize the entries properly without knowing what G is?
Genericity

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 detachable
Initializing variables

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

(+ for me: the “Week 6 of Einführung in die Programmierung” criterion)