ETH Zürich

Chair of Software Engineering

Void safety

these slides contain advanced material and are optional

The inventor of null references

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

By Tony Hoare, 2009

Problems of void-calls

- Entities are either
  - Attached: referencing a valid object
  - Detached: Void (or null)
- Calls on detached entities cause a runtime error
- Runtime errors are bad...

How can we prevent this problem?

Solution to void-calls

A call *f.x (...)* is only allowed, if *f* is statically attached.

- Statically attached: Checked at compile-time
- Dynamically attached: Attached at runtime

- Consistency:

If *f* is statically attached, its possible runtime values are dynamically attached.

Statically attached entities

- Attached types
  - Reference types that cannot be `Void`
  - `x: attached STRING`
- Certified attachment patterns (CAP)
  - Code pattern where attachment is guaranteed
  - `if x /= Void then x.f end` (where x is a local)
- Object test
  - Assign result of arbitrary expression to a local
  - Boolean value indicating if result is attached
  - `if attached x as l then l.f end`

Attached types

- Can declare type of entities as attached or detachable
  - `att: attached STRING`
  - `det: detachable STRING`
- Attached types
  - Can call features: `att.to_upper`
  - Can be assign to detachable: `det := att`
  - Cannot be set to void: ~~`att := Void`~~
- Detachable types
  - No feature calls: ~~`det.to_upper`~~
  - Cannot be assign to attached: ~~`att := det`~~
  - Can be set to void: `det := Void`

Attached types (cont.)

- Entities need to be initialized
  - Detachable: `Void`
  - Attached: assignment or creation
- Initialization rules for attached types
  - Locals: before first use
  - Attributes: at end of each creation routine
  - Compiler uses control-flow analysis
- Types without attachment mark
  - Default can be set in project settings
  - Default for void-safe projects should be **`attached`**

Attached types demo

- EiffelStudio settings
- Declarations
- Error messages
- Initialization

Certified attachment pattern (CAP)

- Code patterns where attachment is guaranteed
- Basic CAPs for locals and arguments
  - Void check in conditional or semi-strict operator
  - Setter or creation

```
capitalize (a_string: detachable STRING)
    do
        if a_string /= Void then
            a_string.to_upper
        end
    ensure
        a_string /= Void implies a_string.is_upper
    end
```

CAP demo

- Different CAPs for locals and arguments
  - Void check in contract
  - Void check in conditional
  - Setter
  - Creator

Object test

- Checking attachment of an expression (and its type)
- Assignment to a local variable
  - Local is not declared and only available in one branch

```
name: detachable STRING

capitalize_name
    do
        if attached name as l_name then
            l_name.to_upper
        end
    ensure
        attached name as n implies n.is_upper
    end
```

Side note on object tests

- Object test can also be used to make a type cast
- The test is **True**, if object conforms to specified type
- Local variable will have specified type

```
name: detachable ANY

capitalize_name
    do
        if attached {STRING} name as l_name then
            l_name.to_upper
        end
    ensure
        attached {STRING} name as n implies n.is_upper
    end
```

Object test demo

- Object test in body
- Object test in assertion
- Object test to test for type

Stable attributes

- Detachable attributes which are never set to void
- They are initially void, but once attached will stay so

```
name: detachable STRING
    note
        option: stable
    attribute
    end

capitalize_name
    do
        if name /= Void then
            name.to_upper
        end
    end
```

Stable demo

- Declaring stable attributes
- CAPs with stable attributes

Arrays

- Arrays can have more storage space then elements
- Empty storage space filled with *default* values
- What is the default for attached types?
  - `a: attached ARRAY [attached STRING]`

- `See Array demo`

Other languages: Spec#

- Research variant of C#
- Adds contracts and non-null types (and more)
- Non-null types are marked with !

```
String s = null;

String! s = „abc“;
```

~~String! s = null;~~

Other languages: JML

- Research variant of Java
- Adds contracts and non-null types (and more)
- Types (except locals) are non-null per default

```
~~String s = null;~~

String s = „abc“;

/*@ nullable @*/ String s = null;
```

References

- Eiffel documentation on void-safety
  - http://docs.eiffel.com/book/method/void-safe-programming-eiffel
- Avoid a Void: The eradication of null dereferencing
  - http://s.eiffel.com/void_safety_paper
- Targeted expressions
  - http://se.ethz.ch/~meyer/publications/online/targeted.pdf