УДК 004.052.42, 004.4'6, 004.423.42, 004.432.2, 004.438 Eiffel, 519.681.2, 519.682.1

Making void safety practical

Alexander Kogtenkov (ETH Zürich, Switzerland; Eiffel Software, USA)

Null pointer dereferencing remains one of the major issues in modern object-oriented languages. An obvious addition of keywords to distinguish between never null and possibly null references appears to be insufficient during object initialization when some fields declared as never null may be temporary null before the initialization completes. The proposed solution avoids explicit encoding of these intermediate states in program texts in favor of statically checked validity rules that do not depend on special conditionally non-null types. Object initialization examples suggested earlier are reviewed and new ones are presented to compare applicability of different approaches. Usability of the proposed scheme is assessed on open-source libraries with a million lines of code that were converted to satisfy the rules.

Keywords: null pointer dereferencing, null safety, void safety, object initialization, static analysis, library-level modularity

1. Introduction

Tony Hoare [5] called his invention of the null reference a "billion-dollar mistake". The reason is simple: most object-oriented languages suffer from a problem of null pointer dereferencing. Even in a type-safe language, if an expression is expected to reference an existing object, it can reference none, or be *null*. Given that the core of object-oriented languages is in the ability to make calls on objects, if there is no object, the normal program execution is disrupted.

Not prevented at compile time, it remains one of the day-to-day issues. My analysis of the public database of cybersecurity vulnerabilities known as Common Vulnerabilities and Exposures (CVE®)[1] operated by *MITRE* and funded by Computer Emergency Readiness Team (*CERT*) reveals that in the past 10 years entries mentioning null pointer dereference bugs appear at consistent rate of about 70 bugs a year. As the database covers only software affecting the whole planet, real economy losses, caused by unlisted projects, are much higher.

To distinguish types of expressions that may return `null` from always returning an object, Raymie Stata [14] proposed a notation `T ?` for Java. Developers of the Checkers Framework[2]

[1] Common Vulnerabilities and Exposures. 2017. URL: `http://cve.mitre.org/` (visited on 2017-04-27).

[2] The Checker Framework 2.1.10. 04/03/2017. URL: `https://checkerframework.org/` (visited on 2017-05-08).

mention that now most static analyzers for Java use annotations `@Nullable` and `@NonNull`. Manuel Fähndrich and Rustan Leino [3] used C# attributes `[NotNull]` and `[MayBeNull]`. In different forms similar marks are used in Eiffel [6] (with type marks **attached** and **detachable**) and Kotlin [7] (with a mark `?`). Unfortunately, sequential initialization of object fields does not permit for non-null fields to be initialized with object references atomically.

Most solutions of the object initialization issue extend type systems to identify incompletely initialized objects. My review of open libraries showed that most code could be made null-safe without new type marks. Instead of tweaking the type system, I introduced compile-time validity rules for the remaining cases. With them, not only all examples from relevant publications [3; 4; 12; 16] could be compiled as expected, but new scenarios became feasible.

Together with removal of annotations for local variables [8], based on typing rules similar to those used in security data flow [17] and known as *flow-sensitive typing* [11], reduced annotation overhead simplifies adaptation of legacy code and makes null-safe programming more accessible.

2. Motivating examples

I Polymorphic call from a constructor. Manuel Fähndrich and Rustan Leino [3] describe a call to a virtual method on `this` in a superclass constructor. Because subclass fields of the object are not initialized yet, accessing them in the polymorphic call causes `NullReferenceException`. Xin Qi and Andrew C. Myers [12] consider a similar example with a class `Point` and its subclass `CPoint` that adds a color attribute.

II Polymorphic callback from a constructor. Accesses to an uninitialized object can be done indirectly. If a superclass constructor passes a reference to the current object as an argument to create another object, this "remote" constructor can call-back on the object where not all fields are initialized yet. A reasonable solution should distinguish between legitimate and non-legitimate calls to "remote" constructors to be sufficiently expressive and sound.

III Modification of existing structures. Convenience of the ability to invoke regular procedures inside a creation procedure can be demonstrated with a mediator pattern [1]. It decouples objects so that they do not know about each other, but still can communicate using an intermediate object, *mediator*. Concrete types of the communicating objects are unknown to the mediator, and, therefore, it cannot create them.

On the other hand, communicating objects know about the mediator and can register according to their role. If the registration is done in their constructors, clients do not need to clutter their code with calls to a special feature *register* after creating every new communicating object. The assignment like `x = new Comm (mediator)` should do both – recording a reference to the mediator and registration of the communicating object.

Registration of a new object may also be required in GUI libraries where a GUI-specific toolkit object has to keep references to the user-created object for event-based communication.

IV Safety violations. In addition to valid cases, authors usually mention examples that should trigger a compiler error (*e.g.*, Alexander J. Summers and Peter Müller [16]). This aims at the original goal: a sound solution should catch potential null dereferencing at compile time.

V Circular references. Another issue arises when two objects reference each other. If the corresponding fields have non-null types, access to them should be protected to avoid retrieving `null` by the code that relies on the field type.

Manuel Fähndrich and Songtao Xia [4] review a linked list example with a sentinel. When a new list is constructed, a special sentinel node is created and it should reference the original list object. In other words, an incompletely initialized list object has to be passed to a node constructor as an argument. An attempt to access field `sentinel` at this point would compromise null safety, so there should be means to prevent such accesses or to make them safe (*e.g.*, by treating field values as possibly null and as referring to uninitialized objects).

VI Self-referencing. This is a variant of circular references when an object references itself rather than another object. Xin Qi and Andrew C. Myers [12] review a binary tree where every node has a parent, and the root is a parent to itself.

At binary node creation, left and right nodes should get a new parent and the parent should reference itself. With any initialization order there are states when the new binary node should be used to initialize either its own field or field `parent` of its left or right nodes before it is completely initialized. Therefore, arbitrary accesses to this node should be protected like in the previous case.

3. Overview

3.1. Language conventions and terminology

Bertrand Meyer [10] pointed out that language rules can simplify or make it more difficult to achieve null safety guarantees. *E.g.*, in Java or C# a superclass constructor has to be called before the subclass constructor. Hence, non-null fields of the subclass cannot be initialized before calling superclass constructor. Without such restrictions, field initialization can be carried out in any suitable order that allows for fixing examples I and II without any new types.

I use Eiffel as an implementation testbed. The language specifies two type marks – **attached** (the default) and **detachable** – to denote non-null and maybe-null types respectively. Current object (`this` in Java and C#) is named **Current** and constructors are called *creation procedures*. They can also be used as regular routines, and are checked twice: as creation procedures for safe object initialization, and as regular procedures. Data members of a class are called *attributes*.

The language standard [6] introduces a notion of a *properly set* variable. For object initialization this means that all attributes of attached types should reference existing objects. By default, a field of a reference type does not reference an existing object, or is **Void**. If **Void** is used as a target of a call, the run-time raises an exception *"Access on void target"*. A compile-time guarantee that a system never causes such an exception is called *Void safety*.

3.2. Solution outline

All examples from the previous section can be divided into 2 major groups:

(A) Examples I to IV: – Can the code be reordered so that all fields are initialized before use?

(B) Examples V and VI: – Can compile-time rules ensure an object with recursive references to itself is not used as a completely initialized one?

The issue in group (A) arises because **Current** object is passed before all attributes of this object are properly set. The simplest rule would be to forbid using **Current** until all attributes are properly set:

Validity rule 1 (Creation procedure, strong)**.** *An expression* ***Current*** *is valid in a creation procedure or in an unqualified feature it (directly or indirectly) calls if all attributes of the current class are properly set at the execution point of the expression.*

The rule is sufficient to deal with group (A) by reordering initialization instructions.

But the rule is too strong for group (B). Of course, if a reference to an incompletely initialized object is leaked, the task to identify such an object becomes almost intractable not only in theory, but also due to complexity of implementing alias analysis correctly [2]. Explicit type annotations [3; 4; 12; 16] move detection of incompletely initialized objects from static analysis methods to the type system. I avoid performing alias analysis and extending the type system by preventing use of incompletely initialized objects in the first place.

The key source of obscurity in an object-oriented environment is polymorphism. Creation procedures are associated with specific classes, hence, no polymorphism is involved here. Even unqualified features they call can be checked for creation validity. The checks will make sure that class fields are not accessed before they are set and **Current** is completely initialized. But qualified calls are still an issue:

- a call on an incompletely initialized object cannot assume all attributes are properly set;
- a qualified call does not allow seeing what operations on an incompletely initialized object are performed.

The solution is to disallow qualified calls when some objects are incompletely initialized:

Validity rule 2 (Creation procedure, weak)**.** *A creation procedure is valid if any of the following is false at the same execution point:*

- ***Current*** *is used before all attributes have been properly set and not all attributes are properly set after that.*
- *The expression at the execution point is one of*
  - *a qualified feature call;*
  - *a creation expression that makes a qualified call.*

Unlike validity rule 1, the weak version assumes there is information, whether creation procedures of other classes make direct or indirect qualified calls. It could be explicitly or implicitly specified in creation procedure signatures, or inferred from code.

4. Related work

Raw types ***(solve examples I and IV with*** 2+ ***annotations).*** Manuel Fähndrich and K. Rustan M. Leino [3] denote attached types with T^- and detachable types with T^+ and propose to add raw types T^{raw-} to be used for partially initialized objects. If class C has an attribute of type T and some entity has type C^{raw-} then a qualified call to this attribute has type T^+

regardless of original attachment status of that attribute. An assignment to an entity of a raw type accepts only a source expression of a non-raw non-null type to ensure that if an object becomes fully initialized, it cannot be uninitialized. Also, by the end of every constructor, every non-null field should be assigned.

Then raw types are refined with class frames corresponding to superclasses. Inside a constructor of a class C, the special entity `this` has type C^{raw-}, and when the constructor finishes, the type becomes C^{-}. In a constructor of a super-class A the type of `this` is $C^{raw(A)-}$. The authors also specify conformance rules in this type system. Unfortunately, rules for super-class constructors, *e.g.*, for $T^{raw(R)-}$, are not directly applicable to languages with multiple class inheritance like Eiffel. And raw types do not support creation of circular references.

An implementation demonstrated that further extensions are required for real code, *e.g.*, to access fields that have been initialized and to indicate that a method initializes certain fields.

Masked types ***(solve examples I to VI with many annotations).*** Xin Qi and Andrew C. Myers [12] address the complete object life cycle. They instrument the type system with so called "masks" representing sets of fields that are not currently initialized. For example, the notation `Node\parent!\Node.sub[l.parent] -> *[this.parent]` for an argument `l` tells that it has a type `Node` and on entry requires that its field `parent` is not set and at the same time fields declared in subclasses of `Node` are not set unless `l.parent` is initialized. On exit the actual argument conforms to the type `Node\*[this.parent]` that indicates that the node object will be completely initialized as soon as its field `parent` is set.

The notation is very powerful and goes far beyond void safety, but even with its complexity authors complain that it is not sufficient for real programs. For information hiding they propose abstract masks updated in descendant classes as required. The idea looks similar to the data groups approach proposed by Rustan Leino in [9]. For modular processing of abstract masks, subclass masks and mask constraints are introduced with union and difference operations.

Like with masked types, validity rule 2 depends on whether class attributes are properly set and a reference to **Current** object escapes before that. Flow-sensitive type analysis is performed without special annotations too. However, with masked types the results are checked against provided specifications, while in my approach they are used to check validity rule conditions.

Free and committed types ***(solve examples I and IV to VI with*** 1+ ***annotations).*** Alexander J. Summers and Peter Müller distinguish [16] just two object states: under initial-

ization and completely initialized. A newly allocated object has a so called "free" type. When an object is deeply initialized, *i.e.*, all its fields are set to deeply initialized objects, it is said to have a "committed" type. The commitment point logically changes the type of an object from free to committed and is defined as the end of a constructor that takes only committed arguments. Possible aliasing between free and committed types is prevented by not having a subtyping relation between them. This differs from the convention for raw types [3].

Validity rule 2 is very close in spirit to the idea of free and committed types. But it relies on a flow-sensitive analysis and ceases free type status when all attributes are set. This allows for handling cyclic data structures without explicit annotations.

A variant of committed and free types is implemented in the Checker Framework with annotations `@UnknownInitialization` and `@UnderInitialization` supporting type frames `@UnderInitialization (A.class)` to tell that all fields specified in a (super)class `A` have been initialized. Authors of the Checker Framework claim that `this` cannot be used in a class constructor as `@Initialized`. This rules out examples II and III.

Other approaches *(solve examples I to VI with* 0 *annotations, non-modular).* Additional annotations are avoided by Bertrand Meyer in [10] using so called "targeted expressions" and creation-involved features. The analysis is somewhat similar to the abstract interpretation approach used by Fausto Spoto [13] and should be applied to the system as a whole, thus sacrificing modularity. This makes it difficult to develop self-contained libraries. The advantage of the approach is in selective detection of variables that are not completely initialized.

5. Formalization

Formalization of validity rules and proofs of their properties are done using the Isabelle/HOL proof assistant to avoid any inconsistencies and omissions. The theories code verified by *Isabelle2016*[3] is available at `https://bitbucket.org/kwaxer/void_safety/` (tag `1.2.5`).

Initialization state Validity rules are formalized using a simplified version of an Eiffel-like abstract syntax. The transfer function $\cdot \gg \cdot$ takes 2 arguments – an expression and a set of attributes V that may be unattached before the expression – and returns a set of attributes

[3]Isabelle2016. 01/16/2016. URL: `http://isabelle.in.tum.de/website-Isabelle2016/` (visited on 2017-05-07).

that may be unattached after the expression. At the beginning of a creation procedure the set of unattached attributes is a set of all current class attributes of attached reference types.

A validity predicate $V \vdash e \surd_c'$ tells if an expression e satisfies validity rule 1 in the context with unset attributes V.

Safe uses of Current If **Current** is never referenced in a creation procedure, there is no issue because the incompletely initialized object is not passed anywhere. If **Current** is referenced when all attributes are set, there is no issue as well: once an object is completely initialized, it remains completely initialized and can be freely used. Finally, if **Current** is referenced when not all attributes of the current class are set, but can escape only at the current execution point (*i.e.*, all previous expressions do not make any qualified calls, thus excluding the possibility to access this incompletely initialized object), it is possible that all attributes are set now and therefore the object is completely initialized regardless of its status when the reference to it escaped. These properties are captured by a function *safe*.

Detection of qualified feature calls For telling if a feature makes a qualified feature call, it is sufficient to analyze the corresponding abstract syntax tree. The function also takes care about qualified feature calls present in the features that are called from a current creation procedure using an unqualified feature call.

Another function is used to compute a set of creation procedures that can be called by the current one. Because the set of classes is known at compile time and is bounded, all creation procedures recursively reachable from the current one can be computed as a least fixed point.

Together with the function that tells whether a creation procedure has immediate qualified calls the function *has_qualified* tells if a creation procedure can lead to a qualified call.

Validity predicate A formal predicate $S,\ V \vdash e \surd_c$ for validity rule 2 is defined using functions *safe* and *has_qualified*. S stands for the current system to retrieve dependencies between creation procedures. The predicate is true as soon as $V \vdash e \surd_c'$ is true, i.e. validity rule 2 is more permissive than validity rule 1.

The predicate is monotone, so it is sufficient to analyze loops and unqualified feature calls just once, because any subsequent iterations or recursive feature calls would be analyzed with a larger set of properly-set attributes.

The soundness proof for object initialization is similar to the one given by Alexander J. Summers and Peter Müller [15] with two major differences. Firstly, the *free* status of a current object does not last until the end of a creation procedure, but only up to the point when all attributes are set, with the reservation that the creation procedure is not called by another one with an incompletely initialized **Current**. Secondly, annotations are replaced with the requirement to avoid qualified feature calls in the context with incompletely initialized objects.

For initialization of **Current** two situations are possible. In the first case all attributes of the current class are set and there are no incompletely initialized objects in the current context. Then the current object is deeply initialized and can be freely used before the creation procedure finishes. In the second case either some attributes of the current class are not properly set or the context has references to objects that are not completely initialized. Because qualified calls are disallowed in these conditions, the uninitialized attributes cannot be accessed and access on void target is impossible. Due to the requirement to set all attributes at the end of a creation procedure, all these objects will have all attributes set, and, taking into account that the only reachable objects are either previously fully initialized or are new with all attributes pointing to the old or new objects, *i.e.*, also fully initialized, all objects become fully initialized in the context where all attributes of the current class are set and no callers passed an uninitialized **Current**.

6. Practical results

Although validity rule 1 looks pretty restrictive, 4254 classes of public libraries have been successfully converted relying on this rule. This comprises 822487 lines of code and 3194 explicit creation procedures. 59% of these creation procedures (1894 in absolute numbers) perform regular direct or indirect qualified calls and might be in danger if not all attributes were set before **Current** was used. However, it was possible to refactor all the classes to satisfy the rule.

On average, 60% of creation procedures make qualified calls. Remaining 40% do not use any qualified calls and set attributes using supplied arguments or by creating new objects. They could be unconditionally marked with annotations as safe for use with incompletely initialized objects.

In contrast to this, just a tiny fraction of all creation procedures – 77 creation procedures from two libraries, or less than 2% – do pass uninitialized objects and take advantage of the

weaker validity rule 2. In other words, if specific annotations were used, at most 5% of them would be useful, the rest would just clutter the code.

The validity rule checks for creation procedures are pretty light. The libraries were compiled with and without checks for validity rule 2 on a machine with 64-bit *Windows 10 Pro*, *Intel® Core™ i7-3720QM*, 16GB of RAM and SSD hard drive using *EiffelStudio 16.11 rev.99675*. For all libraries the slowdown was just 0.7% that seems to be more than acceptable.

7. Conclusion

Proposed solutions for the object initialization issue have the following benefits:

No annotations. Validity rules do not require any other type annotations in addition to attachment marks.

Flexibility. Creation of objects mutually referencing other objects is possible.

Simplicity. The analyses require only tracking for attributes that are not properly set, for use of **Current** and for checking whether certain conditions are satisfied when (direct or indirect) qualified feature calls are performed.

Coverage. It was possible to refactor all libraries to meed the requirements of the rules without changing design decisions. The rules solve all examples from the motivation section.

Modularity. Validity rule 2 depends on properties of creation procedures from other classes. Because these creation procedures are known at compile time, the checks do not depend on classes that are not directly reachable from the one being checked. Therefore, a library can be checked as a standalone entity without the need to recheck it after inclusion in some other project.

Performance. Experiments demonstrate very moderate increase of total compilation time, below 1% on sample libraries with more than 2 millions lines of code.

Incrementality. Fast recompilation is supported if information about reachable creation procedures and whether they perform qualified calls is recorded for every class.

Main drawbacks of the rules are:

Certain coding pattern. Certain initialization order have to be followed.

Disallowing legitimate qualified calls. Lack of special annotations prevents from distinguishing between legitimate and non-legitimate qualified calls. To preserve soundness all qualified calls are considered as potentially risky.

Special convention for formal generics. If a target type of a creation expression is a formal generic parameter, special convention should be used to indicate whether a creation procedure of an actual generic parameter satisfies the validity rule requirements.

References

1. Design Patterns: Elements of Reusable Object-oriented Software / E. Gamma [et al.]. — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995. — ISBN 0-201-63361-2.
2. Effective Dynamic Detection of Alias Analysis Errors / J. Wu [et al.] // Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. — Saint Petersburg, Russia : ACM, 2013. — Pp. 279–289. — (ESEC/FSE 2013). — ISBN 978-1-4503-2237-9. — DOI: 10.1145/2491411.2491439.
3. *Fähndrich M.*, *Leino K. R. M.* Declaring and Checking Non-null Types in an Object-oriented Language // Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications. — Anaheim, California, USA : ACM, 2003. — Pp. 302–312. — (OOPSLA '03). — ISBN 1-58113-712-5. — DOI: 10.1145/949305.949332.
4. *Fähndrich M.*, *Xia S.* Establishing Object Invariants with Delayed Types // Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications. — Montreal, Quebec, Canada : ACM, 2007. — Pp. 337–350. — (OOPSLA '07). — ISBN 978-1-59593-786-5. — DOI: 10.1145/1297027.1297052.
5. *Hoare T.* Null references: The billion dollar mistake // Presentation at QCon London. — 2009.
6. *ISO.* ISO/IEC 25436:2006(E): Information technology — Eiffel: Analysis, Design and Programming Language. — 1st. — Geneva, Switzerland : ISO (International Organization for Standardization), IEC (International Electrotechnical Commission), 12/01/2006.
7. *JetBrains.* Kotlin Language Specification. — 01/31/2017. — URL: https://jetbrains.github.io/kotlin-spec/kotlin-spec.pdf (visited on 2017-01-31).
8. *Kogtenkov A.* Mechanically Proved Practical Local Null Safety // Proceedings of the Institute for System Programming of the RAS. — Moscow, Russia, 2016. — Dec. — Vol. 28, no. 5. —

Pp. 27–54. — ISSN 2079-8156 (Print), 2220-6426 (Online). — DOI: 10.15514/ISPRAS-2016-28(5)-2.

9. *Leino K. R. M.* Data Groups: Specifying the Modification of Extended State // Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. — Vancouver, British Columbia, Canada : ACM, 1998. — Pp. 144–153. — (OOPSLA '98). — ISBN 1-58113-005-8. — DOI: 10.1145/286936.286953.
10. *Meyer B.* Targeted expressions: safe object creation with void safety. — 07/30/2012. — URL: http://se.ethz.ch/~meyer/publications/online/targeted.pdf (visited on 2017-05-08).
11. *Pearce D. J.* On Flow-Sensitive Types in Whiley. — 09/22/2010. — URL: http://whiley.org/2010/09/22/on-flow-sensitive-types-in-whiley/ (visited on 2017-05-07).
12. *Qi X.*, *Myers A. C.* Masked Types for Sound Object Initialization // Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — Savannah, GA, USA : ACM, 2009. — Pp. 53–65. — (POPL '09). — ISBN 978-1-60558-379-2. — DOI: 10.1145/1480881.1480890.
13. *Spoto F.* Precise null-pointer analysis // Software & Systems Modeling. — 2011. — Vol. 10, no. 2. — Pp. 219–252. — ISSN 1619-1366. — DOI: 10.1007/s10270-009-0132-5.
14. *Stata R.* ESCJ 2: Improving the safety of Java. — 12/02/1995. — URL: http://kindsoftware.com/products/opensource/ESCJava2/ESCTools/docs/design-notes/escj02.html (visited on 2017-04-27).
15. *Summers A. J.*, *Müller P.* Freedom before commitment: simple flexible initialisation for non-full types: tech. rep. / ETH Zurich, Department of Computer Science. — Zurich, Switzerland, 2010. — No. 716. — DOI: 10.3929/ethz-a-006904372.
16. *Summers A. J.*, *Müller P.* Freedom Before Commitment: A Lightweight Type System for Object Initialisation // Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. — Portland, Oregon, USA : ACM, 2011. — Pp. 1013–1032. — (OOPSLA '11). — ISBN 978-1-4503-0940-0. — DOI: 10.1145/2048066.2048142.
17. *Volpano D.*, *Irvine C.*, *Smith G.* A Sound Type System for Secure Flow Analysis // Journal of Computer Security. — Amsterdam, The Netherlands, The Netherlands, 1996. — Jan. — Vol. 4, no. 2/3. — Pp. 167–187. — ISSN 0926-227X. — URL: http://dl.acm.org/citation.cfm?id=353629.353648.