Null safety benchmarks for object initialization

A.V. Kogtenkov <kwaxer@mail.ru>
Independent scientist,
Podolsk, Russia

Abstract. 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. There are several proposals to solve the object initialization problem. How can they be compared in practice? Are the implementations sound? This work presents a set of examples distilling out the use cases from publications on the subject and open source libraries and explains the criteria behind. Then it discusses expected results for a selected set of tools performing null safety checks for Eiffel, Java, and Kotlin, and concludes with the actual outcomes demonstrating immaturity of the solutions.

Keywords: null pointer dereferencing; null safety; void safety; object initialization; static analysis; null safety benchmarks.

DOI: 10.15514/ISPRAS-2017-29(6)-7

For citation: Kogtenkov A.V. Null safety benchmarks for object initialization. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017. pp. 135-150 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-7

1. Introduction

To construct a sound null-safe type system, most solutions of the problem add a notion of non-null and maybe-null types, usually expressed with additional type annotations. Such annotations would be sufficient to solve the null safety problem if objects could be created in an atomic operation, so that all fields marked as non-null were initialized with object references. Unfortunately, sequential initialization of the fields breaks the solution. Several proposals solving the object initialization issue suggest extending the type systems further to identify objects that are not completely initialized. Another group of approaches is based on static analysis that does not require any additional annotations at the expense of more sophisticated checks.

The theoretical solutions in the cited works specify a subset of an object-oriented language rather than a full language. The omission of the real-life language constructs

leads to omission of all required checks in the actual implementations. This causes the following issues:

- It is hard to reason about the extended model for a real language because all the consequences of such an extension are unclear and difficult to grasp when they go beyond intuition and when combination of such features requires exhaustive examination of possible interactions. The models used in the theoretical frameworks are limited for exactly this reason: they are not easily scalable and an addition of a new construct increases the size of the associated proofs proportionally, i.e. the factor is multiplicative rather than additive. This can be seen on the mechanically checked formalization of the null safety in the local context where a combination of conditional expressions, loops and exceptions greatly increased the size of the proofs. Moreover, some language features, such as value types, exceptions, concurrency or garbage collection may require a complete redesign of the model with much higher complexity impact.
- The model and the real language are decoupled informally – they are developed by different people – and formally – the proofs for the model and the implementation for the real language are carried out using different tools. There is no guarantee that an implementation is correct with respect to its model.

This leads to slow adoption of the total safety guarantees in production. The total guarantees are replaced by the "partial" ones. E.g., null safety is guaranteed only when the program does not do anything "wrong". Unfortunately, this defeats the whole purpose of the safety guarantees because null dereferencing errors in such "wrong" programs may happen at unexpected places, including trusted ones.

This work is a case study of the null safety guarantees provided by existing implementations rather than by the theoretical findings. It presents a set of examples that can be used to

- check if a particular programming pattern is handled by the implementation;
- perform measurable comparison of different solutions in terms of their soundness, expressiveness and verbosity.

The examples are written in Java, Kotlin, and Eiffel. They are available for independent analysis at https://bitbucket.org/kwaxer/null-safety-benchmark/src/?at=2017-ispras. I evaluate the benchmarks on the *Kotlin* and *Eiffel* compilers and the *Checker Framework* for Java.

The main contributions of this work are:

- development of execution scenarios of object initialization to benchmark different null safety solutions and to provide measurable comparison;
- evaluation of production solutions with null safety guarantees against the benchmarks.

The rest of the paper is organized as follows. Section 2 identifies the key reasons of the object initialization problem and specifies additional requirements to the benchmarks. Section 3 gives an overview of the existing work in the area. Section 4 classifies scenarios to test implementations with null safety guarantees and reviews the structure of the proposed benchmark suite. Section 5 presents the results of the evaluation on some production environments. Section 6 provides a quick summary and concludes with the directions of future work.

2. Overview

2.1 Reasons of the object initialization problem

Null safety is complicated for object initialization. To understand why, I suggest to look at how program execution can lead to the null reference exception. Firstly, the object that causes the problem should not complete its initialization, i.e., some of its fields of non-null types should be null. Secondly, this object should be accessible – either directly or through some variables. Thirdly, the information that its initialization is incomplete should be lost. Otherwise, it would be easy to report the error at compile time. Finally, the reference retrieved from the uninitialized field of the object should be dereferenced to trigger the exception. To summarize, there are the following roots of the problem:

- *Non-atomic initialization* of an object leads to the possibility to have fields with null values even when their type is non-null.
- *Aliasing* allows for accessing the same object from an arbitrary point of the program, in particular, from the code that does not expect an incompletely initialized object.
- *Uncontrollable control flow*, interrupting the regular one, makes sequential reasoning about program execution useless.
- *Dereferencing* of an uninitialized field of the incompletely initialized object triggers the exception.

The solutions extending the type system with new types limit the operations on incompletely initialized objects. The solutions based on static analysis, such as *practical void safety* , specify the conditions, when dereferencing may be unsafe, and forbid such dereferencing altogether.

2.2 Restrictions on the benchmarks

In order to focus only on the object initialization problem, I put the following restrictions on the code in the benchmarks:

- Self-containment – the code should not rely on or assume any null-safe properties of the classes outside the examples.

- Limited null safety scope – the code should reflect only issues with object initialization. Other mechanisms, such as array initialization and initialization of static fields in Java, companion objects in Kotlin and once features in Eiffel, as well as general data flow analysis for null safety are out of scope.

3. Related work

Alexander J. Summers and Peter Müller set the following design goals of the type system they propose to use for null safety:

1. *Modularity:* the type system can check each class type separately.
2. *Soundness:* the type system is safe, i.e., null pointer exceptions are impossible at run-time.
3. *Expressiveness:* the type system handles common initialization patterns. In particular, it allows objects to escape from constructors and supports the initialization of cyclic structures.
4. *Simplicity:* the type system is simple and requires little annotation overhead.

I use this list in section 4 as the base for further fine-grained classification of solution properties. Then, the properties can be evaluated with specific examples, thus allowing for measurable comparison of different implementations. The detailed classification of the solution properties is reviewed in the earlier publication . The review is based on the associated algorithms and descriptions. Compared to this work, it considers expected behavior of the solutions with wider spectrum of properties, because it also includes qualitative ones, such as modularity. On the other hand, that review does not provide actual code to check the tools, nor does it verify that the proposed test cases can be expressed in a specific language.

Every publication on the problem of object initialization has few examples that authors use either to demonstrate issues with their approach, or to explain how the issues are resolved. I collected all the examples from the publications I know about as well as found in class libraries and divided them into the following groups.

3.1 Polymorphic call from a constructor

When a constructor of a superclass is invoked in C#, a call to a virtual method on `this` is considered a bad practice. At this moment, subclass fields of the object are not initialized yet and using them in the polymorphic call is unsafe. Manuel Fähndrich and Rustan Leino describe an example of this situation. In a descendant class, before one of its fields is set, the superclass constructor is called. The constructor invokes a virtual method. The override of the method in the descendant accesses the uninitialized field leading to `NullReferenceException`.

Xin Qi and Andrew C. Myers give a similar example where they consider a class `Point` and its subclass `CPoint` that adds a color attribute.

3.2 Polymorphic callback from a constructor

Fig. 1 shows a sample object structure taken from the portable GUI library *EiffelVision*[1] employing a bridge pattern to support different platforms. A client of the library directly works only with the interface objects. Upon its creation, the interface object creates an implementation object that, on completion of initialization, notifies the interface object via a callback. If at this point some non-null fields of the interface object are unset, access on them causes a null dereferencing error.

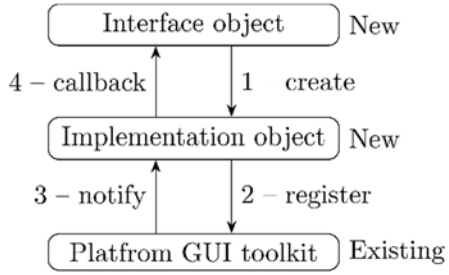

Fig. 1. Object structure in a portable GUI library.

3.3 Modification of existing structures

The ability to invoke regular procedures inside a creation procedure is convenient, e.g., for a mediator pattern . This pattern 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, the mediator cannot create them. A mediator's client is responsible for creating necessary communicating objects instead.

The communicating objects know about the mediator and can register themselves in the mediator according to their role. If the registration is done in the constructors of the communicating objects, the mediator's clients do not need to clutter the code with calls to a special feature *register* every time they create a new communicating object. An assignment like `x = new Comm (mediator)` should do both actions: the recording of the mediator object in the new communicating object, and the registration of the communicating object in the mediator. A chat room adapted from and shown in Fig. 2 is an example implementing a mediator pattern.

[1] https://www.eiffel.org/doc/solutions/EiffelVision%202


```
class ROOM create make feature
        users: ARRAYED_LIST [USER]
        make
                do
                        create users.make (0)
                end
        join (a: USER)
                do
                        users.extend (a)
                end
        send (s: STRING)
                do
                        across users as u loop
                                u.item.receive (s)
                        end
                end
end
```

```
class USER create make feature
        room: ROOM
        make (r: ROOM)
                do
                        room := r
                        r.join (Current)
                end
        send (s: STRING)
                do
                        room.send (s)
                end
        receive (s: STRING)
                do
                        io.put_string (s)
                        io.put_new_line
                end
end
```

Fig. 2. Example of a mediator pattern (in Eiffel).

When the feature *join* is called in the creation procedure *make* of a *USER* object, all fields of the object should be set.

The need to register a newly created object in an existing one is also present in the earlier example with the GUI library in Fig. 1 where the newly created implementation object has to register itself in the existing GUI toolkit object to dispatch events from the underlying GUI toolkit to the implementation object and then to the interface object.

3.4 Circular references

An 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 types and, therefore, expects non-null values. Manuel Fähndrich and Songtao Xia demonstrate the problem on a linked list example with a sentinel.

When a new list is constructed, a special sentinel node is created. The sentinel should reference the original list object. In other words, an incompletely initialized list object has to be passed to the sentinel node constructor as an argument. An attempt to access the field that is expected to reference the sentinel node inside the sentinel constructor would compromise null safety. Therefore, 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).

There are similar circular references in the classes included in the library *Gobo*[2] to model XML documents. According to the XML specification , every document has one root element and every XML element has one parent, coinciding with the document node for the root element.

3.5 Self-referencing

A particular case of circular references concerns an object that references itself rather than another object. Xin Qi and Andrew C. Myers give the example of a binary tree where every node has a parent, and the root is a parent to itself. At a 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 where the new binary node should be used to initialize either its own field or the field `parent` of its left or right nodes before it is completely initialized. Therefore, arbitrary accesses to this node should be protected.

3.5 Safety violations

In addition to valid cases, authors usually mention examples that should trigger a compiler error. This aims at the original goal: a sound solution should catch potential null dereferencing at compile time.

4. Benchmark criteria and null safety suite

The most important goal of the null safety design is soundness. It limits the possibilities to write arbitrary code that is still null safe. The general problem of safe object initialization is undecidable. Therefore, the code can be checked in finite time only when some restrictions are imposed on it.

Soundness and expressiveness work against each other: the simpler the language rules, the less code can be written without violating them. If the rules are too strict, some scenarios found in real software can become extinct. E.g., according to the theory, the *raw types* do not allow for creation of circular structures, the *free and committed types* rule out registration of objects in existing object structures inside constructors, and *practical void safety* disallows any qualified calls, including potentially safe ones, as soon as there are some incompletely initialized objects in the current execution context.

The proposed benchmark suite ignores some important properties of the solutions. Measuring such properties requires significantly different code with tight dependency on the underlying language and involved tools. Consequently, it would be difficult to do the comparison. The ignored criteria include

- Modularity, including scope, telling whether it is sufficient to analyze (recursively) ancestors and suppliers of the class to be checked, and

[2] http://www.gobosoft.com/eiffel/gobo/xml/

  incrementality, telling whether changes to previously checked code require a partial recheck rather than a complete one;
- Simplicity, including ease of use, telling whether few new simple rules are added to the language to make object initialization null safe, and performance, telling the resource consumption (algorithmic space and time complexity) to support the additional checks.

The assessment of different solutions from the theoretical point of view, based on the analysis of the corresponding algorithms, is given in the earlier paper that, in particular, elaborates on the criteria listed above. The current work focuses on the behavior of the actual implementations instead. The benchmarks are applicable only to existing tools and allow for measuring the number of additional type marks that need to be specified besides the marks “non-null” and “maybe-null”.

The test suite consists of two main parts: one for soundness examples and one for expressiveness examples. Because there are certain differences between languages in syntax and semantics, every example is equipped with an accompanying document describing the execution scenario. The document also lists possible variations of the example if present. In the text below, the names of the examples are underlined.

4.1 Soundness

Authors of all the solutions mentioned in section claim them to be sound. Unfortunately, not all aspects of a real programming environment are usually reflected in a formal model. In particular, none of the null safety formalizations reflects garbage collection that is an important channel to compromise safety guarantees.

The roots of the object initialization problem, mentioned in section , are mapped to the programming language constructs as follows:

- *Non-atomic initialization* corresponds to the order of initialization of the object fields intermingled with other computations. This work does not consider languages that support atomic (transactional) object initialization.
- Explicit *aliasing* becomes possible when an object is assigned to a field of an existing object, either passed to the constructor as an argument or directly reachable from the current context, or when the new object is thrown as an exception. Implicit *aliasing* happens when the class declares a finalizer that gets access to the object.
- *Uncontrollable control flow* can be caused by concurrent execution, preemptive execution (with exception and signal handlers), cooperative execution (coroutines).
- *Dereferencing* is done by a qualified call of the form `target.access` where `access` stands for a field or method name and `target` is a name of a reference corresponding to one of uninitialized fields of the object.

The soundness examples demonstrate potential scenarios where execution can lead to a null dereference error at run-time. The language tools should be able to detect and to report the possible error at compile time – this constitutes the null safety guarantees. If the error is not detected, the corresponding implementation is unsound.

If a program context does not expect an uninitialized object, there should be no channels that allow for the object to escape to this context. The following language mechanisms can make such escaping unexpected:

- exceptions;
- concurrency;
- cooperative execution.

These mechanisms are used in the examples in an attempt to demonstrate unsoundness of implementations. This is done by performing an unsafe dereferencing using either unqualified calls (of the form `field_name`) that access fields of the current object, or qualified calls (of the form `expression.field_name`), that access fields of the current or some other object. The values of the fields may be null or (recursively) have null values in non-null fields of referenced objects.

The escaping channels depend on the programming language. The most common ones are discussed next.

4.1.1 Registration in an existing object

A program can register a new object in an existing one. When this is done before the new object is completely initialized, there is a problem: the incompletely initialized object can be accessed via the exiting object because of aliasing. Thus, such registration should be disallowed or no accesses to the fields can assume that they are non-null. The scenario can be further classified by

1) the *source* of the reference to the existing object that can be either (a) passed as an argument to the constructor with a variant that the passed reference could correspond to the newly created (fresh) object rather than a fully initialized one, (b) retrieved from the current execution context (static fields, companion and singleton objects, once functions);
2) the *type* of the object in which the new one is registered: it can be (a) a user-defined one or (b) a built-in one (e.g., an array, a tuple, etc.).

4.1.2 Reclamation of incompletely initialized objects

Finalizers are the methods called before object's memory is reused. The finalizers are registered for calling by the run-time after object's memory is allocated and before the constructor is invoked. If the object initialization does not complete (due to an uncaught exception in the constructor), the finalizer is invoked on the incompletely initialized object. Unless a programmer keeps track of object initialization, there is no way to figure out what state the object is in. Therefore, the current object in a

finalizer should be treated like at the beginning of a constructor. The reclamation example accesses non-null fields that are not initialized and, therefore, should be flagged as erroneous.

4.1.3 Out-of-order object transfer

Most programming languages allow for transferring references to objects bypassing regular control flow. The most familiar mechanism is exceptions. If a new exception object referencing an incompletely initialized one is thrown, the reference to the incompletely initialized object becomes accessible in the code that relies on the type system rules and does not expect uninitialized fields. The transfer example attempts to throw such an exception object.

4.2 Expressiveness

The examples in this group do not lead to null pointer exceptions and can be accepted as valid by the tools performing null safety checks. If the checks are too strict, they can rule out legitimate use cases.

4.2.1 Access to an initialized object

As soon as an object is completely initialized, it can be safely accessed even when it is used inside its constructor. In particular, it is safe to perform

- a callback on this object from the code to where it is passed. This pattern is discussed in section .
- registration of any form (argument, context, fresh) listed in the section about soundness. (See also section .)
- out-of-order transfer as soon as the exception object refers to the completely initialized one. Because the object is completely initialized, it does not cause a problem in the exception handling context.

4.2.2 Access to an uninitialized object

Calls on incompletely initialized objects require special precaution. They may enable somewhat higher code reuse, but are “viral”, because any reference obtained from an incompletely initialized object should be considered as maybe-null and incompletely initialized according to the rule *Field Read* . Therefore, the usability of this coding technique is limited, and the example “uninitialized” is included in the suite merely for completeness.

4.2.3 Circular references

Object structures with references that make a cycle appear to be a common design pattern (see section). The corresponding benchmark checks whether the rules are

flexible enough to allow for self-referencing of one object or mutual-referencing of two objects.

5. Evaluation results

The examples listed in section 4 are used to check the following language tools:

- *Checker Framework* version 2.2.1 from September 29, 2017
- *EiffelStudio* compiler version 17.05 from May 29, 2017
- *Kotlin* compiler version 1.1.50 from September 22, 2017

At the time of writing, the benchmark contains 6 expressiveness examples and 8 soundness examples (registration examples have 2 variants each: one with exceptions and one with threads), all in 3 programming languages, 2574 total lines of code.

Table 1. Evaluation results

(a) Results of expressiveness tests

Tool	Example: cycle: callback	cycle: self	cycle: mutual	uninitialized	registration: argument	registration: context
Checker Framework	$+_{1/2}$	$+_{1/1}$	$+_{2/2}$	$+_{1/2}$	$\oplus_{1/1}$	$\oplus_{1/1}$
EiffelStudio compiler	+	+	+	−	+	+
Kotlin compiler	+	+	+	+	+	+

(b) Results of soundness tests

Tool	Example: registration: argument	registration: fresh	registration: context	uninitialized	transfer
Checker Framework	$\ominus_{1/2}$	$\ominus_{2/2}$	$\ominus_{1/2}$	−	$\ominus_{2/3}$
EiffelStudio compiler	+	+	+	−	+
Kotlin compiler	−	−	−	−	−

Legend:

+ passed as expected	□ passed unexpectedly	$_{m/}$ number of different additional annotations
− failed as expected	□ failed unexpectedly	$_{/n}$ total number of additional annotations

The results for the expressiveness examples are summarized in Table 1(a). The *Checker Framework* employs the theory of *free and committed types*, and requires additional annotations to support the scenarios when **`this`** is passed from within the constructor as an argument to another constructor or method.

The theory states that all the registration examples should fail with compile-time errors, i.e., only 4 out of 6 tests should pass. However, the *Checker Framework* unexpectedly allows for assigning a non-completely initialized object to a field of a completely initialized one either received as an argument or retrieved from a static field. This deviation from the original set of rules might be the reason of the analysis unsoundness as discussed below.

The *practical void safety*, implemented in the *EiffelStudio* compiler, does not handle access on incompletely initialized objects, therefore, as expected, the example "uninitialized" does not pass. The expected score of 5 passing tests out of 6 tests matches the one obtained from the actual runs. Kotlin, on the other hand, has no special restrictions on uses of **`this`**, so all the expressiveness examples can be compiled.

The results for the soundness examples are presented in Table 1(b). In accordance with the documentation, the *Kotlin* compiler does not detect any invalid uses of the reference **`this`** in any of the examples and is unsound with respect to null safety.

As pointed out in section, object reclamation has not been mentioned in theoretical works as a potential source of null dereferencing caused by incomplete object initialization. Therefore, none of the tested tools applies special rules to finalizers. Therefore, all tested tools are unsound in this case.

The failure of the *Checker Framework* to catch errors in all other tests is surprising. Most probably, one bug is caused by missing language constructs in the simplified model language used to prove the soundness of the type system. Indeed, it has no equivalent of the statement **`throw`**. In order to preserve soundness, the statement should not satisfy the typing rules as soon as its argument is not of a committed type.

The origin of the *Checker Framework* bugs in the registration examples is unclear and demonstrates a gap between the typing rules specified in the proposed solution and the actual implementation. As a result, instead of 6 or 7 passing tests out of 8 tests, the *Checker Framework* fails in all soundness tests.

The *EiffelStudio* compiler passes 7 out of 8 soundness tests that matches the expected ratio of the *practical void safety* mechanism.

6. Conclusion and future work

None of the tested solutions guarantee complete null safety. It should be possible to fix the implementations of the *Checker Framework* and the *EiffelStudio* compiler, but at the time of writing the *Kotlin* compiler has no provisions for fixing the object initialization issue and can be thought of as "null safety aware", rather than "null safety complete" product.

The gap between theoretical works and practical implementations is caused by a simplified model of the target language and absence of verification of the implementations against the models.

The work reveals the following areas of future development:

- application of the proposed benchmark to other frameworks claiming null safety guarantees;
- collaboration with tool developers to eliminate the deficiencies found in the tools;
- extension of the suite with new examples to cover more coding patterns found in the field.

References

[1]. Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). Fifth Edition of a Recommendation. W3C, Nov. 2008. URL: http://www.w3.org/TR/2008/REC-xml-20081126/.

[2]. Manuel Fähndrich and K. Rustan M. Leino. Declaring and Checking Non-null Types in an Object-oriented Language. In: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications. OOPSLA'03. ACM, 2003, pp. 302–312. DOI: 10.1145/949305.949332.

[3]. Manuel Fähndrich and Songtao Xia. Establishing Object Invariants with Delayed Types. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications. OOPSLA'07. ACM, 2007, pp. 337–350. DOI: 10.1145/1297027.1297052.

[4]. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., 1995.

[5]. Alexander Kogtenkov. Practical Void Safety. In: Verified Software. Theories, Tools, and Experiments. 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22–23, 2017, Revised Selected Papers. Ed. by Andrei Paskevich and Thomas Wies. Vol. 10712. Lecture Notes in Computer Science. Springer International Publishing, 2017. DOI: 10.1007/978-3-319-72308-2_9.

[6]. Alexander Kogtenkov. Towards null safety benchmarks for object initialization. In: Modeling and Analysis of Information Systems 24.6 (2017).

[7]. A.V. Kogtenkov. Mechanically Proved Practical Local Null Safety. In: Trudy ISP RAN / Proc. ISP RAS, vol. 28, issue 5, pp. 27–54. DOI: 10.15514/ISPRAS-2016-28(5)-2.

[8]. Mediator pattern. 2017. URL: https://en.wikipedia.org/wiki/Mediator_pattern (visited on 2017-11-20).

[9]. Bertrand Meyer. Targeted expressions: safe object creation with void safety. July 30, 2012. URL: http://se.ethz.ch/~meyer/publications/online/targeted.pdf (visited on 2017-05-08).

[10]. Xin Qi and Andrew C. Myers. Masked Types for Sound Object Initialization. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'09. ACM, 2009, pp. 53–65. DOI: 10.1145/1480881.1480890.

[11]. Alexander J. Summers and Peter Müller. Freedom Before Commitment: A Lightweight Type System for Object Initialisation. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. OOPSLA'11. ACM, 2011, pp. 1013–1032. DOI: 10.1145/2048066.2048142.

Эталонные тесты безопасности нулевых ссылок при инициализации объекта

А.В. Когтенков <kwaxer@mail.ru>
Независимый учёный,
Россия, г. Подольск

Аннотация. Разыменование нулевого указателя остаётся одной из основных проблем в современных объектно-ориентированных языках. Очевидное добавление ключевых слов, чтобы различать между всегда ненулевыми и возможно нулевыми ссылками, оказывается недостаточным во время инициализации объекта, когда некоторые поля, объявленные всегда ненулевыми, могут временно быть нулевыми до окончания инициализации. Существует несколько подходов к решению проблемы инициализации объекта. Каким образом их можно сравнить практически? Являются ли реализации обоснованными? Данная работа представляет набор примеров, выделяя сценарии использования из публикаций по теме и библиотек с открытым кодом, и объясняет стоящие за ними критерии. Затем она обсуждает ожидаемые результаты для выбранного набора инструментов, производящих проверки безопасности нулевых ссылок для Eiffel, Java и Kotlin, и завершается фактическими результатами, демонстрирующими незрелость решений.

Ключевые слова: разыменование нулевого указателя; безопасность нулевых ссылок; безопасность пустых ссылок; инициализация объектов; статический анализ; эталонные тесты безопасности нулевых ссылок.

DOI: 10.15514/ISPRAS-2017-29(6)-7

Для цитирования: Когтенков А.В. Эталонные тесты безопасности нулевых ссылок при инициализации объекта. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 135-150 (на английском языке). DOI: 10.15514/ISPRAS-2017-29(6)-7

Список литературы

[1] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). Fifth Edition of a Recommendation. W3C, Nov. 2008. URL: http://www.w3.org/TR/2008/REC-xml-20081126/.

[2] Manuel Fähndrich and K. Rustan M. Leino. Declaring and Checking Non-null Types in an Object-oriented Language. In: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications. OOPSLA'03. ACM, 2003, pp. 302–312. DOI: 10.1145/949305.949332.

[3] Manuel Fähndrich and Songtao Xia. Establishing Object Invariants with Delayed Types. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications. OOPSLA'07. ACM, 2007, pp. 337–350. DOI: 10.1145/1297027.1297052.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., 1995.

[5] Alexander Kogtenkov. Practical Void Safety. In: Verified Software. Theories, Tools, and Experiments. 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22–23, 2017, Revised Selected Papers. Ed. by Andrei Paskevich and Thomas Wies. Vol. 10712. Lecture Notes in Computer Science. Springer International Publishing, 2017. DOI: 10.1007/978-3-319-72308-2_9.

[6] Alexander Kogtenkov. Towards null safety benchmarks for object initialization. In: Modeling and Analysis of Information Systems 24.6 (2017).

[7] A.V. Kogtenkov. Mechanically Proved Practical Local Null Safety. In: Trudy ISP RAN / Proc. ISP RAS, vol. 28, issue 5, pp. 27–54. DOI: 10.15514/ISPRAS-2016-28(5)-2.

[8] Mediator pattern. 2017. URL: https://en.wikipedia.org/wiki/Mediator_pattern (visited on 2017-11-20).

[9] Bertrand Meyer. Targeted expressions: safe object creation with void safety. July 30, 2012. URL: http://se.ethz.ch/~meyer/publications/online/targeted.pdf (visited on 2017-05-08).

[10] Xin Qi and Andrew C. Myers. Masked Types for Sound Object Initialization. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'09. ACM, 2009, pp. 53–65. DOI: 10.1145/1480881.1480890.

[11] Alexander J. Summers and Peter Müller. Freedom Before Commitment: A Lightweight Type System for Object Initialisation. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. OOPSLA'11. ACM, 2011, pp. 1013–1032. DOI: 10.1145/2048066.2048142.