Science of Computer Programming 221 (2022) 102844

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A Java typestate checker supporting inheritance

Lorenzo Bacchiani [a], Mario Bravetti [a,b], Marco Giunti [c], João Mota [c,*], António Ravara [c]

[a] University of Bologna, Italy
[b] Focus Team, INRIA, France
[c] NOVA LINCS and NOVA School of Science and Technology, Portugal

ARTICLE INFO

Article history:
Received 30 November 2021
Received in revised form 12 July 2022
Accepted 13 July 2022
Available online 20 July 2022

Keywords:
Behavioral types
Object-oriented programming
Subtyping
Type-checking
Typestates

ABSTRACT

Detecting programming errors in software is increasingly important, and building tools that help developers with this task is a crucial area of investigation on which the industry depends. Leveraging on the observation that in Object-Oriented Programming (*OOP*) it is natural to define stateful objects where the safe use of methods depends on their internal state, we present Java Typestate Checker (*JATYC*), a tool that verifies Java source code with respect to typestates. A typestate defines the object's states, the methods that can be called in each state, and the states resulting from the calls. The tool statically verifies that when a Java program runs: sequences of method calls obey to object's protocols; objects' protocols are completed; null-pointer exceptions are not raised; subclasses' instances respect the protocol of their superclasses. To the best of our knowledge, this is the first OOP tool that simultaneously tackles all these aspects.

The code (and data) in this article has been certified as Reproducible by Code Ocean: https://codeocean.com/. More information on the Reproducibility Badge Initiative is available at https://www.elsevier.com/physical-sciences-and-engineering/computer-science/journals.

* Corresponding author.
E-mail address: jd.mota@campus.fct.unl.pt (J. Mota).

https://doi.org/10.1016/j.scico.2022.102844

Software metadata

(Executable) software metadata description	
Current software version	3.0
Permanent link to executables of this version	https://github.com/jdmota/java-typestate-checker/releases/tag/3.0
Permanent link to Reproducible Capsule	https://codeocean.com/capsule/5619420/tree/v1
Legal Software License	MIT License
Computing platforms/Operating Systems	Linux, OS X, Microsoft Windows
Installation requirements & dependencies	Java 8, Checker Framework 3.14.0
Link to user manual	https://github.com/jdmota/java-typestate-checker/wiki/Documentation
Support email for questions	jd.mota@campus.fct.unl.pt

Code metadata

Code metadata description	
Current code version	3.0
Permanent link to code/repository used for this code version	https://github.com/ScienceofComputerProgramming/SCICO-D-21-00203
Permanent link to Reproducible Capsule	https://codeocean.com/capsule/5619420/tree/v1
Legal Code License	MIT License
Code versioning system used	git
Software code languages, tools, and services used	Java, Kotlin, Gradle, Checker Framework
Compilation requirements, operating environments & dependencies	Java 8
Developer documentation/manual	https://github.com/jdmota/java-typestate-checker/wiki/Documentation
Support email for questions	jd.mota@campus.fct.unl.pt

1. Introduction

Programming errors such as de-referencing null pointers [1], using objects wrongly (e.g. reading from a closed file; closing a socket that timed out[1]), or forgetting to free resources (e.g. not closing a file reader), result in programs that may malfunction or waste memory unnecessarily. It is, therefore, crucial to develop tools that assist the software development process by detecting these mistakes as early as possible since these bugs occur more often than one might think [2].

In programming languages, some common errors are detected thanks to type systems embedded in compilers [3]. Unfortunately, the subset of errors detected at compile-time in mainstream languages is still limited, and unsafe code that breaks the program's logic, or that could crash at runtime, can yet be compiled (and ran) in most programming technologies. In more detail, our approach avoids some uncaught errors and is motivated by the following observations on the practice of Object-Oriented Programming (*OOP*).

Most OOP languages, including Java, do not statically ensure that methods are called according to a specified protocol, like calling *hasNext* before calling *next* in an iterator. Usually, the protocol is specified in natural language in the documentation, but not statically enforced: this is a source of many errors, like accessing a variable that was not initialized [4]. While some language frameworks support a refined analysis, they require expert users to provide complex specifications, for example, in separation logic [5–7].

To overcome this limitation, tools such as *Mungo* [8] extend Java classes with **typestate** definitions [9] which specify the behavior of instances of those classes in terms of a state machine, and check if sequences of method calls happen in the order prescribed in the typestate [8].

To further enhance the static analysis and **avoid null-pointer exceptions** at runtime, one can use the *Nullness Checker*[2] plugin of the *Checker Framework* [10], a framework that supports adding type systems to the Java language. This plugin enhances the Java's type system so that types are non-nullable by default, except when declared with a *Nullable* annotation. Some modern languages, such as Kotlin, also distinguish non-null types from nullable types,[3] thus avoiding these exceptions. Nonetheless, the approach might give false alarms even when the code is safe,[4] requiring the programmer to provide additional checks that a value is not *null*, following a style known as *defensive programming*, or a number of annotations.[5]

Last but not least, any static analysis for OOP must take into consideration **inheritance** in order to be effective. This is challenging in the context of typestates: given that a class can inherit from another and be used as a type of the superclass, it is crucial to ensure that the behavior specified in the usage protocol of the superclass is also possible in its subclasses. Thus, we need a notion of **subtyping** for protocols, which are akin to session types [11,12] that is, behavioral types representing interactions in Service Oriented Computing [13,14]. Session types subtyping for synchronous communication was

[1] https://github.com/xetorthio/jedis/issues/1747.
[2] https://checkerframework.org/manual/#nullness-checker.
[3] https://kotlinlang.org/docs/null-safety.html.
[4] An example is available at https://tinyurl.com/yvdvrd5j.
[5] An example is available at https://tinyurl.com/5dpkzxdh.

```
1  public class BaseIterator {
2    private String[] items;
3    protected int index;
4    public BaseIterator(String[] items) {
5      this.items = items;
6      this.index = 0;
7    }
8    public boolean hasNext() {
9      return this.index < this.items.length;
10   }
11   public Object next() {
12     return this.items[this.index++];
13   }
14   public int remainingItems() {
15     return this.items.length - this.index;
16   }
17 }
```

Listing 1: *BaseIterator* class.

first studied by Gay and Hole [15]. Lange and Yoshida [16] design an algorithm to check if a session type is a subtype of another was implemented. Bacchiani et al. [17] develop a tool to generate simulation graphs for synchronous session subtyping. When applying synchronous session types subtyping to an OOP language, inputs are method calls and outputs are returned values. Checking for protocol subtyping is only the first step: as we will see in the next sections, we also need other mechanisms to correctly handle inheritance, method overriding and casting.

In this paper, we present *Java Typestate Checker (JATYC)* [18,19], a tool that type-checks Java source code where objects are associated with typestates. Java classes are annotated with typestates which specify the behavior of class instances in terms of available methods and state transitions. With JATYC, well-typed programs have the following properties: objects are used according to their protocols (typestates); protocols reach the *end* state (if the program terminates); null-pointer exceptions are not raised (in the code we can inspect). To ensure these properties, we follow the usual session type approach and force the **linear** use of objects associated with protocols.

JATYC is a new implementation of Mungo [8] that supports inheritance and adds critical features and fixes known issues, like assuming that a *continue* statement jumps to the beginning of the loop's body, thus skipping the condition expression [19], which may produce false negatives. It is freely distributed[6] and is implemented in Kotlin [20] as a plugin for the Checker Framework [10], which is actively maintained and well-integrated with the Java language and toolset.

The major contributions with respect to the current version of Mungo are:

- checking the **absence of null pointer errors**, which is critical to avoid the "The Billion Dollar Mistake" [1];
- checking that the **protocols of objects are completed**, i.e. protocols reach the *end* state, if the program terminates;
- support for **subtyping** thanks to a **synchronous subtyping algorithm**, inspired in the work by Gay and Hole [15], Lange and Yoshida [16], and Bacchiani et al. [17].

A previous version of the tool has been introduced by Mota et al. [18]; the present version enhances it by adding support for **subtyping**.

2. Protocol analysis and nullness checking

To motivate the need for JATYC, consider a *BaseIterator* Java class that allows one to iterate over items of an array. List. 1 shows an implementation.[7]

The intended protocol is defined implicitly by the sequences of method calls that are supported, and by the "states" reached via those calls. To use the *BaseIterator*, one must invoke the *hasNext* method before calling *next*, to ensure that there are remaining items to retrieve. If this contract is not followed, an *IndexOutOfBoundsException* will be thrown.

While the Java compiler accepts code that does not follow this contract, in the next section we will show how to enrich Java classes with *typestate annotations* that allow rejecting programs containing these kinds of behavioral errors at compile-time.

What JATYC allows one to do is to complement the code with protocol specifications and statically make sure these are respected and executed to completion (if the program terminates). Additionally, it ensures no null pointer errors are raised and checks subtyping compliance.

[6] https://github.com/jdmota/java-typestate-checker.
[7] The complete example is available at https://git.io/J1Fta.

```
1  typestate BaseIterator {
2    HasNext = {
3      boolean hasNext(): <true: Next, false: end>
4    }
5    Next = {
6      Object next(): HasNext
7    }
8  }
```

Listing 2: *BaseIterator* protocol.

```
1  import jatyc.lib.Typestate;
2  @Typestate("BaseIterator")
3  public class BaseIterator { ... }
```

Listing 3: *BaseIterator* class with *Typestate* annotation.

```
1  BaseIterator it = new BaseIterator(args);
2  while (!it.hasNext()) {
3    // Error: Cannot call next on state end
4    System.out.println(it.next());
5  }
6  // Error: [it] did not complete its protocol
```

Listing 4: *BaseIterator* use.

Protocol specifications. All instances of a Java class having a typestate are checked in order to enforce the prescribed behavior. The typestate specifications are written in *.protocol* files.[8] List. 2 presents the protocol for the *BaseIterator* (cf. List. 1). It specifies two states, *HasNext* and *Next*, and implicitly includes the *end* state, which is the final state. In the initial state *HasNext*, only the *hasNext* method is available to be called (line 3). If the method returns *true*, the state changes to *Next*; otherwise, the state changes to *end*, where no operations are allowed. In the *Next* state, we are allowed to call the *next* method (line 6), which changes the state to *HasNext*.

To associate a protocol with a Java class, one must include a *Typestate* annotation containing the (relative) path of the protocol file (List. 3). The *.protocol* extension is optional.

Protocol compliance and completion. JATYC ensures that instances of Java classes associated with a typestate obey to the corresponding protocol and reach the *end* state, so that potentially important method calls are not forgotten and resources are freed.

To see an example of incorrect use of *BaseIterator*, consider List. 4, where errors are indicated in the comments. According to the protocol (List. 2), we can only call *next* if the *hasNext* call returns *true*, but the code is doing the opposite (notice the negated condition in line 2). This results in an error stating that we cannot call *next* in state *end* and that the protocol was not completed. Removing the negation fixes both issues.

Nullness checking. Null pointer errors are the cause of most runtime exceptions in Java programs [4,21]: being able to detect these errors at compile-time is therefore crucial. Towards that direction, JATYC offers the following guarantees:

1. Types are non-null by default (contrary to Java's default type system[9]), method calls and field accesses are only performed on non-null types, as in the *Nullness Checker*,[10] and nullable types are marked with the *Nullable* annotation. Given that in the *BaseIterator* class, all the fields are initialized, no potential null-pointer errors are reported with respect to field accesses. Nonetheless, since there is no guarantee that the items of the array are non-null, we need to augment the *next* method with a *Nullable* annotation (List. 5).
2. False alarms regarding accesses to initialized nullable fields (in classes associated to protocols) are ruled out by taking into account that methods are only called in a specific order. For example, imagine that the underlying collection of the iterator was not provided in the constructor but instead via an *init* method that initializes the *items* field (initially marked as nullable).[11] If the protocol specifies that *init* should be called before any method, the tool can ensure that subsequent fields accesses will not raise null pointer errors, without the need for defensive programming (i.e. checking that *items != null*) or additional annotations.

[8] The complete grammar is available at https://git.io/JtMu3.
[9] The fact that null is a value of any type is the source of Java not being type safe. [22].
[10] https://checkerframework.org/manual/#nullness-checker.
[11] A complete example is available at https://git.io/J1FtI.

```
import jatyc.lib.Nullable;
...
public @Nullable Object next() {
  return this.items[this.index++];
}
```

Listing 5: *Nullable* annotation.

```
typestate RemovableIterator {
  HasNext = {
    boolean hasNext(): <true: Next, false: end>
  }
  Next = {
    Object next(): Remove
  }
  Remove = {
    boolean hasNext(): <true: Next, false: end>,
    void remove(): HasNext
  }
}
```

Listing 6: *RemovableIterator* protocol.

3. Inheritance

To add support for *inheritance*, JATYC needs to be endowed with a subtyping algorithm to ensure the protocol of subclasses complies with the protocol of superclasses. Besides the implementation of such an algorithm, the support for these concepts also requires one to deal correctly with *method overriding* and *casting*.

3.1. Synchronous subtyping algorithm

The algorithm for supporting protocol subtyping takes inspiration from the synchronous subtyping one for session types [16,17]. It builds graphs from the protocols to be checked, traverses them by firing common input/output operations and marks each encountered pair of states. Notice that, in our setting, input operations are represented by method calls, while output operations by values returned by them. We mark pairs of states if: (*i*) both are input states and *input contravariance* holds, i.e. the subtype can perform a set of input operations greater or equal to the one of the supertype, (*ii*) both are output states and *output covariance* holds, i.e. the supertype can perform a set of output operations greater or equal to the one of the subtype, (*iii*) both states are *end* states. The algorithm stops when either all reachable pairs have been marked (subtyping holds) or a pair of states does not satisfy any of the above conditions (subtyping does not hold). Since Java does not support inheritance in enumeration classes, we have to consider all enumeration values as potentially returnable and, consequently, all should be included in the protocol. Because of this, output covariance always holds (in our setting all outputs are invariant).

For example, consider the *RemovableIterator* protocol presented in List. 6. It extends the *BaseIterator* protocol (List. 2) by adding the state *Remove* with the new method *remove*. Thanks to the algorithm described above, we can safely state that the *RemovableIterator* is a subtype of the *BaseIterator*. In particular, the *Remove* state respects input contravariance with respect to the *HasNext* state in the supertype.

3.2. Method inheritance

Inheritance makes it possible to reuse methods from superclasses, override some, or add new ones. For example, the code presented in List. 7 shows an example of safe class inheritance: the *RemovableIterator* class implements the protocol presented in List. 2 and extends the *BaseIterator* class. In particular, the *RemovableIterator* class, differently from the *BaseIterator*, uses a **Java List** as the underlying data structure and consequently all the methods need to be overridden in order to access the collection. Moreover, according to its protocol, the class presented in List. 7 implements the new *remove* method.

To correctly support inheritance, we consider the following cases:

1. A class without protocol extending a class without protocol.
2. A class with protocol extending a class with protocol.
3. A class without protocol extending a class with protocol.
4. A class with protocol extending a class without protocol.

The first and second scenarios are trivially handled: the former does not require any inspection since classes by default do not have protocols; and the latter is checked using the subtyping algorithm. In the third case, we let the use of overridden

```
import java.util.*;
import jatyc.lib.*;

@Typestate("RemovableIterator")
public class RemovableIterator extends BaseIterator {
  protected List<Object> items;
  public RemovableIterator(String[] items) {
    super(items);
    this.items = Util.toList(items);
  }
  public boolean hasNext() {
    return this.index < this.items.size();
  }
  public @Nullable Object next() {
    return this.items.get(this.index++);
  }
  public void remove() {
    this.items.remove(--this.index);
  }
  public int remainingItems() {
    return this.items.size() - this.index;
  }
}
```

Listing 7: *RemovableIterator* class.

```
public static void main(String[] args) {
  BaseIterator it = new RemovableIterator(args);
  RemovableIterator rIt =
    (RemovableIterator) iterate(it);
  System.out.printf("Left:%d\n", rIt.remainingItems());
}

public static BaseIterator iterate(@Requires("HasNext") BaseIterator it) {
  while (it.hasNext()) {
    System.out.printf("Item:%s\n", it.next());
  }
  return it;
}
```

Listing 8: *Polymorphic* code example.

methods to be governed by the inherited protocol and newly added methods are treated as *anytime* methods, which can be called at any moment. In the fourth case, we consider all methods in the superclass as *anytime* methods and enforce that these remain so in the subclasses, which implies that these methods cannot be included in protocols of subclasses. Note that any method that does not appear in a protocol is considered an *anytime* method, like the *remainingItems* method in the *BaseIterator* class (List. 1). To ensure safety, *anytime* methods currently can only perform read operations or call other *anytime* methods.

3.3. Casting

The support for inheritance opens the door to *polymorphism* and the need to handle *casting*. For example, consider the code in List. 8.

In a nutshell, we create a *RemovableIterator* object and assign it to a variable of type *BaseIterator*, thus performing an up-cast, and then we pass it to the *iterate* method. This method iterates over all items and returns a *BaseIterator* in the *end* state. Finally, we perform a down-cast and we call the *anytime* method *remainingItems*, which returns zero. Notice that the method *iterate* makes use of the *Requires* annotation from the *jatyc.lib* package, which indicates in which states the object pointed by the parameter is expected to be in.

Casting operations are currently only allowed in the beginning of the protocol (before any method is called) or at the end of the protocol. This limitation is caused by the fact that states belonging to subclasses and superclasses could possibly be in a *many-to-many* relation. For example, it turns out that the *HasNext* state of the *BaseIterator* (List. 2) is in relation with both *HasNext* and *Remove* states of the *RemovableIterator* (List. 6). If we downcast from *BaseIterator* to *RemovableIterator*, we do not know to which state we should cast to (*HasNext* or *Remove*).

```
typestate Iterator {
  HasNext = {
    boolean hasNext(): <true: Next, false: end>,
    drop: end // This marks a state as droppable
  }
  Next = { String next(): HasNext }
}
```

Listing 9: Example of droppable states.

4. Future work

In this paper, we present a subtyping endowed version of JATYC, a tool that makes it possible to run programs with the following properties: (*i*) objects are used according to their protocols (typestates); (*ii*) protocols reach the *end* state (if the program terminates); (*iii*) null-pointer exceptions are not raised (in the code we can inspect). The support for subtyping, albeit with the limitations described, allows the use of fundamental OOP concepts like polymorphism and inheritance.

As future work, we would like to also consider *droppable states* [19] in the subtyping analysis. These states are declared by including the special transition *drop: end*, thus specifying that an object may stop to be used in that state. For example, we can specify an iterator protocol that does not need to reach the *end* state (List. 9). Droppable states require relaxing the subtyping algorithm: we will work on a proper formalization of the current algorithm to correctly deal with droppable states.

Another future direction is to improve the management of polymorphism: currently up/down-castings are only allowed in the beginning of the protocol (before any method is called) or at the end of the protocol. Our goal is to implement a correct algorithm for typestate-mapping that makes it possible to perform castings no matter which state the object to be casted is in.

CRediT authorship contribution statement

Lorenzo Bacchiani: Conceptualization, Investigation, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Mario Bravetti:** Conceptualization, Investigation, Methodology, Supervision, Validation, Writing – original draft, Writing – review & editing. **Marco Giunti:** Conceptualization, Investigation, Methodology, Supervision, Validation, Writing – original draft, Writing – review & editing. **João Mota:** Conceptualization, Investigation, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **António Ravara:** Conceptualization, Investigation, Methodology, Supervision, Validation, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was partially supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No. 778233 (BehAPI) and by NOVA LINCS (UIDB/04516/2020 and UIDB/04516/2020/TRA/BIM/07) via the Portuguese Fundação para a Ciência e a Tecnologia.

References

[1] T. Hoare, Null references: the billion dollar mistake, https://tinyurl.com/eyipowm4, 2009, presentation at QCon London.
[2] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, C. Zhai, Bug characteristics in open source software, Empir. Softw. Eng. 19 (2014) 1665–1705, https://doi.org/10.1007/s10664-013-9258-8.
[3] L. Cardelli, Type systems, ACM Comput. Surv. 28 (1996) 263–264, https://doi.org/10.1145/234313.234418.
[4] N.E. Beckman, D. Kim, J. Aldrich, An empirical study of object protocols in the wild, in: Proc. of European Conference on Object-Oriented Programming (ECOOP), Springer, 2011, pp. 2–26.
[5] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, F. Piessens, VeriFast: a powerful, sound, predictable, fast verifier for C and Java, in: Proc. of NASA Formal Methods (NFM), Springer, 2011, pp. 41–55.
[6] J.C. Reynolds, Separation logic: a logic for shared mutable data structures, in: Proc. of Logic in Computer Science (LICS), IEEE, 2002, pp. 55–74.
[7] S.S. Ishtiaq, P.W. O'hearn, BI as an assertion language for mutable data structures, in: Proc. of Principles of Programming Languages (POPL), 2001, pp. 14–26.
[8] D. Kouzapas, O. Dardha, R. Perera, S.J. Gay, Typechecking protocols with Mungo and StMungo, in: Proc. of Principles and Practice of Declarative Programming (PPDP), ACM, 2016, pp. 146–159.
[9] R. Garcia, É. Tanter, R. Wolff, J. Aldrich, Foundations of typestate-oriented programming, ACM Trans. Program. Lang. Syst. 36 (2014) 12, https://doi.org/10.1145/2629609.
[10] M.M. Papi, M. Ali, T.L. Correa Jr, J.H. Perkins, M.D. Ernst, Practical pluggable types for Java, in: Proc. of Software Testing and Analysis (ISSTA), 2008, pp. 201–212.

[11] K. Honda, V.T. Vasconcelos, M. Kubo, Language primitives and type discipline for structured communication-based programming, in: Proceedings of Programming Languages and Systems - ESOP'98, in: Lecture Notes in Computer Science, vol. 1381, Springer, 1998, pp. 122–138.
[12] H. Hüttel, I. Lanese, V.T. Vasconcelos, L. Caires, M. Carbone, P. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H.T. Vieira, G. Zavattaro, Foundations of session types and behavioural contracts, ACM Comput. Surv. 49 (2016) 3, https://doi.org/10.1145/2873052.
[13] L. Cruz-Filipe, I. Lanese, F. Martins, A. Ravara, V.T. Vasconcelos, The stream-based service-centred calculus: a foundation for service-oriented programming, Form. Asp. Comput. 26 (2014) 865–918, https://doi.org/10.1007/s00165-013-0284-5.
[14] M. Boreale, R. Bruni, L. Caires, R.D. Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V.T. Vasconcelos, G. Zavattaro, SCC: a service centered calculus, in: Proceedings of Web Services and Formal Methods (WS-FM), in: Lecture Notes in Computer Science, vol. 4184, Springer, 2006, pp. 38–57.
[15] S.J. Gay, M. Hole, Types and subtypes for client-server interactions, in: Proc. of Programming Languages and Systems (ESOP), in: Lecture Notes in Computer Science, vol. 1576, Springer, 1999, pp. 74–90.
[16] J. Lange, N. Yoshida, Characteristic formulae for session types, in: Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS), in: Lecture Notes in Computer Science, vol. 9636, Springer, 2016, pp. 833–850.
[17] L. Bacchiani, M. Bravetti, J. Lange, G. Zavattaro, A session subtyping tool, in: Proc. of Coordination Models and Languages (COORDINATION), in: Lecture Notes in Computer Science, vol. 12717, Springer, 2021, pp. 90–105.
[18] J. Mota, M. Giunti, A. Ravara, Java typestate checker, in: Proc. of Coordination Models and Languages (COORDINATION), in: Lecture Notes in Computer Science, vol. 12717, Springer, 2021, pp. 121–133.
[19] J. Mota, Coping with the reality: adding crucial features to a typestate-oriented language, Master's thesis, NOVA School of Science and Technology, 2021, http://hdl.handle.net/10362/125329.
[20] D. Jemerov, S. Isakova, Kotlin in Action, Manning Publications Company, 2017.
[21] J. Sunshine, Protocol programmability, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2013.
[22] N. Amin, R. Tate, Java and Scala's type systems are unsound: the existential crisis of null pointers, ACM SIGPLAN Not. 51 (2016) 838–848, https://doi.org/10.1145/3022671.2984004.