Verifying the Option Type with Rely–Guarantee Reasoning

James Yoo
jmsy@cs.washington.edu
University of Washington
Seattle, Washington, USA

Michael D. Ernst
mernst@cs.washington.edu
University of Washington
Seattle, Washington, USA

René Just
rjust@cs.washington.edu
University of Washington
Seattle, Washington, USA

ABSTRACT

Many programming languages include an implementation of the option type, which encodes the absence or presence of values. *Incorrect* use of the option type results in run-time errors, and *unstylistic* use results in unnecessary code. Researchers and practitioners have tried to mitigate the pitfalls of the option type, but have yet to evaluate tools for enforcing correctness and good style.

To address problems of correctness, we developed two modular verifiers that cooperate via a novel form of rely–guarantee reasoning; together, they verify use of the option type. We implemented them in the Optional Checker, an open-source static verifier. The Optional Checker is the first verifier for the option type based on a sound theory — that is, it issues a compile-time guarantee of the absence of run-time errors related to misuse of the option type. We then conducted the first mechanized study of tools that aim to prevent run-time errors related to the option type. We compared the performance of the Optional Checker, SpotBugs, Error Prone, and IntelliJ IDEA over 1M non-comment, non-blank lines of code. The Optional Checker found 13 previously-undiscovered bugs (a superset of those found by all other tools) and had the highest precision at 93%.

To address problems of style, we conducted a literature review of best practices for the option type. We discovered widely varying opinions about proper style. We implemented linting rules in the Optional Checker and discovered hundreds of violations of the style recommended by Oracle, including in 11% of JDK files that use `Optional`. Some of these were objectively bad code, and others reflected different styles.

KEYWORDS

Pluggable type systems, static analysis, option type, rely–guarantee reasoning

1 INTRODUCTION

An option value is either *present* (containing a value) or *absent* (not containing a value). This concept appears in most modern programming languages. Option values go by many names, such as *Some*/*None* in OCaml, Rust, and Scala, or *Just*/*Nothing* in Haskell. It is an error to access an absent option value. Languages take two approaches regarding such errors.

Some languages require programmers to check the presence of every option value before accessing it. In Haskell, the canonical way to access an option value (`Maybe`) is via pattern matching, with present (`Just`) and absent (`Nothing`) cases. Here is example code from Pandoc [29], a markup language processor:

```
escaped :: CSVOptions -> Parser Char
escaped opts =
  case csvEscape opts of
    Nothing ->
      case csvQuote opts of
        Nothing -> mzero
        Just q -> try $ char q >> char q
    Just c -> try $ char c >> noneOf "\r\n"
```

Significant boilerplate code is required (i.e., the nested pattern-matching structure), even if the programmer knows the `Maybe` value is `Just` (and not `Nothing`). Other functional programming languages are similar to Haskell, including Agda [4], Coq [54], Elm [26], F# [79], Idris [22], OCaml [67], Standard ML [80], and Zig [1]. These languages provide escape hatches that programmers can use to avoid pattern-matching, but they are considered non-idiomatic and may crash at run time.

Other languages, such as C++ [55], Java [2], Nim [84], Rust [106], Scala [64], and Swift [8], avoid boilerplate code by not forcing a program to check for the presence of an option value before unwrapping it. To avoid run-time errors, the programmer must ensure that only present option values are accessed. Such reasoning is complex, tedious, and easy to overlook. When applied naively, it results in an explosion of unreachable boilerplate code.

This paper presents a way to get the best of both worlds: a compile-time guarantee of safety without boilerplate or awkward code, such as the need for eta-expansion [85]. We have developed two modular verifiers that cooperate via a novel form of rely–guarantee reasoning [59] to provide a compile-time guarantee that no absent option value is ever accessed. We have implemented our type systems in the Optional Checker, an open-source formal verification tool and linter for the Java `Optional` type. While our experiments target Java code, the type systems and findings are broadly applicable.

In addition to the need to avoid run-time errors, the option type is subject to style rules (section 3.2) that encourage best practices. It is burdensome for programmers to remember these rules. The Optional Checker automatically checks these rules. If the Optional Checker issues no warnings, the program follows best practice in use of the `Optional` type. We used this feature of the Optional Checker to evaluate the extent to which the Java programming community accepts and follows the style guidelines.

We evaluated the Optional Checker on over 1M non-comment, non-blank lines of code. The Optional Checker uncovered 13 previously unknown defects, 7 of which have been patched to-date. Additionally, we evaluated the Optional Checker against three

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10.
https://doi.org/10.1145/3691620.3695036

other tools that warn about misuse of `Optional`: SpotBugs,Error Prone [41], and the IntelliJ IDEA IDE [58]. The Optional Checker is more precise than the other tools (i.e., it issues fewer false positive warnings), despite being the only sound tool (i.e., it detects all possible `NoSuchElementExceptions` resulting from accessing an absent `Optional` value).

Our primary contributions are:

- A literature review of style guidelines regarding the use of option types, focusing on the Java `Optional` type (section 3).
- Two modular, sound type systems that cooperate via a novel form of rely–guarantee reasoning (section 4).
- An open-source implementation, the Optional Checker. It is a *verifier* that certifies the absence of `NoSuchElementExceptions`. It is also a *linter* that enforces style guidelines (section 5).
- An experimental evaluation of *correctness*. It compares tools that detect errors related to the use of option types (section 7).
- An empirical investigation of *style*: how programmers use the Java `Optional` type in practice (section 8).

The empirical studies use over 1M non-comment, non-blank lines of open-source code (section 6).

2 BACKGROUND

2.1 The Java `Optional` Type

An instance of Java's `Optional` type is a wrapper object that may be absent (representing a null value) or present (representing a non-null value). Accessing an absent `Optional` value at run time results in a `NoSuchElementException`. A search on GitHub for "Optional" "NoSuchElementException" yielded 2,830 results, illustrating that it is an important and common issue.

The `Optional` class defines three constructors:

- `empty()` for creating an absent `Optional`.
- `of(T value)` for creating a present `Optional` containing the given non-null value.
- `ofNullable(T value)` for creating an `Optional` that is absent if the value is null, otherwise is present with the given value.

The parameter type `T` is a generic type argument.

`Optional` defines methods to check for the presence of an `Optional` and to access the wrapped value:

- `isPresent()` returns true if and only if the `Optional` is present.
- `get()` returns the wrapped value if the `Optional` is present, or throws a `NoSuchElementException` if the `Optional` is absent.

The `Optional` API also includes 12 other methods that use `get()` and `isPresent()` internally, such as `orElseThrow()`.

If a programmer is unsure whether an instance of `Optional` is present, the code should check before attempting to unwrap its value. (Most of `Optional`'s methods do this internally.) Consider the following code from Smithy [108], a domain-specific language for defining client–server interfaces.

```
Optional<OperationShape> untagApi = ...
if (untagApi.isPresent()) {
   untagApiVerified = verifyUntagApi(untagApi.get(), ...);
}
```

The expression `untagApi.get()` is executed only if the call to `isPresent()` returns true.

2.2 `Optional` vs. null

In a typical usage scenario, a programmer might refactor a codebase to replace a nullable value with an instance of `Optional`. However, this translation does not eliminate any potential errors without additional programmer effort. Every `NullPointerException` is refactored into a `NoSuchElementException`, which crashes the program just the same. Our verification tool, the Optional Checker, guarantees that no `NoSuchElementException` is introduced.

The designers of `Optional` acknowledge that `Optional` is inherently no more or less safe than null. The key motivation for `Optional` was to prevent programmers from forgetting to handle all possible cases [88]. Use of references "makes it disturbingly easy to simply forget" null checks [9, 45], but the designers believed that the extra syntax required by `Optional` makes developers less likely to forget that two cases exist. The designers did not consider the use of a tool, like the Optional Checker, to prevent programmers from forgetting [93].

In languages such as C++, Java, Nim, Rust, Scala, and Swift, an `Optional` reference can be null. Every use of `Optional` has the possibility of throwing both `NoSuchElementException` *and* `NullPointerException`. This is ironic, given that the goal of `Optional` is to avoid `NullPointerExceptions`. However, tools already exist to prevent null-pointer dereferences [10, 13, 34, 91, 97, 116], and a user could use one of those tools along with the Optional Checker to prevent both types of exception. Therefore, `NullPointerExceptions` are out of scope for this paper.

2.3 Pluggable Type-Checking in Java

Type-checking is a static verification technique that is just as powerful as abstract interpretation or dataflow analysis [24]. Type-checking is a specify-and-verify technique: the programmer writes a specification of code behaviour, and a tool verifies that the code satisfies the specification. Pluggable type-checking [12] permits a programmer to choose which type systems are run on a program. Pluggable type-checking is widely adopted in industry, with usage documented at Amazon [61, 62, 126], Google [107], Meta [98], and Uber [10].

A type qualifier [37] refines a type, restricting its possible run-time values. Consider the declaration `String myRegex;`. A standard Java compiler will admit both `myRegex = "[0-9]"` and `myRegex = "[0"`. The latter assignment is a defect that might lead to a crash or other misbehaviour later in the program. To prevent such defects, a programmer might instead write `@Regex String myRegex;` [113], where "`@Regex String`" is a type comprising a type qualifier `@Regex` and a Java basetype `String`. The `@Regex` type qualifier refines the type of the `String` type into one that represents fewer values: the set of valid regular expressions. The type `@Regex String` is a subtype of `String`.

A Java compiler plug-in can enforce the semantics of the `@Regex` type system. For example, the plug-in would forbid the second assignment in "`String s = ...; @Regex String myRegex = s;`". As another example, the formal parameter of `Pattern.compile()` has declared type `@Regex String`, and so the plug-in would issue a warning at the call `Pattern.compile("[0")`.

If the regex plug-in discovers no violations of the type system, then the programmer has a compile-time guarantee that the program is free from run-time errors stemming from improper use of regular expressions.

Our implementation, the Optional Checker, is a verification tool for the `Optional` type. Its soundness statement is: if a program type-checks, then the program will never access an empty instance of the option type. Section 4.1.6 contains a proof.

3 GOOD STYLE WITH THE `OPTIONAL` TYPE

We performed a literature review via the snowball methodology. We started with searches for "optional" "Java" and created more searches that add "advice", "style", "guideline". We examined the first 1000 hits for each of these searches, and also followed relevant citations in the hits. We retained only documents that contain substantive original text supporting its guidelines. This yielded over 40 results (fig. 1).

Oracle provided no official usage advice about `Optional` when it was introduced in Java 8 [2]. Four and a half years later, the Java 11 documentation [88] for `Optional` stated, "`Optional` is primarily intended for use as a method return type where there is a clear need to represent 'no result,' and where using null is likely to cause errors."

By that time, developers and researchers had developed many, often conflicting, opinions about the effective use of `Optional`. In fig. 1, for nearly every guideline, there is a different author who argues for the exact opposite! We conclude that there is no consensus on the best way to use `Optional`, and quite a few authors (including of code in the JDK, see section 8.2) use `Optional` differently than its authors do.

One problem with Oracle's guideline is inconsistency: it recommends `Optional` only for a few method return types, but null pointer exceptions are harmful everywhere. Therefore, many authors propose expanding use of `Optional`. This viewpoint is also clear in practice (section 8). Another problem with Oracle's guideline is vagueness, with words like "primarily" and "likely".

We speculate that another reason for the discrepancies is that some authors prefer a more procedural programming style, and others prefer a more functional style. This could explain the conflicting advice about calling methods in the `Optional` class.

Absolutism:
Never use Optional [32, 47]
Use Optional everywhere, instead of null [92]
Null:
Don't use null for an Optional variable or return value [15, 42, 57, 63, 66, 70, 88, 89, 111, 114, 119, 124]
Don't check an Optional against null [89]
The content of a present (non-absent) Optional must be non-null [88]
Return values:
Use Optional for *all* return types [87, 110]
Use Optional only for return types that might be null [9, 70, 88, 122]
Use Optional for return types in public APIs [19, 20, 72, 88]
Don't use Optional for return types [15, 115]
Getters:
Don't use Optional for getters [87]
Use Optional only for getters [63]
Formal parameters:
Don't use Optional for parameters [9, 15, 42, 56, 57, 63, 66, 68, 70, 73, 81, 87–89, 95, 96, 111, 120–122]
Permit Optional for parameters [123]
Use Optional for all parameters [110]
Fields:
Don't use Optional for fields [19, 20, 56, 57, 63, 66, 68, 70, 81, 88, 95, 96, 120–122]
Use Optional for all fields [110]
Permit Optional for fields [35, 123]
Collections:
Don't use Collection<Optional<T>> [66, 68, 70, 73, 88, 89]
Don't use Optional<Collection<T>> [15, 57, 63, 66, 70, 87, 89, 120]
Collection<Optional<T>> is permissible [101]
Don't use Optional<Optional<T>> [42, 70]
Methods in the `Optional` class:
Never call isPresent() before get() [15, 66, 73, 96, 119]
Always call isPresent() before get() [57, 89, 111]
Don't use get() [70, 95, 119]
Don't use isPresent() [20, 57, 70]
Use get() only if the Optional is known to be present [66, 70, 96]
Don't use orElse() [66, 89, 119]
Don't use map() if the value is unused, use ifPresent() instead [42]
Local computations:
Don't create and consume an Optional locally [19, 66, 70, 122]
Equality and identity:
Don't use identity operations: ==, synchronization, etc. [42, 66, 81, 120]
Don't use equals() or hashCode() on Optional [124]
Use equals() and hashCode() on Optional [42, 66]

Figure 1: Style guidance about use of Java's `Optional` type.

3.1 Specialized Operations for Option Types

Introducing specialized syntax and operations for option types is one way to address the trade-off between safety and boilerplate code. Examples include the methods noted in section 2.1 and, more generally, monadic operators that return option values via composable "pipelines". These pipelines defer all presence checking to the end of a computation, so that it only has to be done once. Here is an example in Scala:

```
val optReport = for {
    user <- getOptUser(id)
    _ <- authUser(user)
    transactionId <- performTransaction(user)
} yield generateReport(transactionId)
```

This sort of code makes problems harder to localize, because an absent value that appears at the end of the pipeline might have been generated anywhere within — in this case, within any of `getOptUser(id)`, `authUser(user)`, or `performTransaction(user)`.

3.2 Marks' Style Rules

We implemented style rule checking in the Optional Checker. We chose to check the style rules proposed by Stuart Marks [70], a widely-respected authority on the Java `Optional` type who worked on its design and implementation and has given many talks about its idiomatic use [71]. Moreover, his rules cover the largest set of categories in fig. 1, making it the single best set of style rules to implement.

(1) Never, ever, use null for an `Optional` variable or return value.
(2) Never use `Optional.get()` unless you can prove that the `Optional` is present.
(3) Prefer alternative APIs over `Optional.isPresent()` and `Optional.get()`.

(4) It's generally a bad idea to create an `Optional` for the specific purpose of chaining methods from it to get a value.
(5) If an `Optional` chain has a nested Optional chain, or has an intermediate result of `Optional<Optional<T>>`, it's probably too complex.
(6) Avoid using `Optional` in (a) collections, (b) method parameters, and (c) fields.
(7) Don't use an `Optional` to wrap any collection type (`List`, `Set`, `Map`).

The Optional Checker checks these guidelines (they can be disabled individually) and it can be extended to enforce other guidelines. Here are notes on these guidelines:

(1) The Optional Checker performs a best-effort check of this property. For a guarantee of no `NullPointerExceptions`, the programmer should use a specialized nullness-checking tool [10, 13, 34, 91, 97, 116].
(2) This is not a style rule. It is the correctness property that guarantees that no use of `Optional` causes `NoSuchElementException`.
(3) Marks prefers other APIs in the `Optional` class. An example is `o.orElse(default)`, which evaluates to `o`'s value if present, and otherwise evaluates to the given value `default`. Another example is `o.map(fn)`, which evaluates to absent if `o` is absent, and otherwise is equivalent to `Optional.ofNullable(fn(o.get()))`. But Marks notes that sometimes, use of `get()` and `isPresent()` is the best choice.
(4) Creating then consuming an `Optional` is both more verbose and more expensive (in time and space) than using the original nullable value.
(5) Prefer simple, straightforward, readable code.
(6) `Collection<Optional T>` should be just `Collection<T>`: don't put absent `Optionals` in a collection. Fields are generally private and used within a limited scope, where their uses can be locally checked and `Optional` provides limited benefit. `Optional` in method parameters imposes unnecessary clutter and overhead on every caller and on the method body. Use of references rather than `Optional` requires the user to check for null, but forgetting a null check in the method body is unlikely, since the method's contract is stated in its documentation.
(7) Use an empty collection to represent the absence of values.

4 VERIFYING CORRECT USE OF OPTIONAL

To verify correct use of the option type, we designed two modular type systems cooperating via partial rely–guarantee reasoning. Section 4.1 presents a type system for modelling present option values, and section 4.2 presents a type system for modelling non-empty collections. These type systems cooperate via a novel form of rely–guarantee reasoning [59], described in section 4.3.

The type systems are fully specified by the type hierarchies given in this paper. Both are instantiations of qualifier-based type systems. Section 4.1.6 gives a proof of soundness.

We chose to create two simple type systems plus an interaction mechanism. It would be possible to create a single monolithic analysis, but that would be much harder to write and understand. The current design enforces separation of concerns and permits reuse and recombination of both the type systems and the composition mechanism.

4.1 A Type System for Option Values

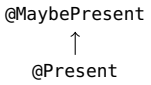


Figure 2: The type hierarchy for possibly-present data.

The *Optional* type system (fig. 2) contains these type qualifiers:

- `@MaybePresent`: the top type, which denotes a possibly-present instance of `Optional`. This type includes absent `Optional` values, present `Optional` values, and `null`. This is the default type (that is, `@MaybePresent Optional` is equivalent to unqualified `Optional`), so programmers need not explicitly write it.
- `@Present`: denotes a definitely-present instance of `Optional`. The type also includes the null value. Section 2.2 discussed how to handle the possibility of `NullPointerExceptions`.

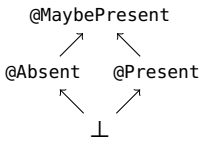
4.1.1 Explicitly-written `@Present` type qualifiers. A programmer can write the type `@Present Optional`. In most cases, doing so is poor style. If an `Optional` value is known to be present, then it is better style to use the non-null value instead.

However, writing `@Present` can be necessary when interfacing with existing code that uses the `Optional` type. An example, which we observed multiple times in our experiments, is when a superclass specifies a method as returning `@MaybePresent Optional` but some subclasses override the method to return `@Present Optional`.

4.1.2 Method Specifications. The Optional Checker differs from ordinary type systems in that it supports method pre- and post-conditions. Its annotations are more like a *specification language* than merely a type system. Its method specifications (which are all soundly checked, not trusted) are implemented as annotations parameterized by Java expressions, and include:

- `@RequiresPresent`: the method contract requires that the given expression evaluates to a present `Optional` on method entry.
- `@EnsuresPresent`: if the method terminates successfully, the given expression evaluates to a present `Optional`.
- `@EnsuresPresentIf`: a conditional method postcondition specification stating that the presence or absence of some expression depends on the return value of the annotated method.

4.1.3 No Representation for Absent Option Values. The type qualifier hierarchies in figs. 2 and 3 lack an `@Absent` qualifier, as shown at right. A reason is that the typing rules for `@Absent` would be identical to those of `@MaybePresent` and `@UnknownNonEmpty`: operations that lead to exceptions (e.g., `get()`) would still be forbidden.



4.1.4 Flow-Sensitive Type Refinement. The Optional Checker is unlike a traditional type system (but like a dataflow analysis or abstract interpretation [5, 6, 25, 60]) in that an expression's type is flow-sensitive. An expression may have different types on different lines of code. Declarations are respected: the refined type is always equal to or a subtype of its declared type.

4.1.5 Lambda Expressions. Standard flow-sensitive type refinement for conditional statements (section 4.1.4) is insufficient for some features of `Optional`. Consider the method `ifPresent(Consumer action)`. (Method `ifPresentOrElse()` is similar.) In the expression `opt.ifPresent(f)`, if `opt` is present, the function `f` is called on the value that is wrapped by `opt`. Within the body of `f`, two facts are true: `opt` is present, and the argument to `f` is a non-null non-optional value.

This feature also enhances other verifiers beyond the Optional Checker. Consider Jodd's method `StringUtil.ifNotNull()` [28]. This method takes a function as an argument. If the function is called, then (1) the value passed to the function is non-null, and (2) some other argument to the method is non-null. This is now expressible, improving the precision of nullness checking [91].

4.1.6 Soundness Considerations. We present a proof sketch of the soundness of our type system for option values, which we implement via the Optional Checker.

Theorem. If a program type-checks under the Optional Checker's type system, then the program will not throw a run-time exception that results from accessing an empty instance of the option type.

Proofs of progress and preservation in our type system can be adapted from prior work on qualifier-based type systems [17, 37–39, 127], but we provide a proof sketch here. Assume the type-checker issues no error.

(1) From the assumption, the run-time type of every expression is a subtype of (or equal to) its compile-time type.
(2) The compile-time type of the receiver of `get()` is `@Present`.
(3) By (1) and (2), every value that any receiver of `get()` evaluates to is present; it is never absent.
(4) Since `get()` is never invoked on an absent value, `get()` never throws a `NoSuchElementException`.

A proof sketch for our type system for non-empty containers (section 4.2) follows the same structure, with its qualifiers replacing that of the type system for option values.

We have no proof of soundness for our implementation. Its trusted computing base includes the following. (1) The implementation of the Optional Checker and the JDK annotations. These were written by one author and reviewed by two additional authors. (2) The implementation of the Checker Framework, which is used in production at many companies. (3) Code used by the Checker Framework, such as javac and the JDK implementation.

4.2 A Type System for Non-Empty Containers

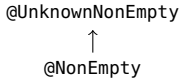


Figure 3: The type hierarchy for container types.

The *Non-Empty* type system (fig. 3) models containers (e.g., lists, sets, maps, iterators, streams) that may or may not be empty. Its type qualifiers consist of:

- `@UnknownNonEmpty`: the top type, which denotes a container value that may or may not contain elements. This is the default type, so programmers need not explicitly write it.
- `@NonEmpty`: denotes a container value that *definitely* contains at least one element.

Many aspects of this type system mirror those of the Optional type system. It recognizes that `new ArrayList()` and `List.of()` create an empty collection, but `List.of(1, 2, 3)` creates a non-empty collection. Methods such as `add()` are annotated with the `@EnsuresNonEmpty` method specification, so after calling `lst.add(x)`, `lst` is `@NonEmpty`. It accounts for side effects through local aliases.

Additional features, such as method specifications (e.g., `@EnsuresNonEmpty`) exist for the Non-Empty type system, mirroring those described for the Optional type system in section 4.1.2.

The Non-Empty type system is effective; a verifier implementing the type system detected two previously-unknown defects [133, 134] resulting in run-time exceptions in plume-util [33], a project comprising 11K non-comment, non-blank lines of code and 5K NCNB lines of tests.

4.3 Partial Rely–Guarantee Reasoning

As described so far, the Optional verifier issues a false positive warning for some programs. As an example, the `Stream.max()` method returns an `Optional` value that is present if and only if the stream is non-empty. Therefore, the following call to `Optional.get()` is safe:

```
List<Integer> numbers = List.of(1, 2, 3);
Optional<Integer> maximum = numbers.stream()
    .max(Integer::compareTo);
maximum.get() // False positive warning: `maximum` is @MaybePresent
```

The verifier issues a false positive warning because the return type of `Stream.max()` is conservatively inferred as `@MaybePresent`, because in general `max()` might return an absent value.

By *relying* on information from Non-Empty types, the Optional verifier can determine that the type of `maximum` is `@Present`. The relied-upon information must itself be verified (*guaranteed*), which happens when the Non-Empty type system runs. This separation of concerns enables modular verification and is the heart of rely–guarantee reasoning.

In typical formulations of modular verification and rely–guarantee reasoning, each verifier is run on the full program. That is impractical for the Optional Checker. Forcing the user to verify the whole program with respect to non-emptiness would impose significant costs and many false positives, since some code's container behavior is inherently subtle. Our insight is that it is enough to verify only *the non-empty properties that are relied upon* by the Optional verifier.

In our partial rely–guarantee approach, the Optional verifier depends only on *explicitly-written* `@NonEmpty` type qualifiers. The Non-Empty verifier verifies that all of those explicit type qualifiers are correct: at run time, the value is never empty. For example, the programmer may write:

```
@NonEmpty List<Integer> numbers = List.of(1, 2, 3);
Optional<Integer> maximum = numbers.stream()
    .max(Integer::compareTo);
maximum.get(); // Legal: `maximum` is @Present
```

The annotations from the Non-Empty type system are machine-checked, as are the annotations from the Optional type system. It is impossible for a programmer to annotate an empty container with `@NonEmpty` in a well-typed program.

Algorithm 1 Collecting methods to verify under the Non-Empty type system.

```
1:  global variables
2:      Callers // Mapping from methods to their callers
3:
4:  procedure MethodsForNonEmptyVerification(CFG)
5:      ToVerify = ∅
6:      for m ∈ CFG.methods do
7:          if HasNonEmptyPrecondition(m) then
8:              ToVerify ← ToVerify ∪ {m} ∪ Callers[m]
9:          else if HasNonEmptyAnnotation(m) then
10:             ToVerify ← ToVerify ∪ {m}
11:     return ToVerify
```

Algorithm 2 Helper methods for algorithm 1.

```
1:  procedure HasNonEmptyPrecondition(m)
2:      return m is annotated with @RequiresNonEmpty or
3:              any of m's formals is annotated with @NonEmpty
4:  procedure HasNonEmptyPostcondition(m)
5:      return m's return type is annotated with @NonEmpty or
6:              m is annotated with @EnsuresNonEmpty or
7:                      @EnsuresNonEmptyIf
8:  procedure HasNonEmptyAnnotation(m)
9:      return HasNonEmptyPrecondition(m) or
10:             HasNonEmptyPostcondition(m) or
11:             any of m's local variables is annotated
12:                     with @NonEmpty
```

Our approach may be viewed as a type of demand-driven analysis [30, 104], though it does not need the back-and-forth associated with most demand-driven analysis: each verifier runs just once.

4.3.1 Pseudocode. Algorithm 1 describes the process by which methods are collected to be verified under the Non-Empty type system. The set ToVerify comprises methods that should be verified with the Non-Empty verifier system due to the presence of explicitly-written annotations from the Non-Empty type system (lines 7 and 9).

The callers of methods that explicitly depend on the Non-Empty type system are also verified. Consider the code below:

```
// Generate a non-empty random sequence of numbers
@NonEmpty List<Integer> randomSequence() { ... }

int getRandomNumber() {
  return randomSequence().stream().findAny().get(); // Legal
}
```

`findAny()` returns any element of a stream; it will return a present `Optional` value when invoked on a non-empty stream. The method `getRandomNumber()` does not explicitly declare a dependence on the Non-Empty type system. However, it must be checked in order to establish the relationship between the presence (or absence) of the `Optional` value returned by `findAny()` and the stream on which it is invoked.

5 IMPLEMENTATION

The Optional Checker is a static analysis tool for Java's `Optional` type that enforces correctness (section 5.1) and style (section 5.2). It is a:

- *Verifier*, which provides a compile-time guarantee that a program will not throw `NoSuchElementException` as a result of misuse of the `Optional` type. This is achieved via our type systems that cooperate via rely–guarantee reasoning (section 4).
- *Linter*, which warns about violations of style rules [70] regarding the use of the `Optional` type. This is achieved via pattern-matching on the AST of a Java program and type analysis.

The Optional Checker comprises 1,276 non-comment non-blank lines of code, 1,263 lines of comments, 1,499 lines of tests, 24 annotations in the JDK (4 copies of the 6 annotations in fig. 4), and 124 annotations on JavaParser, of which 112 are on overrides of its `getParentNode()` method. We consider the idiosyncratic design in JavaParser to be poor style.

To use the Optional Checker, a programmer runs the Java compiler with an additional `-processor` command line flag, e.g., `javac -processor optional Main.java`. Warnings and errors emitted by the Optional Checker are displayed identically to those emitted by a standard Java compiler. Errors indicate cases where a run-time error might be thrown, and warnings indicate style rule violations.

5.1 Enforcing Correctness

To enforce correctness (i.e., ensure the absence of `NoSuchElementException` at run-time), the Optional Checker relies on annotations from the Optional type system written in the JDK.

```
public final class Optional<T> {
  public static <T> @Present Optional<T> of(T value);

  public T get(@Present Optional<T> this);

  @EnsuresPresentIf(result = true, expression = "this")
  public boolean isPresent();

  @EnsuresPresentIf(result = false, expression = "this")
  public boolean isEmpty();

  public T orElseThrow(@Present Optional<T> this);

  @EnsuresPresent("this")
  public T orElseThrow(Supplier exceptionSupplier);
}
```

Figure 4: The specification of Java's `java.util.Optional` class. Other methods in the class need no annotations.

Figure 4 shows the complete specification of the JDK's `Optional` class. It requires only 6 annotations. Classes `OptionalDouble`, `OptionalInt`, and `OptionalLong` have identical annotations.

The Optional Checker needs no special-case typing rules to enforce the correctness property, which is ensured by a formal parameter annotation on `get()`. Ordinary subtyping issues an error at each possibly-erroneous call site. An advantage of using annotations is that any other method can be specified and checked.

The only annotations we have not yet discussed are those on `orElseThrow` (). The zero-argument version throws `NoSuchElementException` if its receiver (`this`) is absent. Method `orElseThrow` () has the identical specification in the JDK (and therefore the same annotations for the Optional Checker) as `get()`, and either can be

implemented as a call to the other. Tool behavior should not depend on which of these two equivalent methods a programmer calls.

By contrast, the one-argument overload `orElseThrow(Supplier)` does not have `@Present` on its receiver type. The programmer explicitly throws an arbitrary exception after checking whether the element is absent. Whatever exception is thrown is intended by the programmer; it is not unexpected or an error.

The `@EnsuresPresent("this")` postcondition on `orElseThrow(Supplier)` indicates that, if the method completes normally, then the receiver is guaranteed to be present.

5.2 Enforcing Good Style

In enforcing good style, the Optional Checker mechanizes Marks' style rules (section 3.2) as automated linting rules that are applied to source code. Programmers can enable/disable individual linting rules.

5.2.1 `Optional` should never be `null`. Style rule #1 (section 3.2) forbids use of null for an `Optional` variable or return value. The Optional Checker enforces this by detecting pseudo-assignments whose right-hand side is the literal `null` and whose left-hand side has `Optional` type. This is a local, best-effort analysis.

The Optional Checker does not forbid every expression of `@Nullable` type on the right-hand side, because application invariants may guarantee that the value is non-null at run time, and the Optional Checker would issue too many false positive warnings. For a full guarantee, a user can run a nullness-checking tool [10, 13, 34, 91, 97, 116].

The Optional Checker interprets the rule to also forbid comparing an `Optional` to the literal null with `==` or `!=`, since the outcome of such a test is (or at least should be!) predictable at compile time.

5.2.2 Forbidding code patterns. Style rules #3, #4, and part of #5 forbid certain code patterns. The Optional Checker enforces these rules by pattern-matching over the AST (abstract syntax tree) of the code.

For example, rule #4 discourages "creat[ing] an Optional for the specific purpose of chaining methods from it to get a value." The Optional Checker raises a warning whenever there is a chain of method calls of the form *creation propagation* elimination*. The creation methods are `empty()`, `of()`, and `ofNullable()`. The propagation methods are `filter()`, `flatMap()`, `map()`, and `or()`. The elimination methods are `hashCode()`, `ifPresent()`, `ifPresentOrElse()`, `isEmpty()`, `isPresent()`, `toString()`, `get()`, `isEmpty()`, `orElse()`, `orElseGet()`, `orElseThrow()`, and methods defined by the `Object` class.

5.2.3 Forbidding Compound Types. Parts of style rules #5, #6, and #7 forbid certain compound types, such as `Optional<Optional<T>>`, `Optional<Collection<T>>`, and `Collection<Optional<T>>`. Such types can arise in two ways. The first way is when programmers write them explicitly; this is easy to check for using a simple type analysis.

The second way is as the type of an intermediate result within an expression. Detecting these is more difficult. Java's type inference algorithm [46], which computes generic type arguments, produces `Optional<? extends Object>` as the type of the expression `Optional.of(Optional.of("hello"))`. This encompasses all the type contexts in which the expression can be legally used. As a result of this fact, it is not possible to utilize a type-based analysis [90] that first resolves the type of every expression and then utilizes those types.

Although the expression `Optional.of(Optional.of("hello"))` has type `Optional<? extends Object>`, the subexpression `Optional.of("hello")` has type `Optional<String>`. Therefore, our analysis decomposes expressions then composes the types of their subexpressions to find the most *specific* type of the expression, as opposed to the most *general* type that a Java compiler computes. (We implemented this as an extension to the Checker Framework, making it available to other type systems.) This yields `Optional<Optional<String>>` for `Optional.of(Optional.of("hello"))`, enabling the Optional Checker to warn about this and other code that creates values of forbidden types.

6 EVALUATION: SUBJECT PROGRAMS

Our experiments use 27 real-world Java programs totalling over 1M non-comment, non-blank lines of code (The subject programs are listed in the supplementary material.) We obtained our dataset by querying GitHub via Sourcegraph [112]. We used 4 patterns in our query: "`import java.util.Optional;`", "`java.util.Optional<`", "`Optional.ofNullable(`", "`.get()`", and limited the output to 10k results. We queried GitHub twice using these patterns, resulting in a total of 349 programs.

We reduced the dataset, retaining only programs that

(1) have at least ten stars on GitHub, indicating that there is a community of users,
(2) had a commit made to the primary development branch (i.e., `main` or `master`) within the last six months, so that our bug reports and patches are less likely to be ignored,
(3) use the Gradle [48] or Maven [7] build system, making it easier to run the Optional Checker, and
(4) compile under Java 11, which is currently the version of Java with the highest adoption rate [103].

resulting in 129 programs from which we arbitrarily selected 27. We did not cherry-pick examples: whenever we considered a program, we put it permanently in our dataset and did not remove it. One program failed to compile with SpotBugs and two failed to compile with Error Prone. We manually inspected source code for the bug patterns relevant to the `Optional` type in these cases.

7 EVALUATION: CORRECTNESS

Our evaluation of the Optional Checker's efficacy in verifying correct use of the option type aims to answer the following research questions:

RQ1 How does the Optional Checker compare to other tools that aim to prevent run-time errors related to the optional type?
RQ2 How much effort is required to use the Optional Checker?

Our methodology for our evaluation of correctness is described in section 7.1. The Optional Checker discovered 13 previously-unknown real-world defects (section 7.2) of which 7 have been patched to-date. Section 7.3 discusses false positive warnings raised by the Optional Checker. Section 7.4 compares 4 tools: SpotBugs [115], Error Prone [41], IntelliJ IDEA [58], and the Optional Checker.

7.1 Methodology

Our experimental procedure uses the same two-step iterative process that a programmer would use to verify a legacy program. Repeatedly:

(1) Run the Optional Checker, which emits compile-time errors for `Optional` operations that may throw a `NoSuchElementException`, such as an invocation of `get()` on a possibly-absent `Optional` value.
(2) Address the Optional Checker's error messages in one of 3 ways. If the error indicates a defect, fix it. If the error stems from insufficient information provided to the Optional Checker, write additional machine-checked specifications. If the error is a false positive (e.g., application invariants guarantee the presence of an `Optional` value), suppress the error.

When a defect could be fixed without major architectural changes, we submitted a patch. Otherwise, we submitted a bug report.

We performed a trivial refactoring in one subject program [23]. We changed "`spec`" to "`help.commandSpec()`" to accommodate a method that used a local variable, which cannot appear in a method precondition:

```
 @RequiresNonEmpty("#1.commandSpec().subcommands()")
 private TextTable createTextTable(CommandLine.Help help) {
    CommandSpec spec = help.commandSpec();
    ...
-   int commandLength = maxLength(spec.subcommands(), ...);
+   int commandLength = maxLength(help.commandSpec().subcommands(), ...
```

7.2 Real-World Errors

7.2.1 File system operations. JavaPackager [40] is a plugin for Maven and Gradle that packages Java applications into native Windows, macOS, or Linux applications. The Optional Checker detected two defects, both with a root cause relating to file system operations. The maintainer accepted our patches for both of these defects [130, 132].

```
...
* @return Found file or null if nothing matches
*/
public static File findFirstFile(File searchFolder, String regex) {
    return Arrays.asList(searchFolder
        .listFiles((dir, name) -> Pattern.matches(regex, name)))
        .stream()
        .map(f -> new File(f.getName()))
        .findFirst()
        .get(); // Error if no file name matches `regex`.
   }
```

In the above code, a `NoSuchElementException` will be thrown at the call to `get()` if no file has a name that matches the regular expression. There is also a mismatch between the method specification and the implementation. Our patch was to modify the code to match the specification. We replaced "`get()`" by "`orElse(null)`".

The second error is an illegal call to `get()` on the result of a file system operation [130]. The code incorrectly assumes that the operation always succeeds, so the `Optional` result is never absent. Our patch replaced the use of `get()` with methods that are safe to use on a possibly-absent `Optional`, such as `ifPresent(consumer)`.

7.2.2 Variable capture. riot [102] is a command-line utility to access Redis data stores. the Optional Checker issued an error for this call to `get()`:

```
public Function<Map<String, Object>, String> build() {
   if (fields.isEmpty()) {
     if (prefix.isPresent()) {
       return m -> prefix.get();
     }
```

`prefix` is a private `Optional` field. If `prefix` is present, then a call to `build` returns a function `m -> prefix.get()`, which references the `prefix` field and which the client may store in a variable. The client may then call the method `prefix(null)`, which sets the `prefix` field to an absent `Optional`. Finally, the client may call the stored function, which will crash at the expression `prefix.get()`. IntelliJ unsoundly missed this defect. The maintainers updated the lambda expression to no longer capture the `prefix` field.

7.2.3 Empty Sequences and Streams. A Java `Stream` [3] represents a sequence of elements. Aggregate operations on a `Stream` return an `Optional` value that is absent if the stream is empty. Consider the code below from Spring Cloud Deployer Kubernetes [18]:

```
Stream<...> conditionsStream = Stream.of(conditions);
Boolean allConditionsMet = conditionsStream
  .reduce((x, y) -> x.and(y))
  .get()
```

The array `conditions` is a field that is set by the constructor; although the constructor has package-private access, there exists a public method that passes its arguments through to the constructor. Therefore, a client can create an object with an zero-length array for `conditions`, leading to an empty `Stream` `conditionsStream`, an absent `Optional` returned by `reduce()`, and a crash at `get()`.

7.2.4 Incorrect guard. pcgen [94] is an RPG character generator. The Optional Checker issued an error for this code:

```
if (current.isEmpty() || !current.equals(newRegion)) {
  ...
  fireDataFacetChangeEvent(id, newRegion.get().toString(), ...);
}
```

The true path in this conditional statement may be exercised when `newRegion` is absent and not equal to `current`. In this case, `newRegion.get()` crashes. Our patch for this defect was to replace the call to `get()` on `newRegion` within the conditional to a call to `ifPresent(supplier)`, which provides safe access to the value of `newRegion`. The maintainers accepted our patch for this defect [131].

7.2.5 Failure to check. The other 8 defects (3 in chunky and pcgen, 1 in hivemq and jib) are simple failures to check the result of a call to a method that we confirmed could return an absent `Optional`. The defect is unconditionally unwrapping the returned `Optional`. We have submitted patches [128, 129] to correct the 3 defects in chunky, which have been accepted by the maintainers. We have opened an issue in the repository for hivemq, and plan to open other issues or submit patches for the remaining defects.

7.3 False Positives

Like every sound tool, the Optional Checker sometimes reports false positive errors. It reported 1 in our experiments.

7.3.1 Call Sequencing. In pcgen, the Optional Checker reported a real latent defect that currently does not cause a failure. We marked this as a false positive report.

The methods in pcgen's `GroupFunction` class are not invoked directly. Rather, they are placed in a list, and then the program iterates over the list executing them one by one. In all the lists that are created, `evaluate()` (which accesses an `Optional<FormatManager>` value) is never called except after `allowArgs()` (which sets the value to present).

The Optional Checker cannot reason about this complex control flow. (And, there is a latent bug: a future maintainer could make a list with `evaluate()` first.)

IntelliJ does not issue a warning. This is due to unsoundness, not cleverness: IntelliJ also does not issue a warning if a list is made with the methods in the wrong order.

7.4 Comparison With Other Tools

We evaluated the Optional Checker against other 3 tools that warn about misuse of the Java `Option` type. Each of them is widely used in industry [52, 53, 118].

SpotBugs [51, 115] and Error Prone [41] are rule-based static analyzers that include rules to mitigate misuse of the option type. Rule-based static analyzers apply sets of pre-defined lexical rules against a program. These rules may express simple syntactic patterns such as the maximum length of a line in a program, to more complicated patterns, such as ensuring methods are called in a specific sequence. For example, SpotBugs' `NP_OPTIONAL_RETURN_NULL` rule [69] forbids null returns for methods with an `Optional` return type. Error Prone includes 8 rules, or, *bug patterns* [43], regarding the use of the option type. Most pertain to style, but the `OptionalNotPresent` bug pattern warns programmers from accessing the value of an `Optional` instance if it is *explicitly* known to be empty (e.g., an invocation of `get()` immediately after a call to `isEmpty()` that returns true). Also like the Optional Checker, it is built on top of the Checker Framework's Dataflow Framework [27].

SpotBugs and Error Prone report specific cases matching the lexical rules or bug patterns they employ with high confidence. However, rule-based approaches often suffer in cases where additional precision may be derived from dataflow information and type inference, which may be difficult or impossible to encode in a set of static lexical rules that match solely on the syntax of a program. Additionally, SpotBugs and Error Prone are bug detectors; their goal is not to verify the absence of errors, but to balance error reduction with the rate of false positives.

The IntelliJ IDEA IDE [58] is the dominant Java integrated development environment [118]. Version 2023.3 contains 9 inspections related to `Optional` [117]. These inspections cover all of the best practices of section 3.2. When enabled, failed checks are presented inline in the editor as warnings and do not cause compilation failures. (By contrast, the Optional Checker and Error Prone can both halt compilation, though each can be configured to issue only warnings.) Some of the `Optional` analyses are disabled by default, but we enabled them all for our experiments.

Table 1 reports the tools' performance.

7.4.1 Annotations as Checked Specifications. The Optional Checker enables programmers to specify information about the presence of `Optional` values. For example, IntelliJ issued a false positive warning at a call to the method `getLocalScopeName` in pcgen [94]:

Table 1: Defects reported by 4 static analysis tools. Annos is the number of machine-verified annotations we wrote as a specification in programs.

Tool	True positives	False positives	Precision	Recall	Annos
SpotBugs	0	0	n/a	0%	0
Error Prone	0	0	n/a	0%	0
IntelliJ IDEA	11	5	69%	85%	0
Optional Checker	13	1	93%	100%	6

```
@Override
public Optional<String> getLocalScopeName() {
  return Optional.of("PC.STAT");
}
```

This method is an override of a base method that returns an absent `Optional`, as in section 4.1.1. A single annotation to the signature of the overridden method informs the Optional Checker of its specification:

```
public @Present Optional<String> getLocalScopeName()
```

Running the Optional Checker guarantees that the method body satisfies the specification. These checked specifications are relied upon by the Optional Checker to establish facts about the presence of `Optional` values at call sites.

7.4.2 Soundness. SpotBugs does not provide a guarantee of soundness and does not analyze byte-code for illegal instances of `get()`. It detects cases where a null value is returned from a method that is marked to return an `Optional<T>`, which violates style rule #1.

Error Prone only issues errors for cases where it is *certain* that a `NoSuchElementException` will be thrown, such as in the code below from its catalogue of bug patterns [44]:

```
if (!o.isPresent()) {
  return o.get(); // this will throw a NoSuchElementException
}
```

IntelliJ is the tool in our evaluation that came closest to detecting the same defects that the Optional Checker found. However, it is not sound. Four unsoundnesses are variable capture (section 7.2.2), call sequencing (section 7.3.1), not warning when `orElseThrow()` is invoked on a possibly-present `Optional` (section 5.1), and, in jib, a sequence of operations with `Future` and `Optional`.

IntelliJ's unsoundness was previously unknown, and would not have been known without building a sound tool. IntelliJ's precision was also unknown. The Optional Checker's higher precision is a qualitative, not just quantitative, difference. Only the Optional Checker meets usability guidelines proposed by Google [107] that require 90% precision.

8 EVALUATION: STYLE

This section compares the style guidelines of section 3.2 with Java practice as exemplified by the subject programs of section 6.

RQ3 To what extent do real-world Java programs adhere to style rules regarding the `Optional` type?

Table 2: Style violations related to the Java `Optional` type; section 3.2 provides a description for each table header. Rule #2 is a violation of correctness, which appears in table 1.

	1	3	4	5	6.a	6.b	6.c	7
SpotBugs	1	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Error Prone	0	1	1	1	n/a	n/a	n/a	n/a
IntelliJ IDEA	3	14	2	11	30	61	102	n/a
Optional Checker	7	6	43	1	22	93	67	0

Section 8.1 describes our methodology Table 2 overviews of our results, which are discussed in section 8.2.

8.1 Methodology

Similar to our investigation of correctness, we ran 4 tools (SpotBugs [115], Error Prone [41], IntelliJ IDEA [58], and the Optional Checker) on our dataset of subject programs from section 6, and we examined their style warnings. This study provides a view, across over over 1M, of how programmers use the `Optional` type and whether they violates style guidelines.

The goal of this section is not to compare or criticize the tools. The tools may interpret the rules slightly differently. The correspondence between tool output and style rules is approximate; for instance, IntelliJ has inspections that apply to both `Optional` and `Stream`, which cannot be disabled separately. Furthermore, the tools are all open-source and can be tweaked to produce exactly the same diagnostic output as one another.

8.2 Results

Violations of rule #3, "Prefer alternative APIs over `Optional.isPresent()` and `Optional.get()`," were likely due to programmers who were unfamiliar with the `Optional` API or who preferred a procedural to a functional programming style. Both `Optional` and functional programming take a bit of getting used to.

Rule #4, "It's generally a bad idea to create an Optional for the specific purpose of chaining methods from it to get a value," was often exuberantly violated. A single expression would create an `Optional`, call `Optional` methods, and then call `get()`. We hypothesize that programmers enjoyed the fluent APIs that emulate a functional programming style. Newcomers to a feature are also known to overuse it. A simple violation of rule #4 is:

```
dfcl = Optional.ofNullable(dfcl).orElse(new
      DataFacetChangeListener[0]);
// Simpler version:
if (dfcl == null) {
  dfcl = new DataFacetChangeListener[0];
}
```

Further examples appear in the supplementary material.

We observed a similar (over-)use of streams, with them being created from a list, manipulated, and turned back into a list when a list method would be simpler and more efficient. In other cases, use of `Stream` helps to clarify and simplify code.

Violations of rule #6, "Avoid using `Optional` in (a) collections, (b) method parameters, and (c) fields", were by far the most frequent. They come from programmers who wish to take advantage of `Optional` in more parts of their code than just some return statements, as noted in section 3.2. Eight classes in the JDK's `base` module violate Oracle's guidelines by using `Optional` for formal parameters, usually for multiple methods. This is 11% of the 72 files that use `Optional`. The `base` JDK module also violates other style guidelines: it contains `Optional` local variables that not related to any `return` statement, and uses of `get()` without a check.

8.3 `Optional` Fields and Parameters

hollow [82] is a library developed by Netflix with the goal of rapidly providing consumer objects with read-only access to in-memory datasets from a single producer object. the Optional Checker emitted 7 warnings, mostly due to fields of `Optional` type.

In one instance [83], field `Optional<Boolean> isPinned;` was read in the code below via the getter `isPinned()`:

```
if (requestedVersionInfo.isPinned() == null || ...) {
   return;
}
if (!(requestedVersionInfo.isPinned().isPresent() && ...) {
   return;
}
boolean isPinned = requestedVersionInfo.isPinned().get();
...
```

This is a safe use of the `Optional` type in that it will not throw a run-time exception. However, it exemplifies the complexity introduced by the use of an optional value. After 3 levels of indirection (i.e., a nullness check, a presence check, and a call to `get()`), the programmer finally has safe access to the wrapped `Boolean` value.

9 LIMITATIONS AND THREATS TO VALIDITY

The Optional Checker can only issue guarantees for the source code that it has access to at compile-time. This excludes unannotated libraries, including native libraries.

The Optional Checker itself may have defects; that is, our implementation of the Optional Checker might be buggy. Our extensive test suite (larger than the implementation, see section 5) mitigates this threat.

In order to count errors and style violations, our experimental infrastructure used Java's `@SuppressWarnings` annotation. Our counts may be undercounts because Java only permits `@SuppressWarnings` on declarations — not statements or expressions. Sometimes one `@SuppressWarnings` encompassed a section of code with (say) multiple errors, but we counted it only once because of the single `@SuppressWarnings` annotation.

Our results undercount the benefit of the Optional Checker, because we used it on programs after a large amount of implementation and testing had occurred. Many defects that the Optional Checker would have discovered have already been detected and patched. Use of formal verification tools earlier in the software development cycle is more beneficial.

A field study is needed to determine whether programmers find the Optional Checker beneficial.

9.1 Beyond Java

We chose Java because most tools that aim to prevent misuse of the option type are written for Java. This enables us to compare our work to other tools. There is an abundance of high-quality open-source subject programs written in Java.

If our subject programs are not representative, then our results will not generalize to other Java programs. If Java programs use

`Optional` differently than in other programming languages, then our results will not generalize to other languages.

Our approach — type systems and partial rely–guarantee reasoning — is equally applicable to other statically-typed languages. The syntax would be different. For example, C# and C++ would use attributes where Java uses annotations. Any language could use stylized comments.

For a dynamically-typed language like Python, users could use the Mypy static type checker [99], or the Optional Checker could be rewritten (retaining the same semantics) as an abstract interpretation rather than a type system.

For any language, specifications must be written for the option type, analogous to our JDK annotations.

10 RELATED WORK

10.1 Optional Typing vs. the Optional Type

This paper is about verifying use of the type *named* `Optional`, and not about optional typing. In the literature, "optional typing" most often means "pluggable typing" [12] or "soft typing" [16]. (Adding to potential confusion, our implementation uses pluggable type qualifiers, such as `@Present`.) In that nomenclature, an optional type is one that may or may not be written, such as types in TypeScript [11], or in any dynamically-typed languages such as JavaScript [125], Python [100], Ruby [21], or Racket [36]. A natural outgrowth of optional typing is gradual typing [109], in which programmers write more and more types as development proceeds.

10.2 Inferring Specifications for the Option Type

Infer [76] is a static analysis tool for Java, C/C++, and Objective-C, which runs in production at Meta. Infer employs an incremental static analysis that augments separation logic [105] with bi-abduction [14], which enables a mechanism by which specifications are able to be inferred from unannotated source code. This enables Infer to scale to large programs without the need for programmer intervention (e.g., additional type annotations). Infer also makes use of Pulse; a compositional bug-catching analyzer [65] that uses incorrectness logic [86] to support additional checks beyond those that are provided by its standard analysis.

Like the Optional Checker, Infer aims to guarantee the absence of run-time crashes resulting from the access of empty option values. The `OPTIONAL_EMPTY_ACCESS` [77] and `OPTIONAL_EMPTY_ACCESS_LATENT` [78] issue types (provided by its static analysis based on incorrectness logic) attempt to prevent the access of empty option values. However, neither of these issue types is implemented for Java.

Unlike the Optional Checker, Infer does not lint a program to detect unstylistic use of the option type; programmers must run another tool in addition to Infer to obtain confidence that their use of the option type does not violate style guidelines.

Infer is unsound by design [75], which was a deliberate trade-off in order to maximize the value given to developers over soundness or completeness concerns. Additionally, Infer is non-deterministic, rooted in its analysis of mutually-recursive functions and further exacerbated when executed in parallel [74]. This leads to inconsistencies in defect detection, with defects being discovered in one run but not another.

10.3 Frameworks

CodeQL is a framework for developing static analysis tools, much like the Checker Framework. We were unable to find an `Optional` analysis written in CodeQL.

Jqual [49] is another framework — a translation of Cqual [37] into Java — that is implemented as a plug-in for the Eclipse development environment [31]. It has been extended from type checking to type inference [50]. We were unable to find an `Optional` analysis written for Jqual or Cqual. Jqual lacks important features such as flow-sensitivity [38] object-sensitivity, generics, method polymorphism, inner classes, etc.

11 CONCLUSION

This paper investigated the real-world use of the option type, focusing on the problems of *correctness* and *style*.

To address problems of correctness, we developed the Optional and Non-Empty type systems that cooperate via a novel form of rely–guarantee reasoning. Together, they verify use of the option type. We implemented them in the Optional Checker, an open-source static verifier that is the first verifier for the option type based on a sound theory. The Optional Checker achieved a precision of 93%—higher than any other tool.

To address problems of style, we conducted a large-scale, mechanized study of the Optional Checker and prior tools on over 1M lines of open-source code. We also noted which widely-promulgated style guidelines are most often flouted by Java programmers: those relating to local creation and consumption of a single `Optional` value, and those relating to use of `Optional` at locations other than method return types.

DATA AVAILABILITY

To support open science, our implementation, experimental scripts, annotated programs, and results are available at https://zenodo.org/doi/10.5281/zenodo.11522277.

ACKNOWLEDGMENTS

We thank Martin Kellogg and the anonymous reviewers for their comments. This research was supported in part by DARPA contract FA8750-20-C-0226.

REFERENCES

[1] 2024. *Zig Language Reference.* https://ziglang.org/documentation/master/#Optionals
[2] Java Platform Standard Ed. 8. 2014. Optional (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html. The Java Platform documentation for the Optional type.
[3] Java Platform Standard Ed. 8. 2014. Stream (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html. The Java Platform documentation for the Stream type.
[4] The Agda Team. 2024. *Language Reference - Agda 2.6.4.3 documentation.* https://agda.readthedocs.io/en/v2.6.4.3/language/index.html
[5] F.E. Allen and J. Schwartz. 1974. *Determining the Data Relationships in a Collection of Procedures.* IBM Research Report RC 4989 (22125). IBM T.J. Watson Research Center.
[6] Frances E. Allen. 1974. Interprocedural Data Flow Analysis. In *IFIP 1974: Proceedings of the 6th IFIP Congress.* Stockholm, Sweden, 398–402.

[7] Apache Software Foundation. 2023. Maven - Welcome to Apache Maven. https://maven.apache.org/index.html. Accessed 2023.

[8] Apple Inc. 2023. Optional. https://developer.apple.com/documentation/swift/optional. Accessed 2023.

[9] Baeldung. 2020. Guide To Java 8 Optional. https://www.baeldung.com/java-optional.

[10] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. NullAway: Practical type-based null safety for Java. In *ESEC/FSE 2019: The ACM 27th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Tallinn, Estonia, 740–750.

[11] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*. Springer-Verlag, Berlin, Heidelberg, 257–281. https://doi.org/10.1007/978-3-662-44202-9_11

[12] Gilad Bracha. 2004. Pluggable type systems. In *RDL 2004: Workshop on Revival of Dynamic Languages*. Vancouver, BC, Canada.

[13] Dan Brotherston, Werner Dietl, and Ondřej Lhoták. 2017. Granullar: Gradual nullable types for Java. In *CC 2017: 26th International Conference on Compiler Construction*. Austin, TX, USA, 87–97.

[14] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*.

[15] Noel Rodríguez Calle. 2023. Java Optional and best practices. https://refactorizando.com/en/java-optional-and-best-practices/.

[16] Robert Cartwright and Mike Fagan. 1991. Soft Typing. In *PLDI '91: Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*. Toronto, ON, Canada, 278–292.

[17] Brian Chin, Shane Markstrum, and Todd Millstein. 2005. Semantic type qualifiers. In *PLDI 2005: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. Chicago, IL, USA, 85–95.

[18] Spring Cloud. 2023. resolve. https://github.com/spring-cloud/spring-cloud-deployer-kubernetes/blob/0b84199d17788e1d05e5f3bab5df91ff0863b1e3/src/main/java/org/springframework/cloud/deployer/spi/kubernetes/PredicateRunningPhaseDeploymentStateResolver.java#L48. Accessed 2023.

[19] Stephen Colebourne. 2015. Java SE 8 Optional, a pragmatic approach. https://blog.joda.org/2015/08/java-se-8-optional-pragmatic-approach.html.

[20] Stephen Colebourne. 2017. Java SE 8 Best Practices: A personal viewpoint. In *Jfokus Developers Conference*. Jfokus, Stokholm, Sweden. https://www.jfokus.se/jfokus17/preso/Java-SE-8-best-practices.pdf.

[21] Ruby Community. 2024. Ruby Core Reference. (2024). https://ruby-doc.org/3.2.2/

[22] The Idris Community. 2020. *Language Reference - Idris 1.3.3 documentation*. https://docs.idris-lang.org/en/latest/reference/index.html

[23] Picocli Contributors. 2024. Picocli: a mighty tiny command line interface. https://github.com/remkop/picocli/blob/b03121b07eafaa094f634a09109f77df4b9cb4c0/picocli-examples/src/main/java/picocli/examples/customhelp/GroupingDemo.java#L101-L111. Accessed 2024.

[24] Patrick Cousot. 1997. Types as abstract interpretations. In *POPL '97: Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Paris, France, 316–331.

[25] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*. Los Angeles, CA, 238–252.

[26] Evan Czaplicki. 2023. Maybe - core 1.0.5. https://package.elm-lang.org/packages/elm/core/latest/Maybe. Accessed 2023.

[27] Dataflow framework 2023. A Dataflow Framework for Java. https://checkerframework.org/manual/checker-framework-dataflow-manual.pdf. Accessed 2023.

[28] Jodd Developers. 2023. StringUtil - Jodd Util. https://util.jodd.org/utilitites/stringutil. Accessed 2023.

[29] Pandoc Developers. 2023. Pandoc: The universal markup converter. https://github.com/felixonmars/pandoc/blob/0134b6332323f3230fb174ebf3bc85c1520228b2/src/Text/Pandoc/CSV.hs#L67C1-L75C1. Accessed 2023.

[30] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. 1995. Demand-driven Computation of Interprocedural Data Flow. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*.

[31] Eclipse [n.d.]. Eclipse Project. http://www.eclipse.org/. http://www.eclipse.org/

[32] Michael D. Ernst. 2022. Nothing is better than the Optional type. Really. Nothing is better. *Java Magazine* (Dec. 2022).

[33] Michael D. Ernst. 2024. plume-util. https://github.com/plume-lib/plume-util. Accessed 2024.

[34] Inc. Facebook. 2019. Infer : Eradicate. https://fbinfer.com/docs/checker-eradicate/.

[35] FasterXML, LLC. 2020. Jackson Project Home @github. https://github.com/FasterXML/jackson.

[36] Matthew Flatt and PLT. 2024. *The Racket Reference* (8.13 ed.). https://docs.racket-lang.org/reference/

[37] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. 1999. A theory of type qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*. Atlanta, GA, USA, 192–203. https://doi.org/10.1145/301618.301665

[38] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. 2006. Flow-insensitive type qualifiers. *ACM TOPLAS* 28, 6 (Nov. 2006), 1035–1087.

[39] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. 2002. Flow-sensitive type qualifiers. In *PLDI 2002: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. Berlin, Germany, 1–12.

[40] fvarrui. 2023. fvarrui/JavaPackager. https://github.com/fvarrui/JavaPackager. Accessed 2023.

[41] Google. 2018. Error Prone. https://errorprone.info/.

[42] Google. 2023. Bug Patterns. https://errorprone.info/bugpatterns.

[43] Inc. Google. 2023. Error Prone Bug Patterns. https://errorprone.info/bugpattern. Accessed 2023.

[44] Inc. Google. 2023. OptionalNotPresent. https://errorprone.info/bugpattern/OptionalNotPresent. Accessed 2023.

[45] Google Corporation. 2015. Using and avoiding null. https://github.com/google/guava/wiki/UsingAndAvoidingNullExplained.

[46] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. 2023. Java SE Specifications. https://docs.oracle.com/javase/specs/jls/se21/jls21.pdf. , 789-819 pages. Accessed 2023.

[47] W. Brian Gourlie. 2015. Java 8's new Optional type is worthless. https://medium.com/@bgourlie/java-8-s-new-optional-type-is-worthless-448a00fa672d.

[48] Gradle Inc. 2023. Gradle. https://gradle.org. Accessed 2023.

[49] David Greenfieldboyce and Jeffrey S. Foster. 2005. Type qualifiers for Java. http://www.cs.umd.edu/Grad/scholarlypapers/papers/greenfiledboyce.pdf.

[50] David Greenfieldboyce and Jeffrey S. Foster. 2007. Type qualifier inference for Java. In *OOPSLA 2007, Object-Oriented Programming Systems, Languages, and Applications*. Montreal, Canada, 321–336.

[51] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. In *OOPSLA Companion: Companion to Object-Oriented Programming Systems, Languages, and Applications*. Vancouver, BC, Canada, 132–136.

[52] Sonatype Inc. 2024. Maven Central: com.google.code.findbugs. https://central.sonatype.com/namespace/com.google.code.findbugs. Accessed 2024-08-11.

[53] Sonatype Inc. 2024. Maven Central: error_prone_annotations. https://central.sonatype.com/artifact/com.google.errorprone/error_prone_annotations/versions. Accessed 2024-08-11.

[54] INRIA 2023. Standard Library | The Coq Proof Assistant. https://coq.inria.fr/doc/V8.17.1/stdlib/Coq.Init.Datatypes.html#option. Accessed 2023.

[55] ISO. 2012. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization.

[56] Java Developer Central. 2019. A Complete Guide to Java Optional. https://javadevcentral.com/a-complete-guide-to-java-optional.

[57] JetBrains. 2020. List of Java inspections. https://www.jetbrains.com/help/idea/2020.1/list-of-java-inspections.html.

[58] JetBrains. 2023. IntelliJ IDEA - the leading Java and Kotlin IDE. https://www.jetbrains.com/idea/. Accessed 2023.

[59] Cliff B. Jones. 1983. Tentative steps toward a development method for interfering programs. 5, 4 (Oct. 1983), 596–619.

[60] John B. Kam and Jeffrey D. Ullman. 1976. Global data flow analysis and iterative algorithms. *JACM* 23, 1 (Jan. 1976), 158–171.

[61] Martin Kellogg, Manli Ran, Manu Sridharan, Martin Schäf, and Michael D. Ernst. 2020. Verifying Object Construction. In *ICSE 2020, Proceedings of the 42nd International Conference on Software Engineering*. Seoul, Korea, 1447–1458.

[62] Martin Kellogg, Martin Schäf, Serdar Tasiran, and Michael D. Ernst. 2020. Continuous compliance. In *ASE 2020: Proceedings of the 35th Annual International Conference on Automated Software Engineering*. Melbourne, Australia, 511–523.

[63] Semyon Kirekov. 2020. Java Optional is not so obvious. https://levelup.gitconnected.com/java-optional-is-not-so-obvious-263d9559dd41.

[64] LAMP/EPFL. 2023. Option. https://dotty.epfl.ch/api/scala/Option.html. Accessed 2023.

[65] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.* (2022).

[66] Anghel Leonard. 2018. 26 Reasons Why Using Optional Correctly Is Not Optional. *Dzone.com* (Nov. 2018).

[67] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, KC Sivaramakrishnan, and Jérôme Vouillon. 2023. OCaml library: Option. https://v2.ocaml.org/api/type_Option.html. Accessed 2023.

[68] Przemyslaw Magda. 2017. Optional Anti-Patterns. *Dzone.com* (July 2017). https://dzone.com/articles/optional-anti-patterns.

[69] SpotBugs Maintainers. 2024. Bug descriptions – spotbugs 4.8.6 documentation. https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#np-method-

with-optional-return-type-returns-explicit-null-np-optional-return-null. Documentation for SpotBugs' rule to prevent null returns for option values.
[70] Stuart Marks. 2016. Optional: The mother of all bikesheds. In *vJUG24*. https://stuartmarks.wordpress.com/2016/09/27/vjug24-session-on-optional/.
[71] Stuart Marks. 2017. Optional - The Mother of All Bikesheds. https://www.youtube.com/watch?v=Ej0sss6cq14. Accessed 2024-08-20.
[72] Stuart Marks and Brian Goetz. 2015. API Design with Java 8 Lambda and Streams. In *JavaOne 2015*. Oracle Corporation, San Francisco, CA, USA.
[73] Mervyn McCreight. 2019. A look at the Optional datatype in Java and some anti-patterns when using it. freeCodeCamp https://www.freecodecamp.org/news/optional-in-java-and-anti-patterns-using-it-7d87038362ba/.
[74] Meta. 2024. facebook/infer: Issue #1110. https://github.com/facebook/infer/issues/1110. Accessed 2024-08-18.
[75] Meta. 2024. facebook/infer: Issue #427. https://github.com/facebook/infer/issues/427#issuecomment-240687267. Accessed 2024-08-18.
[76] Meta. 2024. Infer. https://fbinfer.com. Accessed 2024-08-18.
[77] Meta. 2024. Infer: OPTIONAL_EMPTY_ACCESS. https://fbinfer.com/docs/all-issue-types/#optional_empty_access. Accessed 2024-08-18.
[78] Meta. 2024. Infer: OPTIONAL_EMPTY_ACCESS_LATENT. https://fbinfer.com/docs/all-issue-types/#optional_empty_access. Accessed 2024-08-18.
[79] Microsoft Corp. 2023. Options - F# | Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/options. Accessed 2023.
[80] Robin Milner, Mads Tofte, and Robert Harper. 1990. *The Definition of Standard ML*.
[81] Hopewell Mutanda. 2019. Java 8 Optional Usage and Best Practices. *Dzone.com* (July 2019). https://dzone.com/articles/java-8-optional-usage-and-best-practices.
[82] Netflix. 2023. netflix/hollow. https://github.com/Netflix/hollow. Accessed 2023.
[83] Netflix. 2023. versionDetected. https://github.com/Netflix/hollow/blob/5ee483170e2902dc694a47ceeb4dc7a4a901d284/hollow/src/main/java/com/netflix/hollow/api/consumer/metrics/AbstractRefreshMetricsListener.java#L82. Accessed 2023.
[84] Nim Developers. 2024. *std/options*. https://nim-lang.org/docs/options.html
[85] nLab authors. 2023. eta-conversion. https://ncatlab.org/nlab/show/eta-conversion. Revision 14.
[86] Peter W. O'Hearn. 2019. Incorrectness logic. *Proc. ACM Program. Lang.* (2019).
[87] Daniel Olszewski. 2018. Java 8 Optional best practices and wrong usage. http://dolszewski.com/java/java-8-optional-use-cases/.
[88] Oracle Corporation. 2018. Class Optional<T>. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Optional.html.
[89] Indrek Ots. 2019. Misusing Java's Optional type. https://blog.indrek.io/articles/misusing-java-optional/.
[90] Jens Palsberg. 2001. Type-based analysis and applications. In *PASTE 2001: ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*. Snowbird, Utah, USA, 20–27.
[91] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*. Seattle, WA, USA, 201–212. https://doi.org/10.1145/1390630.1390656
[92] Nicolai Parlog. 2022. Where to use Optional — Inside Java Newscast #19. https://nipafx.dev/inside-java-newscast-19/.
[93] Nicolai Parlog. 2023. The Design of Optional. https://nipafx.dev/design-java-optional/. Accessed 2023.
[94] PCGen. 2023. PCGen/pcgen. https://github.com/PCGen/pcgen/blob/4c001b3526e3e5deef49c23750e563c578d4f2ed/code/src/java/pcgen/rules/persistence/DynamicLoader.java#L97C1-L101C4. Accessed 2023.
[95] Perforce JRebel Labs. 2020. Java 8 Best Practices Cheat Sheet. https://www.jrebel.com/resources/java-8-best-practices.
[96] Manh Phan. 2019. Best practice for Optional in Java. https://ducmanhphan.github.io/2019-12-06-Best-practice-for-Optional-in-Java/.
[97] Artem Pianykh, Ilya Zorin, and Dmitry Lyubarskiy. 2022. Retrofitting null-safety onto Java at Meta. https://engineering.fb.com/2022/11/22/developer-tools/meta-java-nullsafe/.
[98] Artem Pianykh, Ilya Zorin, and Dmitry Lyubarskiy. 2022. Retrofitting null-safety onto Java at Meta. https://engineering.fb.com/2022/11/22/developer-tools/meta-java-nullsafe/. Accessed 2024-08-20.
[99] The Mypy Project. 2014. mypy - Optional Static Typing for Python. https://mypy-lang.org. Accessed 2024-08-20.
[100] Python Software Foundation. 2024. *The Python Language Reference* (Python 3.12.3 ed.). https://docs.python.org/3/reference/index.html
[101] A N M Bazlur Rahman. 2023. Optional in Java: A Swiss Army knife for handling nulls and improving code quality. https://foojay.io/today/author/bazlur-rahman/.
[102] redis developer. 2023. redis-developer/riot. https://github.com/redis-developer/riot/blob/e831410a93613059d614c1692877db2faccb9598/core/riot-core/src/main/java/com/redis/riot/core/convert/IdConverterBuilder.java#L46-L50. Accessed 2023.
[103] New Relic. 2023. 2023 State of the Java Ecosystem. https://newrelic.com/resources/report/2023-state-of-the-java-ecosystem#most-used-java-versions. Accessed 2023.
[104] Thomas Reps. 1994. Solving demand versions of interprocedural analysis problems. In *Compiler Construction*. Springer Berlin Heidelberg, 389–403.
[105] J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74. https://doi.org/10.1109/LICS.2002.1029817
[106] Rust Foundation. 2023. std::option - Rust. https://doc.rust-lang.org/std/option/index.html. Accessed 2023.
[107] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at Google. *CACM* 61, 4 (Mar. 2018), 58–66.
[108] Amazon Web Services. 2023. smithy-lang/smithy. https://github.com/smithy-lang/smithy/blob/309862e54cd5582b3e3456845cf9e6c9fd873a4d/smithy-aws-traits/src/main/java/software/amazon/smithy/aws/traits/tagging/TaggableResourceValidator.java#L125-L128. Accessed 2023.
[109] Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *SFP 2006: Workshop on Scheme and Functional Programming (SFP)*. Portland, OR, USA, 81–92.
[110] Volodymyr Sobotovych. 2022. Using java.util.Optional everywhere. To be or not to be? https://wheleph.gitlab.io/posts/2022-07-09-using-optional-everywhere/.
[111] SonarSource S.A. 2020. Java static code analysis: Unique rules to find Bugs, Vulnerabilities, Security Hotspots, and Code Smells in your Java code. https://rules.sonarsource.com/java/.
[112] Sourcegraph. 2023. Sourcegraph Search. https://sourcegraph.com/search. Accessed 2023.
[113] Eric Spishak, Werner Dietl, and Michael D. Ernst. 2012. A type system for regular expressions. In *FTfJP: 14th Workshop on Formal Techniques for Java-like Programs*. Beijing, China, 20–26.
[114] SpotBugs Community. 2021. Bug descriptions - spotbugs 4.8.2 documentation. https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#np-method-with-optional-return-type-returns-explicit-null-np-optional-return-null.
[115] SpotBugs maintainers. [n.d.]. SpotBugs. https://github.com/spotbugs/spotbugs. SpotBugs source code repository.
[116] F. Spoto. 2010. The Nullness Analyser of Julia. In *LPAR 2010: Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. Dakar, Senegal, 405–424.
[117] JetBrains s.r.o. 2023. IntelliJ IDEA 2023.3 List of Java inspections. https://www.jetbrains.com/help/idea/list-of-java-inspections.html. Accessed 2023.
[118] Stack Overflow. 2023. Stack Overflow Developer Survey 2023. https://survey.stackoverflow.co/2023/#most-popular-technologies-new-collab-tools. Accessed 2023.
[119] Mohamed Taman. 2020. 12 recipes for using the Optional class as it's meant to be used. *Java Magazine* (June 2020). https://blogs.oracle.com/javamagazine/post/12-recipes-for-using-the-optional-class-as-its-meant-to-be-used.
[120] Mohamed Taman. 2020. The Java Optional class: 11 more recipes for preventing null pointer exceptions. *Java Magazine* (July 2020). https://blogs.oracle.com/javamagazine/post/12-recipes-for-using-the-optional-class-as-its-meant-to-be-used.
[121] The Bored Dev. 2020. Please stop the Java Optional mess! (How To Use Optional in Java). https://theboreddev.com/please-stop-the-java-optional-mess/.
[122] Emanuel Trandafir. 2022. 4 reasons why you should use Java Optional — or not? https://medium.com/javarevisited/4-reasons-why-you-should-use-java-optional-or-not-4e44d51a9645.
[123] VMware Tanzu. 2023. Spring Home. https://spring.io/.
[124] Ben Weidig. 2019. Better null-handling with Java Optionals. https://belief-driven-design.com/better-null-handling-with-java-optionals-da974529bae/.
[125] Allen Wirfs-Brock and Brendan Eich. 2020. JavaScript: The First 20 Years. *Proc. ACM Program. Lang.* 4, HOPL, Article 77 (jun 2020), 189 pages. https://doi.org/10.1145/3386327
[126] Chad Woolf, Byron Cook, and Tom McAndrew. 2019. AWS re: Inforce 2019: Automate Compliance Verification on AWS Using Provable Security (GRC301). https://www.youtube.com/watch?v=BbXK_-b3DTk. Accessed 2024-08-20.
[127] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115 (1994), 38–94.
[128] James Yoo. 2023. Avoid possibility of crash for GeneralTab.java and PluginManagerController.java. https://github.com/chunky-dev/chunky/pull/1678. Accessed 2024-09-07.
[129] James Yoo. 2023. Fix for possible crash in SceneChooserController.java. https://github.com/chunky-dev/chunky/pull/1677. Accessed 2024-09-07.
[130] James Yoo. 2023. Prevent run-time exception when image mounting fails. https://github.com/fvarrui/JavaPackager/pull/379. Accessed 2024-09-01.
[131] James Yoo. 2023. Prevent run-time exception when region update occurs. https://github.com/PCGen/pcgen/pull/7009. Accessed 2024-09-07.
[132] James Yoo. 2023. Update FileUtils#findFirstFile to conform to specifications. https://github.com/fvarrui/JavaPackager/pull/376. Accessed 2024-09-01.

[133] James Yoo. 2024. Avoid NoSuchElementException in Math-Plume.modulusStrictInt. https://github.com/plume-lib/plume-util/pull/371. Accessed 2024-09-07.

[134] James Yoo. 2024. Avoid NoSuchElementException in Math-Plume.modulusStrictLong. https://github.com/plume-lib/plume-util/pull/368. Accessed 2024-09-07.