Granullar: Gradual Nullable Types for Java

Dan Brotherston
University of Waterloo
Waterloo, Canada
daniel.brotherston@uwaterloo.ca

Werner Dietl
University of Waterloo
Waterloo, Canada
werner.dietl@uwaterloo.ca

Ondřej Lhoták
University of Waterloo
Waterloo, Canada
ondrej.lhotak@uwaterloo.ca

Abstract

Object-oriented languages like Java and C# allow the `null` value for all references. This supports many flexible patterns, but has led to many errors, security vulnerabilities, and system crashes. Static type systems can prevent null-pointer exceptions at compile time, but require annotations, in particular for used libraries. Conservative defaults choose the most restrictive typing, preventing many errors, but requiring a large annotation effort. Liberal defaults choose the most flexible typing, requiring less annotations, but giving weaker guarantees. Trusted annotations can be provided, but are not checked and require a large manual effort. None of these approaches provide a strong guarantee that the checked part of the program is isolated from the unchecked part: even with conservative defaults, null-pointer exceptions can occur in the checked part.

This paper presents Granullar, a gradual type system for null-safety. Developers start out verifying null-safety for the most important components of their applications. At the boundary to unchecked components, runtime checks are inserted by Granullar to guard the verified system from being polluted by unexpected `null` values. This ensures that null-pointer exceptions can only occur within the unchecked code or at the boundary to checked code; the checked code is free of null-pointer exceptions.

We present Granullar for Java, define the checked-unchecked boundary, and how runtime checks are generated. We evaluate our approach on real world software annotated for null-safety. We demonstrate the runtime checks, and acceptable compile-time and run-time performance impacts. Granullar enables combining a checked core with untrusted libraries in a safe manner, improving on the practicality of such a system.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Extensible languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures; D.3.4 [*Programming Languages*]: Processors—Compilers; D.1.5 [*Programming Techniques*]: Object-oriented Programming

Keywords pluggable type systems, gradual type systems, nullness, runtime checks

CC'17, February 5–6, 2017, Austin, TX, USA
© 2017 ACM. 978-1-4503-5233-8/17/02...$15.00
http://dx.doi.org/10.1145/3033019.3033032

1. Introduction

Main-stream object-oriented programming languages like Java and C# allow the dedicated `null` value for all references. This language design choice supports flexible code patterns, but has led to many errors, security vulnerabilities, and system crashes, leading Tony Hoare to call it his "Billion Dollar Mistake"[1].

Pluggable type systems enforce additional properties on top of an underlying type system. These type systems "plug" into the underlying type system and enforce an "optional" property. Pluggable type systems have been used to enforce a variety of properties, in particular null safety [1, 6, 8, 12].

Gradual type systems, as formalized by Siek and Taha first for functional [19] and then for object-oriented [18] languages bridge the gap between dynamic and static typing within a single program.

When using an additional type system like null-safety within an existing language, often only part of the program has been annotated with static types. Unchecked code may arise either because a developer has not yet added the necessary nullness annotations to their entire program, or because the program uses unannotated third party libraries.

Granullar extends a static pluggable type system that enforces null-safety, which builds on the theory of Freedom-before-Commitment [20], with a gradual type system that inserts nullness runtime checks to safely interact with unchecked software components. In combination, this guarantees that null-pointer exceptions cannot occur in checked code.

Our goal is the incremental integration of a type system for null-safety into real-world software. Developers start out verifying null-safety for the most important components of their applications—the "checked" code. However, the application depends on byte-code only libraries or modules that are not being checked (yet)—the "unchecked" code. At the boundary between checked and unchecked code, runtime checks are inserted to guard the verified system from being polluted by unexpected `null` values.

Existing systems safely handle completely checked applications. However, interactions with unchecked code are handled using defaults. Even with conservative defaults for the unchecked code, unexpected null-pointer exceptions can occur in the checked code—violating the safety assumption.

We present Granullar[2], a gradual nullness type system to safely interact with unchecked code, an implementation for Java, and an evaluation.

Outline. The structure of this paper is as follows:

- Section 2: overview of Granullar discussing the basic type system and existing defaulting options;

[1] `http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare`

[2] *Gra*dual *Null*ness *A*ugmented *R*untime

- Section 3: discussion of all interactions across the checked-unchecked boundary and runtime checks to ensure safety of the checked part;
- Section 4: implementation of Granullar as an extension of a non-gradual nullness type system for Java;
- Section 5: evaluation of Granullar to demonstrate the runtime checks and compile-time and the run-time performance impacts.

Finally, Section 6 discusses the most related work and Section 7 concludes.

2. Granullar Overview

Granullar is an extension of an existing null-safety type system [20], implemented for Java by the Nullness Checker of the Checker Framework [6, 15]. This paper focuses on extending the basic null-safety aspects, ignoring object initialization. In its basic form, this system uses two type annotations: `@Nullable` to mark reference types that might be `null`, and `@NonNull` to mark reference types that are guaranteed to never be `null`. For checked code, this system uses `@NonNull` as the default for reference types (as suggested by earlier work [4]) and uses flow-sensitive local type inference to minimize the annotation overhead in method bodies. For clarity, all annotations in checked code are explicit. The type hierarchy of this system is simple: `@Nullable` is the supertype of `@NonNull`. The type rules ensure that whenever the programming language uses a reference, e.g. for a field access or method invocation, the reference is `@NonNull`.

2.1 Defaults

Checked code can reference unchecked code and needs to determine a type signature for the unchecked code. There are three options:

Conservative defaults: The safest option is to use conservative defaults for all unchecked code, that is, use `@NonNull` as the type of method parameters and for field writes, and use `@Nullable` for method return types and for field reads. This will result in an error for every use of unchecked code which might violate an assumption of either the checked code or the unchecked code. However, this option will produce the largest number of false positives and will deter developers.

Liberal defaults: Another option is to make the types as permissive as possible, that is, use `@Nullable` as the type of method parameters and for field writes, and use `@NonNull` for method return types and for field reads. This will optimistically allow all uses of unchecked code and assume that a `@NonNull` value is returned from method invocations and field reads. This will result in no compile-time warnings about the use of unchecked code, but can result in null-pointer dereferences in both the unchecked *and* checked part of the code.

Trusted annotations: Finally, the software developer is able to provide trusted annotations that should be used for unchecked code. These annotations might be derived by manual inspection of the documentation or the code, or by static or dynamic type inference. This option gives the developer the flexibility to determine suitable annotations for unchecked code. However, as these annotations are not checked against the unchecked code, they might be incorrect, possibly leading to null-pointer dereferences in both the unchecked *and* checked part of the code.

Let us illustrate these options on the code in Figure 1. Class `Checked` is annotated with the null-safety checker, but needs to maintain a reference to unchecked class `Unchecked`, on which it calls method `work`. Let us illustrate each one of the options in sequence:

```
1  class Checked {
2    @NonNull Unchecked uc = new Unchecked();
3
4    @NonNull String call(@Nullable Object pc) {
5      @NonNull String s = uc.work(this, pc);
6      return s.toString();
7    }
8
9    @NonNull String bar(@NonNull Object pb) {
10     return pb.toString();
11   }
12 }
13
14 class Unchecked {
15   String work(Checked c, Object pw) {
16     return pw.toString() + c.bar(null);
17   }
18 }
```

Figure 1: Checked/unchecked example.

Conservative defaults: The default signature will be

```
@Nullable String work(@NonNull Checked c,
                      @NonNull Object pw)
```

This will prevent the method invocation on line 5, as parameter `pc` is `@Nullable`; also, the return type of the method is incompatible with local variable `s`. However, even once these two obstacles are resolved, there is another issue: unchecked method `work` calls back into checked code by invoking `Checked::bar`. The invocation of `bar` passes `null` to the `@NonNull` parameter `pb`. This violates the typing of the checked code and leads to a null dereference in code that is supposed to be null safe.

Liberal defaults: The default signature will be

```
@NonNull String work(@Nullable Checked c,
                     @Nullable Object pw)
```

This will result in no compile-time errors by the null-safety checker. However, there will be a null-pointer dereference in `work`, because the parameter `pw` is dereferenced. This default limits the annotation effort, but gives no guarantees.

Trusted annotations: Let us assume that the developer specified the signature as

```
@NonNull String work(@NonNull Checked c,
                     @Nullable Object pw)
```

The result will be similar to the liberal defaults: no compile-time warnings, but runtime exceptions. The trusted annotations are only used to type-check the checked part of the code; the unchecked part is not verified against these annotations.

None of these options can give a guarantee about null safety for the checked part, as illustrated with the example for conservative defaults. Note that the unchecked part could also update a non-null field with a null value. Eventually, the checked part could cause a null-pointer exception, without any unchecked code remaining on the call stack. Such violations of null safety are extremely hard to debug.

The gradual extension we propose provides type-safety in the checked portion of the program and generates no more false positives than liberal defaults. Violations are detected at the boundary between checked and unchecked code, preventing violations of null safety to propagate into the checked part.

2.2 Gradual Type System

The non-gradual type system is extended into a gradual type system by adding a new type annotation, `@Dynamic`, representing the

unknown or "dynamic" values originating from unchecked code. The `@Dynamic` type annotation is a subtype of `@Nullable` and a supertype of `@NonNull`. This new `@Dynamic` type annotation will be used as the default for unchecked code. Runtime checks will be generated whenever a `@Dynamic` reference is pseudo-assigned to a `@NonNull` reference: instead of raising a subtyping error at compile time, the runtime system ensures that the given value is actually non-null. See Section 3 for a discussion of the runtime system. All pseudo-assignments to `@Dynamic` are allowed, giving complete freedom to interacting with unchecked code. We say pseudo-assignment to include both explicit assignment instructions and implicit dataflow due to parameter passing and return values.

Note that `@Dynamic` is not written by developers within their program. It is only used to designate references coming from unchecked code.

For the example from Figure 1, the signature will use the new `@Dynamic` type as follows:

```
@Dynamic String work(@Dynamic Checked c,
                     @Dynamic Object pw).
```

This will result in the generation of a runtime check for all invocations of that method that expect a `@NonNull` result, as in the invocation on line 5.

Additionally, we create a duplicate of method `Checked::bar` that checks the argument values at runtime. This version will be used by unchecked code, which might provide invalid arguments. By generating this additional version, we guard against calls from unchecked code into checked code that might violate static types. Null safety violations are detected at the boundary between the checked and the unchecked code and prevented from propagating into the checked code. This allows detecting violations earlier than at the eventual dereference of the null value.

Granullar compiles the checked part of the code with an enhanced null-safety checker. It generates additional code, in the checked part only, to handle the interaction with unchecked code. At execution time, the checked code ensures that the unchecked code doesn't violate null safety. In our implementation for Java, the Granullar Nullness Checker is a plug-in to the standard OpenJDK Java compiler and the generated bytecode runs on a standard JVM.

Overall, by using Granullar, the developer has the ease of use of a liberal defaulting scheme, while maintaining type safety within the checked parts of the code. By vigorously protecting the checked-unchecked boundary, nullness-related bugs can be found earlier and pin-pointed to the appropriate unchecked method, instead of causing a pollution of the checked state.

3. Granullar Runtime Checks

This section identifies how unchecked code interacts with checked code, when to generate runtime checks, and what additional methods to generate to ensure null safety of the checked code while maintaining performance. The examples use Java, but the discussed issues apply similarly to other object-oriented programming languages.

In our setting, the boundary between checked and unchecked code is at a compilation unit level. A source file is either processed by the gradual nullness type system or it is not. As a result, classes are either entirely checked, or entirely unchecked.

Section 3.1 describes how values are checked at runtime. This section then discusses the boundary in order of increasing complexity of the interactions: checked code invoking an unchecked static method (Section 3.2); unchecked code invoking a checked static method (Section 3.3); dynamic method binding (Section 3.4); fields (Section 3.5); and interfaces (Section 3.6). Section 3.7 discusses how to overcome restrictions placed on object constructors, and finally, Section 3.8 discusses how to accommodate arrays and generics in the system.

```
1 class NonNullRuntime {
2   static Object checkValue(Object value) {
3     if (value == null) {
4       [error]
5     }
6     return value;
7   }
8 }
```

Figure 2: Runtime check whether `value` is non-null. The developer can choose how to handle errors, e.g. by raising a `NullPointerException` or logging the error.

Each type of relationship is illustrated by a code example and discussion.

3.1 Runtime Tests

When a `@Dynamic` type is pseudo-assigned to a `@NonNull` type, a runtime test ensures that the value is actually non-null. To simplify the generation of tests and the customization of the test implementation, the test logic is a dedicated method. The test method takes a parameter, tests it, and returns the original value back to the caller. Figure 2 shows the `checkValue` method; the implementation can choose how to handle errors: do nothing, log the violation, print a stack trace, or abort the execution. Note that even just logging this location provides valuable information: it logs when unchecked code violated a null safety property. A later null-pointer exception might not contain enough information to trace the violation back to this location. Logging the violations will support debugging efforts.

When inserting the check method into an expression, the return value is cast back to the original type of the `@Dynamic` argument. This ensures that the expression binds to the same operators and methods as the original code. Conceptually, the method can be thought of as having a type parameter `T` that is used for the return and parameter types. However, modifying the abstract syntax tree was easier using the above approach with casts.

3.2 Checked Code Invoking an Unchecked Static Method

When checked code invokes an unchecked static method, the return type at compile time will be `@Dynamic`. A pseudo-assignment of this result to `@NonNull` will result in the creation of a runtime check to ensure the returned value is indeed non-null. Because the invoked method is static, dynamic method binding does not need to be considered.

Figure 3 gives example code to illustrate this scenario.

3.3 Unchecked Code Invoking a Checked Static Method

Checked and unchecked code can also interact through parameters of checked methods. In this situation, a checked method is called by unchecked code and passed unchecked values as arguments. At this point, runtime values that are incompatible with the static parameter types could be passed, invalidating the type assumptions of the checked method. Runtime checks are required to prevent this from occurring. However, at other times, checked code may call this same checked method. In this case, runtime tests are unnecessary, since the type of the arguments has already been checked at compile time. In order to maximize performance of the checked portions of the code, runtime type checks must be avoided here.

To achieve this, for each checked static method `method_name`, a new method called `method_name_safe` is created. The new method contains the original method body in its entirety. The original method receives a new body which contains a runtime check on each of its non-null parameters followed by a call to the safe version. Then all method invocations in the checked portion of the program which refer to checked code are converted to call the safe version of

```
class A {
  static Data foo() {
    [unchecked code, might return null]
  }
}
```

(a) Unchecked class `A`.

```
class B {
  void bar() {
    @NonNull Data myData = A.foo();
    [dereference myData somewhere much later]
  }
}
```

(b) Checked class `B` that calls unchecked method `A::foo`. Note how the unchecked return value might pollute the checked part of the program, causing a null dereference far removed from the call.

```
class B {
  void bar() {
    @NonNull Data myData = (Data)
      NonNullRuntime.checkValue(A.foo());
    [dereference myData somewhere much later]
  }
}
```

(c) Transformed class `B` with added runtime check. Pollution of the checked code is prevented at the boundary to the unchecked code.

Figure 3: Checked code calling an unchecked method. See Section 3.2.

the method. This enables checked code to use the fast path, which incurs no penalty from executing runtime checks, while unchecked code remains unmodified, and thus calls the original method, which now contains runtime checks. Therefore, we have runtime checks enabling type safety between unchecked code and checked code, while not incurring any penalty in runtime performance of checked code. Again, because we are only considering static methods so far, dynamic method binding doesn't complicate the example.

Figure 4 gives example code that illustrates this scenario.

3.4 Dynamic Method Binding

This section discusses how to handle dynamic method dispatch. An unchecked class can inherit from a checked class or the other way around. In both situations, dynamic method binding might result in crossing the checked-unchecked boundary at runtime. We need to account for dynamic method binding when generating additional safe methods. The other complexity in this scenario is that the compiler cannot know at compile time if a checked class will ever be extended by an unchecked class. Our transformations assume an open world and account for possible future extensions of the class hierarchy.

Figure 5a and Figure 5b show two typical classes in an inheritance relationship. Class `G` is checked, but its subclass `H` is not. This situation cannot be detected at compile time of class `G`, as the type hierarchy can grow (that is, a class may be extended) after it has been compiled.

In this situation, there are two possible calls to the method `foo`. A call can be on an object of type `G`, `g.foo`, which is a dynamically bound method call; the call will be resolved at runtime to either a checked or unchecked object. A call can also be to the super implementation, `super.foo`, which is a statically bound call; the call will be resolved at compile time. Both of these method calls can originate either from within the checked portion of the code or within the unchecked portion of the code.

Since there is an inheritance relationship between `G` and `H`, neither class can continue to be considered entirely checked or unchecked.

```
class C {
  static void bar(@NonNull Object param) {
    [dereference param somewhere much later]
  }

  void foo() {
    C.bar(new Object());
  }
}
```

(a) Checked class `C`.

```
class D {
  void foo() {
    C.bar(null);
  }
}
```

(b) Unchecked class `D` calling checked method `C::bar`. The argument `null` violates the checked parameter type.

```
class C {
  static void bar(@Dynamic Object param) {
    C.bar_safe(
      NonNullRuntime.checkValue(param));
  }

  static void bar_safe(@NonNull Object param) {
    [dereference param somewhere much later]
  }

  static void bar() {
    C.bar_safe(new Object());
  }
}
```

(c) Checked class `C` after transformation to achieve safe calling from unchecked code.

Figure 4: Unchecked code calling a checked method. See Section 3.3.

```
class G {
  @NonNull Object foo(@NonNull Object param) {
    ...
    return new Object();
  }

  void otherMethod() {
    this.foo(new Object());
  }
}
```

(a) Checked class `G`.

```
class H extends G {
  @Override
  Object foo(Object param) {
    ...
    return null;
  }

  void otherMethod2() {
    this.foo(new Object());
  }
}
```

(b) Unchecked class `H` inheriting from checked class `G`.

Figure 5: Unchecked code calling a dynamically-bound, checked method. See Section 3.4.

```
class G {
  @NonNull Object foo(@Dynamic Object param) {
      this.foo_safe(
          NonNullRuntime.checkValue(param));
  }

  @NonNull Object foo_safe(
      @NonNull Object param) {
    ...
    return new Object();
  }

  void otherMethod() {
    this.foo_safe(new Object());
  }
}
```

Figure 6: Checked class `G` after a simple, but incorrect, transformation.

The transformations of checked classes must consider dynamic method binding.

Incorrect transformation. Figure 6 shows the transformation that would result from the system described in the previous section. It protects all calls to `foo` by creating a new method `foo_safe` which performs the original function body, and giving the original method a new function body which performs a runtime check and then calls the safe version.

The function calls in the unchecked code now call the method with the runtime tests to validate the parameters, and thus are checked for safety. The super method also calls the correct method, as the `@Override` method is unchecked, but calls the checked portion.

However, when considering dynamic method binding, a change in behaviour has occurred. In the checked portion in function `G.otherMethod`, the function call to `foo` could dispatch to `H.foo` if the runtime type of the object `this` is actually `H`. However, since we have transformed the function name to `foo_safe`, that dispatch will no longer occur. In fact, no dynamic method dispatch could occur into unchecked code from checked code since unchecked code will never contain `_safe` methods.

Correct transformation. To overcome this incorrect dynamic method binding, a runtime check is required to determine if the runtime object is an instance of a checked class. If it is, the method should dispatch to the `_safe` version, otherwise it should dispatch to the unmodified version. This runtime check cannot be avoided and will add to the overhead of all method calls, except for those where the class or method is marked final within checked code, and thus cannot be overridden.

An efficient method to implement the runtime check is to flag each class which has been type checked with a marker, and check for that marker field on the class in the runtime check. Instead of transforming each `foo` call to `foo_safe`, a new method `foo_maybe` is created, which performs the runtime check and dispatches to either `foo` or `foo_safe`, depending on the actual runtime class.

The `_maybe` method also provides a location to perform any required runtime checks on the method return values, such that the check only needs to be executed on values coming from unchecked code.

Figure 7 shows the result of the final transformation. The `foo_maybe` method checks for the existence of the field `checked_flag` which indicates that the actual runtime class of the object has been checked, and thus, it is safe to dispatch to `foo_safe` instead of `foo`.

```
class G {
  private static final
  Void checked_flag = null;

  @NonNull Object foo(@Dynamic Object param) {
    return this.foo_safe(
      NonNullRuntime.checkValue(param));
  }

  @NonNull Object foo_maybe(
      @Dynamic Object param) {
    if ([this has checked_flag]) {
      return this.foo_safe(param);
    } else {
      return NonNullRuntime.checkValue(
        this.foo(param));
    }
  }

  @NonNull Object foo_safe(
      @NonNull Object param) {
    ...
    return new Object();
  }

  void otherMethod() {
    this.foo_maybe(new Object());
  }
}
```

Figure 7: Checked class `G` after the full transformation.

3.5 Fields

Another point of interaction between checked and unchecked code is through fields. Reading from an unchecked field will result in an unknown value. The same strategy as used for unchecked method return types applies here. The type of any unchecked field is `@Dynamic` and a runtime test will be generated for a conversion from `@Dynamic` to `@NonNull` types.

However, this strategy does not extend to unchecked writes of checked fields. Unchecked code may write a value into a field that is incompatible with the field's checked type. There is no way of performing the same transformations as used for methods. Since fields cannot execute code upon use, there is no way for the compiler to instrument fields to perform any check when they are written to. Field accesses in Java and most other languages are entirely passive and don't execute code. Thus, there is no way for checked code to be informed of a write to a field.

Granullar treats any `@NonNull` field whose visibility allows it to be modified by unchecked code as unchecked itself. This conservatively provides type safety because a field which can be written by unchecked code could contain a `null` value, and there is no way for the type checker to determine if this occurs in unchecked code. Since there is no way to determine if accessible fields are actually written to, all reads must contain a check. Unfortunately, this causes runtime tests to be executed even if the field has never been touched by unchecked code. This is not ideal from a performance perspective. However, this would only apply to `@NonNull` fields which are modifiable from unchecked code. In Java, this means non-private, non-final fields, which should be used sparingly.

3.6 Interfaces

The situation with interfaces is similar to classes. However, interfaces have no method implementations, so only when combined with a class is there checked code. However, dynamic method dis-

```
interface I {
  @NonNull Object foo(@NonNull Object param);
}

class J implements I {
  @NonNull Object foo(@NonNull Object param) {
    ...
  }

  void otherMethod(@NonNull I i) {
    i.foo(new Object());
  }
}
```

(a) Checked interface `I` and implementing class `J`

```
class K implements I {
  Object foo(Object param) {
    ...
  }

  void otherMethod2(I i) {
    i.foo(null);
  }
}
```

(b) Unchecked class `K` implementing interface `I`.

Figure 8: Checked interfaces. See Section 3.6.

patch causes some complexity when combined with interfaces. See Figure 8 for an example.

In this example, the interface `I` is provided, and implemented by both checked class `J` and an unchecked class `K`. In this case, the interface is provided in checked code. Both classes contain a method call to a method in the interface, on an object typed only as `I`, that at runtime could be an instance of either `J` or `K`. Note that `K::otherMethod2` passes `null` to a checked method that expects a `@NonNull` parameter.

In order to have the call that originates from unchecked code checked at runtime, the same transformation as for other methods must be done. However, adding methods `foo_safe` and `foo_maybe` to the interface `I` would break unchecked code that does not implement these methods.

The solution to this problem depends on the features of the programming language. In Java 8, default interface methods can be used. This feature allows an interface to have a default implementation of a method, which provides an implementation using only the interface methods. This allows new methods to be added to an interface without requiring all implementing classes to be updated. The code in Figure 9 shows the result of this transformation.

Using the default method feature, the interface is modified to have the `foo_safe` and `foo_maybe` methods without breaking the implementation in `K`, and also allowing the method call in `J::otherMethod` to be transformed to `i.foo_maybe`. The call of `I.foo` in unchecked class `K` passes `null` as argument; if the parameter `i` is bound to an instance of checked class `J`, this mistake is caught by the runtime check in `J::foo`.

Note that `foo_safe` has an exception thrown as the method body in the default implementation. This implementation will never be called, since `foo_maybe` will decide between `foo` and `foo_safe` depending on the presence of the `checked_flag`, which will only be present if the implementing class has been type checked and thus contains an implementation of the `foo_safe` method. However, the `foo_maybe` method is implemented in the interface, because it is possible that it may be called on an instance of `K` (an unchecked class), and thus must work correctly in that context.

```
interface I {
  @Dynamic Object foo(@Dynamic param);

  default
  @NonNull Object foo_safe(
      @NonNull Object param) {
    throw RuntimeException();
  }

  default
  @NonNull Object foo_maybe(
      @NonNull Object param) {
    if ([this has checked_flag]) {
      return this.foo_safe(param);
    } else {
      return NonNullRuntime.checkValue(
          this.foo(param));
    }
  }
}

class J implements I {
  private static final
  Void checked_flag = null;

  @NonNull Object foo(@Dynamic Object param) {
    return this.foo_safe(
        NonNullRuntime.checkValue(param));
  }

  @NonNull Object foo_safe(
      @NonNull Object param) {
    ...
  }

  void otherMethod(@NonNull I i) {
    i.foo_maybe(new Object());
  }
}
```

Figure 9: Checked code for `I` and `J` after transformations for `foo`.

Another configuration to consider is an unchecked interface that can be implemented by checked and unchecked classes. However, since the interface itself is unchecked, it is impossible to transform it as shown previously. If we perform the transformation of methods in a checked class, all method calls with a receiver reference that is typed with the class type can be transformed safely, but method calls with a receiver reference typed with the interface type cannot be transformed and the only option is to call the original method. This call will incur runtime checks, even if the class is checked. However, this behaviour is desirable since the method parameters won't be checked statically, because the interface will only have `@Dynamic` method and parameter types. Thus, for unchecked interfaces, calls will always incur runtime checks for the parameters, even if the method is actually implemented in checked code.

3.7 Constructors

In many programming languages, constructors are not named and thus cannot have `_safe` versions. This prevents the transformations described in this section from being directly applied.

Consider Figure 10: class `L` has a constructor with one `@NonNull` parameter; the constructor is invoked both from checked and unchecked code.

There must be a way to differentiate between safe and unsafe constructors. Classes can have multiple constructors, but they are not disambiguated by name, only by the parameter list. Therefore, the transformation must create a new constructor with a different

```
class L {
  L(@NonNull Object param) {
    [dereference param somewhere much later]
  }

  void otherMethod() {
    L foo = new L(new Object());
    ...
  }
}
```

(a) Checked class `L` with a constructor that expects a `@NonNull` argument.

```
class M {
  void otherMethod2() {
    L foo = new L(null);
    ...
  }
}
```

(b) Unchecked class `M`. Note how `L` is invoked with an invalid argument.

Figure 10: Checked constructors. See Section 3.7.

```
class L {
  L(@Dynamic Object param) {
    L((SafeConstructorMarkerDummy) null,
      (Object)
        NonNullRuntime.checkValue(param));
  }

  L(@Nullable SafeConstructorMarkerDummy dp,
      @NonNull Object param) {
    [dereference param somewhere much later]
  }

  void otherMethod() {
    L foo =
      new L((SafeConstructorMarkerDummy) null,
            new Object());
    ...
  }
}
```

Figure 11: Checked class `L` after the transformations.

parameter list. However, it is possible for an end user to create a constructor with a conflicting parameter list, which would result in a compile time error. So, ideally, the transformation should use a parameter type that is impossible for the end user to use in their program. For the purposes of this system, it suffices to choose a type which the user is unlikely to use. An empty dummy class, `SafeConstructorMarkerDummy`, is added to the type system runtime code for this purpose. Adding this parameter adds some overhead to calling a constructor as an additional argument must be passed, but the value can simply be `null`.

Figure 11 shows the original constructor (with the original parameter list), which now simply calls the test function on every parameter and passes the returned values (with appropriate casts) as arguments to the new constructor along with a null value cast to the marker type `SafeConstructorMarkerDummy`. Additionally, the new constructor is shown, which has the additional marker parameter.

This achieves the same transformations in constructors as we have for methods. Since constructors do not have dynamic dispatch, they do not require a `_maybe` method. This is true for all methods which don't use dynamic dispatch, including private methods, static methods, constructors, super method calls, and `this` and `super` constructor calls.

3.8 Arrays and Generics

Arrays and generics introduce another point of complexity to the type system. Both the array type and the element type can have pluggable type annotations. For a null-safety type system, there can be non-null and nullable arrays of non-null or nullable elements. Similarly, type arguments in a generic type carry independent type annotations and may also have nullness annotations, for example, a non-null list of nullable elements.

Adapting gradual typing features to check arrays and generics is not immediately straightforward. There are several different ways to deal with them.

No checking: The simplest way to deal with arrays and generics is simply to allow assignment of arrays with `@Dynamic` elements into arrays with `@NonNull` elements without checking. This would not generate any new errors for end users to deal with, but it does open a hole in the type system, allowing unchecked values to pollute the checked portion of the program.

Runtime checks at point of assignment: To ensure safety in the checked portion of the program, runtime checks could be inserted into the code as is done for other types, in order to verify at runtime that the value conforms to the type. This poses some problems however.

Arrays can be checked by iterating over each element and testing them in succession. However, this could introduce substantial runtime cost if the array is large. Since the checks are added by the compiler, it is not explicit in the code where these checks would execute, and could have unexpected performance impacts for the developer.

Generic types on the other hand, have no such standard strategy for checking. Every generic type would need a custom runtime check function. It is possible to automatically generate this function in the compiler by observing where the generic type parameter is used in the generic class, but only for types which are themselves checked for nullness. Checks for types which are found in unchecked code could not be generated because the type parameters are erased, and are not available in bytecode. The end user would need to provide the checks for each type and the correctness of these checks cannot be automatically verified.

Restrict assignment: Finally, it could be made a type error to assign a `@Dynamic` array element type or type argument to a `@NonNull` array element type or type argument. This would force the end user to explicitly write a `@Dynamic` type annotation for array elements or type arguments up until the point of use. In some cases, where type inference or flow refinement is being used, it might be possible to do this automatically by flowing the `@Dynamic` type backwards, from an assignment back to the declaration.

This would result in many more type errors when adopting a type system, and would increase the end user effort in doing so. Further, it makes explicit the use of the `@Dynamic` type. Without this feature, the end user never has to know about or use the `@Dynamic` type. By restricting assignments, the user would need to know about and understand when to use this type annotation. This would also make the boundary between checked and unchecked code much more fluid, as the user, by writing `@Dynamic`, would essentially move the boundary into the statically checked code. This would make it more difficult to reason about where checks are inserted, and what the performance implications of those checks are.

For the initial prototype, liberal defaults were used, and arrays and generics were unchecked. However, this was deemed insufficient and improvements were explored.

Unfortunately there does not seem to be any one optimal solution; a hybrid approach appears to provide the best balance of type safety and usability. For arrays, since it is possible, checking each element to provide maximum type safety is the best option. If the end user wishes to force the check to occur at a different location, they can write the `@Dynamic` qualifier on the array element type to allow an assignment without a check.

However, since no general check for generics is possible, they must either be unchecked, decreasing safety, or restricted, which decreases usability. More work is needed to evaluate whether restricting generic assignment is worth the usability burden to gain type safety.

4. Implementation

This section discusses the implementation details of Granullar. Section 4.1 discusses how the Checker Framework's nullness type system works, Section 4.2 explains how Granullar is integrated with this nullness type system, and Section 4.3 discusses several implemented performance optimizations.

4.1 The Nullness Type System

Granullar is implemented in Java as an extension of the Nullness Checker of the Checker Framework [6, 15]. The Checker Framework operates as an annotation processor for the OpenJDK™Java compiler. An annotation processor is employed to process Java Annotations using the Annotation Processing Framework. Annotations, indicated with the `@` symbol preceding the name and parameters for an annotation, can appear in Java code preceding any type name. Nullness of types within an end developer's code are specified using annotations such as `@Nullable` or `@NonNull` on their types.

By passing the Nullness type checker's annotation processor as an argument to the standard OpenJDK™Java compiler, an end user invokes the additional type checking.

4.2 Granular

Granullar consists of 2885 lines of code (as measured by cloc v1.6[3]). It is implemented on top of the existing Nullness type checker to type check the checked portion of the code. Augmented with a `@Dynamic` type, the nullness system also identifies where to insert runtime checks. During the annotation processing phase, Granullar invokes undocumented portions of the OpenJDK™Java compiler's private classes in order to insert new methods and code into the abstract syntax tree (AST) parsed from the end developer's Java file.

Since Granullar builds standard Java class files which can run in an unmodified Java Virtual Machine (JVM), the only runtime dependency is the `NonNullRuntime` class from Figure 2, which consists of only 8 lines of Java code. The end user can run the generated bytecode with any version of this class. For example this would allow a developer to test with a version which throws exceptions, but to deploy in production a version which merely logs errors.

4.3 Optimizations

In order to improve runtime performance, values which have a `@Nullable` type annotation do not receive a runtime test. This provides two optimizations. First, there are fewer runtime tests, since some values do not need tests. Second, the test itself can be simpler. Initially, the actual compile time type was passed to the test, so that the test can check the runtime value against the compile time type. However, there are only ever two options, either `null` values are allowed, or disallowed. We can therefore use the presence of the test to indicate `null` values are not permitted and the test need only check the value, not the type.

5. Evaluation

The evaluation seeks to answer three questions about Granullar. Section 5.1 describes the experimental setup. Section 5.2 experimentally evaluates the behaviour. Section 5.3 describes the performance characteristics. Finally, Section 5.4 discusses the usability and applicability of Granullar to the motivating scenarios in the context of alternative options for improving type safety.

5.1 Setup

The evaluation uses two existing projects, Daikon[4] and plume-lib[5], that are in a client-library relationship. The projects consist of 169,902 and 13,797 lines of Java code, respectively (as measured by cloc v1.6).

Both projects are fully annotated with the nullness type system. To simulate an environment where only a part of a project is annotated and checked, the part of the program that is intended to be unchecked is compiled with the standard Java compiler, which ignores the nullness type system. The part of the program to be checked is compiled using the nullness type system.

Two different configurations are used to simulate different motivating examples. In the first, plume-lib is checked using the gradual nullness type system, and Daikon is compiled with the standard Java compiler. This represents a scenario where a medium size library uses the gradual type system, and a project using the library does not.

The second configuration splits plume-lib into two parts, a small utility component called `ArraysMDE` and the remaining code which uses the utility. This represents the scenario where a project is transitioning to being checked by a new type system, but some parts of the project have not been annotated yet.

Both projects, plume-lib and Daikon, have extensive unit test suites, can exercise code paths in each project. Starting with a code base which is already checked by the nullness type system allows us to select an arbitrary checked/unchecked boundary for our evaluation. A future evaluation could investigate the effectiveness of Granullar during development.

5.2 Experiments

We demonstrate that Granullar fulfills two properties. First, a program with no null-safety errors should have no runtime checks fail when partially checked by the gradual type system. Second, a program with runtime null pointer errors that cross the checked-unchecked boundary should result in a runtime check failing at the boundary.

Initial testing was a set of twenty synthetic tests, where an artificial boundary was created in a Java code snippet, and a null value made to cross this boundary. A portion of the code is compiled with Granullar, and the remainder with the standard javac compiler. Each test is executed and it is verified that the null value is caught at the boundary check.

To test real world scenarios, both plume-lib/Daikon configurations described in Section 5.1 are compiled and the associated unit test suites are run. Since the entire project is already checked using the nullness type system, this verifies the first correctness property, that no runtime boundary checks fail when there are no errors.

Next, a few null value errors are manually inserted into the unchecked portion of each configuration, such that the null value

[3] `http://cloc.sourceforge.net/`

[4] `http://plse.cs.washington.edu/daikon/`

[5] `http://mernst.github.io/plume-lib/`

	Before	Optim.
Methods added	1014	1014
Constructors added	104	104
Runtime checks inserted	1616	1352

Figure 12: Runtime tests inserted in plume-lib at compile time.

	plume-lib		Daikon	
	Before	Optim.	Before	Optim.
`_maybe` calls	27,408	27,408	4,087,041	4,087,035
Runtime checks	1,944	1,533	162,976	135,611

Figure 13: Numbers of executed `_maybe` calls and runtime checks.

	plume-lib	Daikon
Standard Java	1.158	5.590
Unoptimized Granullar	1.202	7.974
Optimized Granullar	1.196	7.696

Figure 14: Runtime of test suite execution in seconds.

crosses the checked-unchecked boundary. The unit test suite is used to verify that these errors are caught by the boundary checks, and that they do not propagate further into the checked portion of the code.

Together, these experiments show that the system performs correctly under the scenarios tested, both in synthetic comprehensive tests, and in real world software.

5.3 Performance

The evaluation of performance aims to characterize the compile time and runtime performance overhead of the gradual nullness type system. The specific performance overhead depends on the size of the checked-unchecked boundary. At compile time, this is defined by how many runtime checks and new methods are inserted. At runtime, however, the size of the boundary is instead defined by how many times threads of execution cross the boundary, and thus invoke runtime checks, as well as how many times the dynamically dispatched method is invoked. Since the overhead is dependent on this aspect, the goal is only to characterize it for the given motivating scenarios.

To measure the performance, both of the configurations described in Section 5.1 are compiled, and the unit test suite is timed. The tests are run 100 times and the results averaged. Additionally, performance with and without the optimizations described in Section 4.3 is measured, along with the performance of the unit test suite without any runtime checks (compiled using the standard Java compiler). The test machine is running Ubuntu 14.04 LTS on a VMWare Workstation 10 virtual machine. The hardware has a six core AMD FX 6100 processor with 8 GB of RAM.

Inserting gradual checks adds an almost 40% performance penalty to the compiler over the non-gradual nullness type system, which in turn is several times slower than the standard Java compiler.

Runtime, however, is more complex. As mentioned, the performance overhead depends substantially on the size of the boundary. Figure 14 shows that when a small component of plume-lib is unchecked, a very small boundary is created, and the performance overhead is only 4%. However, when plume-lib is checked, and Daikon is unchecked, a very large boundary is formed that results in a larger performance overhead of 42%.

Figure 14 also shows that adding the optimizations only reduces the performance overhead to 3% and 38% respectively. The reason for this limited improvement is clear from the numbers in Figure 13: while the optimizations do reduce the number of runtime type checks invoked, the majority of the overhead is caused by the `_maybe` method calls.

Additionally, the reduction in the number of checks invoked is very small. This is due to the fact that most types in a program are `@NonNull`, and thus cannot be optimized away. This is made clear in Figure 12, which shows the number of tests actually inserted at compile time: even with the optimizations, most tests are still needed. Even so, this optimization allows a more efficient test that reduces overhead even when a test is required.

This suggests the best target for optimization are the `_maybe` methods. The Java Virtual Machine (JVM) spends substantial effort optimizing dynamic method calls, both in optimizing away virtual method calls, as well as using a virtual dispatch table to execute them quickly. Since our `_maybe` method hard-codes its own dispatching logic, it cannot benefit from these optimizations.

5.4 Usability

We compare gradual nullness with adopting conservative defaults as an option for increasing type safety when annotating only part of a program with null-safety annotations. Using conservative defaults means that any unchecked method can only be provided the most restrictive type, `@NonNull`, as an argument, and is assumed to return the most permissive type, `@Nullable`.

By choosing these defaults, the type system increases the safety when calling unchecked methods. However, type safety holes remain. If an unchecked method calls a checked method, there is no way for conservative defaults to prevent a null value from being supplied to a `@NonNull` parameter. The example from Figure 1 demonstrates this weakness in practice. To overcome this would require all externally visible methods to accept `@Nullable` parameters. This would severely limit the usefulness of the system.

Even accepting this limitation, adopting a conservative defaulting policy over the standard defaulting policy increases the developer effort for annotating or adding null value checks to their code. To evaluate this, we removed existing annotations and used conservative defaults for plume-lib and Daikon. This resulted in 1,581 and 11,316 type errors in each project, respectively.

A later analysis of the types involved showed that approximately 15% of the tests inserted into plume-lib involved either generic or array types. Further, 31% of the tests executed involved these types as well. This suggests that handling array and generic types well is important.

In conclusion, gradual checking using Granullar requires no additional developer effort to support, does not have the correctness limitations of conservative defaulting, and instead gives a runtime guarantee that the checked code is not polluted with unwanted `null` values.

6. Related Work

Modern languages make it easier to write programs without `null` values, for example, by using `Optional` in Java 8, `Option` in Scala, `Maybe` in Haskell, Flow, and Elm, and null-safe types and operators in Kotlin and Groovy. However, a lot of code is still being written that uses `null` without taking advantage of these language features.

In this section, we discuss some closely related work on pluggable type systems (Section 6.1), gradual type systems (Section 6.2), and nullness type systems (Section 6.3).

6.1 Pluggable Type Systems

Bracha [3] defines pluggable typing by two features: type annotations must be optional in the language syntax, and types that *are* specified must have no effect on the runtime semantics of the language. Bracha argues that this definition leads to type systems that function as a plug-in to the language. An end developer can use

any number of available type systems in a given program. Bracha suggests that this would allow traditionally exotic research type systems to see broad use and wider adoption.

JavaCOP is a framework developed by Andreae et al. [1] to provide a declarative rule based language to define pluggable type systems in Java. They posit that their rule based system is easy for type system designers, and closely matches traditional syntax directed type rules.

Papi et al. introduced the Checker Framework that implements a pluggable type system as a plug-in for the OpenJDK Java compiler [15]. The authors' design goal was to ensure that it was easy for type system designers to build simple type systems, but still possible to build powerful type systems. The authors provide four type systems as a demonstration of their framework, including a nullness type system. The authors also compare the framework with other pluggable type systems including JavaCOP [1] and JQual [12].

Ekman et al. built a pluggable null-safety type checker for Java [8] based upon the null-safety type system of Fähndrich and Leino [9]. They used inference and whole program analysis to determine safe types to assign to unannotated code. Their system is based on the JastAdd Extensible Java Compiler [7] and demonstrated that compiler's applicability to pluggable type systems.

6.2 Gradual Type Systems

Siek and Taha introduced their version of gradual typing in functional languages [19] and then expanded it to object-oriented languages [18]. The authors present a formal type system that supports gradual typing, and provide a formal language calculus. The authors define the run-time semantics of their language via a conversion to a language with runtime casts. This idea of inserting runtime casts or checks is the basis of our work.

Gronski et al. propose a language and a type system called SAGE [13] that combines the Dynamic type with refinement types to help overcome the undecidability of refinement types. They eschew traditional compile time guarantees, instead inserting run time casts. However, when a cast fails at runtime, this failure can be fed back into the compiler, so future compilations can catch the cast error at compile time.

Schwerter et al. [2] develop a gradual-effect system based on a generic-effect framework that models effects with a set of privileges, which represent allowable effects during evaluation of an expression, and predicates, which check that an expression has the correct permissions. The authors note difficulty in dealing with dynamic behaviour, as seen in Gordon et al.'s JavaUI work [11], as motivation for using gradual checking. They introduce the notion of an unknown permission, and adapt Siek and Taha's consistency relation to the concept of sets of permissions.

Gradual type systems for Javascript allow the transition from dynamically typed Javascript to languages with stronger guarantees. TypeScript recently added null- and undefined-aware types[6], which provide additional null-safety to TypeScript programmers. Flow similarly has Maybe Types[7]. Neither system provides features comparable to Granullar.

Kotlin is a JVM language that provides several improvements over Java, while staying interoperable with Java. Kotlin adds null-safety using type annotations[8] and uses platform types[9] to interact with unchecked Java types. Kotlin produces runtime checks at dereferences of platform types, which can happen much later than when the value crossed the checked-unchecked boundary. There is also no discussion of unchecked Java code violating Kotlin's null safety.

Aspect-Oriented Programming [14] can be used to instrument code with cross-cutting concerns. One could think of Granullar as instrumenting the checked-unchecked boundary with the additional runtime checks. However, no current AOP system supports pluggable type systems to specify where instrumentation should happen.

6.3 Nullness Type Systems

One of the earliest null-safety type systems for object-oriented languages [9], introduced by Fähndrich and Leino, can detect null-dereferences statically when code is annotated with nullness annotations. A type, `C`, is split into either `C?`, indicating *possibly-null* values, or `C!`, indicating *non-null* values.

Summers and Müller[20] evaluate several different object initialization schemes. *Raw types*, as introduced by Fähndrich and Leino for their non-null type system [9], were not sufficiently expressive to allow the initialization patterns that Summers and Müller envisioned. *Delayed Types*, by Fähndrich and Xia [10] could be only two of: sound, sufficiently expressive, or easy to use, but not all three. *Attached Types*, which are found in Eiffel, have a specification which is unsound, but an implementation that is (by experiment) sound but not expressive, and also not modular. Finally, *Masked Types*, developed by Qi and Myers [16], provide soundness, modularity, and exceptional expressiveness, but Summers and Müller felt that they were too complex. The Nullness Checker, on which Granullar builds, uses Freedom-before-Commitment [20] to enforce correct object initialization in checked code. Granullar currently only enforces null-safety at the checked-unchecked boundary. Checking object initialization is left as future work.

Servetto et al. [17] propose placeholders and placeholder types as an alternative to null pointers for initializing circular data structures. Placeholders are similar to null values when initializing a circular structure, but are guaranteed only to exist within a local scope. When combined with placeholder types to restrict the usage of placeholders, they can be guaranteed never to cause a `PlaceholderException`. This reduces the need for traditional null values in Java.

Chalin and James [4] presented a large case study arguing for non-null references as the default. The Nullness Checker uses `@NonNull` as the default, except for locations that can be flow-sensitively refined (e.g. local variables). However, these defaults only apply for checked code. In Section 2 we discuss possible defaults for unchecked code and argue why we take a gradual approach with Granullar.

Cornu et al. [5] present Casper, an approach that uses "causality traces" and "ghost objects" to trace null-pointer exceptions to their root cause. In contrast, Granullar protects the checked-unchecked boundary and detects violations at that boundary.

7. Conclusions

This paper presents Granullar, a gradual nullness type system for Java that improves type safety in partially annotated programs. It is motivated by limitations observed in using the non-gradual nullness type system in other projects. This paper presents the difficulties which the object-oriented nature of Java-like languages imposes on gradual typing, and how to overcome these challenges.

Along with the implementation, this paper presents an evaluation exploring correctness, performance overhead, and applicability to the motivating examples. The evaluation demonstrates correctness

[6] `https://www.typescriptlang.org/docs/release-notes/typescript-2.0.html`

[7] `https://flowtype.org/docs/five-simple-examples.html#nullable-types`, `https://flowtype.org/docs/nullable-types.html`

[8] `https://kotlinlang.org/docs/reference/null-safety.html`

[9] `https://kotlinlang.org/docs/reference/java-interop.html#null-safety-and-platform-types`

using both synthetic tests and errors inserted into real world software. It demonstrates effectiveness of the type system for increasing type safety in partially annotated software, even when compared with conservative defaulting, and with less annotation overhead for the developer. Finally, the evaluation considers the performance overhead in real world programs, again modeled on the motivating examples. It shows that both runtime overhead as well as compile time overhead are acceptable in some instances.

7.1 Future Work

One of the goals of gradual typing is using static type information to improve performance. We are investigating optimizations like removing null-pointer checks from the JVM.

Additionally, instrumenting bytecode would provide more feasible options for ensuring type safety of fields: instead of performing runtime checks for all field reads, instrument the unchecked bytecode to perform the checks on all writes.

Further improving performance of Granullar is another avenue of future research. The majority of the overhead is incurred by the dynamic method dispatch and a substantial opportunity for optimizations exists here. By leveraging the JVM's `invokedynamic` instruction, and providing our `_maybe` method as the bootstrap function, we want to benefit from the optimization work done in the JVM.

Handling more programming language features and other programming languages is also interesting future work. We outlined possible work to better support arrays and generics. Interfaces currently rely on Java 8 default interface methods. Support for interfaces in languages that do not provide default interface methods remains open.

Finally, a full formalization and proof of the correctness of Granullar is left as future work.

Acknowledgments

We thank the reviewers for their constructive comments. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada. This material is based upon work supported by the United States Air Force under Contract No. FA8750-15-C-0010. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Air Force and the Defense Advanced Research Projects Agency (DARPA).

References

[1] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 57–74. ACM, 2006.

[2] F. Bañados Schwerter, R. Garcia, and É. Tanter. A theory of gradual effect systems. In *International Conference on Functional Programming (ICFP)*, pages 283–295. ACM, 2014.

[3] G. Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

[4] P. Chalin and P. R. James. Non-null references by default in java: Alleviating the nullity annotation burden. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 227–247. Springer, 2007.

[5] B. Cornu, E. T. Barr, L. Seinturier, and M. Monperrus. Casper: Automatic tracking of null dereferences to inception with causality traces. *Journal of Systems and Software*, 122:52–62, 2016.

[6] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. W. Schiller. Building and using pluggable type-checkers. In *International Conference on Software Engineering (ICSE)*, pages 681–690. ACM, 2011.

[7] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–18. ACM, 2007.

[8] T. Ekman and G. Hedin. Pluggable checking and inferencing of nonnull types for Java. *Journal of Object Technology*, 6(9):455–475, 2007.

[9] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 302–312. ACM, 2003.

[10] M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 337–350. ACM, 2007.

[11] C. S. Gordon, W. Dietl, M. D. Ernst, and D. Grossman. Java UI: effects for controlling UI object access. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 179–204. Springer, 2013.

[12] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 321–336. ACM, 2007.

[13] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242. Springer, 1997.

[15] M. M. Papi, M. Ali, T. L. Correa Jr, J. H. Perkins, and M. D. Ernst. Pluggable type-checking for custom type qualifiers in Java. Technical report, MIT CSAIL, 2007.

[16] X. Qi and A. C. Myers. Masked types for sound object initialization. In *Principles Of Programming Languages (POPL)*, pages 53–65. ACM, 2009.

[17] M. Servetto, J. Mackay, A. Potanin, and J. Noble. The billion-dollar fix. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 205–229. Springer, 2013.

[18] J. Siek and W. Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 2–27. Springer, 2007.

[19] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

[20] A. J. Summers and P. Müller. Freedom before commitment: a lightweight type system for object initialisation. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1013–1032. ACM, 2011.