Scala with Explicit Nulls

Abel Nieto
University of Waterloo, Canada
anietoro@uwaterloo.ca

Yaoyu Zhao
University of Waterloo, Canada
y437zhao@edu.uwaterloo.ca

Ondřej Lhoták
University of Waterloo, Canada
olhotak@uwaterloo.ca

Angela Chang
University of Waterloo, Canada
yue.chang@edu.uwaterloo.ca

Justin Pu
University of Waterloo, Canada
justin.pu@edu.uwaterloo.ca

Abstract

The Scala programming language makes *all* reference types *implicitly nullable*. This is a problem, because `null` references do not support most operations that do make sense on regular objects, leading to runtime errors. In this paper, we present a modification to the Scala type system that makes nullability *explicit* in the types. Specifically, we make reference types *non-nullable* by default, while still allowing for nullable types via *union types*. We have implemented this design for explicit nulls as a fork of the Dotty (Scala 3) compiler. We evaluate our scheme by migrating a number of Scala libraries to use explicit nulls. Finally, we give a *denotational semantics* of *type nullification*, the interoperability layer between Java and Scala with explicit nulls. We show a soundness theorem stating that, for variants of System F_ω that model Java and Scala, nullification preserves values of types.

2012 ACM Subject Classification Software and its engineering → General programming languages; Theory of computation → Denotational semantics; Theory of computation → Type theory; Software and its engineering → Interoperability

Keywords and phrases Scala, Java, nullability, language interoperability, type systems

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.25

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at `https://doi.org/10.4230/DARTS.6.2.14`.

Funding This research was supported by the Natural Sciences and Engineering Research Council of Canada and by the Waterloo-Huawei Joint Innovation Lab.

Acknowledgements We would like to thank Sébastien Doeraene, Fengyun Liu, Guillaume Martres, and Martin Odersky for their feedback on our explicit nulls design.

1 Introduction

Scala inherited elements of good design from Java, but it also inherited at least one misfeature: the `null` reference. In Scala, like in many other object-oriented programming languages, the `null` reference can be typed with *any reference type*. This leads to runtime errors, because

© Abel Nieto, Yaoyu Zhao, Ondřej Lhoták, Angela Chang, and Justin Pu;
licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).
Editors: Robert Hirschfeld and Tobias Pape; Article No. 25; pp. 25:1–25:26

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

null does not (and cannot) support almost *any* operations. For example, the program below tries to read the length field of a string, only to find out that the underlying reference is null. The program then terminates with the infamous NullPointerException[1].

```
val s: String = null                     // ok: String is a nullable type
println(s"s has length " + s.length) // throws a NullPointerException
```

Errors of this kind are very common, and can sometimes lead to security vulnerabilities. Indeed, "Null Pointer Dereference" appears in position 14 of the *2019 CWE Top 25 Most Dangerous Software Errors*, a list of vulnerability classes maintained by the MITRE Corporation [21]. As of November 2019, a search for "null pointer dereference" in MITRE's vulnerability database[2] returned 1429 entries.

The root of the problem lies in the way that Scala structures its type hierarchy. The null reference has type Null, and Null is considered to be a subtype of any reference type. In the example above, Null is a subtype of String, and so the initializer val s: String = null is allowed. We could say that in Scala, (reference) types are *implicitly nullable*. The alternative is to have a language where nullability has to be *explicitly* indicated. For example, we can re-imagine the previous example in a system with explicit nulls (the notation String|Null stands for the *union type* "String or Null"):

```
val s: String = null        // error: Null is not a subtype of String
val s: String|Null = null // ok: s is explicitly marked as nullable
println("s has length " + s.length)     // error: String|Null does not have a 'length' field
if (s != null) println("s has length " + s.length) // ok: we checked that s is not null
```

In a world with explicit nulls, the type system can keep track of which variables are potentially null, turning runtime errors into compile-time errors.

Our contributions, implemented on top of the Dotty (Scala 3) compiler and currently under consideration for inclusion in Scala 3, are as follows:

- We retrofitted Scala's type system with a mechanism for tracking nullability, using union types. To improve usability of nullable values in Scala code, we also added a simple form of flow typing to Scala.
- So that Scala programs can interoperate with Java code, where nulls remain implicit, we present a *type nullification* function that turns Java types into equivalent Scala types.
- We evaluate the design by migrating multiple Scala libraries to explicit nulls. The main findings are that most of the effort in migrating Scala code to explicit nulls comes from Java interoperability, and that the effort is significant for some libraries.
- Finally, we formalize type nullification using variants of System F_ω that have been augmented to model implicit and explicit nulls. Using denotational semantics, we prove a *soundness* theorem for nullification, saying that nullification preserves values of types.

2 A New Type Hierarchy

To understand the special status of the Null type, we can inspect the current Scala type hierarchy, shown in Figure 1. Roughly, Scala types can be divided into *value* types (subtypes of AnyVal) and *reference* types (subtypes of AnyRef). The type Any then stands at the top of the hierarchy, and is a supertype of both AnyVal and AnyRef (in fact, a supertype of

[1] https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html
[2] https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=NULL+Pointer+Dereference

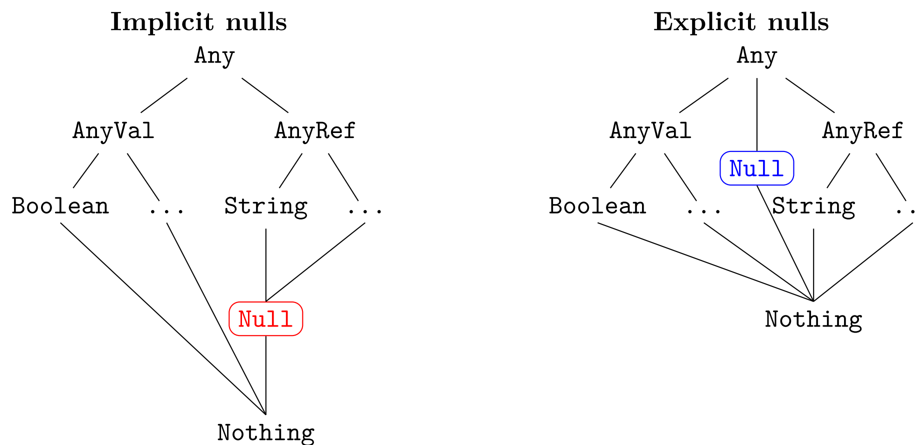

Figure 1 Alternative Scala type hierarchies with implicit and (our design) explicit nulls.

every other type). Conversely, `Nothing` is a subtype of all types. Finally, `Null` occupies an intermediate position: it is a subtype of all *reference* types, but not of the value types. This justifies the following typing judgments:

```
val s: String = null // ok: String is a reference type
val i: Int = null    // error: Int is a value type
```

This is what makes `nulls` in Scala *implicit*. In order to make `nulls` *explicit*, we need to dislodge the `Null` type from its special position, so that it is no longer a subtype of all reference types. We achieve this by making `Null` a direct subtype of `Any`. This new type hierarchy, which underlies our design, is also shown in Figure 1. With the new type hierarchy we get new typing judgments:

```
val s: String = null      // error: reference types like String are no longer nullable
val i: Int = null         // error: Int is a value type
val sn: String|Null = null // ok: Null <: String|Null
```

`String|Null` is a *union type*. In general, the union type `A|B` (read "`A` *or* `B`") contains *all* values of both `A` and `B`, as indicated by the subtyping judgements `A <: A|B` and `B <: A|B`. Union types are a new feature present in Dotty but not in Scala 2 and, as the example shows, they allows us to encode nullability.

The explicit nulls hierarchy is still unsound in the presence of uninitialized values:

```
1 class Person {
2   val name: String = getName()
3   def getName(): String = "Person" + name.length // 'name' is null here
4 }
5 val p = new Person() // throws a NullPointerException
```

Because, after allocation, the fields of Scala classes are initialized to their "default" values, and the default value for reference types is `null`, when we try to access `name.length` in line 3, `name` is `null`. This produces a `NullPointerException`. While ensuring sound initialization is an interesting challenge, it is not one we tackle in this paper. Developing a sound initialization scheme for Scala, while balancing soundness with expressivity, remains future work. We review some of the existing approaches in Section 7.

2.1 Fixing a Soundness Hole

Even though explicit nulls do not make the Scala type system sound (e.g. there remain null-related soundness holes related to incomplete initialization of class fields), they *do* remove the specific source of unsoundness identified by Amin and Tate [3]. This class of bugs, reported in 2016 and still present in Scala and Dotty, happens due to a combination of implicit nullability and type members with arbitrary lower and upper bounds. For example, the example presented by Amin and Tate [3] crucially relies on being able to construct a term t that has *both* type e.g. `LowerBound[Int]` *and* `UpperBound[String]`, two type applications unrelated by subtyping. Because of implicit nullability, `null` has both types, which makes the unsoundness possible. With our explicit nulls design, the typing above is no longer possible, so the runtime error becomes a compile-time error.

3 Java Interoperability

One of Scala's strengths is its ability to seamlessly use Java libraries. Because both languages are compiled down to Java Virtual Machine (JVM) *bytecode* [18], Java libraries appear to Scala code as any other Scala library would. The interaction can also happen in the opposite direction: Java code can use Scala libraries.

Because reference types remain implicitly nullable in Java, we need a way to "interpret" Java types as Scala types, where nullability is explicit. For example, if a Java method returns a `String`, then the Java type system will allow `null` as a return value. If we use said method from Scala, we need to interpret the method's type as `String|Null`.

In the opposite direction, when Java code uses Scala libraries, the problem is simpler because Java types are less precise than Scala types. In particular, both the Scala types `String` and `String|Null` can be interpreted as the Java type `String`, which includes the null value.

3.1 Type Nullification

Type nullification is the process of translating Java types to their Scala equivalents, in the presence of explicit nulls. By equivalent, we mean that if type nullification sends type A to type B, the values of A and B must be the same. Below are two examples of the behaviour we want from nullification:

- The values of the $\texttt{String}_{\texttt{Java}}$ type[3] are all finite-length strings (e.g. `"hello world"` and `""`), plus the value `null`. By contrast, the values of $\texttt{String}_{\texttt{Scala}}$ are *just* all finite-length strings (but not `null`). This means that nullification must map $\texttt{String}_{\texttt{Java}}$ to $\texttt{String}_{\texttt{Scala}}\texttt{|Null}$.
- Similarly, we can think of a Java method with signature
  $\texttt{String}_{\texttt{Java}}$ `getName(`$\texttt{String}_{\texttt{Java}}$ `s)`
  as representing a function from $\texttt{String}_{\texttt{Java}}$ to $\texttt{String}_{\texttt{Java}}$ (i.e. `getName:` $\texttt{String}_{\texttt{Java}} \to \texttt{String}_{\texttt{Java}}$). Suppose that $f \in \texttt{String}_{\texttt{Java}} \to \texttt{String}_{\texttt{Java}}$. Notice that f can take `null` as an argument, and return `null` as a result. This means that nullification should return $\texttt{String}_{\texttt{Scala}}|\texttt{Null} \to \texttt{String}_{\texttt{Scala}}|\texttt{Null}$ in this case.

Here is why "preserves values of a type" is a useful correctness criterion for nullification. Suppose that nullification instead *underapproximated* a type's values. For example, we could turn $\texttt{String}_{\texttt{Java}}$ into $\texttt{String}_{\texttt{Scala}}$. We might then call e.g. the `length` method on the

[3] We write $\texttt{T}_{\texttt{Java}}$ and $\texttt{T}_{\texttt{Scala}}$ for Java's and Scala's view of the same type `T`, respectively.

$$F_{\text{null}}(R) = R|\texttt{Null} \qquad \text{if } R \text{ is a reference type} \qquad \text{(FN-Ref)}$$
$$F_{\text{null}}(R) = R \qquad \text{if } R \text{ is a value type} \qquad \text{(FN-Val)}$$
$$F_{\text{null}}(T) = T|\texttt{Null} \qquad \text{if T is a type parameter} \qquad \text{(FN-Par)}$$
$$F_{\text{null}}(C\texttt{<}R\texttt{>}) = C\texttt{<}A_{\texttt{null}}(R)\texttt{>}|\texttt{Null} \qquad \text{if } C \text{ is Java-defined} \qquad \text{(FN-JG)}$$
$$F_{\text{null}}(C\texttt{<}R\texttt{>}) = C\texttt{<}F_{\texttt{null}}(R)\texttt{>}|\texttt{Null} \qquad \text{if } C \text{ is Scala-defined} \qquad \text{(FN-SG)}$$
$$F_{\text{null}}(A\&B) = (A_{\text{null}}(A)\&A_{\text{null}}(B))|\texttt{Null} \qquad \text{(FN-And)}$$

$$A_{\text{null}}(R) = R \qquad \text{if } R \text{ is a reference type} \qquad \text{(AN-Ref)}$$
$$A_{\text{null}}(T) = T \qquad \text{if } T \text{ is a type parameter} \qquad \text{(AN-Par)}$$
$$A_{\text{null}}(C\texttt{<}R\texttt{>}) = C\texttt{<}A_{\texttt{null}}(R)\texttt{>} \qquad \text{if } C \text{ is Java-defined} \qquad \text{(AN-JG)}$$
$$A_{\text{null}}(C\texttt{<}R\texttt{>}) = C\texttt{<}F_{\texttt{null}}(R)\texttt{>} \qquad \text{if } C \text{ is Scala-defined} \qquad \text{(AN-SG)}$$
$$A_{\text{null}}(R) = F_{\text{null}}(R) \qquad \text{otherwise} \qquad \text{(AN-FN)}$$

F_{null} is applied to the types of fields, and argument and return types of methods of *every* Java-defined class. We try the rules in top-to-bottom order, until one matches.

Figure 2 Type nullification functions.

$\texttt{String}_{\texttt{Scala}}$, only to find out that the underlying reference was `null`. Another way of saying this is that underapproximations are *unsound for reads*. Similarly, consider what would happen were nullification to *overapproximate* types. For example, we could map $\texttt{String}_{\texttt{Java}}$ to Scala's `Any`. This is sound for reads, because we cannot call `length` on an `Any`. However, if the Java type were to appear *contravariantly*, e.g. as a method argument, then the Scala code could pass an `Any` (a value of any type), where the Java code expects a $\texttt{String}_{\texttt{Java}}$, leading to runtime errors. That is, overapproximations are *unsound for writes*. This leads us back to our desired goal of preserving values of types. For now, we only argue informally that nullification preserves values of types. Section 6 formalizes this idea using denotational semantics and proves soundness of the rules on a core calculus.

Nullification can be described with a pair of mutually-recursive functions ($F_{\text{null}}, A_{\text{null}}$) that map Java types to Scala types. The functions are defined in Figure 2 and described below. But first, a word about how nullification is applied. The Dotty compiler can load Java classes in two ways: from source or from bytecode. In either case, when a Java class is loaded, we apply F_{null} to the types of fields and the argument and result type of methods. The resulting class with modified fields and methods is then made accessible to the Scala code. Below is some intuition and example for the different nullification rules.

Case (FN-Ref and FN-Val) These two rules are easy: we nullify reference types but not value types, because *only* reference types are nullable in Java. Here is an example Java class and its translation (given in Java syntax enhanced with union types and a `Null` type):

```
// Java class
class C {
  String s;
  int x;
}
```

```
// After nullification
class C {
  String|Null s;
  int x;
}
```

Case (FN-Par) Since type parameters are always nullable in Java, we need to nullify them as well.

```
// Java class
class C<T> {
  T foo() {...}
}
```

```
// After nullification
class C<T> {
  T|Null foo() {...}
}
```

For example, if we have `c: C<Boolean>`, then `c.foo()` now returns a `Boolean|Null`, as opposed to just `Boolean` like it used to.

Case (FN-JG) This rule handles generics `C<T>`, where `C` is a class defined in Java (*Java-defined*). The rule is designed to reduce the number of redundant nullable types we need to add. Let us look at an example:

```
// Java
class Box<T> { T get(); }
class BoxFactory<T> {
  Box<T> makeBox();
}
```

```
// After nullification
class Box<T> { T|Null get(); }
class BoxFactory<T> {
  Box<T>|Null makeBox();
}
```

Suppose we have a `BoxFactory<String>`. Notice that calling `makeBox` on it returns a `Box<String>|Null`, not a `Box<String|Null>|Null`, because of **FN-JG**. This seems at first glance unsound, because the box itself could contain `null`. However, it is sound because calling `get` on a `Box<String>` returns a `String|Null`.

Generalizing from the example, we can see that it is enough to nullify the type application `C<T>` as `C<T>|Null`. That is, it is enough to mark the type as nullable only at the top level, since uses of `T` in the body of `C` will be nullified as well, *if `C` is Java-defined*. Notice that the correctness argument relies on our ability to patch all Java-defined classes that transitively appear in the argument or return type of a field or method accessible from the Scala code being compiled. All such classes must be visible to the Scala compiler in any case, and will thus be nullified, so this requirement is satisfied by the implementation.

In fact, the rule is a bit more complicated than we have explained so far. The full rule is $F_{\text{null}}(C\texttt{<}R\texttt{>}) = C\texttt{<}A_{\texttt{null}}(R)\texttt{>|Null}$. Notice that in fact we *do* transform the type argument, but do so using A_{null} instead of F_{null}. A_{null} is a version of F_{null} that does not add `|Null` at the top level. A_{null} is needed for cases where we have nested type applications, and it is explained in more detail below. Here is a sample application of F_{null} to a nested type application, assuming that C, D, and `String` are all Java-defined:

$$
\begin{aligned}
F_{\text{null}}(C\texttt{<}D\texttt{<String>>}) &= C\texttt{<}A_{\texttt{null}}(D\texttt{<String>})\texttt{>|Null} \\
&= C\texttt{<}D\texttt{<}A_{\texttt{null}}(\texttt{String})\texttt{>>|Null} \\
&= C\texttt{<}D\texttt{<String>>|Null}
\end{aligned}
$$

Notice how we only add `|Null` at the outermost level. This minimizes the number of changes required to migrate existing Scala code with Java dependencies.

Case (FN-SG) This rule handles the mirror case, where the Java code refers to a generic C<T> in which C is a class defined in Scala (*Scala-defined*). For example, assuming that `Box` is Scala-defined, we get:

```
// Java code that refers to Scala class Box
class BoxFactory<T> {
  Box<T> makeBox();
}
```

```
// After nullification
class BoxFactory<T> {
  Box<T|Null>|Null makeBox();
}
```

Notice that unlike the previous rule, **FN-SG** adds `|Null` to the type argument, and not just to the top level. This is needed because nullification is only applied to Java classes, and not to Scala classes. We then need a way to indicate that, in the example, the returned `Box` may contain `null`.

Case (FN-And) This rule just recurses structurally on the components of the type. Even though Java does not have intersection types, we sometimes encounter them during nullification, because the Scala compiler desugars some Java types using intersections. For example, the Java type `Array[T]`, where `T` has no supertype, is represented in Scala as `Array[T & Object]`.

As previously mentioned, A_{null} is a helper function that behaves mostly like F_{null}, but never nullifies types at the top level. A_{null} is useful because we want to avoid adding superfluous |`Null` unions whenever possible.

4 Flow Typing

To improve usability of nullable types, we added a simple form of flow-sensitive type inference to Scala [15]. The general idea is that sometimes, by looking at the control flow, we can infer that a value previously thought to be nullable (due to its type) is no longer so.

4.1 Supported Cases

Below we list the cases supported by flow typing. In the examples, the notation `???` stands for an unspecified expression of the appropriate type[4]:

- *Branches of an if-expression.* If an if-expression has a condition `s != null`, where `s` satisfies some restrictions (see below), then in the `then` branch we can assume that s is non-nullable.

```
val s: String|Null = ???
if (s != null) {
  val l = s.length // ok: s has type String in the 'then' branch
}
val l = s.length    // error: s has type String|Null
```

  We can reason similarly about the `else` branch if the test is `p == null`.
- *Logical operators.* We also support the logical operators `&&`, ||, and ! in conditions: e.g. given a condition `if (s != null && s2 != null)`, we infer that *both* `s` and `s2` are non-null in the `then` branch.
- *Propagation within conditions.* We support type specialization within a condition, taking into account that `&&` and `||` are short-circuiting: e.g. in the condition `s != null && s.length > 0`, the test `s.length` is type correct because the right-hand side of the condition will only be evaluated if `s` is non-null.
- *Nested conditions.* Our inference works in the presence of arbitrarily-nested conditions: given the condition `!(a = null || b = null) && (c != null)`, we infer that all of `a`, `b`, and `c` are non-null in the `then` branch.
- *Early exit from blocks.* If a statement conditionally performs an early exit from a block based on whether a value is `null`, we can soundly assume that the value is non-null from that point on. This is the case for both `return` statements and exceptions:

```
if (s == null) return 0
return s.length // ok: s inferred to have type String from this point on
```

  In general, if we have a block $s_1, \ldots, s_i, s_{i+1}, \ldots, s_n$, where the s_i are statements, and s_i is of the form `if (cond) exp`, where `exp` has type `Nothing`, then depending on `cond`, we might be able to infer additional nullability facts for statements $s_{i+1}, \ldots, s_n$. Here,

[4] `???` is actually valid Scala code, and is simply a method with return type `Nothing`.

the condition `cond` can contain nested conditions such as those discussed in the previous point. The reason is that type `Nothing` has no values, so an expression of type `Nothing` cannot terminate normally (it either throws or loops). It is then safe to assume that statement s_{i+1} executes only if `cond` is `false`.
There is one extra complication here, which is that Scala allows forward references to method definitions, which combined with nested methods can lead to non-intuitive control flow. In our implementation, we have logic for detecting forward references and disabling flow typing in such cases to preserve soundness. In the presence of forward references, we discard the more precise type inferred for a specific program point by flow typing and fall back to the flow-insensitive declared type that is conservatively sound at all program points.

4.1.1 Stable Paths

Scala has four kinds of definitions: `vals`, `lazy vals`, `vars`, and `defs`. `vals` are eagerly evaluated and immutable. `lazy vals` are like `vals`, but lazily evaluated and then memoized. `vars` are eagerly evaluated and mutable. Finally, `defs` are lazily evaluated, but not memoized, so they are used to define methods.

We use flow typing on `vals` and `lazy vals`, but not on `vars` or `defs`. Using naive flow typing on a `var` would be unsound, because the underlying value might change between the moment it is tested (where it might be non-null) and the later use of the `var` (where it might be again `null`). Similarly, flow typing on `defs` would be problematic, because a `def` is not guaranteed to return the same value after every invocation.

In general, given a path $p = v.s1.s2.\dots.s_n$, where v is a local or global symbol, and the s_i are selectors, it is safe to do flow inference on p *only if* p is *stable*. That is, all of $v, s1, \dots, s_n$ need to be `vals` or `lazy vals`. If p is stable, then we know that p is immutable and so the results of a check against `null` are persistent and can be trusted.

4.2 Inferring Flow Facts

The goal of flow typing is to discover *nullability facts* about stable paths that are in scope. A *fact* is an assertion that a specific path is non-null at a given program point.

At the core of flow typing, we have a function $\mathcal{N} : \texttt{Exp} \times \texttt{Bool} \to \mathbb{P}(\texttt{Path})$. $\mathcal{N}$ takes a Scala expression e (where e evaluates to a boolean) and a boolean b, and returns a set of paths known to be non-nullable if e evaluates to b. That is, $\mathcal{N}(\texttt{e}, \texttt{true})$ returns the set of paths that are non-null if e evaluates to `true`, and $\mathcal{N}(\texttt{e}, \texttt{false})$ returns the set of paths known to be non-null if e evaluates to `false`. $\mathcal{N}$ is defined in Figure 3.

We can use $\mathcal{N}$ to support the flow typing scenarios we previously outlined:

- Given an if expression `if (cond) e1 else e2`, we compute $F_{\text{then}} = \mathcal{N}(\texttt{cond}, \texttt{true})$ and $F_{\text{else}} = \mathcal{N}(\texttt{cond}, \texttt{false})$. The former gives us a set of paths that are known to be non-null if `cond` is true. This means that we can use F_{then} when typing e_1. Similarly, we can use F_{else} when typing e_2.
- To reason about nullability *within* a condition `e1 && e2`, notice that e_2 is evaluated *only* if e_1 is `true`. This means that we can use the facts in $\mathcal{N}(\texttt{e1}, \texttt{true})$ when typing e_2. Similarly, in a condition `e1 || e2`, we only evaluate e_2 if e_1 is *false*. Therefore, we can use $\mathcal{N}(\texttt{e1}, \texttt{false})$ when typing e_2.
- Given a block with statements `if (cond) e; s`, where e has type `Nothing`, or a block of the form `if (cond) return; s`, we know that s will only execute if `cond` is false. Therefore, we can use $\mathcal{N}(\texttt{cond}, \texttt{false})$ when typing s.

$$
\begin{aligned}
\mathcal{N}(\texttt{p == null}, \texttt{true}) &= \{\} \\
\mathcal{N}(\texttt{p == null}, \texttt{false}) &= \{p\} \text{ if } \texttt{p} \text{ is stable} \\
\mathcal{N}(\texttt{p != null}, \texttt{true}) &= \{p\} \text{ if } \texttt{p} \text{ is stable} \\
\mathcal{N}(\texttt{p != null}, \texttt{false}) &= \{\} \\
\mathcal{N}(\texttt{A \&\& B}, \texttt{true}) &= \mathcal{N}(\texttt{A}, \texttt{true}) \cup \mathcal{N}(\texttt{B}, \texttt{true}) \\
\mathcal{N}(\texttt{A \&\& B}, \texttt{false}) &= \mathcal{N}(\texttt{A}, \texttt{false}) \cap \mathcal{N}(\texttt{B}, \texttt{false}) \\
\mathcal{N}(\texttt{A || B}, \texttt{true}) &= \mathcal{N}(\texttt{A}, \texttt{true}) \cap \mathcal{N}(\texttt{B}, \texttt{true}) \\
\mathcal{N}(\texttt{A || B}, \texttt{false}) &= \mathcal{N}(\texttt{A}, \texttt{false}) \cup \mathcal{N}(\texttt{B}, \texttt{false}) \\
\mathcal{N}(\texttt{!A}, \texttt{true}) &= \mathcal{N}(\texttt{A}, \texttt{false}) \\
\mathcal{N}(\texttt{!A}, \texttt{false}) &= \mathcal{N}(\texttt{A}, \texttt{true}) \\
\mathcal{N}(\texttt{\{s1; ...; sn; cond\}}, \texttt{b}) &= \mathcal{N}(\texttt{cond}, \texttt{b}) \\
\mathcal{N}(\texttt{e}, \texttt{b}) &= \{\} \text{ otherwise}
\end{aligned}
$$

Figure 3 Flow facts inference. Correctness follows from De Morgan's laws.

4.3 Asserting Non-Nullability

For cases where flow typing is not powerful enough to infer non-nullability, we added a `.nn` ("assert non-nullable") method to cast away nullability from any term.

```
var s: String|Null = ???
val l = s.nn.length // ok: .nn method casts away nullability
```

In general, if `e` is an expression with type `T|Null`, then `e.nn` has type `T`. The `nn` method is defined as an *extension method*. This is a kind of implicit definition that makes `nn` available for any receiver of type `T|Null`. `nn` does a *checked cast*, so `e.nn` fails with an exception if the receiver `e` evaluates to `null`.

5 Evaluation

In this section, we empirically evaluate the expressiveness of the explicit nulls system and the effort required to migrate existing Scala programs to it. We test the popular belief that Scala programs tend not to use `null` references much themselves except for interaction with Java code. The explicit nulls system requires a program to explicitly specify what is to be done if a `null` reference arises at each program location where it is not ruled out statically; we quantify how many such locations there are in typical Scala programs.

We perform our evaluation on the programs in the Dotty community build,[5] a suite of Scala programs that have been ported from Scala 2 to compile with the Dotty compiler (without explicit nulls), and are regularly tested as part of the Dotty regression tests. The community build programs are summarized in Table 1.

We divide our evaluation into three parts. First, in Section 5.1, we evaluate `null` references possibly coming from interaction with Java code. Second, in Section 5.2, we evaluate the effectiveness of flow-sensitive typing in ruling out the possibility of `null` references. Third, in Section 5.3, we examine other causes of null-related compilation errors that are not related to interaction with Java and are not ruled out by flow typing.

[5] `https://github.com/lampepfl/dotty/tree/master/community-build/test/scala/dotty/communitybuild`

Table 1 Community build libraries.

Name	Description	Size (LOC)	Files
`scala-pb`	Scala protocols buffer compiler	37,029	275
`squants`	DSL for quantities	14,367	222
`fastparse`	Parser combinators	13,701	80
`effpi`	Verified message passing	5,760	60
`betterfiles`	IO library	3,321	29
`algebra`	Algebraic type classes	3,032	75
`scopt`	Command-line options parsing	3,445	28
`shapeless`	Type-level generic programming	2,328	18
`scalap`	Class file decoder	2,210	22
`semanticdb`	Data model for semantic information	2,154	49
`intent`	Test framework	1,866	48
`minitest`	Test framework	1,171	32
`xml-interpolator`	XML string interpolator	993	20
`stdLib213`	Scala standard library	31,723	588
`scala-xml`	XML support	6,989	115
`scalactic`	Utility library	3,952	53
Total		134,041	1,714

5.1 Evaluation of Java interaction

We evaluate the interaction with Java code by counting the number of compilation errors in several variants of the explicit nulls system. The error counts per thousand lines of code for each program and each variant are shown in Table 2.

The **Baseline** column shows the error counts for the explicit nulls system as described in this paper so far. There is significant variance between the different programs, from two or fewer errors per thousand lines of code in more abstract, Scala-like programs, to tens of errors per thousand lines of code in more low-level programs, particularly those that interact significantly with Java. We conjecture that interaction with Java is the main cause of the errors, and evaluate several variations of the system to test this conjecture.

Our first attempt to reduce the number of errors is with nullness annotations in Java code. The **Annotations** column shows the error counts when the Scala programs are compiled with a variant of the Java standard library with annotations specifying that the return values of certain methods cannot be `null`. The annotations are taken from the Checker Framework Project [23], which publishes an annotated version of the Java standard library, with nullness annotations on 4414 methods and 1712 fields in 847 classes. There are many different standards for annotating Java code with nullability; our implementation supports reading 12 such annotation formats and additional formats can be added easily. On some of the programs with high error counts, the annotations reduce the error count significantly, by up to half on `scalap`, but on others, they make little difference, such as on `ScalaPB`. One reason for this is that some programs interact with Java code other than the standard library, and the other Java libraries are not annotated. Another reason is that although the Checker Framework provides thousands of annotations, it still leaves a large part of the standard library unannotated, and the Scala programs interact with these unannotated methods. Annotating the entire standard library would be a huge effort, and even then, more annotations would be needed for any other Java libraries that a Scala program interacts with.

Table 2 Error frequency by configuration in errors per thousand LOC. The mean is weighted by the number of LOC in each program. The **Baseline** column reflects the configuration described in this paper so far. The **Annotations** column adds annotations to the Java standard library to specify methods that do not return `null`. The **JavaNull** column reflects a configuration in which method selections are (unsoundly) allowed on possibly `null` references returned by Java methods. The **Non-null Ret.** column reflects a configuration in which all calls of Java methods are (unsoundly) assumed to never return a `null` reference. The **Ann. No Flow** column reflects a configuration like the **Annotations** column, except with the flow typing discussed in Section 4 disabled.

	Baseline	**Annotations**	**JavaNull**	**Non-null Ret.**	**Ann. No Flow**
scalactic	72.37	57.19	57.19	3.04	57.19
betterfiles	43.36	38.54	37.04	7.23	38.54
stdLib213	37.26	34.01	33.54	17.24	34.36
ScalaPB	24.98	24.76	24.76	1.38	24.76
minitest	18.79	13.66	12.81	6.83	13.66
scalap	15.84	7.69	7.24	1.81	7.69
scala-xml	13.59	11.45	11.30	9.30	11.88
semanticdb	12.07	7.43	6.04	1.39	7.43
intent	8.57	6.97	6.97	0.54	6.97
scopt	5.52	4.93	4.64	2.32	4.93
xml-interpolator	2.01	2.01	2.01	2.01	2.01
shapeless	1.72	0.00	0.00	0.00	0.00
fastparse	1.61	1.53	1.53	1.46	1.53
effpi	1.39	1.39	1.04	0.00	1.56
algebra	0.33	0.33	0.33	0.00	0.33
squants	0.00	0.00	0.00	0.00	0.00
Mean	20.79	18.96	18.74	5.56	19.07

Another conjecture is that it is common to chain calls to Java methods. For example, if `s` is of type `String`, we may call `s.trim.toUppercase`, where `trim` is a Java method on strings that returns another string, on which we wish to call the Java method `toUppercase`. Such a pattern is rejected by the explicit nulls system if trim can return a null reference, since a null reference does not have a `toUppercase` method, but if this pattern is common, it may be pragmatic to allow it, even if it is potentially unsound. We evaluate a variant of the explicit nulls system that adds a special `JavaNull` annotation to mark `Null` types returned from Java methods. The Dotty type system treats these annotated `Null` types the same as any other `Null` types, with the exception that a method in a class `C` can be called on a receiver of type `C | Null` if the `Null` has the special annotation. This variant permits the sequence of calls `s.trim.toUppercase`, since the nullable return type of `trim` has the special `JavaNull` annotation. Note that this pragmatic design decision sacrifices soundness. The error counts for this variant of the explicit nulls system, together with the standard library annotations from the Checker Framework, are shown in the **JavaNull** column. Although the `JavaNull` annotation does reduce error counts for some programs, the reduction is small. This suggests that there are important things other than method selections that Scala programs do with the values returned from Java methods, and thus the `JavaNull` annotation to enable method selections is not sufficient to significantly reduce error counts.

Finally, we measure an upper bound on the reduction in error count that can be achieved by annotating Java methods that return non-null values. We evaluate a configuration of the explicit nulls system that assumes that *every* call to a Java method returns a non-null

value. This is equivalent to annotating every possible Java method with a non-null return type annotation. It is also equivalent to an extreme case of the special `JavaNull` annotation, which exceptionally allows method selection on nullable values returned from Java methods: if we were to allow `JavaNull` types in all places that currently require non-null types, rather than only in method selections, this would be equivalent to assuming that return values of Java methods cannot be `null`.

The resulting error counts are shown in the **Non-null Ret.** column. The impact of this configuration is very large: it causes a major reduction in error counts in all of the programs that still have large numbers of compilation errors. The `scalactic` library, which had over 72 errors per thousand lines of code in the baseline configuration, has only just over 3 errors per thousand lines of code. The mean error count goes from about 21 in the Baseline configuration and about 19 in the Annotations and JavaNull configurations down to 6 in the Non-null Ret. configuration. These results show that the conservative assumption that Java methods might return null is by far the most frequent cause of compilation errors in the explicit nulls system. Furthermore, once these errors are removed, fewer than ten errors per thousand lines of code remain in all programs except the Scala standard library. This is quite a small number, and we consider it reasonable to expect that Scala programmers can fix the remaining errors by hand.

5.2 Evaluation of Flow-sensitive typing

In this section, we evaluate the usefulness of the flow-sensitive typing design that was described in Section 4. We have turned off flow-sensitive typing, so that each variable has a single type everywhere it is in scope, independent of any nullness tests, and again count the number of compilation errors. The error frequency with flow-sensitive typing turned off is shown in the last column of Table 2, **Ann. No Flow**. The configuration uses the annotations from the Checker Framework to specify which methods in the Java standard library return non-null values; therefore, this column is directly comparable to the **Annotations** column. The **Annotations** configuration was selected as the most precise variant of typing calls to Java methods that is still sound (assuming the Checker Framework annotations are correct).

Flow-sensitive typing makes a difference in three of the benchmarks, `stdLib213`, `scala-xml`, and `effpi`, and even there, the difference is small relative to the total number of errors. One possible reason that flow-sensitive typing has such a small impact could be that our specific flow-sensitive analysis is not sufficiently precise, so compilation errors are reported even in code that tests that references are not null; however, we will see in the next section that this is not the case. Examining the code of the community build programs, we observe that Scala programs rarely expect to encounter `null` references and thus rarely test for them, and when they do, they often use a different idiom than a test of the form `if(x != null) ...`. Specifically, many of the programs pass possibly `null` values to the constructor of the `Option` class, which turns a `null` value into the `None` object and a non-null value into an instance of `Some`. This common idiom does not require flow-sensitive typing to ensure safety.

5.3 Evaluation of other causes of nullness errors

The error counts in the **Non-null Ret.** column, where we assume that Java methods never return `null`, are low enough that it is quite feasible to manually fix the programs to remove the compilation errors. We have done this for all the programs except `stdLib213` and classified the individual causes of each compilation error. We exclude `stdLib213` not

Table 3 Error classification. Libraries were migrated under **Non-null Ret.** configuration. Normalized count is in errors per thousand LOC.

Error Category	Total Count	Count per 1000 LOC
Declaration of nullable field or local symbol	74	0.81
Use of nullable field or local symbol (`.nn`)	52	0.57
Overriding error due to nullability	46	0.5
Generic *received* from Java with nullable inner type	19	0.6
Generic *passed* to Java requires nullable inner type	6	0.07
Incorrect Scala standard library definition	4	0.04
Limitation of flow typing	1	0.01
Total	202	2.21

	Modified	Total	%
LOC	484	91,337	0.53
Files	88	958	9.19

only because it has the highest error rate per thousand lines of code, but also because with 31,723 lines of code, it also has a high absolute number of errors. This analysis enables us to determine the common causes of the remaining compilation errors.

The number of errors in each category is shown in Table 3. We now explain the categories.

- *Declaration of nullable field or local symbol.* These are cases where the Scala code declares a `var` or `val` (as a field, or locally within a method) that is provably nullable because the code explicitly assigns `null` to it. For example, we might have a class field that is immediately initialized to `null`. The fix for this error is to change the type to a nullable type to reflect that the variable does (sometimes) contain a `null` reference.
- *Use of nullable field or local symbol (`.nn`).* This is the dual of the previous category. After we change the type of a variable that is sometimes null to a nullable type, all uses of that variable become nullable. Each existing use of that variable in a context that requires a non-null value then results in a compilation error, since the variable could be null. The fix for this error is to dynamically check and cast away the nullability using `.nn`.
- *Overriding error due to nullability.* This error happens when a Scala class overrides a Java-defined method that takes a reference type as an argument. Because nullification makes the argument types of the overridden method nullable, the argument types in the overriding method must also be made nullable to match the signature of the overridden method.
- *Generic* received *from Java with nullable inner type.* Sometimes we encounter a Java method that returns a generic with a nullified inner type. The common example are Java methods returning arrays of reference types. For example, the `split` method of the `String` class returns an `Array[String]`, which is nullified to `Array[String|Null]|Null`. This, in turn, leads to errors in Scala code that reads elements of this array and expects them to be non-null.
- *Generic* passed *to Java requires nullable inner type.* This happens when a Java method expects as argument a generic of some reference type (usually an `Array`). We fix these errors using `asInstanceOf` casts. An improvement to the type inference algorithm that was added to the Dotty compiler after our evaluation fixes most of these errors.[6]

[6] https://github.com/lampepfl/dotty/pull/8635

- *Incorrect Scala standard library definition.* This class contains type errors that could be prevented by modifying some definition in the Scala standard library to use a more precise type. For example, the `Option.apply` method is parameterized by a type `T`, takes an argument of type `T`, and returns a value of type `Option[T]`. If the argument is null, it returns `None`; otherwise, it returns the argument wrapped in `Some`, but it never returns `Some(null)`. When this method is called on a nullable argument, for example of type `String|Null`, its return type is `Option[String|Null]`, but a more precise return type would be `Option[String]`. These errors could be fixed by future versions of the Scala standard library.
- *Limitation of flow typing.* These are cases where our implementation of flow-sensitive typing is not precise enough to model the null checks that occur in the program and prove that a value cannot be `null`. We only found one error in this class, which is due to an undiagnosed bug in our implementation that is not yet fixed.

5.4 Summary

Our results confirm the common belief that `null` references are used rarely in Scala code except for interaction with Java. For the uses of `null` that are unrelated to Java, our system reports very few compilation errors, and few changes were needed to make the community build programs compile with the explicit nulls system.

However, a large number of nullness errors are caused by values returned from calls to Java methods. Scala programmers have several options for handling these return values. The first option is to harden Scala programs to always expect and handle possible `null` references returned from Java methods. The second option is to annotate Java methods known to never return `null` references. Both these options require a significant effort. On the other hand, a third option, which requires minimal effort, is the optimistic assumption that most Java method calls will not return `null`. This option is the status quo, the state of the existing Scala code, which is not required by the compiler to explicitly consider the possibility of `null` values. There is no free lunch: there are many places in Scala code where a Java method could return `null`; one either makes the considerable effort to check and annotate or harden each such place, or one accepts the risk of a `null` reference occurring at one of those places at run time.

6 Denotational Semantics of Nullification

Type nullification is the key component that interfaces Java's type system, where `null` is implicit, and Scala's type system, where `null` is explicit. In this section, we give a theoretical foundation for nullification using denotational semantics. Specifically, we present λ_j and λ_s, two type systems based on a variant of System F_ω restricted to second-order type operators. In λ_j, nullability is implicit, as in Java. By contrast, in λ_s nullability is explicit, like in Scala. Nullification can then be formalized as a function that maps λ_j types to λ_s types. Following a denotational approach, we give a set-theoretic model of λ_j and λ_s. We then prove a soundness theorem stating that the meaning of types is largely unchanged by nullification.

We choose System F_ω as the basis for our formalization, rather than object-oriented calculi such as DOT [2, 27, 25] or Featherweight Generic Java [16], because type application is the challenging case for nullification. Since nullification turns Java types into Scala types, it does not need to handle many Scala-specific types (e.g. path-dependent types), so DOT is not needed for the formalization. Similarly, Featherweight Generic Java has features like inheritance that do not interact with nullification.

$S, T ::=$	**λ_j Types**
$\texttt{int}_j$	int
$\texttt{String}_j$	string
$S \times_j T$	product
$S \to_j T$	function
$\Pi_j(X :: *_n).S$	generic
$\mathrm{App}_j(S, T)$	type application
X	type variable

$\sigma, \tau ::=$	**λ_s Types**
$\texttt{Null}$	null
$\texttt{int}_s$	int
$\texttt{String}_s$	string
$\sigma + \tau$	union
$\sigma \times_s \tau$	product
$\sigma \to_s \tau$	function
$\Pi_s(X :: *).\sigma$	generic
$\mathrm{App}_s(\sigma, \tau)$	type application
X	type variable

Figure 4 Types of λ_j and λ_s. Differences are highlighted.

6.1 System F_ω, λ_j, and λ_s

We will model the Java and Scala type systems as variants of System F_ω [14, 26], the higher-order polymorphic lambda calculus. System F_ω supports universal quantification on types: e.g. we can type the (polymorphic) identity function as $\Pi X.X \to X$. The variant that we use has *second-order type operators*, which means that in the type operator $\Pi X.S$, X ranges over all types that are *not themselves type operators*. By contrast, in the unrestricted version of the calculus, X can range over other type operators. By restricting type operators, we incur a loss of expressivity: notably, we can no longer typecheck recursive data structures (which are ubiquitous in both Java and Scala). On the other hand, giving a denotational semantics for the restricted variant is much easier, because one can use a naive set-based model. More importantly, the main difficulty in designing nullification was handling Java generics. Given a generic such as `List<T>`, Java only allows instantiations of `List` with a *reference* type *that is not itself generic*. For example, `List<String>` is a valid type application, but `List<List>` is not. This is precisely the kind of restriction imposed by our version of System F_ω.

That said, System F_ω is too spartan: it does not distinguish between value and reference types, does not have records (present in both Java and Scala), and does not have union types (needed for explicit nulls). To remedy this we can come up with slight variations of System F_ω that have the above-mentioned features. We call these λ_j ("lambda j") and λ_s ("lambda s"), and they are intended to stand for the Java and Scala type systems, respectively. Figure 4 shows the types of these two calculi. From now on we will focus solely on the types and will forget about terms, because nullification is a function from types to types.

λ_j extends System F_ω with integers, strings, and products (which stand in for objects). Type applications are written $\mathrm{App}_j(S, T)$.

λ_s differs from λ_j by adding a `Null` type, type unions (written $\sigma + \tau$), and by making types be explicitly nullable, just like our version of Scala. Explicit nullability is indicated via *kinds*, as explained below.

6.1.1 Kinding Rules

In subsequent sections, we will assign meaning to types. However, we can only interpret types that are *well-kinded*. Intuitively, we need a way to differentiate between a type like $\Pi_j(X :: *_n).X$, where all variables are bound, from $\Pi_j(X :: *_n).Y$, where Y is free and so cannot be assigned a meaning.

The *kinding* rules in Figure 5 fulfill precisely this purpose. The judgment $\Gamma \vdash_j T :: K$ (resp. $\Gamma \vdash_s \sigma :: K$) establishes that type T has kind K under context Γ, and is thus well-kinded in λ_j (resp. λ_s). The different kinds K describe: nullable types ($*_n$), non-nullable types ($*_v$),

$\boxed{\Gamma \vdash_j S :: K}$

$\Gamma \vdash_j \texttt{int}_j :: *_v$ (KJ-Int)

$\Gamma \vdash_j \texttt{String}_j :: *_n$ (KJ-String)

$$\frac{\Gamma \vdash_j S :: * \quad \Gamma \vdash_j T :: *}{\Gamma \vdash_j S \times_j T :: *_n} \text{(KJ-Prod)}$$

$$\frac{\Gamma \vdash_j S :: * \quad \Gamma \vdash_j T :: *}{\Gamma \vdash_j S \to_j T :: *_v} \text{(KJ-Fun)}$$

$$\frac{\Gamma, X :: *_n \vdash_j S :: K}{\Gamma \vdash_j \Pi_j(X :: *_n).S :: *_n \Rightarrow K} \text{(KJ-Pi)}$$

$$\frac{\Gamma \vdash_j S :: *_n \Rightarrow K \quad \Gamma \vdash_j T :: *_n}{\Gamma \vdash_j \mathrm{App}_j(S, T) :: K} \text{(KJ-App)}$$

$$\frac{\Gamma(X) = *_n}{\Gamma \vdash_j X :: *_n} \text{(KJ-Var)} \qquad \frac{\Gamma \vdash_j S :: *_n}{\Gamma \vdash_j S :: *} \text{(KJ-Null)} \qquad \frac{\Gamma \vdash_j S :: *_v}{\Gamma \vdash_j S :: *} \text{(KJ-NonNull)}$$

$K ::=$	**Kinds**
$*_n$	kind of nullable types
$*_v$	kind of non-nullable types
$*$	kind of proper types
$*_n \Rightarrow K$	kind of type operators (λ_j)
$* \Rightarrow K$	kind of type operators (λ_s)

$\Gamma ::=$	**Contexts**
$\varnothing$	empty context
$\Gamma, X :: *_n$	nullable type binding

$\boxed{\Gamma \vdash_s \sigma :: K}$

$\Gamma \vdash_s \texttt{int}_s :: *_v$ (KS-Int)

$\Gamma \vdash_s \texttt{String}_s :: *_v$ (KS-String)

$\Gamma \vdash_s \texttt{Null} :: *_n$ (KS-NullType)

$$\frac{\Gamma \vdash_s \sigma :: K_1 \quad \Gamma \vdash_s \tau :: K_2 \quad K_1, K_2 \in \{*_n, *_v, *\}}{\Gamma \vdash_s \sigma + \tau :: K_1 \oplus K_2} \text{(KS-Union)}$$

where $K \oplus K = K$, $K_1 \oplus K_2 = K_2 \oplus K_1$, $K \oplus *_n = *_n$, and $*_v \oplus * = *$

$$\frac{\Gamma \vdash_s \sigma :: * \quad \Gamma \vdash_s \tau :: *}{\Gamma \vdash_s \sigma \times_s \tau :: *_v} \text{(KS-Prod)}$$

$$\frac{\Gamma \vdash_s \sigma :: * \quad \Gamma \vdash_s \tau :: *}{\Gamma \vdash_s \sigma \to_s \tau :: *_v} \text{(KS-Fun)}$$

$$\frac{\Gamma, X :: * \vdash_s \sigma :: K}{\Gamma \vdash_s \Pi_s(X :: *).\sigma :: * \Rightarrow K} \text{(KS-Pi)}$$

$$\frac{\Gamma \vdash_s \sigma :: * \Rightarrow K \quad \Gamma \vdash_s \tau :: *}{\Gamma \vdash_s \mathrm{App}_s(\sigma, \tau) :: K} \text{(KS-App)}$$

$$\frac{\Gamma(X) = *}{\Gamma \vdash_s X :: *} \text{(KS-Var)} \qquad \frac{\Gamma \vdash_s \sigma :: *_n}{\Gamma \vdash_s \sigma :: *} \text{(KS-Null)} \qquad \frac{\Gamma \vdash_s \sigma :: *_v}{\Gamma \vdash_s \sigma :: *} \text{(KS-NonNull)}$$

.

Figure 5 Kinding rules of λ_j and λ_s. Differences are highlighted.

proper (non-generic) types ($*$), and *type operators* (generics). In λ_j, type operators have kinds of the form $*_n \Rightarrow K$, modelling the fact that type arguments in Java *must* be reference types (e.g. `List<boolean>` is not well-kinded). By contrast, in λ_s (and in Scala), generics can also take value types as arguments (e.g. `List[Boolean]`), so type operators have kinds of the form $* \Rightarrow K$.

The second role of the kind system is to track the *nullability* of types. Here, the difference between λ_j and λ_s is witnessed, for instance, by the KS-String rule: while in λ_j, strings are nullable ($\vdash_j \texttt{String}_j :: *_n$), strings in λ_s are non-nullable ($\vdash_s \texttt{String}_s :: *_v$). In λ_s, like in Scala, nullability can be recovered via type unions: e.g. $\vdash_s \texttt{String}_s + \texttt{Null} :: *_n$.

The rule KS-Union computes the kind of type unions. If either σ or τ contains the null value, then their union $\sigma + \tau$ also contains the null value, so $K \oplus *_n = *_n$. If one of σ or τ definitely does not contain the null value (i.e., is of kind $*_v$) and the other may or may not contain the null value (i.e., is of kind $*$), then their union $\sigma + \tau$ also may or may not contain the null value, so $*_v \oplus * = *$.

Two other rules that we want to highlight are KJ-Null and KJ-NonNull (and their λ_s counterparts). These rules give us a limited form of "subkinding", so that $\vdash_j T :: *_v$ or $\vdash_j T :: *_n$ imply $\vdash_j T :: *$.

▶ **Definition 6.1** (Base kinds). *We say K is a* base *kind if $K \in \{*, *_n, *_v\}$.*

6.2 Denotational Semantics

Before we can prove properties of nullification, we need a *semantics* for our types and kinds. That is, so far, types and kinds are just syntactic objects, and kinding rules are syntactic rules devoid of meaning. For this task of assigning meaning we turn to the machinery of *denotational semantics*. The technical presentation is based on the treatment of predicative System F in Mitchell [20].

Here is a summary of the rest of this section. First, we construct set-theoretic models for both calculi. In this case, a model is just a family of sets that contains denotations of types and kinds. We then show how to map kinds and types to their denotations in the model. The mapping is roughly as follows: kinds $\longrightarrow$ families of sets, proper types $\longrightarrow$ sets, and generic types $\longrightarrow$ functions from sets to sets. Finally, we prove a *soundness* lemma for kinding rules that says that if a type is well-kinded, then its denotation is defined and, further, it is contained in the denotation of the corresponding kind: i.e. $\Gamma \vdash_j T :: K \implies [\![T]\!]_j\eta \in [\![K]\!]_j$. The proofs of all results in this section can be found in the first author's thesis [22].

6.2.1 Semantic Model

▶ **Definition 6.2** (String literals). `strings` *denotes the set of finite-length strings.*

The model for λ_j is a pair $\mathcal{J} = (U_1, U_2)$ of *universes* (families of sets).

U_1 is the universe of proper types. It is the least set containing $\{\texttt{null}\}, \mathbb{Z}$, and `strings` that is closed under union, product, and functions (i.e. if u and v are in U_1, then the set of all functions between u and v, written u^v, is also in U_1). Additionally, we define two families of sets that contain nullable and non-nullable types, respectively: $U_1^{\text{null}} = \{u | u \in U_1, \texttt{null} \in u\}$, and $U_1^{\text{val}} = \{u | u \in U_1, \texttt{null} \notin u\}$. Notice that both U_1^{null} and U_1^{val} are subsets of U_1, and that $U_1 = U_1^{\text{null}} \cup U_1^{\text{val}}$.

The universe U_2 is a superset of U_1 that, additionally, contains all generic types. First, we define a family of sets $\{U_2^i\}$, for $i \geq 0$: $U_2^0 = U_1$, and $U_2^{i+1} = U_2^i \cup \{f : U_1^{\text{null}} \to U_2^i\}$. Then we set $U_2 = \bigcup_{i \geq 0} U_2^i$.

The model for λ_s is very similar to the previous one. It is a pair $\mathcal{S} = (U_1, U_2')$, where U_1 is as defined before. U_2' is almost the same as U_2, except that we set $U_2'^{i+1} = U_2'^i \cup \{f : U_1 \to U_2'^i\}$. Highlighted is the fact that generics in λ_s take arguments from U_1, as opposed to U_1^{null}.

6.2.2 Meaning of Kinds

▶ **Definition 6.3** (Number of arrows in a kind). *Let K be a kind. Then $arr(K)$ denotes the number of arrows ($\Rightarrow$) in K.*

▶ **Definition 6.4** (Meaning of kinds). *We give meaning to λ_j and λ_s kinds via functions $[\![]\!]_j$ and $[\![]\!]_s$, respectively. These functions are inductively defined on the structure of a kind K.*

λ_j

$$
\begin{aligned}
[\![\texttt{int}_j]\!]_j\eta &= \mathbb{Z}\\
[\![\texttt{String}_j]\!]_j\eta &= \{\texttt{null}\} \cup \texttt{strings}\\
[\![S \times_j T]\!]_j\eta &= \{\texttt{null}\} \cup ([\![S]\!]_j\eta \times [\![T]\!]_j\eta)\\
[\![S \to_j T]\!]_j\eta &= [\![S]\!]_j\eta^{[\![T]\!]_j\eta}\\
[\![\Pi_j(X :: *_n).S]\!]_j\eta &= \lambda(a \in U_1^{null}).[\![S]\!]_j(\eta[X \to a])\\
[\![\mathrm{App}_j(S,T)]\!]_j\eta &= [\![S]\!]_j\eta([\![T]\!]_j\eta)\\
[\![X]\!]_j\eta &= \eta(X)
\end{aligned}
$$

λ_s

$$
\begin{aligned}
[\![\texttt{Null}]\!]_s\eta &= \{\texttt{null}\}\\
[\![\texttt{int}_s]\!]_s\eta &= \mathbb{Z}\\
[\![\texttt{String}_s]\!]_s\eta &= \texttt{strings}\\
[\![\sigma \times_s \tau]\!]_s\eta &= [\![\sigma]\!]_s\eta \times [\![\tau]\!]_s\eta\\
[\![\sigma \to_s \tau]\!]_s\eta &= [\![\sigma]\!]_s\eta^{[\![\tau]\!]_s\eta}\\
[\![\Pi_s(X :: *).\sigma]\!]_s\eta &= \lambda(a \in U_1).[\![\sigma]\!]_s(\eta[X \to a])\\
[\![\mathrm{App}_s(\sigma,\tau)]\!]_s\eta &= [\![\sigma]\!]_s\eta([\![\tau]\!]_s\eta)\\
[\![\sigma + \tau]\!]_s\eta &= [\![\sigma]\!]_s\eta \cup [\![\tau]\!]_s\eta\\
[\![X]\!]_s\eta &= \eta(X)
\end{aligned}
$$

Figure 6 Type denotations for λ_j and λ_s. Differences are highlighted.

$$
\begin{aligned}
[\![*_n]\!]_j &= U_1^{null}\\
[\![*_v]\!]_j &= U_1^{val}\\
[\![*]\!]_j &= U_1\\
[\![*_n \Rightarrow K]\!]_j &= \{f : U_1^{null} \to [\![K]\!]_j\}
\end{aligned}
\qquad
\begin{aligned}
[\![*_n]\!]_s &= U_1^{null}\\
[\![*_v]\!]_s &= U_1^{val}\\
[\![*]\!]_s &= U_1\\
[\![* \Rightarrow K]\!]_s &= \{f : U_1 \to [\![K]\!]_s\}
\end{aligned}
$$

We can show that the meaning of kinds is contained within the corresponding model.

▶ **Lemma 6.5** (Kinds are well-defined). *If K is a λ_j kind, then $[\![K]\!]_j \subseteq U_2^{arr(K)}$. If K is a λ_s kind, then $[\![K]\!]_s \subseteq U_2'^{arr(K)}$.*

6.2.3 Meaning of types

We now give denotations for types. To handle types that are not closed, we make the denotation functions take *two* arguments: the type whose meaning is being computed and an *environment* that gives meaning to the free variables. Additionally, we make the simplifying assumption that types have been alpha-renamed so that there are no name collisions.

▶ **Definition 6.6** (λ_j Environments). *A λ_j environment $\eta : Var \to U_1^{null}$ is a map from variables to elements of U_1^{null}. The empty environment is denoted by $\emptyset$. An environment can be extended with the notation $\eta[X \to a]$, provided that X was not already in the domain.*

▶ **Definition 6.7** (λ_s Environment). *A λ_s environment $\eta : Var \to U_1$ is a map from variables to elements of U_1.*

▶ **Definition 6.8** (Environment Conformance). *An environment η (from λ_j or λ_s)* conforms *to a context Γ, written $\eta \vDash \Gamma$, if $dom(\eta) = dom(\Gamma)$.*

▶ **Definition 6.9** (Meaning of types). *We define the meaning of types via functions $[\![]\!]_j : Types_{\lambda_j} \to Env_{\lambda_j} \to U_2$ and $[\![]\!]_s : Types_{\lambda_s} \to Env_{\lambda_s} \to U_2'$. These are shown in Figure 6.*

▶ **Example 6.10.**

$$
\begin{aligned}
[\![\Pi_j(X :: *_n).X]\!]_j\emptyset &= \lambda(a \in U_1^{null}).[\![X]\!]_j\emptyset[X \to a]\\
&= \lambda(a \in U_1^{null}).\emptyset[X \to a](X)\\
&= \lambda(a \in U_1^{null}).a\\
&= id
\end{aligned}
$$

That is, the denotation of $\Pi_j(X :: *_n).X$ is the identity function that maps sets (types) in U_1^{null} to themselves.

The following lemma says that the kinding rules correctly assign kinds to our types.

▶ **Lemma 6.11** (Soundness of kinding rules). *The following hold:*
- $\Gamma \vdash_j T :: K$ *and* $\eta \models \Gamma$ *implies* $[\![T]\!]_j\eta \in [\![K]\!]_j$
- $\Gamma \vdash_s T :: K$ *and* $\eta \models \Gamma$ *implies* $[\![T]\!]_s\eta \in [\![K]\!]_s$

6.3 Type Nullification

Now that we have formal definitions for both λ_j and λ_s, we can also formally define type nullification. Recall that type nullification makes nullability *explicit* as we go from a type system where `null` is implicit (λ_j's) to one where `null` is explicit (λ_s's). For example, $(\texttt{int}_j) \times_j (\texttt{String}_j \to_j \texttt{String}_j)$ becomes $(\texttt{int}_s) \times_s (\texttt{String}_s + \texttt{Null} \to_s \texttt{String}_s + \texttt{Null})$.

That is, *type nullification is a function that turns* λ_j *types into* λ_s *types.* In the implementation (described in Section 3.1), we decided *not to nullify* arguments in type applications. That is, given a Java class `List<T>`, type applications such as `List<String>` are translated as `List<String>`, and not as `List<String|Null>`. The motivation for special casing type arguments is maximizing backwards-compatibility. Because of the different treatment for types based on whether they are in an argument position or not, we will model nullification as a *pair* of functions $(F_{\text{null}}, A_{\text{null}})$. These are defined below.

▶ **Definition 6.12** (Type nullification).

$$
\begin{aligned}
F_{null}(\texttt{int}_j) &= \texttt{int}_s \\
F_{null}(\texttt{String}_j) &= \texttt{String}_s + \texttt{Null} \\
F_{null}(X) &= X + \texttt{Null} \\
F_{null}(S \to_j T) &= F_{null}(S) \to_s F_{null}(T) \\
F_{null}(\Pi_j(X :: *_n).S) &= \Pi_s(X :: *).F_{null}(S) \\
F_{null}(App_j(S,T)) &= App_s(F_{null}(S), A_{null}(T)) \\
F_{null}(S \times_j T) &= (F_{null}(S) \times_s F_{null}(T)) + \texttt{Null}
\end{aligned}
\qquad
\begin{aligned}
A_{null}(\texttt{int}_j) &= \texttt{int}_s \\
A_{null}(\texttt{String}_j) &= \texttt{String}_s \\
A_{null}(S \times_j T) &= F_{null}(S) \times_s F_{null}(T) \\
A_{null}(S \to_j T) &= F_{null}(S) \to_s F_{null}(T) \\
A_{null}(App_j(S,T)) &= App_s(F_{null}(S), A_{null}(T)) \\
A_{null}(X) &= X
\end{aligned}
$$

As the name suggests, A_{null} handles types that are arguments to type application, and F_{null} handles the rest. A_{null} differs from F_{null} in that it does not nullify types at the outermost level (see e.g. the $\texttt{String}_j$ case).

▶ **Definition 6.13** (Context nullification). *We lift nullification to work on contexts, turning* λ_j *contexts into (syntactic)* λ_s *contexts.*

$$
\begin{aligned}
F_{null}(\varnothing) &= \varnothing \\
F_{null}(\Gamma, X :: *_n) &= F_{null}(\Gamma), X :: *
\end{aligned}
$$

▶ **Definition 6.14** (Kind nullification). *We also lift nullification to work on kinds, turning* λ_j *kinds into* λ_s *kinds.*

$$
\begin{aligned}
F_{null}(K) &= K && \text{if } K \text{ is a base kind} \\
F_{null}(*_n \Rightarrow K') &= * \Rightarrow F_{null}(K') && \text{otherwise}
\end{aligned}
$$

6.4 Soundness

We can finally prove a *soundness* result for type nullification. But what should soundness *mean* in this case? One plausible, but as it turns out, *incorrect*, definition is that nullification leaves the meaning of types *unchanged*.

▶ **Conjecture 6.15** (Soundness – Incorrect). *Let* $\Gamma \vdash_j T :: K$*, and let* η *be an environment such that* $\eta \models \Gamma$*. Then* $[\![T]\!]_j\eta = [\![F_{null}(T)]\!]_s\eta$*.*

This conjecture is false because the meaning of generics differs between λ_j and λ_s. In both cases, generics are denoted by functions on types, but the *domains of the functions are different*:

- $[\![\Pi_j(X :: *_n).S]\!]_j \eta = \lambda(a \in U_1^{null}).[\![S]\!]_j(\eta[X \to a])$
- $[\![\Pi_s(X :: *).S]\!]_s \eta = \lambda(a \in U_1).[\![S]\!]_s(\eta[X \to a])$

That is, λ_j generics take arguments that are in U_1^{null} (nullable arguments) and λ_s generics have wider domains and take arguments from *all* of U_1. This matches the behaviour in Java and Scala, where the generic class `List<A>` gets "mapped" by nullification to the Scala class `List[A]`. `List<int>` is then *not valid* in Java (because `int` is not a nullable type), but `List<int>` *is* valid in Scala.

We can recover soundness via the following observation. If `G` is a Java generic, even though $[\![G]\!]_j$ and $[\![G]\!]_s$ are not equal, for any *valid* type application `G<T>` in Java, $[\![G < T >]\!]_j = [\![F_{\text{null}}(G < T >)]\!]_s$. That is, our soundness theorem will say that nullification leaves *fully-applied* generic types unchanged. This is just as well because *users can only manipulate values of type `G<T>` and never values of type `G` directly.*

Before we state the soundness theorem we need a few ancillary definitions.

▶ **Definition 6.16** (Similar Types). *Let $S, T \in U_1$. Then we say S is similar to T, written $S \sim T$, if $S \cup \{\texttt{null}\} = T \cup \{\texttt{null}\}$.*

That is, two types denotations are *similar* if they contain the same elements, except for possibly `null`. Note that $\sim$ is symmetric.

▶ **Definition 6.17** (Similar Type Vectors). *Let $\vec{S} = (S_1, \ldots, S_n)$ and $\vec{T} = (T_1, \ldots, T_n)$ be vectors of types, where $\vec{S}$ and $\vec{T}$ have the same number of elements. Then $\vec{S} \sim \vec{T}$ if they are similar at every component.*

▶ **Definition 6.18** (Similar Environments). *Let η, η' be environments (either from λ_j or λ_s). Then η is similar to η', written $\eta \sim \eta'$, if $dom(\eta) = dom(\eta')$ and for all type variables X in the domain, we have $\eta(X) \sim \eta'(X)$.*

Note this relation is also symmetric.

▶ **Definition 6.19** (Similar Kinds). *Let K_1 and K_2 be two base kinds. Then $K_1 \rightsquigarrow K_2$ is defined by case analysis.*

$$* \rightsquigarrow * \quad (\text{SK-Prop})$$

$$*_v \rightsquigarrow *_v \quad (\text{SK-NonNull})$$

$$*_n \rightsquigarrow *_n \quad (\text{SK-Null1})$$

$$*_n \rightsquigarrow *_v \quad (\text{SK-Null2})$$

$$*_n \rightsquigarrow * \quad (\text{SK-Null3})$$

The rules in Definition 6.19 capture what happens to the kind of a type after being transformed by A_{null}. For example, $\texttt{String}_j$ has kind $*_n$ in λ_j, but $A_{\text{null}}(\texttt{String}_j) = \texttt{String}_s$ has kind $*_v$ in λ_s. This is described by rule (SK-Null2). The $\rightsquigarrow$ relation is not symmetric. This reflects the fact that A_{null} turns $*_v$ types into $*_v$ types, but can turn a $*_n$ type into a $*_v$ type.

▶ **Lemma 6.20.** *If K is a base kind, then $K \rightsquigarrow K$.*

Before proving soundness, we need to prove a weaker lemma that says that nullification preserves well-kindedness. This lemma is necessary because if T is well-kinded and nullification turns T into $F_{\text{null}}(T)$, the latter must be well-kinded as well.

▶ **Lemma 6.21** (Nullification preserves well-kindedness). *Let $\Gamma \vdash_j T :: K$ and $\Gamma' = F_{null}(\Gamma)$. Then*

1. $\Gamma' \vdash_s F_{null}(T) :: F_{null}(K)$.
2. *If K is a base kind, there exists a kind K' with $K \rightsquigarrow K'$ such that $\Gamma' \vdash_s A_{null}(T) :: K'$.*

▶ **Definition 6.22** (Curried type application). *If f is a function of m arguments and $\vec{x} = (x_1, \dots, x_m)$, we use the notation $f(\vec{x})$ to mean the curried function application $f(x_1)(x_2)\dots(x_m)$. In the degenerate case where f is not a function (i.e. $m = 0$), we set $f(\vec{x}) = f$.*

We can finally show soundness. We need to strengthen the induction hypothesis to talk about *both* F_{null} and A_{null}.

▶ **Theorem 6.23** (Soundness of type nullification). *Let $\Gamma \vdash_j T :: K$. Let η, η' be environments such that $\eta \vDash \Gamma$ and $\eta \sim \eta'$. Then the following two hold:*

1. *If K is a base kind, then*
   - **a.** $[\![T]\!]_j\eta = [\![F_{null}(T)]\!]_s\eta'$ *and*
   - **b.** $[\![T]\!]_j\eta \sim [\![A_{null}(T)]\!]_s\eta'$.
2. *If K is a type application with $arr(K) = m$, let $\vec{x}$ and $\vec{y}$ be two m-vectors of elements of U_1^{null} and U_1, respectively, with $\vec{x} \sim \vec{y}$. Then $[\![T]\!]_j\eta(\vec{x}) = [\![F_{null}(T)]\!]_s\eta'(\vec{y})$.*

The first assertion in the soundness theorem says that the meaning of base (non-generic) types is unchanged by nullification. For example, the denotations of $\texttt{String}_j$ and $F_{\text{null}}(\texttt{String}_j) = \texttt{String}_s + \texttt{Null}$ are equal. The second assertion says that if we start with a generic type that takes K arguments and apply it *fully* (i.e. apply it to K arguments), and *then* we apply nullification, the meaning of the type application is also unchanged, provided that the original type application is well-kinded in λ_j. For example, in λ_j we can represent generic pairs by $Pair = \Pi_j(X :: *_n).\Pi_j(Y :: *_n).X \times_j Y$. The theorem says that if $Pair_{(T_1,T_2)} = \text{App}_j(\text{App}_j(Pair, T_1), T_2)$ is well-kinded in λ_j (i.e. both T_1 and T_2 are in $*_n$), then the meaning of $Pair_{(T_1,T_2)}$ is also unchanged by nullification. That is, $[\![Pair_{(T_1,T2)}]\!]_j = [\![F_{\text{null}}(Pair_{(T_1,T_2)})]\!]_s$.

6.5 Discussion

As Section 3.1 points out, both underapproximations and overapproximations in nullification would lead to unsoundness, so "preserves elements of types" is a useful soundness criterion for type nullification.

That the meaning of types with base kinds remains unchanged is important, because program values always have base kinds. The meaning of generics *is* changed by nullification. This reflects the fact that, in λ_s and Scala, type arguments can be either value or reference types, while in λ_j and Java only reference types can be used. The soundness theorem (Theorem 6.23) in this section shows that *fully-applied* generics (which have base kinds) remain unchanged. Extrapolating, this means that Java types corresponding to fully-applied generics (e.g. `ArrayList<String>`), can be represented *exactly* in Scala. The other direction does not hold; e.g. the Scala type `List[Int]` cannot be represented directly in Java (because `Int` is a value type). Instead, `List[Int]` must be translated as `List<Integer>` or `List<Object>`, where `Integer` is the Java type for boxed integers. The type translation from Scala to Java (erasure) is not modelled in this section and remains as future work.

7 Related Work

The related work we have identified can be divided into four classes:

- Type systems for nullability in modern, widely used programming languages.
- Schemes to guarantee sound initialization. These have been mostly implemented as research prototypes, or as pluggable type systems.
- Pluggable type systems that are not part of the "core" of a programming language, but are used as checkers that provide additional guarantees (in our case, related to nullability).
- Denotations of types.

7.1 Nullability in the Mainstream

Kotlin is an object-oriented, statically-typed programming language for the JVM [17]. Kotlin's flow typing handles both `vars` and `vals`, while our system currently only supports `vals`. Additionally, Kotlin can recognize nullability annotations not just at the top-level, but also within type arguments to generics. Nullability in Kotlin is expressed with a type modifier: the reference type `T` is non-nullable, but `T?` is nullable. By contrast, in our design explicit nullability is achieved through a combination of union types and a new type hierarchy. The two approaches are comparable in their expressiveness, but in a language with support for union types (such as Scala), our approach expresses nullability as a derived concept and avoids introducing new kinds of types.

Kotlin handles Java interoperability via *platform types*. A platform type, written `T!`, is a type with *unknown* nullability. Kotlin turns all Java-originated types into platform types. Given a type `T!`, Kotlin allows casting it (automatically) into both a `T?` and a `T`. The cast from `T!` to `T?` always succeeds, but the cast from `T!` to `T` might fail at runtime, because the Kotlin compiler automatically inserts a runtime assertion that the value being cast is non-null. We chose to represent types flowing into Scala code from Java using union types and the `JavaNull` annotation. In this way we avoid introducing a new kind of type (platform types) into the already-crowded Scala type system. Another reason for diverging from the platform types approach is soundness. Kotlin allows (unsound) member selections on platform types, just like we do in Scala via `JavaNull`, but platform types are even more permissive. For example, Kotlin automatically casts a value of platform type `String!` to the non-nullable type `String`; by contrast, in our design the type `String | JavaNull` is *not* a subtype of `String`, so the cast needs to be applied manually. We can think of platform types as a generalization of `JavaNull` that allows not only member selections, but also subtyping with respect to non-nullable types. We wanted to strike a balance between soundness and usability in our design, so we opted for a more restrictive approach than Kotlin's in the handling of Java-originated types.

Ceylon is another object-oriented, statically-typed language that also targets the JVM [12]. Ceylon has union and intersection types, like Scala, and represents explicit nullability via union types [13]. The main difference between Ceylon's design and ours is the handling of interoperability with Java. Ceylon, like Kotlin, takes an "optimistic" approach where all Java-originated types used in Ceylon code are assumed to be non-nullable (and checked as such at runtime, with automatically-generated assertions). By contrast, we assume that all Java reference types are nullable, and so develop a type nullification function that turns Java types, including generics, into equivalent Scala types. To our knowledge, we are the first to formally describe type nullification, as well propose and prove a correctness criteria for it (nullification preserves values of types).

Swift is a statically-typed programming language, originally designed for the iOS and macOS ecosystems [4]. Swift has a `nil` reference, which is similar to `null` in Scala [5]. Types in Swift are non-nullable by default, but one can use *optionals* to get back nullability. For example, the type `Int?`, an optional, is either an integer or `nil`. Optionals can be *force unwrapped* using the `!` operator, which potentially raises a runtime error if the underlying optional is `nil`. Swift also has a notion of *optional binding*, which is similar to the monadic `bind` operator in Haskell [30], but specialized for optionals. Additionally, Swift has *implicitly unwrapped optionals*, which are similar to Kotlin's platform types. That is, the type `Int!`, an implicitly unwrapped optional, need not be forced unwrapped explicitly before a value can be retrieved, but if the underlying value is `nil`, it will produce a runtime error.

C# has reference types that are non-nullable by default. Compared to our design, C# offers more fine-grained control over where explicit nulls are enabled. In our system, explicit nulls can only be enabled or disabled at the project level. In C#, the user can additionally opt in to explicit nulls for *specific code regions* via "pragmas" (program metadata).

7.2 Sound Initialization

Even with a type system that has non-nullable types, there is a possibility of unsoundness because of *incomplete initialization*. This can happen, for example, due to dynamic dispatch, or *leaking* of the `this` reference from the constructor to helper methods. The problem is that in an unitialized (or *partially* uninitialized) object, the *invariants* enforced by the type system need not hold yet. Specifically, fields that are marked as non-null might *nevertheless* be `null` (or contain nonsensical data) because they have not yet been initialized.

Over the years, many solutions have been proposed for the sound initialization problem, usually involving a combination of type system features and static analyses. These prior designs include *raw types* [10], *masked types* [24], *delayed types* [11], the *Freedom Before Commitment* scheme [29], and X10's *hardhat* design [31]. These last two schemes have been identified as the bases for a sound initialization scheme for Scala [19].

7.3 Pluggable Type Checkers

Another line of work that is relevant to nullability is *pluggable type checkers*. A pluggable type checker is a custom-built typechecker that *refines* the typing rules of a host system [23].

The *Checker Framework* [23] is a framework for building pluggable type checkers for Java. Users have the option of writing their typecheckers in a *declarative* style, which requires less work (they do not need to write Java code) but is less expressive, or in a *procedural* style, where the checker can have arbitrarily complex logic, but is therefore harder to implement. One of the checkers that comes "pre-packaged" with the framework is the Nullness Checker. In fact, "the Nullness Checker is the largest checker by far that has been built with the Checker Framework" [9]. As of 2017, the Nullness Checker implemented a variant of the Freedom Before Commitment scheme, as well as support for flow typing and multiple heuristics to improve the accuracy of its static analysis [7, 9]. Dietl et al. [9] conducted an extensive evaluation of the Nullness Checker in production code, finding multiple errors in the Google Collections library for Java.

The *Granullar* project [7] combines the null checker from the Checker Framework with techniques from *gradual typing* [28]. Granullar allows the user to migrate only part of a project to use null checks. To that effect, the code under consideration is divided into *checked* and *unchecked* regions. Nullability checks are done statically within the checked region, using the Freedom Before Commitment scheme implemented by the Checker Framework.

No checks are done for the unchecked portion of the code. However, Granullar *insulates* the checked region from unsafe interactions with the unchecked region by inserting *runtime* non-null checks at the boundary.

NullAway [6] is a nullness checker for Android applications developed at Uber. NullAway is implemented as a pluggable type system on top of the Error Prone framework [1]. NullAway trades away soundness for efficiency. Specifically, the tool is *unsound* in multiple ways: its initialization checks ignore the problem of leaking the `this` reference, all unchecked methods are assumed to return *non-null* values, and flow typing assumes that all methods are *pure* and *deterministic*. In exchange for the unsoundness, NullAway has a lower build-time (2.5x) and annotation overheads than similar tools (2.8 - 5.1x) [6]. After extensive empirical evaluation [6], NullAway's authors note that the unsound assumptions do not lead to nullability errors in practice.

7.4 Semantics of Nullification

The model of System F_ω that we used is based on the one given by Mitchell [20] for System F (which, in turn, is based on Bruce et al. [8]). The denotations for sums and product types are standard in the literature.

There is one deviation from Mitchell [20] in how we construct denotations for generics. The standard way is to say that the denotation of a generic type is an (infinite) Cartesian product, whereas we use a simple function on types. That is, instead of saying $[\![\Pi_s(X :: *).S]\!]_s\eta = \prod_{a\in U_1}[\![S]\!]_j(\eta[X \to a])$, we define $[\![\Pi_s(X :: *).S]\!]_s\eta = \lambda(a \in U_1).[\![S]\!]_s(\eta[X \to a])$. The reason for the discrepancy is that λ_j and λ_s have type applications at the type level (e.g. $\mathrm{App}_j(S, T)$), whereas in System F, type applications are terms (e.g. $t\ [T]$). If we use the variant with the Cartesian product, then $[\![\mathrm{App}_s(\Pi_s(X :: *).X, \mathtt{int}_s)]\!]_s\emptyset$ would be an *element* of $[\![\mathtt{int}_s]\!]_s$ (an element of $\mathbb{Z}$). However, what we need for the soundness theorem is that $[\![\mathrm{App}_s(\Pi_s(X :: *).X, \mathtt{int}_s)]\!]_s\emptyset$ be *equal* to $[\![\mathtt{int}_s]\!]_s$, hence the second definition.

The novelty of our work is the use of denotational semantics for reasoning specifically about nullification. We are not aware of any related work that formalizes and proves soundness of nullification.

8 Conclusions

In this paper, we described a modification to the Scala type system that makes nullability explicit in the types. Reference types are no longer nullable, and nullability can be recovered using type unions. Because interoperability with Java is important, a type nullification phase translates Java types into Scala types. A simple form of flow typing allows for more idiomatic handling of nullable values. We implemented the design as a modification to the Dotty compiler.

To evaluate the implementation of explicit nulls, we migrated Scala programs from the Dotty community build to use the new type system. The results confirm that Scala code uses null references sparingly and that our system requires few modifications to the Scala internals of existing programs, with the significant exception of the places where Scala code interacts with Java code. The Java type system does not provide information about which references could be null. A Scala programmer faces an inevitable choice: One option is to annotate the Java code or to defensively check for the possibility of a null reference at every call to a Java method (and the type system can enforce such checks), but this requires considerable effort in programs that contain many such calls. Another option is to configure the type system to

optimistically but unsoundly assume that Java methods do not return null, which makes migration to the type system easy, but retains the possibility of null dereference exceptions if the Java methods violate the assumption.

We also showed how the intuitive reasoning about nullification based on sets can be given a solid formal footing, via denotational semantics. First, we presented λ_j and λ_s, two type systems based on System F_ω restricted to second-order type operators. These type systems formalize the implicit and explicit nature of `null` in Java and Scala, respectively. We then gave simple set-theoretic models for λ_j and λ_s, which in turn allow us to define denotations for types and kinds. We formalized nullification as a function from λ_j types to λ_s types. Finally, we proved a soundness theorem that says that nullification leaves the meaning of types largely unchanged.

References

1 Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. Building Useful Program Analysis Tools Using an Extensible Java Compiler. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 14–23. IEEE, 2012.

2 Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 249–272, 2016.

3 Nada Amin and Ross Tate. Java and Scala's Type Systems are Unsound: The Existential Crisis of Null Pointers. In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 838–848. ACM, 2016. doi:10.1145/2983990.2984004.

4 Apple Inc. About swift. [Online; accessed 5-November-2019]. URL: https://docs.swift.org/swift-book/.

5 Apple Inc. Swift language guide. [Online; accessed 5-November-2019]. URL: https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html.

6 Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. Nullaway: Practical Type-Based Null Safety for Java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 740–750. ACM, 2019.

7 Dan Brotherston, Werner Dietl, and Ondřej Lhoták. Granullar: Gradual Nullable Types for Java. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 87–97. ACM, 2017.

8 Kim B Bruce, Albert R Meyer, and John C Mitchell. The Semantics of Second-Order Lambda Calculus. *Information and Computation*, 85(1):76–134, 1990.

9 Werner Dietl, Stephanie Dietzel, Michael D Ernst, Kivanç Muşlu, and Todd W Schiller. Building and Using Pluggable Type-Checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690. ACM, 2011.

10 Manuel Fähndrich and K. Rustan M. Leino. Declaring and Checking Non-Null Types in an Object-Oriented Language. In Ron Crocker and Guy L. Steele Jr., editors, *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, pages 302–312. ACM, 2003. doi:10.1145/949305.949332.

11 Manuel Fähndrich and Songtao Xia. Establishing Object Invariants with Delayed Types. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 337–350. ACM, 2007. doi:10.1145/1297027.1297052.

12 Gavin King. The Ceylon Language. [Online; accessed 30-May-2020]. URL: `https://ceylon-lang.org/`.

13 Gavin King. Using Java From Ceylon. [Online; accessed 30-May-2020]. URL: `https://ceylon-lang.org/documentation/1.2/reference/interoperability/java-from-ceylon/`.

14 Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

15 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing Local Control and State Using Flow Analysis. In *European Symposium on Programming*, pages 256–275. Springer, 2011.

16 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

17 Kotlin Foundation. Kotlin programming language. [Online; accessed 5-November-2019]. URL: `https://kotlinlang.org/`.

18 Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

19 Fengyun Liu, Aggelos Biboudis, and Martin Odersky. Initialization Patterns in Dotty. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*, pages 51–55. ACM, 2018.

20 John C Mitchell. *Foundations for Programming Languages*, volume 1. MIT press Cambridge, 1996.

21 MITRE. 2019 CWE Top 25 Most Dangerous Software Errors. [Online; accessed 17-November-2019]. URL: `https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html`.

22 Abel Nieto Rodriguez. Scala with explicit nulls. Master's thesis, University of Waterloo, 2019.

23 Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical Pluggable Types for Java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212. ACM, 2008.

24 Xin Qi and Andrew C. Myers. Masked Types for Sound Object Initialization. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 53–65. ACM, 2009. `doi:10.1145/1480881.1480890`.

25 Marianna Rapoport, Ifaz Kabir, Paul He, and Ondřej Lhoták. A simple soundness proof for dependent object types. *PACMPL*, 1(OOPSLA):46:1–46:27, 2017.

26 John C Reynolds. Towards a Theory of Type Structure. In *Programming Symposium*, pages 408–425. Springer, 1974.

27 Tiark Rompf and Nada Amin. Type soundness for dependent object types (DOT). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 624–641, 2016.

28 Jeremy G Siek and Walid Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

29 Alexander J. Summers and Peter Müller. Freedom Before Commitment: a Lightweight Type System for Object Initialisation. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 1013–1032. ACM, 2011. `doi:10.1145/2048066.2048142`.

30 Philip Wadler. Monads for Functional Programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.

31 Yoav Zibin, David Cunningham, Igor Peshansky, and Vijay Saraswat. Object Initialization in X10. In *European Conference on Object-Oriented Programming*, pages 207–231. Springer, 2012.