Transitioning to Explicit Nulls in Scala

by

Harris Lau

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2026

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Scala, being a language that compiles to the JVM, suffers from the problem that JVM bytecode references are implicitly nullable. Every dereference is therefore a potential null pointer exception. Scala's explicit nulls system removes this hazard at the type level: the null type is no longer a subtype of the reference types, and a nullable value must be declared as a union of its base type with null. The change is sound, but it is also global, and it invalidates a large body of existing Scala code as well as every assumption that Scala programs make about the nullability of the Java libraries they call. A null-safe type system that nobody migrates to is of little use, so the practical question is not whether explicit nulls is correct, but whether the ecosystem can be moved onto it.

This thesis describes and evaluates the four compatibility features that the Scala compiler provides to answer that question. The non-null cast lets a programmer assert non-nullness at a use site, using an intersection with a singleton type to preserve path-dependent typing. Flow typing tracks, for each block of code, a pair of asserted and retracted variable sets composed by sequencing and alternation operators, with a distinguished value for blocks that always terminate abruptly; this admits the common idiom of checking a variable against null before dereferencing it. Flexible types give a value obtained from Java a deliberately unsound pair of bounds that allows it to be used as either nullable or non-nullable, resolving an interoperation problem that neither a fully sound nor a fully permissive translation of Java types can solve for invariant type constructors such as arrays. Finally, the unsafe nulls language import relaxes null-related type checking within a lexical scope, and its counterpart safe nulls restores it, allowing null safety to be adopted incrementally at the granularity of a project, a file, or a single block.

We evaluate these features on two sets of real Scala projects. On the Community Build of 40 widely used libraries, 15 compile under explicit nulls with no changes at all; among the rest, most of the porting effort consists of widening declared types to nullable unions (45% of changed lines) and inserting non-null assertions (23%). An ablation study shows that flow typing and flexible types each carry a substantial and non-overlapping share of the migration burden: disabling flow typing surfaces 424 additional compilation errors across 17 projects, while disabling flexible types surfaces 1,579 across 31 projects. The two features are orthogonal, flow typing addressing null checks within Scala code and flexible types addressing values obtained from Java, and neither substitutes for the other. On the Open Community Build of 1,928 projects, enabling explicit nulls alone causes 854 projects to fail to compile; enabling it together with unsafe nulls reduces this to 16, a compatibility rate of 99.17%, with the remaining failures confined to two narrow and well-understood causes. These results indicate that explicit nulls, taken together with its compatibility features, is mature enough to be enabled by default in a future major release of the Scala compiler.

Acknowledgements

I would like to thank my advisor, Ondřej Lhoták, for his guidance during my research. His patience in answering any questions I asked and openness to explore research directions I am interested in were pivotal to my learning progress. I am very grateful to be under his tutelage.

I would like to thank my fellow labmates at the Programming Languages Group for their companionship. Being new to Canada made finding my way through life difficult, but with them I could always feel like I belonged.

I would like to thank my parents for their unyielding love and support. Throughout my life they have nurtured my interest in Computer Science and never once hesitated to support me when I needed help. I could not be in this position without them.

I would like to thank my partner, Joyce, for her love and support as well. For every time I faced frustrations or difficulties, she would be there cheering me on every step of the way.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Scala is a multi-paradigm programming language that sees both industry use and academic research interest [23]. It can compile to many targets, one of which is the JVM. As such, it has inherited a design problem originally present within Java.

1.1 Nullability

Similar to many object-oriented languages, in Java, any reference may be null. Tony Hoare once described this as the 'billion dollar mistake' [19]. Every time a variable is dereferenced, there is a chance that it is actually null, which makes the runtime emit a `NullPointerException`.

```
def foo(s: String): Int =
    s.length        // s could be null!

foo(null)           // NullPointerException!
```

Scala aims to remedy this issue by modifying the type system such that null is no longer a subtype of reference types, but instead a separate type that only has the top type as its supertype.

```
// Under the explicit null system
def foo(s: String): Int =
    s.length        // Now guaranteed to be safe

foo(null)           // Error: foo takes String, not Null
```

After this change, reference types are no longer implicitly nullable. Instead, to define a nullable variable, the user must declare the type to be a union of the base type and the null type. Naturally, this causes a lot of existing code to no longer be compatible with the new type system. As such, the Scala compiler provides a wide set of features to increase the compatibility of old code with the new type system to encourage the transition to explicit nulls.

1.2 Interoperation

One of Scala's main design goals is to be fully compatible with Java code, i.e. Scala code can call any Java function, and vice versa. As such, extra care must be taken when handling types from Java programs, as all Java reference types are implicitly nullable. With the move to the explicit null system, there are two main challenges regarding compatibility. First, we want to ensure old, implicitly null Scala code can still work with the explicit system. Second, we want to ensure newly-written code can conveniently call existing Java and Scala libraries.

1.3 Compatibility

Even though the new type system provides improved null safety, it would be useless if users never switch to this new system. And in reality, switching to the explicit nulls system causes a significant portion of existing projects to fail to compile. This is a big hindrance to adoption. As such, the Scala compiler provides compatibility features to ease the transition to the explicit nulls system.

There are four major compatibility features provided by the compiler. First is the type casting utility `.nn`, which allows selective narrowing on nullable variables. Second is flow sensitive typing, which narrows the type of variables based on checks present within the program code. Third are flexible types, a unique kind of type with an inverted subtyping relationship that allows for streamlined interoperation with Java types. And last is the unsafe nulls language feature, which forgoes correctness in favor of special checks within the compiler that maximize compatibility, with the benefit of being able to localize type system effects, giving granularity over the explicit nulls migration.

1.4 Contributions

This thesis evaluates the effectiveness of the migration features implemented within the Scala compiler.

We benchmark the effectiveness of these features using the Scala Community Build, a set of projects that contain common code patterns and are widely used by other Scala projects. We attempt to compile the projects under the new type system and use the number of compilation errors as a measure of the disruptiveness of the explicit null changes. Compared to previous work on explicit nulls in Scala [22], our work shows a significant improvement in compatibility due to improved interoperation with Java code. We demonstrate that flow typing and flexible types can save the programmer 73% of the effort on fixing null-related errors on a set of 40 projects. We also show that with the unsafe nulls language feature, 99% of projects out of 1,928 can switch to the explicit null type system with no compilation errors, with only 16 projects needing manual changes.

1.5 Thesis Outline

This thesis provides a comprehensive overview of the transition of Scala from the implicitly nullable type system to the explicitly nullable type system. Chapter 2 provides background information on the Scala type system and the architecture of the compiler. Chapter 3 details the nullability related features provided by the compiler. Chapter 4 evaluates the features and their effectiveness in assisting the transition to explicit nulls. Chapter 5 describes prior work related to this thesis. Chapter 6 is the conclusion, noting possible future work and summarizing our contributions.

Chapter 2

Background

In this chapter, we will explain the background relevant to the explicit null changes in the Scala compiler.

2.1 The Scala Compiler

The Scala compiler is split into distinct phases. In this section, we will highlight the parts that are relevant to our work. The code example below is a simplified overview of the compiler structure.

```
def phases: List[List[Phase]] =
    frontendPhases ::: picklerPhases ::: transformPhases ::: backendPhases

/** Phases for parsing and typing */
protected def frontendPhases: List[List[Phase]] =
    List(new Parser) ::             // Compiler frontend: scanner, parser
    List(new TyperPhase) ::         // Compiler frontend: namer, typer
    ...

/** Phases dealing with TASTY tree pickling and unpickling */
protected def picklerPhases: List[List[Phase]] =
    List(new Pickler) ::            // Generate TASTy info
    List(new Inlining) ::           // Inline and execute macros
    ...

/** Phases transformating pickled trees to backend trees */
protected def transformPhases: List[List[Phase]] = ...

/** Generate the output of the compilation */
```

```
protected def backendPhases: List[List[Phase]] = ...
```

The compiler is split into four major lists of phases. The frontend phases handle turning text from source files into typed abstract syntax trees. The pickler phases turn typed trees into TASTy binary format, Scala's high-level interchange format that preserves complete representation of the source code. The transform phases lower the high level features into trees that are suitable for backend code generation. Notably, one of the phases is Erasure, which erases all types down to JVM types. The backend phases turn the lowered trees into JVM bytecode.

Since the work in this thesis is about modifications to the type system, with the types being erased during lowering, most of the transform phases and backend phases are not affected. Instead, the most relevant parts are the frontend and pickler phases.

2.1.1 Compilation Sources

The Scala compiler accepts a few different types of inputs as sources for compilation. First and foremost are `.scala` source files, which the parser processes and outputs as untyped abstract syntax trees.

The compiler also accepts sources that have already been previously compiled. For Scala 2 and Java, this would correspond to `.class` files. This is necessary to figure out the signatures and interfaces of the libraries that are being called. Using a classfile parser, the compiler produces the types of the relevant function calls and members directly. The compiler will mark these types as originating from Java code. Related are `.java` source files, in which case a built-in Java parser is used to produce untyped trees. These trees will also be marked as originating from Java code.

Last but not least, the compiler also accepts `.tasty` files, which are Scala 3's binary representation for compiled files. It uses Typed Abstract Syntax trees as its internal representation.

We expand on the mechanics of interoperating with implicitly nullable types in section 3.3.

2.1.2 TASTy Files

In Scala 2, when a program is compiled, the compiler produces a JVM class file. When a Scala 3 program is compiled, the compiler produces two outputs for each source file: a standard JVM class file and a TASTy file. TASTy, which stands for Typed Abstract Syntax Trees, is a binary format unique to Scala 3 that stores the complete typed AST of the source program. The pickler phase is what generates the TASTy files.

Class files suffer from the issue of type erasure, where generic type parameters are stripped at compile time, so `List[String]` and `List[Int]` both become `List` in the

bytecode. This impairs the compiler's ability to infer type information from Java libraries and compiled Scala 2 files.

Compared to class files, which contain executable bytecode generated after the transformation phases of the compiler, TASTy files reflect the original Scala program before any transformations. TASTy files record all the additional information in the Scala language, including type parameters, inferred types, and compiler flags. Crucially, this means we have the option to also enable nullification for Scala 3 source files that were compiled under implicit nulls by recompiling them from their TASTy representation. We discuss the implications of this decision in section 3.3.4.

2.1.3 Typer

The Typer is where Scala turns untyped abstract syntax trees (ASTs) into typed ASTs. In Scala, the untyped ASTs only contain the syntax of the program. It is the Typer's job to infer everything the compiler knows about the semantics of the program, and encode the knowledge within the types. It does this by recursively traversing the tree structure, assigning types to expressions, and checking that declared types are consistent and inferring types where necessary.

For each node in the AST, the Typer performs type inference [24] and type checking. Type inference is determining the type of an expression when it is not explicitly declared. For example, in `val x = "Hello"`, `x` will have its type inferred as `String`. Type checking is verifying that an inferred or declared type of an expression is compatible with its expected type. For example, if a function expects a `String` but receives a `String | Null`, the Typer will emit an error.

Compatibility between types is determined through subtyping. Under explicit nulls, `Null` is no longer a subtype of `String`, so the Typer will now reject some code that was previously accepted. We describe the mechanics of subtyping in more detail in section 2.2.5.

Contexts

To handle the complex interactions of the recursive traversal of ASTs, the Typer passes a `Context` object through every typing function. The `Context` contains information about the current lexical scope, such as names that are in scope, what imports and settings are active and more. When the Typer enters an inner block, it can augment the outer context with additional info to reflect the change in scope.

We utilize this recursive context passing structure in section 3.2. We augment the context with nullability information that tracks the effects of each code block on the nullability of the variables in scope. As the Typer traverses through the ASTs, it also threads the nullability information through the context and computes the nullability information for each block.

Member Selection

A key operation the Typer handles is member selection, i.e. expressions of the form `x.f` where `f` is a field or method on `x`. To type such an expression, the Typer looks up `f` in the type of `x`. If the type of `x` is a union `T | Null`, the type `Null` has no members, so the lookup would fail.

In section 3.2, we use the information from flow typing to allow member selections on nullable types when appropriate.

2.1.4 Inlining

The Scala compiler supports inline definitions declared with the `inline` modifier. An inline method is not compiled into a regular function; instead, its body is expanded directly at every call site during compilation. As inlining occurs before any of the transform phases, this allows for the later phases to see the expanded code directly, enabling optimization that would be impossible if it remained an opaque call. This is utilized later in section 3.1, where a helper function for casting nullable variables is denoted as `inline`.

2.2 Types in Scala

Scala has a rich and diverse type system that allows for expression of complex structures. The key type system change in explicit nulls is simple but has big effects on compilation as a whole. This section will provide background on the changes made to implement explicit nulls and the types relevant to the compatibility features.

2.2.1 Type Hierarchy

Types in Scala can be organized into a type hierarchy. On top of the hierarchy, we have the top type `Any`, which is a supertype of all types. At the bottom of the hierarchy we have `Nothing`, which is a subtype of all types. All other types are either a subtype of `AnyVal` (value types), which includes primitives like `Int`, `Boolean` and `Unit`; or `AnyRef` (reference types).

Before explicit nulls, the `Null` type was a subtype of any reference type. To enforce null safety, we no longer make `Null` the subtype of all reference types; instead the only supertype of `Null` becomes `Any`. Figure 2.1 shows the hierarchy of the two systems.

2.2.2 Union and Intersection Types

In the explicit system, all reference types are non-nullable by default. As such, we need to use union types to express nullable types. Union types represent the lowest common

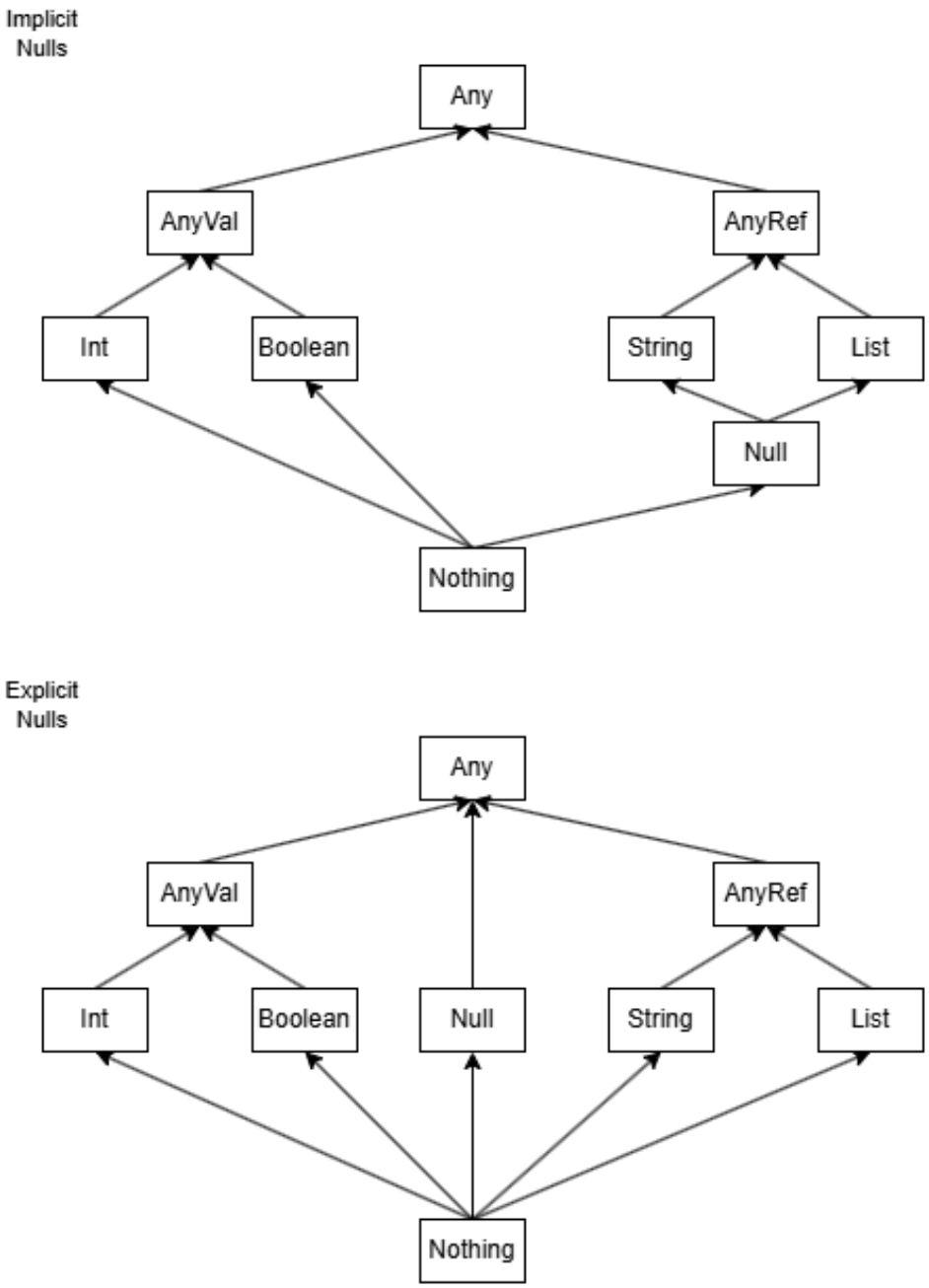

Figure 2.1: The Type Hierarchy of Implicit and Explicit Nulls

supertype of the two sides of the union. As such, attempts to select on nullable unions will fail since the lowest common supertype is `Any`, providing null safety. Intersections are the inverse of union types, representing the highest common subtype of the two sides of the intersection. We demonstrate union and intersection types in the following example.

```
val s1: String        = "Hello"
```

```
val s2: String        = null        // Disallowed, Null is not String
val s3: String | Null = null        // Allowed
s3.toUpperCase()                    // Disallowed, toUpperCase is not
                                    // a member of String | Null
val s4: (String | Null) & String = "Hello"
s4.toUpperCase()                    // Allowed through the intersection
```

In the code, the declaration of `s2` will emit an error since the expression `null` has type `Null`, but we are trying to assign it to a variable of type `String`. The declaration of `s3` is allowed since it has a union type including `Null`. However, we can see that selections on `s3` are disallowed. As `s3` is nullable, trying to dereference it may lead to a `NullPointerException`. Within the explicit null type system, the compiler does not allow selections on variables that can be null, preventing null errors at runtime.

We demonstrate intersection types in the declaration of `s4`. We intersect the union type with `String`. This results in `s4` once again returning to the type `String`, allowing selections on the variable. We will utilize intersections further in section 3.1 to create a polymorphic utility function that casts expressions to their non-nullable equivalent.

2.2.3 Singleton Types

A singleton type `x.type` represents the type of one specific value `x`. The singleton type `x.type` has exactly one inhabitant: `x` itself. Singleton types become useful when the identity of a specific object needs to be preserved through type-level operations. This is apparent with intersection types: a value of type `x.type & T` is both the specific object `x` and a value of type `T`, with neither fact discarded.

```
val a: Animal = new Cat()
val b: Animal = new Dog()

def isSame(x: a.type): Unit = ()

isSame(a)  // Allowed | a has type a.type
isSame(b)  // Error   | b is not a.type
```

We will make use of singleton types in section 3.1 when defining the return type of the non-null cast.

2.2.4 Path-Dependent Types

In Scala, types can be members of objects just as methods and values can. These are called abstract type members. When a type member is accessed through a specific object

instance, the resulting type is path-dependent, meaning its identity depends on which object it was accessed through.

```
trait Animal:
  type Food
  def eat(f: Food): Unit
  def favoriteFood: Food

val cat: Animal = new Cat()
val dog: Animal = new Dog()
```

Here, `Food` is an abstract type member. Accessing it through a specific instance gives a **path-dependent type**; `cat.Food` is the food of that specific cat, and `dog.Food` is the food of that specific dog. These are distinct types that the compiler tracks separately:

```
val catFood: cat.Food = cat.favoriteFood   // Allowed
val dogFood: dog.Food = cat.favoriteFood   // Error: cat.Food != dog.Food
```

The distinction of path dependent types will also be relevant with the non-null cast in section 3.1.

2.2.5 Subtyping

The Scala compiler uses subtyping as the relation that governs when one type can be used in place of another. We write `A <: B` to mean `A` is a subtype of `B`. Subtyping is reflexive in Scala, meaning each type is a subtype of itself. However, subtyping is **not always** transitive: if `A <: B` and `B <: C`, the Typer does not guarantee that `A <: C`. The type hierarchy described in the previous section gives the basic subtyping structure. `Nothing <: T` and `T <: Any` for all types `T`, and subclasses are subtypes of their superclasses.

Subtyping rules straightforwardly extend to unions as well. Each component is a subtype of the union, and a union is a subtype of another type if and only if both components are subtypes of said type.

```
A <: A | B        B <: A | B

A | B <: C   iff A <: C and B <: C
```

A useful consequence of these rules is that `String <: String | Null`, because `String <: String`. This is very commonly used when widening expressions to nullable types.

An intersection is a subtype of both of its components:

`A & B <: A` and `A & B <: B`

This means if `x` has type `T`, a value of type `x.type & T` can be used wherever `x.type` or `T` is expected, preserving both its identity and type. This is once again relevant in section 3.1.

Chapter 3

Implementation

In this chapter, we will detail the compatibility features that are implemented within the Scala compiler.

3.1 Non-null Cast

In cases where the programmer gets an expression that is typed as nullable, but can guarantee that the value is actually non-null, the programmer may wish to type it as non-nullable instead. In the following example, `nullIfNegative` has a return type of `String | Null`. But when passed a positive number, the function always returns a non-null `String`. As such, the programmer needs to put it in a cast to assign it to a non-null variable. In the code example below, v is guaranteed to be non-null, but the programmer needs to use the `asInstanceOf` function to cast it to a non-null String.

```
def nullIfNegative(num: Int): String | Null
val v: String = nullIfNegative(10).asInstanceOf[String]
```

As there are a lot of patterns where the programmer is able to guarantee a non-null value where the compiler cannot, we generalize this pattern and provide it as a utility function in the standard library for Scala as follows.

```
extension [T](x: T | Null) inline def nn: x.type & T =
    if x.asInstanceOf[Any] == null then
        scala.runtime.Scala3RunTime.nnFail()
    x.asInstanceOf[x.type & T]

val v: String = nullIfNegative(10).nn
```

The `extension` keyword adds methods to an existing type. In this case, the target type is defined to be any union type `T | Null`. The method is defined as inline; inlining happens early during compilation, which allows later phases of the compiler to optimize, improving efficiency. The function also checks that the value is actually non-null when invoked, and throws an exception if it is called on a null value. If the user does not want the exception behavior – for example for further optimization – they can use the `uncheckedNN` function instead, in which case a null value will pass through the cast unchecked. If assigned to a non-null variable, it will unsoundly contain a null value and throw a null pointer excception when accessed.

Special attention must be paid to the return type of the function, which is defined as `x.type & T`, where `x.type` is a singleton type that refers to the identity of the specific instance of `x` that the function is called on.

```
extension [T](x: T | Null)
    inline def nnPlain: T = x.asInstanceOf[T]

trait Animal:
    type Food
    def eat(f: Food): Unit
    def favoriteFood: Food

def feed(a: Animal | Null): Unit =
    a.nnPlain.eat(a.nnPlain.favoriteFood)     // Error
```

In the above example, the `nnPlain` method returns the type `T` directly instead of the intersection type. We then define a trait `Animal` that has an abstract type member `Food`. When the type `Food` is referenced in other methods within `Animal`, it implicitly means `this.Food`. Abstract type members must be implemented with a concrete type when the trait is extended. In this case, the use of `nnPlain` causes the function call on eat to fail, suggesting that the argument is not of the correct type. The `eat` function must take an argument that has the specific `Food` of the variable `a`, but `nnPlain` casts the type to `Animal#Food`, which causes the compiler to reject it. As such, it is necessary to preserve the path of the argument by using `x.type & T`; this saves the identity of the specific `x.type` and removes the nullness by intersecting with `& T`.

3.2 Flow-Typing

In a lot of common code, nullable variables are often checked before being accessed, either through an if statement, or by explicitly assigning to them right before accessing. In these cases, it is plainly clear to the programmer that the access is valid. It would be inconvenient for the programmer to have to cast these values. As such, the Scala compiler implements flow-typing to statically determine nullability where possible.

```
val s: String | Null = ???
if s != null then {
    s.toUpperCase()
}
```

For example, even though the variable s has the type `String | Null`, it is used after an explicit comparison. This means that if execution ever reaches the method call, s must be not null. Therefore, we want the compiler to allow such selections.

3.2.1 Nullability Information

The compiler implements flow-typing by keeping track of the effect of each block of code on the nullability of variables. A key distinction is that the information is not about the state of the program at any point of execution, but the effects of a section of code on the state of the program. The information that is kept track of is separated into two sets. The asserted set represents the set of variables that are proven to be non-null after the non-exceptional execution of a piece of code, either due to being directly assigned a non-null value, or being after an explicit null check e.g. `v != null`. The retracted set represents the set of variables that **may** be null after the execution of a piece of code, most commonly because the code assigns a nullable value to the variable.

A reasonable question to ask is why the information is separated into two sets, instead of just one set of variables that is added to when asserted, and removed from when retracted. There are instances where the compiler needs to conservatively estimate the effects from a code block. This is due to the possibility of abrupt termination [2]. In any block of code, it is possible that an exception is thrown, after which execution does not continue through the whole block. This means that the asserted facts gathered from a block may not all have been executed. Conversely, all the retractions may have happened before the exception. To illustrate this problem, see the following example of a while loop:

```
var s: String | Null = "Hello"
while (someCondition) {
    s = null
    // possible break
    s = "Hello"
}
s.toUpperCase()
```

If we only use one set, the information from the while block will be that s is still within the set of non-null variables, since even though the first assignment removes s from the asserted set, the second assignment once again returns it to the set. However, if the while loop was broken out of before the second assignment, s will be null when it reaches the

method call. With the one set approach, there is no method to only isolate the retractions from a block. With the two set approach, when terminating normally, the compiler can take both sets of information; when terminating abruptly, the compiler can take just the retracted set.

We also supply each block with a `Context`. The `Context` contains nullability information about what came before the tree currently being evaluated. This is important for nullability information to be transferred across different variables:

```
var y: String | Null = "Hello"
{
  var x: String | Null = y
  x.toUpperCase()
}
```

In the code example, we assign y to the variable x, then we attempt to invoke a method on x. The nullability information in the outer block determines that y will be non-null. By passing it as context into the inner block, it allows the Typer to further infer that x will also be non-null. After that, the method call on x will be allowed.

3.2.2 Information Composition

To compose effects of multiple blocks together, we define the `seq` operator. Figure 3.1 represents when two pieces of code are connected sequentially. In block A, the effect of the code is that `x` is asserted as non-null and the nullability of `y` is retracted. In block B, `y` is asserted as non-null and `x` is retracted. We then compute the effect of the overall block. The new asserted set is computed by subtracting the retracted set of B from the asserted set of A, then unioning it with the asserted set of B. The new retracted set is computed through combining the retracted sets of the two blocks. We do not subtract the asserted set of block B because the block may throw an exception before the assertion happens.

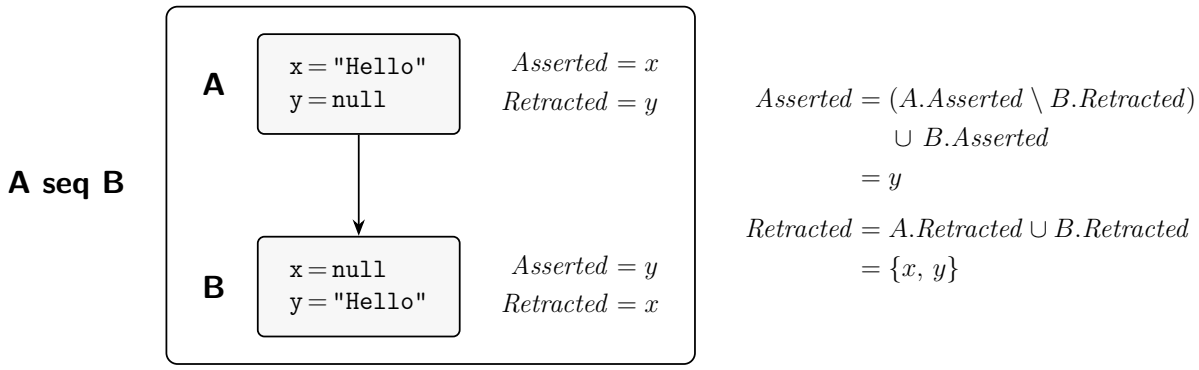

Figure 3.1: Sequencing two sets of nullability information

When dealing with branching statements, it is incorrect to use the `seq` operator because the code flows through more than one possible path. As such, we define the `alt` operator in Figure 3.2. The `alt` operator combines the nullability information from two separate branches of code. The new asserted set is the intersection of the two asserted sets, and the retracted set is the union of the two retracted sets.

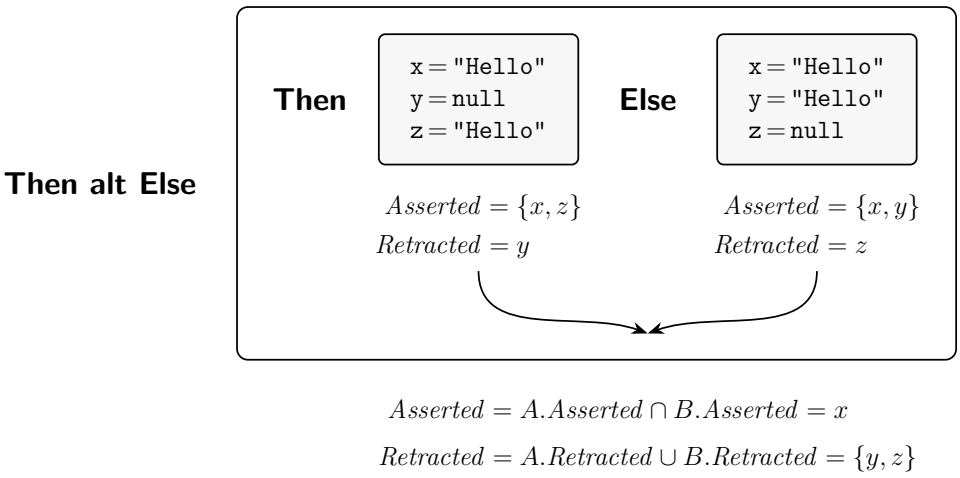

Figure 3.2: Alternating two sets of nullability information

3.2.3 Non-termination

Now consider an example where we have a branching statement, but one side of the branch never terminates normally. Some examples of code that never terminate normally are infinite loops and throw statements. How should the `alt` operator handle such cases? A naive first choice may be to discard both asserted sets entirely and just return a union of retracted sets. However, recall that the definition of the asserted set is the variables that are guaranteed to be non-null when a block terminates **normally**. This means that if one side of the branch **always** terminates abruptly, we can preserve the asserted set of the non-abrupt side into the resulting information. To represent the notion of **guaranteed** abrupt termination, we introduce a new special value for the asserted set.

<Abrupt> is a special value that represents code blocks that the compiler can guarantee terminate abruptly. This allows for the `seq` and `alt` operators to be more precise. Figure 3.3 demonstrates the behavior when the `alt` operator encounters an <Abrupt> value.

Figure 3.4 demonstrates the `seq` case, we reason that when viewed as a whole, a code block sequenced after an abruptly terminating code block is also itself abruptly terminating. So the effect of `Abrupt seq B` or `A seq Abrupt` also results in an <Abrupt> asserted set. However, the behavior of the retracted set is preserved, and we union as usual.

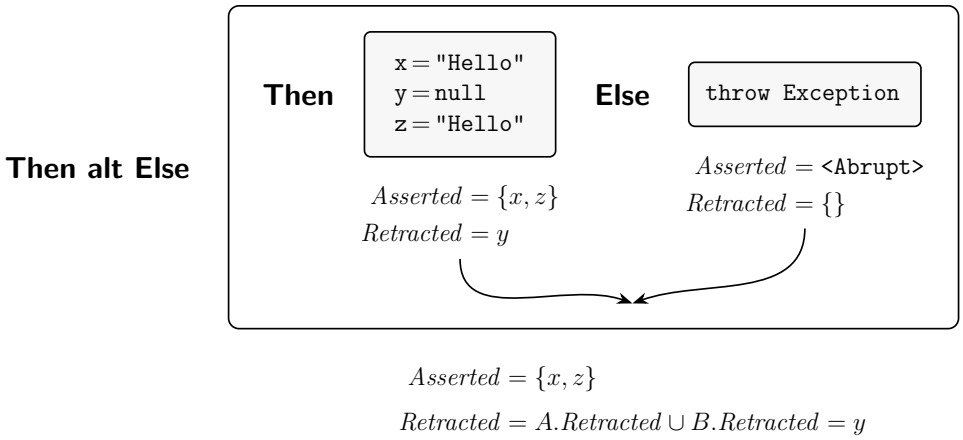

Figure 3.3: Alternating with a non-terminating block

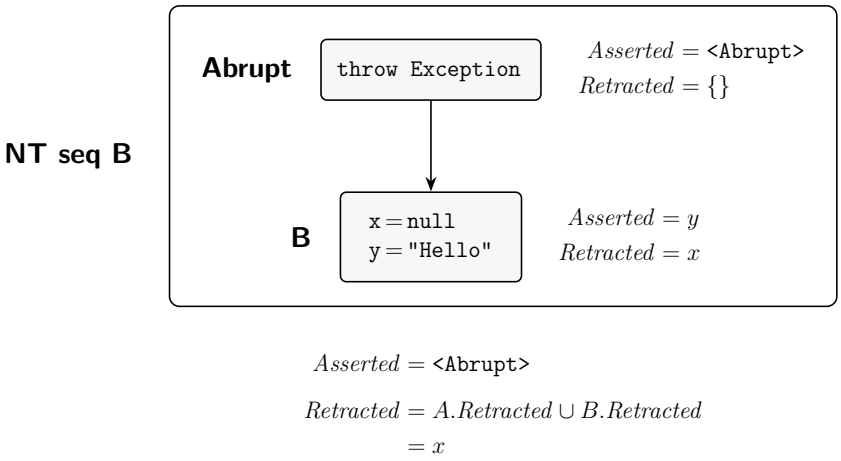

Figure 3.4: Sequencing after a non-terminating block

3.2.4 More on Contexts

Until now, when describing computations, we mostly assumed that a code block starts with no information. Indeed, the top level of the program starts with an empty `Context`. However, when typing in general, a block can be passed an outside context. Whenever the Typer recurses into an inner block, it needs to determine what is the correct context to pass to the inner block. When completing a block, it should also pass out a set of information representing the nullability effect of the block itself. To illustrate this mechanism, we use the following example of an if-then-else block.

Figure 3.5 illustrates the logic for managing contexts within an if-then-else block. For each sub-block, it requires an input context and outputs the effect for the block itself. Starting at the top, we pass the outer context to the if block directly as its context, as the execution of the if-then-else block begins at the if block. The if block also produces a set of

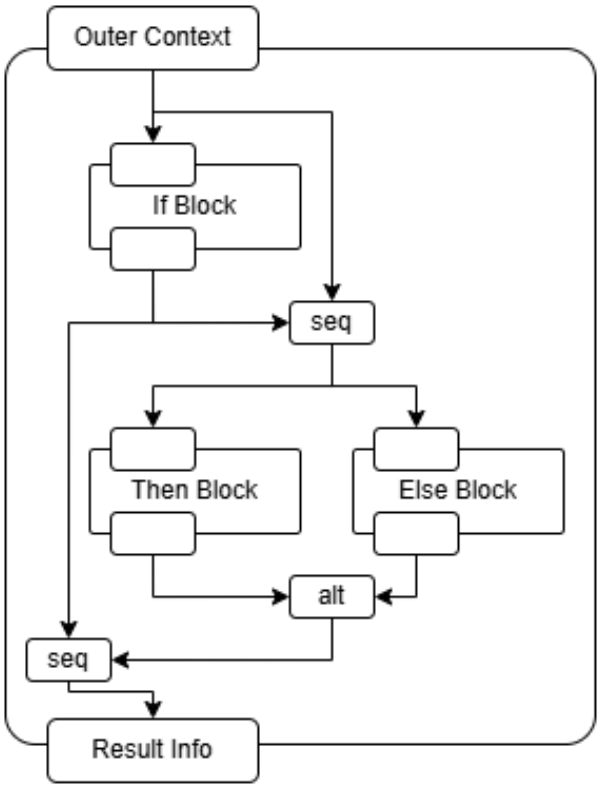

Figure 3.5: Flow Graph for an If-Then-Else block

nullability information (as variable assignments could be present within the if block). Then we must consider the context we pass to the then and else blocks. Recall that the effect output by a block only describes the block itself, and does not include anything before it. Therefore, the correct behavior here is to pass `Outer seq If` to represent the effect of the outer code being followed by the if block before entering the then and else blocks. Since `seq` is order dependent, we distinguish the order by the entry position: the top is the first block, while the side is the second block. Last but not least, we must compute the effect of the if-then-else block as a whole to pass as output. As before, we only wish to describe the effect of this block, irrespective of the outer context. As such, the computation we perform is `If seq (Then alt Else)`, which represents the overall code flow of the If block, then either one of Then or Else block.

With the ability to track nullability information for blocks of code, the Typer can now perform flow-typing on possibly null variables. When the Typer encounters a field access (e.g. `x.f`) or an identifier (e.g. just a variable use `s`), it will check the asserted set within its given `Context` for the prefix. If the prefix exists within the asserted set, according to our definition, the variable must be non-null. As such, the prefix will be cast to its non-null equivalent. This allows for valid selections on nullable variables.

3.2.5 Try-Catch-Finally

It is harder to determine the correct behavior for a try-catch-finally block. First, we determine the actual possible program flows within a try-catch-finally block. In general, it is possible for any point in the try block to emit an exception, after which it may be handled by a case in the catch block. If there is a corresponding catch block, the code within it could either resolve the exception, or further throw an exception, after which execution passes to the finally block. If there is no corresponding catch block, execution passes from the try block to the finally block directly. When the finally block executes, depending on the source of its entry, it either proceeds to the remaining program, or if the exception was not handled, propagates the exception up the stack. The flow of execution is represented in Figure 3.6, where exceptional cases exit to the right, and normal resolutions exit downward.

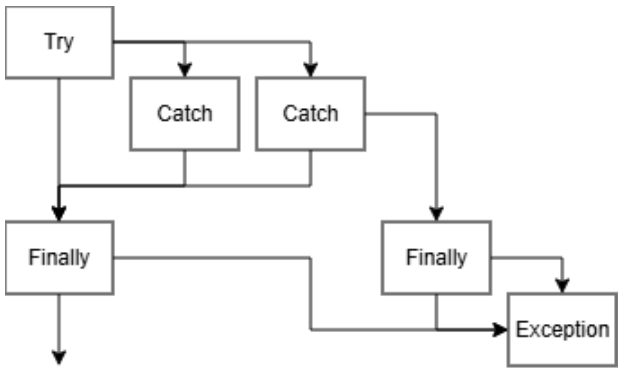

Figure 3.6: Flow Graph for a Try-Catch-Finally block

We now describe the logic behind computing the nullability information in a try-catch-finally block. For each sub-block, we will also specify the nullability information provided to it. In figure 3.6, there are cases where we only wish to utilize the retracted information from a block (like the while-loop example in section 3.2.1). We represent only taking the retracted information by extending the arrow from the right-hand side of the output block. If we wish to take the whole set, we extend from the left-hand side.

We start at the try block, to which we supply the nullability information of the outer context; since it is the entry point, no additional change is needed. Then we move on to the catch blocks. For each catch block, we provide it context with the outer context sequenced with the retracted set of the try block. This is because we do not know at which point within the try block an exception is emitted, therefore we must conservatively estimate the retractions from the try block.

Since catch blocks are mutually exclusive, we combine the information between them by chaining `alt` operations. We then compute the overall effect of the try-catch block. The full expression for the try-catch block should be `Try alt (Try.retracted seq Catches)`,

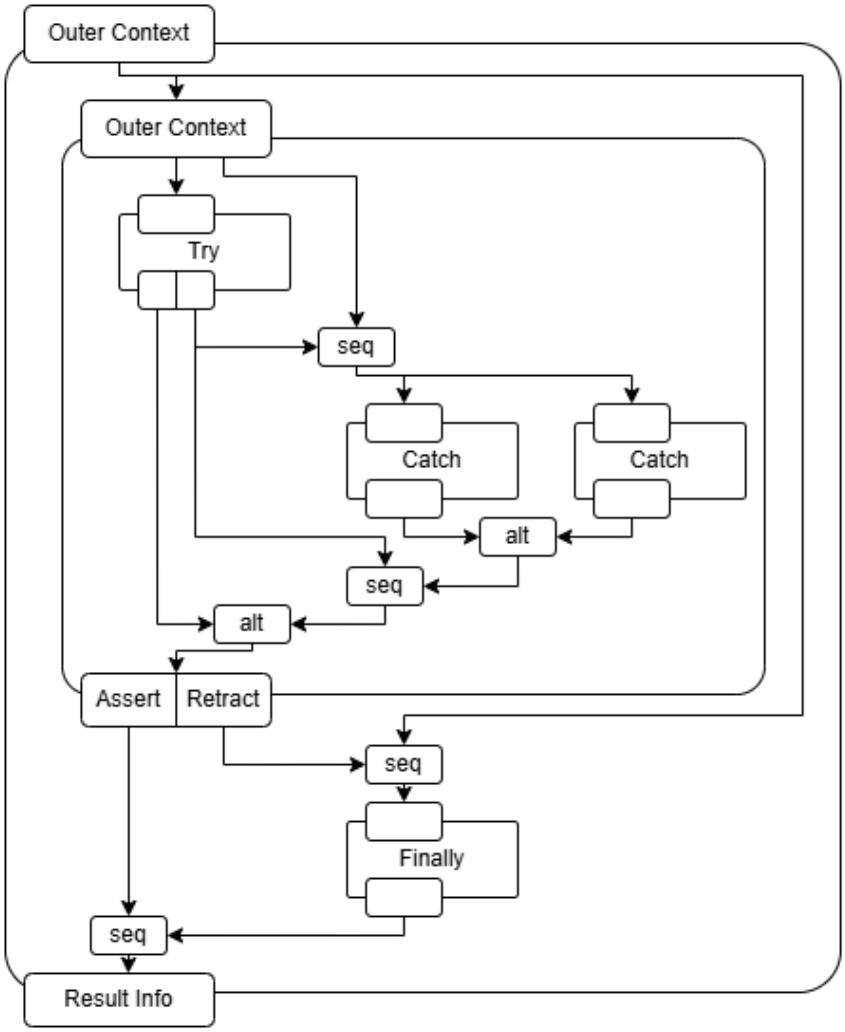

Figure 3.7: Info Flow Graph for a Try-Catch-Finally block

which represents the normal completion of the try block alternated with the recovery from a catch block.

For the finally block, we must be careful about the special nature of its execution. There are two possible contexts, one where the try-catch completed normally, and the other where the try-catch completed abruptly. To not allow invalid selections to occur within the finally block, we choose to provide the finally block with a conservative context, i.e. we only use the retracted information from the try-catch block, and none of the assertions. We seq the retracted information after the outer context, then pass it to the finally block.

Last, we compute the effect of the try-catch-finally block as a whole. We `seq` the effect of the try-catch block with the effect of the `Finally` block, giving us the overall effect of the try-catch-finally block.

```
var v: String | Null = ???
try {
    maybeThrowException()
    v = ""
    maybeThrowException()
} catch {
    case e: Exception =>
        val x = v.length() // Error
        cleanup()
        throw e
} finally {
    val y = v.length()     // Error
}
val z = v.length()         // Allowed
```

As an example, in the above code block, we assign a non-null value to `v` in the try block. We try to select on `v` in the catch block, the finally block and after the finally block. The only operation that is accepted is the last one. As an exception may be thrown before the assignment in the catch block, the value cannot be guaranteed to be non-null in the catch and finally block. However, since the only catch block always terminates abruptly, we can guarantee that if we reach `z`, the value will always be non-null.

3.2.6 Variable Analysis

So far, we have accepted that the state of variables can be tracked as part of flow typing information. However, that is not always true. Consider the following example:

```
var x: String | Null = ???
def y =
    x = null

if x != null then
    // y can be called here
    val a: String = x // Disallowed
```

In this example, the variable `x` is captured in a closure in function `y`. As such, facts about its nullability could be invalidated at any point. Therefore we restrict the set of variables that we keep track of to specific requirements.

First, they must not be assigned to within a closure like the above example. Second, the variable must also not be read inside a nested method. To see why that is, refer to the following example:

```
var x: String|Null = ???
  if (x != null) {
    def f: String = {
      val y: String = x // error: the use of x is out of order
      y
    }
    x = null
    val y: String = f // danger
  }
```

Third, we do not track mutable fields of classes. A mutable field of a class could be reassigned by any code that has a reference to the object, as in the following code.

```
class Box:
  var x: String | Null = "hi"

  def foo(): Unit =
    if (x != null) {
      callOtherMethods()
      val a: String = x   // error: x can be mutated between the
                          // check and the use, with no way to
                          // verify which Box's x is assigned to
    }
```

Our flow-typing logic is unable to account for such mutations.

Nevertheless, it is a very common pattern to have mutable fields within classes, and to perform null checks before accessing them. There may be cases where the programmer knows that a variable's value is stable (i.e. will not be nullified through aliases), but the compiler is unable to deduce it as such. In these cases, we provide the `@stableNull` annotation for the user to indicate to the compiler that a variable is stable, and enable flow-typing for said variables. If a variable in an object is annotated `@stableNull` at its declaration site, the analysis treats it like a local variable. In particular, it unsoundly assumes that the variable will not be nullified through aliases.

3.3 Interoperation and Flexible Types

3.3.1 Nullification

With the ability to interoperate with Java types and distinguish them from Scala types, the compiler needs to deal with the implicit nullability of Java types. Consider the following example:

```
// Convenient approach   (String|Null => String)
val s1: String|Null = j.takeAndReturnString("Hi")
val s2: String      = j.takeAndReturnString(null)    // Unsound

// Sound approach        (String => String|Null)
val s3: String|Null = j.takeAndReturnString("Hi")
val s4: String      = j.takeAndReturnString(null)    // Error
val s5: String      = j.takeAndReturnString(
                          null.asInstanceOf[String]
                        ).asInstanceOf[String]
```

In the above code, we call a Java method `takeAndReturnString()`. According to our Java parser, it has a type of `String => String`. However, since it is a Java type, we do not know whether or not the function takes and returns a nullable or non-nullable string. Ideally, it would be the programmer's task to determine the correct type, and the compiler should allow the programmer to denote all four possibilities of the argument and return type. We then show two approaches to translating the Java type to a Scala type.

The first approach is the convenient approach, where we do not nullify the return type, and invert the nullification behavior in the argument position, so `takeAndReturnString` would be given the type `String|Null => String`. Variable `s1` has the type `String|Null` and passes a non-null `String`, whereas variable `s2` has the type `String` and passes `null` as argument to the function. Both of these lines pass the Typer and compile. However, as established, reference types in Java are implicitly nullable, which means that in reality `takeAndReturnString()` could return `null`, which makes `s2` unsound. Furthermore, the function may have been implemented only expecting a non-null `String`, which means it may throw errors if it receives `null` as an argument. As such, if we use the convenient approach, we would be prone to null pointer errors.

We then introduce the sound approach, where we nullify return types into their null union, and invert the nullification behavior in the argument position. Under this approach, `takeAndReturnString` is given the type `String => String | Null`. As seen in variable `s3`, passing in a non-null argument and assigning the result to a nullable string works as expected. In example `s4`, passing a nullable argument to the function and assigning to a non-null string causes the compiler to issue an error, warning the user that there is a risk of a null pointer error down the line. However, there are cases where the programmer could guarantee that a Java method call is non-null, perhaps due to having access to the source code or documentation. In such cases, the user needs to cast the argument and expression down to a String explicitly.

Even though it would be ideal to use the sound approach to minimize the possibility of `NullPointerException`s, in reality, Scala code very often interoperates with Java code while making assumptions that the return types are non-nullable, and often these assumptions are actually correct. It would be unreasonable to require users to constantly insert casts and explicit checks to work with Java code. So it would stand to reason that

the convenient approach should be used. However, when accounting for variance, there actually exist code patterns that neither the sound approach nor the convenient approach can account for.

3.3.2 Variance

When a type constructor such as `List[T]` or `Array[T]` is parameterized by a type `T`, the subtyping relationship between the resulting types depends on the variance declared on the type parameter. There are three cases.

Covariance (`+T`): if `A <: B` then `F[A] <: F[B]`. The subtyping relationship is preserved in the same direction. Lists are covariant, so `List[String] <: List[Any]` since `String <: Any`.

Contravariance (`-T`): if `A <: B` then `F[B] <: F[A]`. The subtyping relationship is reversed. Function types are contravariant in their argument. `Any => Unit` is a subtype of `String => Unit`, because a function that accepts any value certainly accepts a `String`.

Invariance (no notation): `F[A] <: F[B]` only if both `A <: B` and `B <: A`. No subtyping relationship is induced in either direction. Unlike Java, arrays are invariant in Scala.

```
val strings: List[String] = List("a", "b")
val anys: List[Any]       = strings      // Allowed: List is covariant

val arrStrings: Array[String] = Array("a", "b")
val arrAnys: Array[Any]       = arrStrings // Error: Array is invariant
```

Whereas Lists are covariant because they are immutable, arrays must be invariant because they support both reading and writing. If `Array[String]` were a subtype of `Array[Any]`, then the following code would be unsound:

```
val arrAnys: Array[Any] = arrStrings   // hypothetically allowed
arrAnys(0) = 42                        // writes an Int into an Array[String]
val s: String = arrStrings(0)          // runtime crash
```

3.3.3 Invariance

A unique problem arises when trying to convert a Java `Array` to Scala. Recall that we want to allow the programmer the freedom to denote any Java type as either nullable or non-nullable.

```
// Convenient approach   (Array[String])
val s1: Array[String|Null] = j.getStringArray()    // Error
val s2: Array[String]      = j.getStringArray()    // Allowed

// Sound approach        (Array[String|Null] | Null)
val s3: Array[String|Null] = j.getStringArray().nn // Allowed
val s4: Array[String]      = j.getStringArray().nn // Error
```

In the above example, we consider the two approaches of adapting a Java type to Scala. The `getStringArray()` method returns an `Array[String]`, but since it is implemented in Java, the type does not tell us whether or not the array of strings it returns could contain null references. For an `Array[T]` to be a subtype of `Array[S]`, `T` must both be a supertype and subtype of `S`.

In the first approach, we do not nullify any types or type arguments from Java. `s1` is prohibited. Since `String` is not a supertype of `String | Null`, `Array[String]` is not a subtype of `Array[String | Null]`. Even the convenient approach cannot allow the programmer to denote the type as a nullable array of strings.

In the second approach, we nullify all types and type arguments coming from Java. So `j.getStringArray()` is typed as `Array[String|Null]|Null`. In this case, `s4` is not an allowed code pattern. Since `String | Null` is not a subtype of `String`, it follows that `Array[String | Null]` is not a subtype of `Array[String]`. This disallows the programmer from easily denoting an array of non-null strings coming from Java.

As we can see, neither the convenient approach nor the sound approach allows the programmer to denote both the nullable and non-nullable arrays. Here we introduce flexible types. A flexible type is an abstract type with inverted type bounds. A `FlexibleType[T]`, written as `(T)?`, is both a supertype of `T | Null` and a subtype of `T`. This typing relationship is inherently unsound, but allows for better interoperation with Java code.

```
type FlexibleType[T] >: T | Null <: T
// Flexible approach     (Array[String?]?)
val s5: Array[String|Null] = j.getStringArray()
val s6: Array[String]      = j.getStringArray()
```

Since flexible types can act as both `T | Null` and `T`, both denotations of the Array are allowed by the compiler. The logic behind why the subtyping is allowed is not immediately apparent. We detail both cases as follows:

For `Array[String?]` to be a subtype of `Array[String|Null]`, `String?` must be both a supertype and subtype of `String | Null`. `String?` is a supertype of `String | Null` by definition. Scala determines a type to be a subtype of a union if it is a subtype of either side of the union. Since `String?` is a subtype of `String` by definition, the Typer

also determines `String?` to be a subtype of `String | Null`. As such, `Array[String?]` is a subtype of `Array[String|Null]`.

For `Array[String?]` to be a subtype of `Array[String]`, `String?` must be both a supertype and subtype of `String`. `String?` is a subtype of `String` by definition. Since `FlexibleType` is defined in the Typer as a bounded type, when determining if `String?` is a supertype of `String`, the Typer will compare with the lower bound of the `FlexibleType`, which is `String | Null`. And since `String <: String | Null`, `Array[String?]` is a subtype of `Array[String]`.

As an aside, there are nullability annotations in Java. They are not enforced by the compiler, but IDEs will respect the annotations and issue warnings if they are violated. As such, when reading Java functions, the Scala compiler also respects any `@Nullable` or `@NonNull` annotations. In those cases, the compiler will not put them in a flexible type, instead putting `@Nullable` types in a null union, and leaving `@NonNull` types just as their base type.

3.3.4 Flexible Types for Implicitly-nullable Scala

As mentioned in section 2.1.2, Scala 3 introduces a new intermediate representation in TASTy files. TASTy files store the complete abstract syntax tree of the source program, enabling better compile-time optimizations, avoiding the issue of type erasure.

With the enriched information from TASTy files, we are able to discern whether or not the source file was compiled under explicit nulls. For source files coming from implicitly nullable code, we can try to apply flexible types to the types in the TASTy file. This comes with the benefit of allowing for smoother interoperation, and also resolving the similar issue of being unable to denote nullability for invariant type constructors.

Unfortunately, the added complexity of TASTy files also comes with downsides. The Scala type system allows for expressions of higher-kinded types, but attempting to apply flexible types to higher-kinded types would result in a kind mismatch, as flexible types should only be applicable to simple-kinded types. However, even if we try to work around the issue by only selectively flexifying simple-kinded types, the existence of type variables further complicates the issue. It is possible for type variables in Scala to be either simple- or higher-kinded; although it is a rare code pattern, it makes it extremely difficult to apply flexible types to type variables consistently without also accidentally applying them to higher-kinded variables.

Taking into account the benefits and downsides, we deem that it is not worth the effort to apply flexible types to TASTy sourced code. Most old Scala code that is commonly depended upon is either from Scala 2, or does not have the edge cases that require flexible types. In the rare cases that there are difficulties in interoperation, the unsafe nulls feature is also available as a workaround.

3.4 Unsafe Nulls

The last and most powerful feature the Scala compiler provides for compatibility is the `unsafeNulls` language import. In the scope of where this feature is imported, the compiler enables specific checks within the Typer that allow for the code to behave similarly to the implicit type system. Note that this is distinct from reverting back to the implicit type system; the change to the explicit type system is a **global** change that modifies the subtyping relationship between the `Null` type and reference types, whereas the unsafe null import is a **local** flag that relaxes type checking in such a way that unsafe null accesses are allowed.

In tandem with the `unsafeNulls` import, we provide the `safeNulls` counterpart, which reverts the effects of an `unsafeNulls` import within its scope. The scope of a language import can range from an entire project down to individual source files, classes, methods, or even subexpressions. As a result, this allows for increased granularity in controlling what parts of the program we want to be null safe, and which parts we want to allow to be null unsafe.

```
// PROJECT LEVEL:  project is compiled with both -Yexplicit-nulls
//                 and -language:unsafeNulls flags
// FILE LEVEL: we want safe behavior in this file
import scala.language.safeNulls

def main(): Unit = {
  val s: String = null // Error
}

def unsafeCode() = {
  // SCOPE LEVEL: a troublesome code pattern is present, we
  //              revert to unsafeNulls
  import scala.language.unsafeNulls
  val s: String = null // Allowed
}
```

In codebases where it may take a significant amount of effort to migrate to the explicit null system, this import is especially useful. The user can compile the project with both the `-Yexplicit-nulls` and the `-language:unsafeNulls` flags, which switches the compiler to the explicit nulls type system, but enables relaxed type checking. This allows for a majority of previously implicitly-nullable code to continue compiling. But when writing new code, the user can re-enable the safe-null behavior by importing `safeNulls` at the beginning of the file. This behavior can even be nested further: if some compatibility issue arises within this file, the user can once again import `unsafeNulls` to enable relaxed typing. In the unlikely case that the user imports both within the same statement, `safeNulls` takes precedence.

We now describe how the `unsafeNulls` import changes typing checks to enable unsafe null operations.

First, when selecting on a value with type `T | Null`, members of `T` can also be found.

Second, if `T1` is not a subtype of `T2` under the explicit null type system, but `T1` is a subtype of `T2` under the implicit null type system, then a value of `T1` can be used as a value of `T2`.

Third, if `T1` is not a subtype of `T2` under the explicit null type system, but `T1` is a subtype of `T2` under the implicit null type system, then a value of `T1` can access extension methods and implicit conversions of `T2`.

Lastly, if a block that does not have `safeNulls` enabled produces a value that does not conform to the expected type of its outer block, and the outer block has `safeNulls` enabled, then the compiler will cast the last expression to the expected type if it conforms under unsafe nulls.

Chapter 4

Evaluation

In this chapter, we will evaluate the effects of the explicit nulls changes on existing programs. There are two parts to this analysis. First, the disruptiveness of explicit nulls. What projects fail to compile under explicit nulls, and for what reason? Second, the effectiveness of compatibility features. For projects that fail to compile, how much change is needed to port the project to the new system? How much is each feature needed respectively?

4.1 Experimental Setup

To evaluate the effect of explicit nulls on existing programs, we will be using sets of existing Scala projects as a benchmark. For this experiment, we use two sets of projects. The Community Build is a selected set of 40 widely used Scala projects integrated into the Scala compiler repository's continuous integration process and acts as an integration test for minor compiler changes. The Open Community Build is a set of ∼2000 open source Scala projects that cover a wide range of code styles and use cases. It is used to test major releases of the compiler to prevent regressions.

4.1.1 Disruptiveness

To measure the disruptiveness of explicit nulls, we enabled explicit nulls on the Community Build, and attempted to port the projects so that they compile under explicit nulls. Then we measured the amount of code changes needed for each project to estimate the amount of effort needed. Our porting methodology was best-effort. We first updated all types that are supposed to be nullable by widening their types accordingly. We only inserted casts where necessary. Most of the porting was done by hand, while some was assisted by LLMs. The ported code is available online in dotty-staging repositories.[1]

[1] https://github.com/HarrisL2/scala3/tree/safe-null-community-build/community-build/community-projects

We also aim to measure the disruptiveness of enabling explicit nulls and unsafe nulls at the same time. Ideally this would have the lowest amount of impact, and be compatible with as many projects as possible. As such, we will run it on the Open Community Build to test it on a wide range of projects, then examine what projects fail to compile and for what reason.

4.1.2 Effectiveness

To investigate the effectiveness of flow-typing and flexible types in assisting porting code, we implement a flag that disables each feature respectively and rerun the Community Build. We use the number of compilation errors when a feature is disabled as an indication of the amount of effort saved by said feature, as without that feature the programmer would have to spend additional time modifying the code to get the project to compile.

4.2 Community Build

Table 4.1 contains the number of changes that were needed for each project. Notably, there are 15 projects that do not need any changes to compile under explicit nulls, namely cats-mtl, coop, discipline, discipline-munit, discipline-specs2, effpi, libretto, Monocle, munit-cats-effect, perspective, scalacheck-effect, scalatestplus-scalacheck, scalatestplus-testng, shapeless-3 and simulacrum-scalafix. Some of these projects did not need code changes because they are small in size and did not have any Java interop or null idioms such as initialization and deallocation. However, projects like Monocle, scalatestplus-testng and effpi did in fact have Java interop.

It should be noted that the numbers of errors shown here do not directly correlate to the amount of code changes required. Due to the incremental nature of compilation, fixing some compilation issues in one file could reveal issues in another file, leading to a lower error count than the total number of changes required. Conversely, one erroneous line could cause multiple compilation errors, leading to a higher error count than the number of changes needed to fix the issue.

The projects that needed the most changes — sconfig, cats-effect-3, scala-stm, scala-parallel-collections and scala-xml — are low-level libraries that are performance sensitive. They heavily utilize nulls to encourage garbage collection and to optimize space usage. For example, sconfig uses null as a value for unset properties. This means that to port the code appropriately, many type signatures need to be widened to a nullable type.

There are some outlier projects that contained code patterns that required special attention. ScalaPB is a code generator for protocol buffers. It had code generators that generated Scala code with implicitly nullable variables. The port needed to update the code generator to cast the variables correctly instead of porting the generated files. Afterwards,

Project	Errors before porting	Files Changed	Added	Deleted
sconfig	246	61	478	475
cats-effect-3	18	41	294	261
scala-stm	90	38	227	221
scala-parallel-collections	95	18	184	176
scala-xml	61	34	173	167
ScalaPB	116	61	172	165
Lucre	12	12	85	75
scodec	1	23	57	59
izumi-reflect	4	5	42	17
munit	8	8	38	37
cats	18	9	33	33
fastparse	20	8	29	29
parboiled2	10	8	20	19
specs2	6	6	13	10
intent	1	5	12	12
scala-java8-compat	1	2	10	10
scalaz	3	6	9	9
scalacheck	3	3	7	5
scala-parser-combinators	1	1	6	6
scodec-bits	2	2	4	3
endpoints4s	1	1	3	2
minitest	7	2	2	2
scalap	3	2	2	1
verify	6	2	2	2
xml-interpolator	2	1	2	2
cats-mtl, coop, discipline, discipline-munit, discipline-specs2, effpi, libretto, Monocle, munit-cats-effect, perspective, scalacheck-effect, scalatestplus-scalacheck, scalatestplus-testng, shapeless-3, simulacrum-scalafix	0	0	0	0

Table 4.1: Community Build statistics

the updated generator is used to regenerate the code. The number of lines changed in the above table includes both the generator and the generated files.

Lucre is a Software Transactional Memory library that also supports state tracking and event handlers. The library uses an implicitly nullable upstream dependency (ScalaSTM) which is written in Scala 3, meaning that we do not nullify types. The function that is called has a set of default arguments that are invalid under explicit nulls. We demonstrate the issue in the following code example:

```
// ============================================================
// upstream.scala | compiled WITHOUT -Yexplicit-nulls
// ============================================================
def foo(s: String = null): String // Default null argument

// ============================================================
// consumer.scala | compiled WITH -Yexplicit-nulls
// ============================================================
val v = foo() // Compilation Error
val v2 = foo(null.asInstanceOf[String]) // Workaround
{
  import scala.language.unsafeNulls
  val v3 = foo() // Convenient
}
```

The default arguments are defined in the dependency, which is then type checked under implicit nulls. As such, the null default argument is allowed and compiles. But when called in the actual project, the default arguments will be evaluated under explicit nulls at the call site, which causes the compiler to issue a type mismatch error. We are unable to change the code of the dependency, therefore we can only use workarounds.

It is possible to work around this error by supplying the argument explicitly within a cast. However, this gets very tedious if there are multiple default arguments. A more convenient workaround is to import `unsafeNulls` within the scope of the function call, which is the approach that we took.

We also analyze the categories of changes that were needed in Table 4.2. The most common type is widening types to their nullable union. This includes parameter types, return types and variables. Together, they account for around 45% of the changes. Next are `.nn` assertions at 23%. There are many uses of nullable variables where the compiler is not able to guarantee a non-null value, therefore `.nn`s need to be inserted. These two together already make up a majority of the changes.

There is also a significant amount of `.asInstanceOf` casts. These are often seen in low-level libraries where a variable that is otherwise non-null is assigned null to encourage the garbage collector to deallocate it. Related are variables that start as null, get assigned

#	Change type	Lines	Share	What it is
1	`| Null` **type annotation**	1162	45%	widening a declared type to nullable
2	`.nn` **assert-non-null**	595	23%	asserting a value is non-null at a use site (`x.f` → `x.nn.f`)
3	`.asInstanceOf` **cast**	216	8%	casting a nullable variable to the expected type
4	**null guard**	72	3%	new explicit null checks for flow typing (e.g. `v == null` / `v != null`)
5	`uninitialized`	37	1%	special expression for variables that only start as null
	other (logic / imports / whitespace / reformatted signatures)	504	20%	miscellaneous changes

Table 4.2: Breakdown of change types

a non-null value and never become null again. In these cases, the language provides a special expression `uninitialized` for such variables.

In Table 4.3 we list the change distributions for a select group of projects. Notably, ScalaPB did not have `.nn` changes, due to its nature as generated code. cats-effect-3 is a special case where a large number of casts were used. This is because the project assigned null to variables to indicate an object no longer in use. During most uses of said variable, it is verifiably non-null, and it would not make sense to widen the type of the variable at declaration only to accommodate the possibility of its deallocation at the end of its use. As such, we adapt the code by wrapping the final null assignment in a cast to the desired type to get past the Typer.

4.3 Ablation Study

For our ablation study, we use the **ported** version of Community Build projects as a baseline, then we disable either flow typing or flexible types during compilation. This will show the number of errors that each respective feature saves the programmer from fixing. Again, it should be noted that the number of errors for these projects is a baseline; more errors could be hidden by a project's compilation process.

Project	| Null	.nn	cast	guard	uninit
sconfig	353	99	6	13	0
ScalaPB	118	0	45	0	0
scala-stm	116	86	12	7	0
scala-xml	110	23	2	3	0
cats-effect-3	91	43	96	2	1
scala-parallel-collections	72	57	7	3	24
Lucre	25	23	14	4	0
cats	24	8	1	0	0
fastparse	23	5	1	0	0
munit	20	13	1	0	0
izumi-reflect	7	12	0	0	0
scodec	2	35	1	5	1

Table 4.3: Breakdown of change types for selected projects

4.3.1 Flow Typing

Out of the 40 community projects, 17 projects depended upon flow typing to successfully compile. Table 4.4 shows the number of errors for each project and table 4.5 shows the types and distributions of errors present.

The projects with the highest number of errors per kLoC are sconfig, scala-parallel-collections and scala-stm. This aligns with expectations; these are low-level libraries that heavily utilize nullable code for optimization purposes. This means if they are implemented correctly, the code naturally contains patterns that check for nullness before accessing variables that could be null. This allows for flow typing to reduce the need for explicitly casting variables.

The most common error is a member selection on a nullable union, which is to be expected, since the main code pattern that flow typing aims to assist in is checking for null before accessing members of an object.

The second most common error is a type mismatch. This is most often caused by checking for nulls before passing an argument to a function. Without flow typing, these arguments are no longer narrowed and cause a type mismatch.

The third most common error is no overload method matching the call. This is essentially the same error as the type mismatch above. However it surfaces differently since the compiler is unable to resolve which method to call before type checking the argument.

The last error is that an applied value takes no parameters. This is caused by trying to call a function when the callee is possibly nullable. When the callee is possibly nullable, the compiler does not allow it to be called as a function, and emits this error.

Project	Error	Error/kLoC
sconfig	124	10.1
scala-parallel-collections	41	6.0
scala-stm	25	4.0
ScalaPB	144	3.8
scala-xml	10	2.6
munit	8	1.9
Lucre	39	1.5
scala-java8-compat	6	1.3
fastparse	3	1.1
parboiled2	4	1.1
intent	1	1.0
scodec-bits	2	0.6
izumi-reflect	2	0.4
cats	11	0.3
scalacheck	1	0.3
scalaz	2	0.06
cats-effect-3	1	0.05
TOTAL	424	

Table 4.4: Error count with Flow Typing disabled

Error Code	Count	Share
[E008] member-selection on `T | Null`	294	69%
[E007] type mismatch (`T|Null` supplied where `T` required)	110	26%
[E134] no overload / extension applies	11	3%
[E050] applied value “does not take parameters”	9	2%
TOTAL	424	100%

Table 4.5: Breakdown of flow-typing errors by error code

The following code snippet demonstrates the four error types and how flow typing resolves them.

```
def takesString(x: String): Int = x.length
object Show:
  def of(x: String): String = x
  def of(x: Int): String = x.toString

def s(s: String | Null): Unit =
```

```
  if s != null then
    s.length        // [E008] member-selection on `String | Null`
    takesString(s)  // [E007] nullable value where `String` is required
    Show.of(s)      // [E134] no overload matches `String | Null`

// [E050] applied value "does not take parameters"
// `f` stays `(Int => Int) | Null`.
def f(f: (Int => Int) | Null): Unit =
  if f != null then f(1)
```

A notable example from the Community Build is from sconfig:

```
def computeCachedConfig(
    loader: ClassLoader,
    key: String,
    updater: Callable[Config]
): Config = {
  var cache: LoaderCache | Null = null
  try {
    cache = LoaderCacheHolder.cache
  } catch {
    case e: ExceptionInInitializerError =>
      throw ConfigImplUtil.extractInitializerError(e)
  }
  cache.getOrElseUpdate(loader, key, updater)
}
```

In this code, the nullable variable `cache` is written to within a try-catch block. The value `LoaderCacheHolder.cache` is guaranteed to be non-null, but reading it may throw an `ExceptionInInitializerError`. The catch block then catches this error, passes it through `extractInitializerError` and throws the result again. As such, if the try-catch block completes non-exceptionally, `cache` is guaranteed to be non-null. Without flow typing, the compiler would be unable to establish this fact.

4.3.2 Flexible Types

Out of 40 projects, 31 of them needed flexible types to compile. Table 4.6 shows the number of errors present in each project and table 4.7 shows the types and distributions of errors present.

The projects with the highest error/kLoC are scalatestplus-testng and scalatestplus-scalacheck. They contain Scala wrappers for JUnit APIs; this means they heavily feature

Project	Error	Error/kLoC
scalatestplus-testng	48	132.2
scalatestplus-scalacheck	25	60.7
scalacheck	106	27.1
scodec-bits	75	22.4
sconfig	242	19.7
intent	18	18.5
ScalaPB	629	16.8
minitest	11	16.7
verify	20	15.4
scala-java8-compat	68	15.1
scalap	33	14.9
munit	59	13.7
scala-xml	43	11.2
parboiled2	24	6.3
scala-stm	36	5.7
izumi-reflect	19	4.0
scala-parser-combinators	4	3.6
munit-cats-effect	1	2.8
scodec	5	1.9
scala-parallel-collections	10	1.5
scalaz	40	1.2
effpi	6	1.1
specs2	20	1.0
libretto	6	0.9
cats-effect-3	16	0.8
fastparse	2	0.7
endpoints4s	6	0.4
Monocle	2	0.3
perspective	1	0.1
cats	3	0.1
Lucre	1	0.04
TOTAL	1,579	

Table 4.6: Error count with Flexible Types disabled

interoperation with Java code. Hence, they benefit most from the convenience offered by flexible types.

ScalaPB has the highest number of errors, accounting for 40% of total errors. This is due to the large volume of generated files within the project; any generated file that has interop with Java will see errors when flexible types are disabled.

Error Code	Count	Share
[E008] member-selection on `T | Null`	840	53%
[E007] type mismatch (`T|Null` supplied where `T` required)	678	43%
[E134] no overload / extension applies to `T | Null`	47	3%
[E050] applied value "does not take parameters"	14	1%
TOTAL	1,579	100%

Table 4.7: Breakdown of flexible-types errors by error code

As can be seen, the most common errors are similar to those for flow typing. However, the mechanism behind their cause is slightly different. Instead of the target being a variable that is narrowed by flow typing, it is instead a symbol from a Java source file.

In the following code, we demonstrate examples of the 4 errors as well.

```
// JLib.java
public class JLib {
  public String name() { return "hi"; }
  public scala.Function0<String> thunk() { return null; }
}

def takesString(x: String): Int = x.length
object Show:
  def of(x: String): String = x
  def of(x: Int): String = x.toString

def use(j: JLib): Unit =
  j.name().length        // [E008] member-selection on `String | Null`
  takesString(j.name())  // [E007] nullable value where `String` is required
  Show.of(j.name())      // [E134] no overload matches `String | Null`
  j.thunk()()            // [E050] does not take parameters
```

4.3.3 Orthogonality of Errors

Notably, none of the errors between the flow typing and flexible types overlap. This is due to the orthogonal nature of the two features. Flow typing only works after an explicit check on a variable, whereas flexible types only affect variables that do not explicitly have their nullability checked. We illustrate the distinction in the following code:

```
var flexible                = javaMethod // type is (String)?
var ascribed: String | Null = javaMethod // loses flexible type
```

```
flexible.toUpperCase() // Allowed because of flexible types
if (flexible != null) {
  flexible.toUpperCase() // Flexible types supersede flow typing
}

if (ascribed != null) {
  ascribed.toUpperCase()  // Allowed because of flow typing
}
```

Flexible types only take effect when the programmer does not explicitly type a value from Java, and leaves the compiler to infer it, as seen in the variable `flexible`. Methods can be called on it, and the program is still allowed to use flow typing patterns on it. Even when flow typing is disabled, flexible types still allow the code to compile.

However, for all Scala values and Java values that the programmer ascribes a type to, flexible types no longer take effect. In these cases, the only way to use the variable would be to do an explicit check before accessing it, which allows flow typing to take effect.

As such, we can conclude that flow typing and flexible types both provide distinctive benefit in assisting the programmer in writing explicit null code. Flow typing streamlines narrowing of nullable variables through explicit checks and conditionals, and flexible types assist in interoperation with Java libraries. Disabling flow typing surfaced 424 compilation errors while disabling flexible types surfaced 1,579, meaning that flexible types are around 3.7× more impactful than flow typing in assisting with implicitly nullable code. Totaling up the ablation errors and the baseline errors before porting, we can deduce that flow typing and flexible types together account for 73% of the total errors, meaning that having flow typing and flexible types enabled saves the programmer from fixing 73% of errors.

4.4 Open Community Build

The Open Community Build is an extended set of 1,928 projects used to benchmark major versions of the compiler. To evaluate the effectiveness of the `unsafeNulls` language feature on enabling explicit nulls while minimizing code changes, we enable both explicit nulls and unsafe nulls globally within the compiler by default, and run the Community Build on it. The results are promising for compatibility. Out of the 1,928 projects, only 16 projects have compilation errors that need to be fixed. In comparison, without unsafe nulls, 854 projects fail to compile. In Table 4.8, we list the 16 failing projects and their error counts.[2]

[2] https://scala3.westeurope.cloudapp.azure.com/dashboard/compare?baseScalaVersion=3.9.0-RC1-bin-20260522-7db439f-NIGHTLY&targetScalaVersion=3.10.0-RC1-bin-20260529-08125eb

Project	Error	Type
beangle/commons	2	Macro
zio-archive/zio-connect	2	Macro
xuwei-k/httpz	2	Unit
beangle/serializer	1	Macro
devsisters/shardcake	1	Macro
gaelrenoux/tranzactio	1	Macro
kaizen-solutions/trace4cats-zio-extras	1	Macro
karelcemus/play-redis	1	Macro
laserdisc-io/log-effect	1	Macro
narma/tranzactio	1	Macro
scalamock/scalamock	1	Macro
senia-psm/zio-test-akka-http	1	Macro
vitaliihonta/scala-ql	1	Macro
zio-archive/zio-metrics-legacy	1	Macro
guardian/fastly-api-client	1	Unit
scalameta/munit	1	Unit

Table 4.8: Open Community Build Project Errors

13 out of the 16 projects failed to compile due to a match error from macro-related code. A major effect of enabling explicit nulls is that types from Java are wrapped in flexible types, and unsafe nulls does not mitigate this effect. Macro-related code needs to work closely with compiler internal types, which results in errors. Below is a simplified code example:

```
def inspect(typeRepr: TypeRepr) =
  typeRepr match
    case ref: TypeRef => ...
    case app: AppliedType => ...
    case or: OrType => ...
    ...
    case _ => throw Exception("Unsupported type")
```

`TypeRepr` is a class from the Scala compiler's reflection API that allows macros to work with the compiler's internal representation of types. In the above example, the code attempts to match on a given type. However, with explicit nulls, it does not have a corresponding case to handle a flexible type, and throws an error instead. Of course, not all projects that use `TypeRepr` fail by throwing an exception; some projects may have a default case that does not emit exceptions. In those projects, the failure may happen later down the line due to the flexible type not being processed properly. The fix for these projects would heavily depend on what the project was aiming to do with the type in

the first place. In most cases, flexible types can be treated as a special version of type bounds, so programs can recurse onto its base type or its upper bounds for subtyping related operations.

The other 3 failing projects come from an edge case interaction between nullifying Java symbols and the Unit type. In Scala, `Unit` is a special type that has only one value, '`()`'. When the compiler finds a block that expects a `Unit` return type but does not have `()` as its last expression, it will insert it automatically. We demonstrate the issue caused by explicit nulls in the following example:

```
public abstract class AbstractClass<T> {
  public abstract T foo();
}

val implementation = new AbstractClass[Unit] {
  override def foo() = {
    123
  }
}
```

In the Java file, we define an abstract class with a method `foo` that returns the generic type `T`. When we use the class in Scala, we instantiate the type with `Unit`. This results in the `foo` function having an expected return type of `(Unit)?`. The body of the function ends with a non-unit value, but the compiler does not insert `()` at the end automatically because the return type is not **exactly** `Unit`. As such, this results in a compilation error. This error can be fixed by either adding `()` at the end of the function body, or by typing the implementation of `foo` explicitly as `Unit`.

4.5 Summary

The plan for adoption of explicit nulls within the Scala ecosystem is to enable both explicit nulls and unsafe nulls in the next major compiler version. As shown in the Open Community Build, this change affects a very small minority of projects, at 0.83%. Additionally, those projects only need to make minor code adjustments to get the code to compile.

The benefit of this change will be to allow users to use the `safeNulls` import with greatly increased granularity compared to the all-or-nothing global type system change of explicit nulls. As shown in the compiler Community Build, with the assistance of compatibility features in flow typing and flexible types, a significant amount of effort can be saved when attempting to port code to the new system. Flexible types are 3.7× more impactful than flow typing in terms of saving work, but they both save work on different

code patterns. Flexible types save effort when interoperating with Java, and flow typing reduces the need for casting when working with nullable idioms.

Should the developer of an existing library want the benefits of the new system, but without porting the existing code, they can just selectively enable `safeNulls` in new files they create. Their new code will seamlessly work with unported files, while still benefiting from the null-safe type system.

Overall, the results of the Community Builds show that the explicit null system and the compatibility features have reached a stable and usable state.

Chapter 5

Related Work

In this chapter, we will list prior works that are related to this thesis and compare and contrast it with our work.

5.1 Null Safety in Other Languages

Scala is not the only language to have to deal with nullability. In this section, we survey other programming languages and show how they handle nullability compared to Scala.

5.1.1 Kotlin

Kotlin is an important comparison candidate to Scala. They both compile to the JVM. As such, Kotlin also makes a concerted effort to eliminate nullability errors [5].

Unlike Scala, union types are not natively supported in Kotlin. To denote a nullable type, the user adds a `?` after the type, as in `Type?`. Attempting to access values and methods on a nullable variable will cause a compilation error.

Kotlin also has flow sensitive typing, which they call smart casts [9]. Nullable variables are cast to their non-null equivalents after logical checks in the program. However, Kotlin does not handle mutable class fields in smart casts. To use a nullable mutable class field, the user needs to put it in a local val copy. Scala provides the `@stableNull` annotation in these cases to save effort.

Kotlin's equivalent to the `.nn` cast is the not-null assertion operator (`!!`), which allows the programmer to call methods on a nullable value, but throws an exception if it is null. In addition to this operator, Kotlin also provides the safe call operator (`?.`), which returns null if called on a null value, and the Elvis operator (`?:`), which allows the programmer to provide an alternative value to the expression if the selection target is null. However,

Kotlin does not provide a variant of the `!!` operator that does not throw an exception when casting, whereas Scala provides `.uncheckedNN`.

Being a JVM based language that also values Java interop, Kotlin also needs to deal with nullability from Java. Kotlin uses platform types [1][20], which are similar to Scala's flexible types. Kotlin is stricter than Scala regarding runtime checks. Assigning a null value that has a platform type to a variable that has a non-null type will throw a `NullPointerException`; this prevents variables that are typed as non-null from ever holding null. In Scala, the assignment is allowed, and an exception is only thrown when the program attempts to invoke a function on the null value.

Kotlin was designed with explicit nulls in mind from the beginning, whereas Scala only began adopting the explicit system well into development. This means that Kotlin faces less difficulty in adoption and compatibility with old code. In addition to improving null safety, Scala's switch to the explicit system also needs to focus on ease of migration for old Scala code.

5.1.2 Java

It is also worth discussing the efforts to deal with nullability in Java itself, since it is the source of Scala's interop issue.

Retrofitting non-null types onto existing object-oriented languages has a long history. Fähndrich and Leino [18] showed how to add non-null types to a Java-like language, identifying object initialization as the key difficulty. Chalin and James [15] found in an empirical study that on average 75% of reference declarations in real Java programs are intended to be non-null by design, motivating the non-null-by-default semantics that Scala's explicit nulls system also adopts. Ekman and Hedin [17] implemented non-null checking and inference as a modular extension to an extensible Java compiler.

Java itself does not enforce nullability at the type level; instead, tools like the Checker Framework [25], NullAway [11], and IDE inspectors read `@Nullable`/`@NonNull` annotations as conventions. These are not part of the type system, so they provide no compile-time guarantee from the compiler itself. Scala explicitly reads these annotations (as described in section 3.3) and uses them to assign concrete types instead of flexible types, giving null safety for well-annotated Java libraries. JSpecify [7] is an ongoing collaborative effort to standardize Java nullability annotations. As it matures, Scala's support and interop for Java will also benefit.

5.1.3 TypeScript

TypeScript is a transpiled language that adds types to JavaScript [12]. Notably, its type system is deliberately unsound in places, trading soundness for ease of adoption by existing JavaScript codebases [12] – a trade-off similar in spirit to Scala's flexible types. To enforce

null safety, it provides the `strictNullChecks` option [8]. Similar to Scala, nullable types are declared through a union with the null type. TypeScript also implements flow typing to allow for narrowing nullable variables. Users can also assert a variable to be non-null through the postfix `!` operator. This operator does not perform a runtime check, similar to Scala's `uncheckedNN`.

TypeScript is also able to interoperate with JavaScript. However, JavaScript is untyped, which means TypeScript can only infer types to a limited degree. First, the compiler looks for a `.d.ts` file that gives the types to the JavaScript file. Then the compiler tries to infer types from the source code itself, either through literals or nullability annotations. In the worst case, TypeScript needs to fall back to just assigning the `any` type to variables coming from JavaScript. Users can choose to have the compiler warn about this behavior by enabling the `noImplicitAny` flag.

TypeScript currently does not support scoped adoption of the strict null system. This means that if some dependency of the project does not compile under the strict rules, it will be difficult for the user to use the feature. In contrast, Scala has the `unsafeNulls` and `safeNulls` imports that allow for fine grained control of null-related type checking, which encourages adoption.

5.1.4 C#

C# 8.0 introduced nullable reference types, using a `string?` syntax similar to Kotlin's approach [6]. C# also provides flow typing on nullable variables through explicit checks, assignments and assertions like Scala.

Because nullable reference types were introduced later in development, C# also allows gradual adoption by enabling nullable reference types with `#nullable enable` at the file or project level. Furthermore, the language allows the user to enable the null safety features separately. The user can choose to enable nullable types first, and not have the compiler warn about unsafe accesses, or they can choose to enable warnings for unsafe accesses, and have the compiler assume all reference types are nullable. This allows users to split their migration into two more manageable parts.

When the outward facing API of the library is the focus, the user should choose to enable nullable types first to stabilize the nullability contract of the API. Because warnings are off, the user can defer fixing the implementation until the API is settled.

When null safety is the focus, the user can choose to enable warnings first to have the compiler strictly infer all reference types to be nullable, eliminating the possibility of `NullReferenceException`s as much as possible. Afterwards, the user can switch on the type system to annotate the actual nullable types as such.

5.1.5 Dart

Dart introduced sound null safety in version 2.12 [10]. Like Kotlin and C#, reference types are non-null by default, and a nullable type is written with a trailing `?`, as in `String?`. Dart performs flow analysis to promote a nullable variable to its non-null type after a check, and it provides the same family of null-aware operators seen in the other languages: the null assertion operator (`!`), which throws if the value is null, and the safe call (`?.`) and if-null (`??`) operators. Dart also offers a `late` modifier for non-nullable fields whose initialization is deferred, checked with a runtime guard on first read.

Unlike TypeScript, Dart's null safety is sound: it combines static checking with the runtime type checks the language already performs, so a non-null type is guaranteed never to hold null. Consequently, Dart's `!` operator always performs a runtime check and has no unchecked counterpart analogous to Scala's `uncheckedNN`.

Dart is closest to Scala in that it retrofitted null safety onto an existing language with a large ecosystem rather than being designed with it from the start. Migration is controlled at the granularity of a library through its language version, and during the transition Dart could execute mixed-mode programs that combine migrated and unmigrated libraries under an unsound null safety mode, until the whole program had opted in [4]. A `dart migrate` tool automated much of the annotation effort by inferring where `?` should be added. This library-level opt-in is coarser than Scala's `unsafeNulls` and `safeNulls` imports, which operate at the file and block level, but it plays the same role in easing incremental adoption.

5.1.6 C++

C++ does not have an explicitly nullable type system, but still has null pointers present in the language. There are two ways that programmers can manage nullability in C++. The first method is to use references, which are guaranteed to refer to a valid, existing object. The second method is to use const pointers, which eliminate the possibility of mutation after checking, removing the risk of dereferencing a null pointer.

5.1.7 Comparison

Table 5.1 compares the nullability features provided by the languages. It should be noted that Scala sits at a unique intersection where it needs to cater to both interoperation with a less null-safe language and also incremental adoption from an implicitly nullable type system in a previous iteration of the language.

Table 5.1: Table comparing null compatibility features of languages

	Scala	Kotlin	Java	TypeScript	C#	Dart
Flow Typing	✓	✓	✓	✓	✓	✓
Interoperation	✓	✓		✓		
Incremental Adoption	✓		✓		✓	✓

5.2 Pluggable Type Systems

Pluggable type systems [13] are highly related to our work. They are an alternative to typical mandatory type systems. Compared to typical type systems, the distinguishing feature of pluggable type systems is that they can be enabled or disabled in addition to the base type system when needed. This allows for greater expressivity in the base language, free from the mandatory constraints. Related to the previous section, there are many pluggable type systems for Java that focus on improving null safety. We list relevant projects below.

5.2.1 Checker Framework

The Checker Framework [25] is a popular pluggable type system for Java that allows third parties to write their own checkers. One of the checkers that comes pre-packaged with the Checker Framework is the Nullness Checker. The Nullness Checker is sound in the sense that if it issues no warnings for a given program, then running that program will never throw a null pointer exception [3]. The Nullness Checker utilizes four key annotations:

- `@NonNull` – applied to a member field's type, indicates the field is never null after instantiation completes; applied to a static field, indicates it is never null after the containing class is initialized. It is rarely written explicitly because it is the default in most positions.
- `@Nullable` – a type annotation that makes no commitments about whether the value is null – equivalently, the type includes the null value. The checker issues an error if null is assigned to an expression of `@NonNull` type.
- `@PolyNull` – a polymorphic qualifier for methods whose nullness behavior depends on their arguments.
- `@MonotonicNonNull` – for fields that start out possibly-null but, once set, are never reset to null again. Because the checker works intraprocedurally, when such a field is first read within a method, it cannot be assumed non-null, but after a check within that method, subsequent accesses can be treated as `@NonNull`, even after external method calls.

When enabling the Nullness Checker, the default behavior assumes all variables are non-null. Most of the migration effort will be on identifying the variables that are actually null and annotating them with `@Nullable`. This default is well supported empirically: Chalin and James [15] found that roughly three quarters of reference declarations in real Java code are intended to be non-null. Dietl et al. [16] report on the practical experience of building and deploying checkers with the framework on over two million lines of code, finding hundreds of real bugs, and observe that the annotation burden is the main cost of adoption – the same cost that Scala's migration features aim to reduce.

5.2.2 NullAway

NullAway is a type-based null-safety checker for Java built as a plugin to Error Prone, a static analysis tool for Java [11]. Users found that strict checkers like the Nullness Checker were difficult to deploy on large-scale software projects due to significant build-time overhead or a high annotation burden. NullAway was designed to sacrifice total soundness for reduced overhead. Evaluation of NullAway showed a significantly lower build-time overhead of 1.15x compared to the Nullness Checker at 2.8x to 5.1x [11].

5.3 Gradual Typing

Gradual typing is a type system approach that allows for both static typing and dynamic typing but enforces types at runtime, letting a single language mix statically-checked and dynamically-checked code and letting the programmer choose which system applies to which part of the program [27][26].

The migration problem that gradual typing addresses is closely related to ours. Typed Scheme [29] was designed explicitly to support the gradual migration of untyped Scheme programs to a typed sister language, module by module – analogous to how Scala's `unsafeNulls` allows migrating a codebase to explicit nulls file by file. However, the runtime enforcement that sound gradual typing requires comes at a cost: Takikawa et al. [28] showed that in Typed Racket, partially migrated configurations can suffer slowdowns of an order of magnitude or more due to boundary checks. Scala's explicit nulls system avoids this cost by keeping enforcement purely static; flexible types insert no runtime checks, at the price of weaker guarantees at the Java boundary.

The theoretical underpinning of this trade-off has also been studied for nullability specifically. Nieto et al. [21] developed a blame calculus for explicit nulls, modeling the interaction between explicitly nullable and implicitly nullable code, and proved that "explicitly nullable programs can't be blamed": nullability errors at the boundary originate from the implicitly nullable side.

5.3.1 Granullar

The Granullar project extends a static pluggable type system that enforces null-safety with a gradual type system that inserts nullness runtime checks to safely interact with unchecked software components [14].

Whereas pluggable systems described earlier are concerned with completely checked applications, Granullar further improves upon the concept by focusing on the interaction boundary between the checked and unchecked code. Previous work handles unchecked code by using defaults, but even conservative typing does not fully guarantee safety. Granullar inserts runtime checks at the program boundary to prevent unsound nullable values from polluting checked code.

5.4 Summary

The survey of related work across languages reveals key similarities. Flow-sensitive typing is present across all null-safe languages surveyed. Kotlin's smart casts, TypeScript's control flow analysis and C#'s nullable analysis all cover similar use cases. This shows that in most nullable languages, correct code already performs explicit checks before accessing nullable variables. The implementation of flow sensitive typing helps reduce the programmer's need to cast variables after null checks.

Interoperation with a less null-safe language is a challenge unique to Scala, Kotlin and TypeScript among the languages surveyed. Scala's flexible types and Kotlin's platform types are both deliberately unsound features that sacrifice correctness for convenience, whereas TypeScript must go even further and assign the top type due to the limitations of JavaScript, the underlying language.

Incremental adoption is a concern that the surveyed languages handle with varying degrees of granularity. Kotlin, having been designed with explicit nulls from the start, faces the least migration burden. C# addresses it through a two-phase project-level switch separating type annotation from warning enforcement. TypeScript's `strictNullChecks` is all-or-nothing at the project boundary. Dart, which like Scala retrofitted null safety onto an existing ecosystem, allows opt-in at the granularity of a library and supports mixed-mode execution during migration. Scala's `unsafeNulls` and `safeNulls` imports operate at a finer granularity than any of these, allowing adoption to be controlled at the file and block level within the same codebase. Granullar's approach of inserting runtime checks at the boundary between checked and unchecked code represents a complementary strategy, though it relies on dynamic enforcement where Scala's explicit nulls targets static guarantees.

Taken together, Scala's explicit nulls system occupies a distinctive position in this space. It is the only surveyed approach that simultaneously addresses retrofitting null safety onto an existing language with a large codebase, interoperation with Java through

flexible types, and fine-grained incremental adoption through scoped null safety imports. The evaluation in Chapter 4 examines how well these mechanisms hold up against real-world Scala projects.

Chapter 6

Conclusion

Null pointer exceptions remain a pervasive source of runtime failures in languages that inherit Java's implicit nullability. This thesis has examined Scala's transition from an implicitly nullable type system to an explicitly nullable one, focusing on the compatibility features that make this transition practical for real-world projects. The central challenge is to adopt the new explicit system in a way that does not abandon the existing ecosystem of Scala and Java libraries that rely on implicit nullability.

6.1 Summary of Contributions

We implemented and evaluated four compatibility features within the Scala compiler to ease the migration to explicit nulls.

The **non-null cast** (`.nn`) provides a safe, checked utility for asserting that a nullable value is non-null at a specific use site. Its return type `x.type & T` is designed to preserve path-dependent type information, avoiding the type errors that arise with a naive cast to `T`. The companion `uncheckedNN` provides an unchecked variant for performance-sensitive contexts.

Flow typing tracks the nullability effects of each block of code through a pair of asserted and retracted variable sets, composed using the `seq` and `alt` operators. The `<Abrupt>` sentinel allows the analysis to correctly handle early-exit patterns in conditionals and try-catch-finally blocks. The `@stableNull` annotation extends flow typing to mutable class fields when the programmer can guarantee stability.

Flexible types address the fundamental interoperability problem with Java's invariant type constructors. A flexible type `(T)?` with bounds `T | Null <: (T)? <: T` allows a Java-sourced value to be used as either nullable or non-nullable, satisfying both denotations of invariant containers such as arrays. Additionally, the compiler reads `@Nullable` and `@NonNull` Java annotations to assign concrete types in place of flexible types where annotations are present.

The **unsafe nulls** language feature provides the biggest escape hatch: within the scope of `import scala.language.unsafeNulls`, the compiler relaxes null-related type checks to match the behavior of the implicit system. Its counterpart `safeNulls` re-enables checking within a nested scope. Together they allow null safety to be controlled at the project, file, and block level, giving programmers precise control over the migration boundary.

6.2 Summary of Findings

We evaluated these features against two sets of Scala projects. On the Community Build of 40 projects, 15 projects required no changes whatsoever to compile under explicit nulls. Of those that did require changes, the most common modifications were widening type declarations to include `| Null` (45% of changes) and inserting `.nn` assertions (23%).

The ablation study showed that disabling flow typing caused 424 additional compilation errors across 17 projects, while disabling flexible types caused 1,579 errors across 31 projects — a 3.7× difference. Crucially, these error sets are orthogonal: flow typing eliminates errors arising from explicit null checks on Scala variables, while flexible types eliminate errors arising from Java interoperation. Both features are necessary, and neither is a substitute for the other.

On the Open Community Build of 1,928 projects, enabling both explicit nulls and unsafe nulls globally resulted in only 16 projects failing to compile — a compatibility rate of 99.17%. The failures fell into two categories: macro-related code that lacks pattern match cases for flexible types, and an edge case where a type parameter instantiated with `Unit` gets nullified to `(Unit)?` suppressing the compiler's automatic insertion of `()`. Both are narrow, well-understood failure modes rather than fundamental limitations of the approach.

Taken together, these results demonstrate that explicit nulls, combined with the compatibility features described in this thesis, has reached a level of maturity suitable for broader adoption. The planned next step — enabling explicit nulls and unsafe nulls together by default in a future major compiler version — is supported by the Open Community Build results.

References

[1] Calling Java from Kotlin | Kotlin. Publication Title: Kotlin Help. URL: https://kotlinlang.org/docs/java-interop.html [accessed 2026-07-10].

[2] Chapter 14. Blocks and Statements. URL: https://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html [accessed 2026-07-13].

[3] The Checker Framework Manual: Custom pluggable types for Java. URL: https://checkerframework.org/manual/#nullness-checker [accessed 2026-07-11].

[4] Migrating to null safety | Dart. Publication Title: Dart. URL: https://dart.dev/null-safety/migration-guide [accessed 2026-08-05].

[5] Null safety | Kotlin. Publication Title: Kotlin Help. URL: https://kotlinlang.org/docs/null-safety.html [accessed 2026-07-10].

[6] Nullable reference types - C#. URL: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/null-safety/nullable-reference-types [accessed 2026-07-11].

[7] Nullness Specification | JSpecify. URL: https://jspecify.dev/docs/spec/ [accessed 2026-07-13].

[8] TSConfig Reference - Docs on every TSConfig option. URL: https://www.typescriptlang.org/tsconfig/#strictNullChecks [accessed 2026-07-11].

[9] Type checks and casts | Kotlin. Publication Title: Kotlin Help. URL: https://kotlinlang.org/docs/typecasts.html [accessed 2026-07-10].

[10] Understanding null safety | Dart. Publication Title: Dart. URL: https://dart.dev/null-safety/understanding-null-safety [accessed 2026-08-05].

[11] 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*, ESEC/FSE 2019, pages 740–750, New York, NY, USA, August 2019. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3338906.3338919 [accessed 2026-07-10].

[12] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 257–281, Berlin, Heidelberg, 2014. Springer.

[13] Gilad Bracha. Pluggable Type Systems. January 2004.

[14] Dan Brotherston, Werner Dietl, and Ondřej Lhoták. Granullar: gradual nullable types for Java. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, pages 87–97, New York, NY, USA, February 2017. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3033019.3033032 [accessed 2026-07-10].

[15] Patrice Chalin and Perry R. James. Non-null References by Default in Java: Alleviating the Nullity Annotation Burden. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, pages 227–247, Berlin, Heidelberg, 2007. Springer.

[16] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muşlu, and Todd W. Schiller. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 681–690, New York, NY, USA, 2011. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/1985793.1985889.

[17] Torbjörn Ekman and Görel Hedin. Pluggable checking and inferencing of non-null types for Java. *Journal of Object Technology*, 6(9):455–475, October 2007. URL: https://www.jot.fm/issues/issue_2007_10/paper23.pdf.

[18] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 302–312, New York, NY, USA, 2003. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/949305.949332.

[19] C. A. R. Hoare. Null References: The Billion Dollar Mistake, March 2009. URL: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/.

[20] Java. JVMLS 2015 - Flexible Types in Kotlin, August 2015. URL: https://www.youtube.com/watch?v=2IhT8HACc2E [accessed 2026-07-13].

[21] Abel Nieto, Marianna Rapoport, Gregor Richards, and Ondřej Lhoták. Blame for Null. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:28, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2020.3.

[22] Abel Nieto, Yaoyu Zhao, Ondřej Lhoták, Angela Chang, and Justin Pu. Scala with Explicit Nulls. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:26, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2020.25 [accessed 2026-07-08].

[23] Martin Odersky and Tiark Rompf. Unifying functional and object-oriented programming with Scala. *Communications of the ACM*, 57(4):76–86, April 2014. URL: https://dl.acm.org/doi/10.1145/2591013.

[24] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, page 41–53, New York, NY, USA, 2001. Association for Computing Machinery. URL: https://doi.org/10.1145/360204.360207.

[25] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 201–212, New York, NY, USA, July 2008. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/1390630.1390656 [accessed 2026-07-10].

[26] Jeremy Siek and Walid Taha. Gradual Typing for Objects. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, pages 2–27, Berlin, Heidelberg, 2007. Springer.

[27] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Proceedings of the Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

[28] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 456–468, New York, NY, USA, 2016. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/2837614.2837630.

[29] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 395–406, New York, NY, USA, 2008. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/1328438.1328486.