Scala with Explicit Nulls

by

Abel Nieto Rodriguez

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, 2019

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

The Scala programming language unifies the object-oriented and functional styles of programming. One common source of errors in Scala programs is `null` references. In this dissertation, I present a modification to the Scala type system that makes nullability explicit in the types. This allows us to turn runtime errors into compile-time errors. I have implemented this design for explicit nulls as a fork of the Dotty (Scala 3) compiler. I evaluate the design by migrating a number of Scala libraries to use explicit nulls.

In the second part of the dissertation, I give a theoretical foundation for explicit nulls. I do this in two, independent ways. First, I give a denotational semantics for type nullification, a key part of the explicit nulls design. Separately, I present a core calculus for null interoperability that models how languages with explicit `nulls` (like Scala) interact with languages where `null` remains implicit (like Java). Using the concept of blame from gradual typing, I show that if a well-typed program fails with certain kinds of nullability errors, an implicitly-nullable subterm can always be blamed for the failure.

Acknowledgements

I would like to thank my advisor, Ondřej Lhoták, for his prescient advice on the projects that underlie this thesis, both with "big picture" items and thorny technical details. I would also like to thank the members of my thesis committee, Prabhakar Ragde and Gregor Richards, for their thoughtful comments.

The implementation of Scala with explicit nulls described in this thesis would not have been possible without the help of our undergraduate research assistants. Yaoyu Zhao contributed many new features and bug fixes to our implementation, making it more robust and usable. Angela Chang and Justin Pu carried out most of the migration work described in the empirical evaluation section.

The explicit nulls project also benefited tremendously from the feedback and guidance of members of the LAMP team at EPFL. In particular, I want to thank Martin Odersky, Guillaume Martres, and Fengyun Liu.

Gregor Richards provided valuable advice on how the calculus for null interoperability relates to the larger field of gradual typing. Marianna Rapoport made possible the mechanization of the blame-related proofs, by sharing her wealth of knowledge about Coq and setting up the entire proof infrastructure using Ott and LNGen.

Even though the work I carried out at Microsoft Research during the summer of 2019 is not part of this thesis, I would like to thank my then-manager, David Tarditi, for his mentorship during the internship. David remains a role model for how a senior researcher can effectively lead a research or product team.

The work described in this thesis was supported by the Natural Sciences and Engineering Research Council of Canada.

Dedication

The past September marked ten years since I moved to Canada to start my undergraduate studies. This has turned out to be probably the most consequential event in my life, and would not have been possible without the help of my parents. I thank them for this, and for their lifelong support and love.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Scala is a general-purpose object-oriented and functional programming language [EPFL]. Initially released in 2004, the language has seen both wide industry adoption, as well as interest from academia.

Scala probably owes a large part of its industry success to the idea that Scala is a "better Java". Because Scala programs are compiled to Java Virtual Machine (JVM) bytecode, it is straightforward to use a Java library from Scala code. Additionally, Scala has a number of language features, mostly within its type system, that are not present in Java, but are popular among Scala practitioners. These features, like an expressive module system and type-level programming, turn certain classes of run-time errors into compile-time errors, arguably increasing the robustness of the resulting code. Measuring programming language "popularity" or "use" is a fuzzy undertaking, but the programming language rankings that we do have show Scala in positions 12 to 30 [TIOBE, RedMonk, PYPL] (the TIOBE index shows Scala right behind Fortran! [TIOBE]). Scala's main bi-annual conference, Scala Days, started in 2010 with 150 attendees, and has grown to around 1500 participants for its 2019 edition [ScalaDays].

On the academic side, Scala is interesting mainly because it has a rich type system with unique challenges. These challenges come out of Scala's desire to unify object-oriented and functional programming. The most prominent example of academic work inspired by Scala is probably the *Dependent Object Types* (DOT) calculus, which serves as the theoretical foundation for the language [Amin et al., 2016]. There is a long line of papers studying properties of DOT [Amin et al., 2016, Amin and Rompf, 2017, Rapoport et al., 2017, Wang and Rompf, 2017, Kabir and Lhoták, 2018, Rapoport and Lhoták, 2019, Hu and Lhoták, 2019]. There also has been work on Scala's module system, as well as its "implicits"

mechanism, among other features [Odersky and Zenger, 2005, Odersky et al., 2017].

1.1 The `Null` Problem

Scala inherited elements of good design from Java, but it also inherited at least one misfeature: the `null` reference. In Scala, like in many other object-oriented programming languages, the `null` reference can be typed with *any reference type*. This leads to runtime errors, because `null` does not (and cannot) support almost *any* operations. For example, the program below tries to read the `length` field of a string, only to find out that the underlying reference is `null`. The program then terminates by reporting the infamous `NullPointerException`[1].

```
val s: String = null
println(s"s has length ${s.length}") // throws a NullPointerException
```

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

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

```
val s: String = null // type-error: 'Null' is not a subtype of 'String'.
val s2: String|Null = null // ok: s is explicitly marked as nullable.
if (s2 != null) {
  println(s"s has length ${s.length}") // ok: we checked that s2 is not null.
}
```

In a world with explicit nulls, the type system can keep track of which variables are potentially `null`, and prevent us from making unsafe dereferences. In the last example, we are forced to check that `s2` is not null before accessing it.

[1] https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html

[2] Reachable at https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=NULL+Pointer+Dereference.

In this dissertation, I describe and study a type system just like the one above. It is my hope that this version of Scala with explicit nulls can contribute to the creation of robust software with fewer runtime errors.

1.2 My Thesis

My thesis is that

> we can retrofit the Scala language with explicit nulls. The resulting type system can be formally studied and understood.

This dissertation substantiates the claims above:

1. In Chapter 2, I present a design for a version of Scala with explicit nulls. I have implemented this design as a modification to Dotty, a new compiler for the upcoming version 3 of Scala. The result is a system where many (but not all) nullability-related errors can be detected at compile-time. The changes to the compiler are currently being considered for inclusion in the official compiler codebase. Finally, I evaluated the implementation by migrating to explicit nulls a set of 13 libraries totalling over 90,000 lines of Scala code.

2. In Chapter 3, I look for a formal model of *type nullification*, a key part of the explicit nulls design. Using tools from *denotational semantics*, I show how nullification can be understood as an element-preserving transformation on types.

3. In Chapter 4, I define a core calculus for interoperability between languages with explicit nulls and languages with implicit nulls. This is relevant in the Scala context, because Scala code often interacts with Java code, where `nulls` remain implicit. I characterize the nullability errors that can occur in this calculus using the concept of *blame*, from *gradual typing*. Finally, I show that if a program in this calculus is well-typed, then the *implicitly typed* terms are responsible for certain kinds of evaluation errors. Most of the proofs in this chapter have been mechanized in the Coq proof assistant.

```
class Person(name: String) {
  println(s"created person ${toString()}")

  override def toString(): String = name
}

class Employee(name: String) extends Person(name) {
  var company: String = ""

  def setCompany(company: String): Unit = this.company = company

  override def toString(): String =
    if (company.isEmpty) name else s"${name}@${company}" // 'company' can be uninitialized
}

new Employee("Jane") // throws a 'NullPointerException'
```

Figure 1.1: Uninitialized fields can cause nullability errors.

1.3 Future Work

There is a second class of problems related to nullability, different from the type hierarchy issue we saw before. This problem has to do with initialization, and can be summarized as saying that, even in a type system with explicit nulls, nullability errors are possible in the presence of uninitialized class fields. The example in Figure 1.1 illustrates this situation. We have an `Employee` class that extends (inherits) `Person`. `Employee` has a `company` field, which is kept mutable, because an employee can change employers over time. `company` is initialized to the empty string. However, when a new `Employee` is created, `Employee`'s constructor calls `Person`'s constructor, which in turn calls `toString`. Due to dynamic dispatch, `Employee`'s `toString` is called, at which point a field selection on `company` happens. Because all the above events take place during object construction, the statement initializing `company` has not executed yet, and so `company` is `null` at this point (`null` is the default value for uninitialized fields of reference types in Scala). The expression `company.isEmpty` then generates a `NullPointerException`, even though `null` does not appear in the program, and all fields would seem to be initialized.

The problem of guaranteeing sound initialization is an interesting and challenging one, and Section 2.8.3 surveys some of the existing work in this area. It is, however, not a

problem I tackle in this dissertation. Designing a sound initialization system for Scala remains future work.

Chapter 2

Scala With Explicit Nulls

In this chapter, I describe our design for retrofitting Scala's type system with a mechanism for tracking nullability. The main idea is to make all reference types non-nullable by default. Nullability can then be recovered using union types. So that Scala programs can interoperate with Java code, where `nulls` remain implicit, I present a type nullification function that turns Java types into equivalent Scala types. To improve usability of nullable values in Scala code, I have also added a simple form of flow typing to Scala. I have implemented this design on top of the Dotty (Scala 3) compiler. Finally, I present an evaluation of the design by migrating multiple Scala libraries into the world of explicit nulls.

2.1 The Dotty Compiler

We start this chapter with a brief overview of the Dotty compiler, focusing on the parts of Dotty that are relevant to the explicit nulls project.

Dotty is a research compiler, in development since at least 2013 at École Polytechnique Fédérale de Lausanne (EPFL) [Dotty Team], that will become the reference implementation for the upcoming Scala 3 language (scheduled for release in 2020). The goals of the Dotty project are dual: Dotty will serve as a re-architected version of the current `scalac` compiler (one that is more modular and maintainable); additionally, Dotty is a research platform for designing and trialling experimental Scala features.

Because of these reasons, we decided to implement our explicit nulls project by modifying Dotty. Our current implementation of explicit nulls is being considered for inclu-

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

// Phases dealing with the frontend up to trees ready for TASTY pickling
protected def frontendPhases: List[List[Phase]] =
  // Compiler frontend: scanner, parser, namer, typer
  List(new FrontEnd) ::
  // Sends information on classes' dependencies to sbt via callbacks
  List(new sbt.ExtractDependencies) ::
  // Additional checks and cleanups after type checking
  List(new PostTyper) ::
  // Sends a representation of the API of classes to sbt via callbacks
  List(new sbt.ExtractAPI) ::
  // Set the 'rootTreeOrProvider' on class symbols
  List(new SetRootTree) ::
  Nil
```

Figure 2.1: High-level structure of the Dotty compiler, showing the frontend phases

sion in the official Dotty distribution. The code can be found as a git "pull request" at https://github.com/lampepfl/dotty/pull/6344. As of November 2019, the changes to the compiler in the pull request involved 3627 new lines of code (LOC) spread over 127 files.

2.1.1 Structure of Dotty

The high-level structure of Dotty is shown in Figure 2.1. As usual, the compiler consists of a list of independent *phases* that create and transform abstract syntax tree (AST) nodes [Petrashko et al., 2017]. Most changes required by our design happen in the `FrontEnd` phase, which is in charge of parsing and type-checking.

2.1.2 Types

Scala has a rich type system that includes features from functional and object-oriented programming. On its functional side, Scala includes algebraic data types (ADTs), pattern matching, local type inference, parametric polymorphism (generics), and higher-kinded

types, among others. From the object-oriented world, Scala includes classes, nominal subtyping via inheritance, and definition and use site variance, among others. The reason for listing all these features is to highlight that any changes to the type system (such as those required by explicit nulls) need to interoperate with a potentially very large surface of already existing concepts.

Below I highlight some Scala types that are particularly relevant to explicit nulls.

Union Types

Union types [Pierce, 2002] are handled by Dotty and are not currently available in Scala 2. Informally, given types A and B, the union type $A|B$ contains all the "elements" of A and B. The following programs are then type-correct:

```
val u1: String | Int = "hello"
val u2: String | Int = 42
```

Notice that, unlike ADTs, union types do not have to be *disjoint* (e.g. `String|Object`). Union types can be deconstructed via runtime checks; for example, using pattern matching:

```
val u: String | Int = "hello"

val i: Int = u match {
  case s: String => i.length
  case i2: Int => i2
}
```

After pattern matching is desugared, a test like `case s: String` becomes an explicit runtime tag check that looks like `if u.isInstanceOf[String]`.

Union types are key for our design of explicit nulls, because they allow us to represent nullable types. For example, a local `val` containing a string will have type `String|Null`.

Path-Dependent Types

Path-dependent types are a limited form of dependent types that is unique to Scala. A path-dependent type (of length one) has form

$$a.T$$

where a is a `val`, and T is a *type member* (a type member is like a class field, but it declares a type as opposed to a value). Path-dependent types of length greater than one are also possible; for example, the type $a.b.c.T$ has length three. In this case, we say that $a.b.c$ is a *path*. b and c are class fields. The following example shows that path-dependent types depend on values:

```
class Tree {
  type Node
  val root: Node
}

// Is `n1` an ancestor of `n2` in tree `t`?
def isAncestor(t: Tree, n1: t.Node, n2: t.Node): Boolean

val t1: Tree
val t2: Tree

// Type error: expected `t1.Node`, but got `t2.Node`
isAncestor(t1, t1.root, t2.root)
```

We declare a `Tree` class with a `root` field and a `Node` type member. The type of `root` is `Node`. Next, we declare an `isAncestor` method that checks whether a node is an ancestor of another node. Notice, however, that the type of `n1` and `n2` is *not* `Node`, but `t.Node`, which is a path-dependent type. This path-dependent type guarantees that only nodes that belong to tree `t` can be passed into `isAncestor`. In particular, when we later try to check whether a node of `t1` is an ancestor of a node from a different tree `t2`, the compiler issues a type error, because `t1.Node` and `t2.Node` are different, incompatible types.

The *paths* in path-dependent types are relevant to explicit nulls; in particular, we only do flow typing on *stable* paths. See Section 2.4 for details.

The theoretical foundations of path-dependent types are explored in the Dependent Object Types (DOT) calculus, developed in Amin et al. [2016], Amin and Rompf [2017], Rapoport et al. [2017], Rapoport and Lhoták [2019], among others.

2.1.3 Denotations

Denotations are a Dotty abstraction to express the *meaning* of path-dependent types (and symbols). Consider the following example:

```
object x {
  object y {
    type T = List[String]
  }
}

val z: x.y.T
```

Because types are immutable in Dotty, the type of `z` is the path-dependent type `x.y.T` throughout the compilation process. However, if we want to typecheck a field selection `z.length`, we need a way to go from the path-dependent type to its "underlying" type; that is, to its denotation. In this case, we would say that the denotation of `x.y.T` is `List[String]`, a type application. The reality is a bit more complicated, because denotations, unlike types, are mutable and change over time. For example, after erasure, the denotation of `x.y.T` becomes `List`, since erasure eliminates generics.

Denotations play an important role in our implementation of flow typing.

2.1.4 Symbols

Symbols are a Dotty abstraction for definitions. Alternatively, we can think of symbols as referring to concrete locations in source files. All of the following have associated symbols: classes, fields, methods and their arguments, and local variables. In Scala, there are four main kinds of definitions, each with its own evaluation strategy:

- `vals` are evaluated just once, when defined. Additionally, `vals` are immutable.

  ```
  val x: String = {
    println("eval x") // prints "eval x" once, while `x` is initialized
    "hello"
  }
  x = "world" // error, cannot assign to a val
  ```

- `vars` are like `vals`, but mutable.

  ```
  var x: String = {
    println("eval x") // prints "eval x" once
    "hello"
  }
  x = "world" // ok, vars are mutable
  ```

- `defs` are lazily evaluated every time they are referenced (and `defs` can take arguments, so they can define methods).

```
def x: String = {
  println("eval x")
  "hello"
}
x // prints "eval x"
x // prints "eval x" again
```

- `lazy vals` are lazily evaluated, but then memoized (i.e. they are evaluated at most once). They have a combination of `val` and `def` semantics.

```
lazy val x: String = {
  println("eval x")
  "hello"
}
x // prints "eval x"
x // 'x' is memoized, so this statement does not print "eval x" again
```

The distinction between `vals`, `vars`, and `defs` is relevant to flow typing, since we only infer the types of `vals` and `lazy vals` (only these two classes of symbols can be part of stable paths).

Implicits

Some definitions (e.g. `vals` and method parameters) can be annotated as `implicit`. The `implicit` keyword is overloaded to have different meanings, but one use case is that the compiler will fill in implicit definitions for implicit parameters. For example, we can create a `Context` object that contains state, mark it as `implicit`, and it will then be passed around automatically by the compiler:

```
case class Context(margin: Int)

def lenPlusMargin(name: String)(implicit ctx: Context): Int = {
  name.length + ctx.margin
}

implicit val ctx: Context(margin = 10)
```

```
val l = lenPlusMargin("hello") // desugared into `val l = lenPlusMargin("hello")(ctx)` by the compiler
assert(l == 15)
```

In this case, upon seeing the function call `lenPlusMargin("hello")`, the compiler will notice the missing implicit argument, and search for an implicit definition of type `Context`. It will then fill in `ctx` for the missing argument. This generates the call `lenPlusMargin("hello")(ctx)`.

The theoretical foundations of implicits are described in d. S. Oliveira et al. [2012] and Odersky et al. [2017], among others. Implicits are germane to explicit nulls because they inform our implementation of flow typing (specifically, flow typing within blocks of statements).

2.2 A New Type Hierarchy

To understand the special status of the `Null` type, we can inspect the current Scala type hierarchy, shown in Figure 2.2. Roughly, Scala types can be divided into *value* types (subtypes of `AnyVal`) and *reference* types (subtypes of `AnyRef`). The type `Any` then stands at the top of the hierarchy, and is a supertype of both `AnyVal` and `AnyRef` (in fact, a supertype of every other type). Conversely, `Nothing` is a subtype of all types. Finally, `Null` occupies an intermediate position: it is a subtype of all *reference* types, but not of the value types. This justifies the following typing judgments:

```
val s: String = null // ok, since String is a reference type
val i: Int = null    // error: expected a `Int` but got a `Null`
```

This is what makes `nulls` in Scala *implicit*. In order to make `nulls` *explicit*, we need to dislodge the `Null` type from its special position, so that it is no longer a subtype of all reference types. We achieve this by making `Null` a direct subtype of `Any`. This new type hierarchy, which underlies our design, is also shown in Figure 2.2.

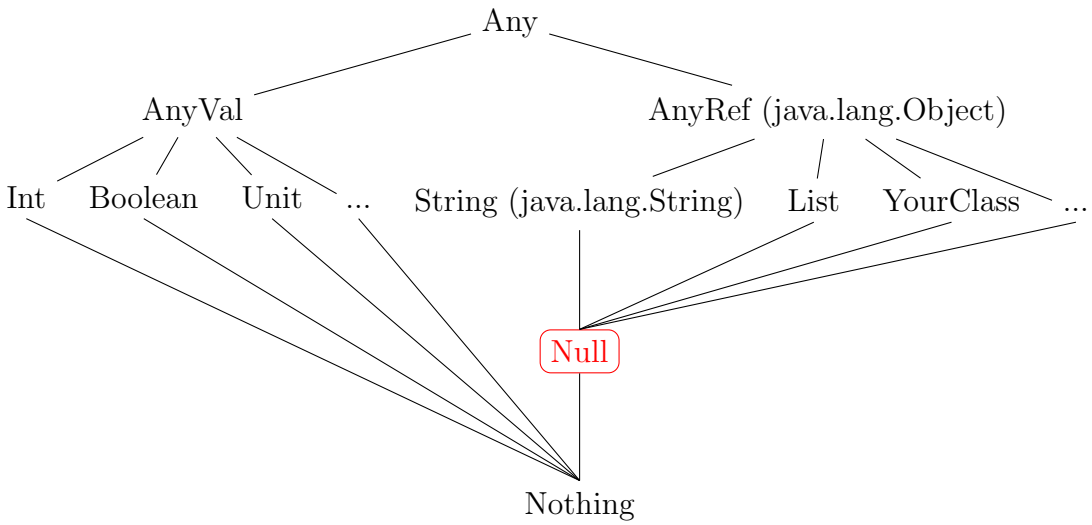
With the new type hierarchy we get new typing judgments:

```
val s: String = null // error: expected a `String` but got a `Null`
val i: Int = null    // error: expected a `Int` but got a `Null`
val sn: String|Null = null // ok: Null <: String|Null
```

We can still express nullable types using union types, as the last example shows.

This new type hierarchy is still unsound in the presence of uninitialized values:

Implicit nulls



Explicit nulls

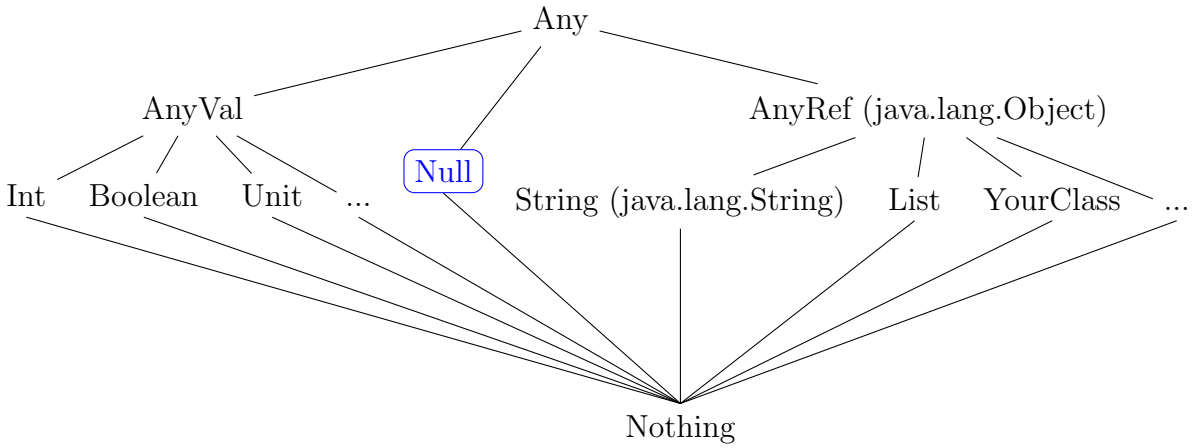


Figure 2.2: Alternative Scala type hierarchies with implicit and explicit nulls

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

Because after allocation the fields of Scala classes are initialized to their "default" values, and the default value for reference types is `null`, when we try to access `name.len` in line 3, `name` is `null`. This produces a `NullPointerException`.

We will accept this source of unsoundness for the purposes of the explicit nulls project. Developing a sound initialization scheme for Scala, while balancing soundness with expressivity, remains future work (also see existing approaches in Section 2.8).

2.2.1 Fixing a Soundness Hole

Even though explicit nulls do not make the Scala type system sound, they *do* remove one specific source of unsoundness that results from combining `null` with type members with *arbitrary lower and upper bounds*. An illustrative example, taken from Amin and Tate [2016], is shown in Figure 2.3. Here is how the unsoundness happens:

- The two traits, `LowerBound` and `UpperBound`, declare a type member `M` and give it an arbitrary lower bound `T` and upper bound `U`, respectively.
- It follows that if we can construct an element of type

  `LowerBound[T] with UpperBound[U]`

  (`with` here means type intersection), that element will have a type member `M`, where `T <: M <: U`. Since subtyping is not always transitive in Scala [Nieto, 2017], this does *not* quite imply that `T <: U`, but we *can* use the following two assignments to turn a `T` into a `U`:

  ```
  val t: T = ...
  val m: M = t // ok: since T <: M
  val u: U = m // ok: since M <: U
  ```

```
object unsound {
  trait LowerBound[T] {
    type M >: T
  }

  trait UpperBound[U] {
    type M <: U
  }

  def coerce[T, U](t: T): U = {
    def upcast(lb: LowerBound[T], t: T): lb.M = t
  val bounded : LowerBound[T] with UpperBound[U] = null
  upcast(bounded, t)
  }

  def main(args: Array[String]): Unit = {
    val zero = coerce[Int, String](0)
  }
}
```

Figure 2.3: The combination of `null` and type members can lead to unsoundness. Example taken from [Amin and Tate, 2016].

- Now notice that `T` and `U` are completely *abstract*, so we could instantiate them to *any* two types, even ones that are not related by subtyping. Example 2.3 sets `T = Int` and `U = String`. Then, if we can construct a reference of type `LowerBound[Int] with UpperBound[String]`, we can use its type member `M` to turn an `Int` into a `String`.

- We would expect that no reference has type

  `LowerBound[Int] with UpperBound[String]`

  , but in fact there is one: `null`. This happens because `null` is an element of *every* reference type. The example uses this "trick" to give the local binding `bounded` this impossible type. In the literature, `bounded.M` is said to have "bad bounds" [Amin et al., 2016].

- With `bounded` in place, the code uses `upcast` to turn an `Int` into a `String`. The result is that the program is type-correct, but running it produces a `ClassCastException`, raised by the JVM. This is unsound! Notice that no downcasts were used in the example.

The soundness hole described above, reported in 2016, is still present in Scala and Dotty. However, it is fixed by our type system with explicit nulls. Specifically, the definition of `bounded` is no longer type-correct, because `null` is not a subtype of `LowerBound[T] with UpperBound[U]`. The definitions of `bounded` and `upcast` need to be adjusted as follows:

```
def upcast(lb: LowerBound[T]|Null, t: T): lb.M = t
val bounded : (LowerBound[T] with UpperBound[U])|Null = null
```

Our typechecker then complains

```
-- [E008] Member Not Found Error ------------------------------
   |    def upcast(lb: LowerBound[T]|Null, t: T): lb.M = t
   |                                               ^^^^
   |    type M is not a member of unsound.LowerBound[T] | Null
```

We just turned a runtime error into a compile-time error. The unsoundness was averted!

```
// Scala
val s = new StringDecorator("hello")
val s2 = s.repeat(2)
val l = s2.length

// Java
class Decorator {
  String s;

  Decorator(String s) { this.s = s; }

  // Returns a copy of `s` concatenated `n` times.
  String repeat(int n) {
    // code implementing `repeat`
  }
}
```

Figure 2.4: Scala code using a Java library

2.3 Java Interoperability

One of Scala's strengths is its ability to seamlessly use Java libraries. For example, Figure 2.4 shows Scala code that uses a Java `Decorator` library (implementing a Decorator pattern [Pree and Gamma, 1995]) to access methods on a `String` that are not available by default.

Notice that there is no Foreign Function Interface (FFI) or other inter-language communication mechanism that the user needs to go through in order to call the Java code. Instead, the Java library appears to the Scala code as any other Scala library would.

The interaction can also happen in the opposite direction: Java code can use Scala libraries. Figure 2.5 shows an example of this.

This interoperability style between Java and Scala is possible because both languages are compiled down to Java Virtual Machine (JVM) *bytecode* (the bytecode is stored in Java class files) [Lindholm et al., 2014].

```
// Java
val ops = new LengthOps
ops.getLength("hello")

// Scala
class LengthOps {
  def getLength(val s: String): Int = s.length
}
```

Figure 2.5: Java code using a Scala library

2.3.1 Interpreting Java Types

When a Java library is used from Scala, the Scala compiler needs to decide which types it is going to assign to the symbols defined in the Java class file. For example, in Figure 2.4, upon detecting that method `repeat` is Java-defined, the Scala compiler needs to decide what the return type of `repeat` will be. Since the Scala compiler knows that the method returns the Java type `String` (this is indicated in the class file), the compiler needs then to "interpret" the Java type as a Scala type. The current implementation of the Scala compiler implements this "type translation" with the identity function. That is, the types are left unchanged. In our example, the return type of `repeat` would be `String`.

Now suppose that we are working with a version of Scala with explicit nulls. Consider what happens if we have the following implementation of `repeat` (from Figure 2.4):

```
// Java
String repeat(int n) {
  return null;
}
```

This implementation is type-correct in Java, because the Java type system remains implicitly nullable. However, if we use `repeat` from Scala like so

```
new StringDecorator("hello").repeat(2).length
```

Then the final call to the `length` method throws a `NullPointerException`, since the receiver (`repeat`'s return value) is `null`. The problem is that even though Scala interprets `repeat`'s return type as `String`, which is non-nullable, the method returns a `null` value. This example shows how interoperability with Java can introduce unsoundness into the Scala type system.

Here is a dual problem that arises because of Scala's naive type translation being the identity. Suppose we have a Java method `isAtLeast` that determines whether a string has at least a certain length, but the method is designed to handle `null` values, so that `isAtLeast(null, n)` is `true` iff $n = 0$.

```
boolean isAtLeast(String s, int n) {
  return (n == 0) || (s != null && s.length >= n);
}
```

In the world of Scala with explicit nulls, the call `isAtLeast(null, 0)` is not type-correct, because `null` is not an element of `String`, which is the type the Scala compiler selects for the first argument of `isAtLeast`. However, this call is semantically well-behaved, since as we saw the method can handle `null` arguments. In this case, the interaction of Java interoperability with explicit nulls prohibits what should be a valid method call.

The solution for both problems is to modify how the Scala compiler interprets the types of Java-defined symbols. In the first case, the signature of `repeat` really should be (written in Scala syntax)

```
def repeat(n: Int): String|Null
```

Similarly, the signature of `isAtLeast` should be

```
def isAtLeast(s: String|Null, n: Int): Boolean
```

In general, when interpreting a Java type, we need to make sure to mark as nullable all *reference types* that appear "in the right places". This we do in the next section.

2.3.2 Type Nullification

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

- The elements of the $\texttt{String}_{\texttt{Java}}$ type are all finite-length strings (e.g. `"hello world"` and `""`), plus the value `null`. By contrast, the element of $\texttt{String}_{\texttt{Scala}}$ are *just* all finite-length strings (but not `null`). This means that nullification must map $\texttt{String}_{\texttt{Java}}$ to $\texttt{String}_{\texttt{Scala}}\texttt{|Null}$.
- Similarly, we can think of a Java method with signature

$\texttt{String}_{\texttt{Java}}$ `getName(`$\texttt{String}_{\texttt{Java}}$ `s)`

as being implemented by a function from $\texttt{String}_{\texttt{Java}}$ to $\texttt{String}_{\texttt{Java}}$ (i.e. `getName:` $\texttt{String}_{\texttt{Java}} \to \texttt{String}_{\texttt{Java}}$). Suppose that $f \in \texttt{String}_{\texttt{Java}} \to \texttt{String}_{\texttt{Java}}$. Notice that f can take `null` as an argument, and return `null` as a result. This means that nullification should return $\texttt{String}_{\texttt{Scala}}|Null \to \texttt{String}_{\texttt{Scala}}|Null$ in this case.

We can go through other kinds of Java types and informally reason about what the result of nullification should be in order to preserve the type's elements. For now, I only provide "preserves elements in a type" as an informal argument for the correctness of nullification. Chapter 3 formalizes this idea using denotational semantics.

Nullification can be described with a pair of mutually-recursive functions $(F_{\text{null}}, A_{\text{null}})$ that map Java types to Scala types. The functions are defined in Figure 2.6 and described below. But first, a word about how nullification is applied. The Dotty compiler can load Java classes in two ways: from source or from bytecode. In either case, when a Java class is loaded, we apply F_{null} to:

- The types of fields.
- The argument types and result type of methods.

The resulting class with modified fields and methods is then made accessible to the Scala code. Below is some intuition and example for the different nullification rules.

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

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

$\xrightarrow{F_{\text{null}}}$

```
// Nullified class as seen by the Scala compiler
class C {
```

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

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

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

Figure 2.6: Type nullification functions

```
  String | Null s;
  int x;
}
```

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

```
class C<T> {
  T foo() {
    return null;
  }
}
```

$\xrightarrow{F_{\text{null}}}$

```
class C<T> {
  T|Null foo() {
    return null;
  }
```

```
}
```

Notice that this rule is sometimes too conservative (leads to backwards-incompatible behaviour), as witnessed by

```
// Scala
class S {
  val c: C[Bool] // C defined as above
  // The line below no longer typechecks, since 'foo' now returns a Bool|Null.
  val b: Bool = c.foo()
}
```

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

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

$\xrightarrow{F_{\text{null}}}$

```
// Nullified class as seen by the Scala compiler
class Box<T> { T|Null get(); }
class BoxFactory<T> { Box<T>|Null makeBox(); }
```

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

Generalizing from the example, we can see that it is enough to nullify the type application `C<T>` as `C<T>|Null`. That is, it is enough to mark the type as nullable only at the top level, since uses of `T` in the body of `C` will be nullified as well, *if `C` is Java-defined*.

Notice that the correctness argument relies on our ability to patch all Java-defined classes that transitively appear in the argument or return type of a field or method accessible from the Scala code being compiled. Since all such classes must be visible to the Scala compiler in any case, and since every Java class visible to the compiler is nullified, we think rule **FN-JG** is sound.

In fact, the rule is a bit more complicated than I have shown so far. The full rule is

$$F_{\text{null}}(C\texttt{<}R\texttt{>}) = C\texttt{<}A_{\texttt{null}}(R)\texttt{>|Null} \qquad \text{if } C \text{ is Java-defined} \qquad \text{(FN-JG)}$$

Notice that in fact we *do* transform the type argument, but do so using A_{null} instead of F_{null}. A_{null} is a version of F_{null} that does not add `|Null` at the top level. A_{null} is needed for cases where we have nested type applications, and it is explained in more detail below.

Here is a sample application of F_{null} to a nested type application, assuming that C, D, and `String` are all Java-defined:

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

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

- **Case (FN-SG)** This rule handles the mirror case, where we have a generic $C\texttt{<}T\texttt{>}$, and C is Scala-defined. For example,

```
// Box is Scala defined
class BoxFactory<T> { Box<T> makeBox(); }
```

$\xRightarrow{F_{\text{null}}}$

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

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

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

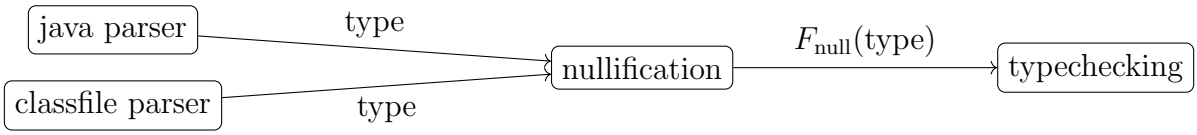
F_{null} vs A_{null}

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

Suppose `getBox` is a Java method that returns a `Box<String>`. Then there might be references to `getBox` from Scala: e.g. `val b: Box[String] = foo.getBox()` or `def withBox(b: Box[String]) = ...`. If nullification turns the return type of `getBox` into `Box<String|Null>|Null`, then the Scala code will require changes when ported to explicit nulls.

Implementation Note

The examples above give the impression that nullification involves somehow modifying the *contents* of a Java class (e.g. its source code). In our implementation, however, nullification only transforms types, and does not change the source code or bytecode of a class. As illustrated below, in the compiler, the nullification module sits between the different parsers and the typechecker. After a Java class is parsed, but before it can be used within the typechecking phase, the types of fields and methods are nullified. That is, nullification is an *online* process that happens during compilation.



2.3.3 The `JavaNull` Type

While experimenting with our implementation of explicit nulls, we found that a common pattern is to chain method or field selectors on a Java-returned value.

```
val s = "hello_world".trim().substring(2).toLowerCase()
```

All of the methods called above are Java-defined, and they all return `String`. After nullification, the return type of the methods becomes `String|Null`. This means that every method selector above will fail to type-check (because the type `String|Null` does not have any methods). Assuming that flow typing is available (flow typing is described in Section 2.4), we would need to change the example above to

```
val ret = someJavaMethod()
val s = if (ret != null) {
  val tmp = ret.trim()
  if (tmp != null) {
    val tmp2 = tmp.substring(2)
    if (tmp2 != null) {
      tmp2.toLowerCase()
    } else ???
  } else ???
} else ???
```

The code is now type-correct, but it has become unacceptably verbose. Additionally, the user needs to explicitly handle every case where a `null` might be returned. Even though the latter is conceptually a good thing, we think it imposes too high of a burden for migrating Scala code with Java dependencies.

Our solution is to give up some soundness to gain on usability. We introduce a special `JavaNull` type alias with "magic powers".

```
type JavaNull = Null
```

`JavaNull` behaves just like `Null`, except it allows (unsound) member selections:

```
val s: String | Null = ???
s.toLowerCase() // error: String | Null has no member named 'toLowerCase'

val s2: String | JavaNull = ???
s2.toLowerCase() // type-correct, but might throw at runtime
```

The original problematic example that chains method calls is now again type-correct, but can throw a `NullPointerException` at runtime if any of the methods returns `null`.

Notice that we need `JavaNull = Null`, and not just `JavaNull <: Null`, since otherwise we cannot e.g. pass an `Array[String|Null]` to a method expecting an `Array[String|JavaNull]`. Types like `Array[String|JavaNull]` arise because we make our nullification function (shown in Figure 2.6) generate union types using `JavaNull` instead of `Null`. That is, the true of form of rule **FN-Ref** is

$$F_{\text{null}}(R) = R|\texttt{JavaNull} \qquad \text{if } R \text{ is a reference type} \qquad \text{(FN-Ref)}$$

This gives us another intuition for `JavaNull`, which is that `JavaNull` denotes `null` values that "flow in" from Java into Scala code. Theoretically, however, it seems problematic that `JavaNull = Null`, but they behave differently. If we were to formalize this part of our explicit nulls system, modelling `JavaNull` would be future work.

2.3.4 More Precise Nullification

In this section, I present several strategies that we use to improve the precision of the baseline nullification function. Some of the optimizations in this section are sound, and others are theoretically unsound, but based on, I will argue, reasonable assumptions.

Java Language Semantics

We can use the Java semantics to argue that certain terms can never be `null`:

- The return type of a constructor should not be nullable, since constructors either fail with an exception, or return a new object (but never return `null`). Given a Java class C, the Scala compiler models its constructor as a method `C <init>()`, where `<init>` is a compiler-internal name identifying constructors. We make sure that nullification recognizes constructors and does not nullify their return types. The type of `new C()` is then `C`, and not `C|JavaNull`.
- Elements of a Java `enum` (enumeration) cannot be null. The Scala compiler represents the enum

  ```
  enum DaysOfWeek {
    MON, TUE, WED, THU, FRI, SAT, SUN
  }
  ```

  as a class with fields `MON`, `TUE`, etc. The type of every field is (in Java) `DaysOfWeek`. We have specialized nullification so that the enum fields are *not* nullified.

Standard Library Metadata

Naive application of type nullification can be too conservative. Consider again the example from Section 2.3.3.

```
val s = "hello world".trim().substring(2).toLowerCase()
```

All of `trim`, `substring`, and `toLowerCase` are methods defined in the Java standard library within the `String` class. Consulting the documentation for these methods, it is clear that even though their return type `String` is a reference type, none of the methods are expected to return `null`. If we trust that the implementations in the Java standard

library are likely to be well-tested and bug-free, then we might want to special case some standard library methods during nullification.

We obtained a version of the Java standard library where methods and fields that are non-nullable were annotated with `@NonNull` annotations. This class file was generated by the Checker Framework Project [Papi et al., 2008]. From the class file, we generated a list of methods and fields in the standard library that can be special cased, because they are annotated as not returning `null`. The list currently contains 4414 methods and 1712 fields spread over 847 classes (see Section 2.7 for more details). For example, all of the methods in the `String` example above are in our list. This means that we recognize the return type of `trim` as `String`, and not `String|JavaNull`.

The aforementioned list of methods and fields is loaded by the compiler and used during nullification to avoid introducing unnecessary nullable types.

Nullability Annotations

Inspired by similar functionality in Kotlin [Kotlin Foundation, b], we modified the Scala compiler to recognize any of several *nullability annotations* that have been developed over the years to support e.g. static analyzers. For example, the class below has a method whose return type is marked as `@NonNull`, and so after nullification the return type of `getName` is `String` and not `String|JavaNull`.

```
// Java
class Person {
  @NonNull
  String getName() { ... }
}
```

Widespread use of nullability annotations reduces the number of types marked as nullable by nullification, and so makes it easier to migrate code with Java dependencies to the world of explicit nulls.

Nullability annotations are recognized without any runtime enforcement. That is, mislabelling e.g. a field as `@NonNull` can lead to a `NullPointerException` if the field *does* end up being `null` during execution.

2.4 Flow Typing

To improve usability of nullable values, we added a simple form of flow-sensitive type inference to Scala [Guha et al., 2011]. The general idea is that sometimes, by looking at the control flow, we can infer that a value previously thought to be nullable (due to its type) is no longer so.

2.4.1 Supported Cases

Below I list the cases supported by flow typing. In the examples, the notation `???` stands for an unspecified expression of the appropriate type[1].

Branches of an If Expression

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

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

We can reason similarly about the `else` branch if the test is `p == null`.

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

The following operators are considered a comparison for the purposes of flow typing: `==`, `!=`, `eq`, and `ne`.

Logical Operators

We also support the logical operators `&&`, ||, and ! in conditions:

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

```
val s: String|Null = ???
val s2: String|Null = ???

if (s != null && s2 != null) {
  val l = s.length    // ok, `s` has type `String`
  val l2 = s2.length // ok, `s2` has type `String`
}

if (s == null || s2 == null) {
  // s: String|Null
  // s2: String|Null
} else {
  val l = s.length    // ok, `s` has type `String`
  val l2 = s2.length // ok, `s2` has type `String`
}

if (!(s == null)) {
  val l = s.length // ok, `s` has type `String`
}
  // s: String|Null
}
```

Propagation Within Conditions

We support type specialization within a condition, taking into account that `&&` and `||` are short-circuiting.

```
val s: String|Null

// In the condition, the test `s.length` is type correct because the right-hand side
// of the condition will only be evaluated if `s` is non-null.
if (s != null && s.length > 0) {
  // s: String
}

// Similar case for `||`.
if (s == null || s.length > 0) {
  // s: String|Null
} else {
```

```
  // s: String | Null
}
```

Nested Conditions

Our inference works in the presence of arbitrarily-nested conditions. Sometimes, the reasoning can be tricky, even for humans!

```
val a: String | Null = ???
val b: String | Null = ???
val c: String | Null = ???

if (!(a == null || b == null) && (c != null)) {
  // `a`, `b`, and `c` all inferred to be non-null
}
```

Early Exit from Blocks

If a statement does an early exit from a block based on whether a value is `null`, we can soundly assume that the value is non-null from that point on.

```
def len(s: String | Null): Int = {
  if (s == null) return 0
  return s.length        // ok, `s` inferred to have type `String` from this point on
}
```

This is also the case with exceptions.

```
def len(s: String | Null): Int = {
  if (s == null) throw new IllegalArgumentException("null argument")
  return s.length        // ok, `s` inferred to have type `String` from this point on
}
```

In general, if we have a block $s_1, \ldots, s_i, s_{i+1}, \ldots, s_n$, where the s_i are statements, and s_i is of the form `if (cond) exp`, where `exp` has type `Nothing`, then depending on `cond` we might be able to infer additional nullability facts for statements $s_{i+1}, \ldots, s_n$. The reason is that type `Nothing` has no values, so an expression of type `Nothing` cannot terminate normally (it either throws or loops). It is then safe to assume that statement s_{i+1} executes only if `cond` is `false`.

There is one extra complication here, which is that Scala allows forward references to be `defs`, which combined with nested methods can lead to non-intuitive control flow.

```
def foo() = {
  val s: String|Null = ???
  bar() // forward reference
  if (s != null) return
  def bar(): Int = {
    // cannot infer s: String
    s.length // type error: `s` has type `String|Null`
  }
}
```

In this example, we test whether *s* is `null` and, if so, return from the outer method `foo`. Since the inner method `bar` is declared *after* the null check is complete, we might be tempted to infer that in the body of `bar`, `s` is non-null, so `s.length` is type-correct. Unfortunately, there is a forward reference to `bar` that happens *before* the null check is done. This means that `bar` could execute with `s` being `null`, so we cannot do flow typing. In our implementation, we have logic for detecting forward references like in the example above, and "disabling" flow typing to preserve soundness. If there is no forward reference, flow typing can be triggered.

```
def foo() = {
  val s: String|Null = ???
  // no forward references
  if (s != null) return
  def bar(): Int = {
    // can infer s: String
    s.length
  }
}
```

Stable Paths

We use flow typing on `vals`, but not on `vars` or `defs`. The example below shows why using (naive) flow typing on a `var` would be unsound.

```
var s: String|Null = "hello"
if (s != null && {s = null; true}) {
  // s == null
```

```
}
```

The expression `{s = null; true}` is valid Scala because it is a block, and blocks are expressions. The value of a block is the value of its last expression; in this case, `true`. This means that when the `then` branch executes, `s` will be `null`, even though we checked against it in the condition.

Similarly, flow typing on `defs` would be problematic, because a `def` is not guaranteed to return the same value after every invocation.

```
var b: Boolean = false
def getS: String | Null = {
  b = !b
  if (b) "hello" else null
}

if (getS != null) {
  getS.length // unsound to infer getS: String, since it would throw a NullPointerException
}
```

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

Unsupported Idioms

Our current support for flow typing is limited and does not include:

- Reasoning about non-stable paths (`vars` and `defs`): conceptually, it should be possible to do flow typing for *local* `vars` that are not *captured* by closures. This would also require "killing" some flow facts once a `var` is re-assigned.
- Generating flow facts about types other than `Null`.

  ```
  val x: String | Null = ???
  if (x.isInstanceOf[String]) {
    // could infer that x: String
  }
  ```

- Support for pattern matching.

```
val s: Animal|Null = ???
s match {
  case Dog(name) => // could infer that 's' is non-null
  case null =>
  case _ => // could infer that 's' is non-null
}
```

- Tracking aliasing between non-nullable paths.

```
val s: String|Null = ???
val s2: String|Null = ???
if (s != null && s == s2) {
  // s: String inferred
  // s2: String not inferred
}
```

2.4.2 Inferring Flow Facts

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

At the core of flow typing we have a function $\mathcal{N} : \texttt{Exp} \times \texttt{Bool} \to \mathbb{P}(\texttt{Path})$. $\mathcal{N}$ takes a Scala expression e (where e evaluates to a boolean) and a boolean b, and returns a set of paths known to be non-nullable if e evaluates to b. That is, $\mathcal{N}(\texttt{e}, \texttt{true})$ returns the set of paths that are non-null if e evaluates to `true`; and $\mathcal{N}(\texttt{e}, \texttt{false})$ returns the set of paths known to be non-null if e evaluates to `false`. $\mathcal{N}$ is defined in Figure 2.7.

Denote the set of stable paths that are non-null if e evaluates to b by $\mathcal{N}_{\text{all}}(\texttt{e}, \texttt{b})$. Clearly, computing $\mathcal{N}_{\text{all}}$ is undecidable, since conditions in Scala can contain arbitrarily complex logic (for example, they can contain statements, class definitions, etc). Below, we argue informally that $\mathcal{N}$ correctly under-approximates $\mathcal{N}_{\text{all}}$.

Lemma 2.4.1 (Soundness of $\mathcal{N}$)**.** $\mathcal{N}(\texttt{e}, \texttt{b}) \subseteq \mathcal{N}_{all}(\texttt{e}, \texttt{b})$

Sketch. By induction on e. We only look at the cases where F_{null} returns a non-empty set.

Case ($\mathcal{N}(\texttt{p == null}, \texttt{false})$) Immediate, provided that p is stable.

Case ($\mathcal{N}(\texttt{p != null}, \texttt{true})$) Immediate, provided that p is stable.

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

Figure 2.7: Flow facts inference

Case ($\mathcal{N}$(`A && B`, `true`)) Here we know that $\mathcal{N}(\texttt{A \&\& B}, \texttt{true}) = \mathcal{N}(\texttt{A}, \texttt{true}) \cup \mathcal{N}(\texttt{B}, \texttt{true})$. By the induction hypothesis, we get that $\mathcal{N}(\texttt{A}, \texttt{true}) \subseteq \mathcal{N}_{\text{all}}(\texttt{A}, \texttt{true})$, and $\mathcal{N}(\texttt{B}, \texttt{true}) \subseteq \mathcal{N}_{\text{all}}(\texttt{B}, \texttt{true})$. Now consider the semantics of conjunction in Scala. If $A \&\& B$ is `true`, then *both* A and B must evaluate to `true`. This means that $\mathcal{N}_{\text{all}}(\texttt{A \&\& B}, \texttt{true})$ includes *at least* the elements in $\mathcal{N}_{\text{all}}(\texttt{A}, \texttt{true}) \cup \mathcal{N}_{\text{all}}(\texttt{B}, \texttt{true})$. The result follows.

Case ($\mathcal{N}$(`A && B`, `false`)) We know that $\mathcal{N}(\texttt{A \&\& B}, \texttt{false}) = \mathcal{N}(\texttt{A}, \texttt{false}) \cap \mathcal{N}(\texttt{B}, \texttt{false})$. Again from the Scala semantics, we know that if $A \&\& B$ evaluates to `false`, either A evaluates to `false` *or* B evaluates to `false`. This implies that $\mathcal{N}_{\text{all}}(\texttt{A \&\& B}, \texttt{false})$ contains *at least* every element in $\mathcal{N}_{\text{all}}(\texttt{A}, \texttt{false}) \cap \mathcal{N}_{\text{all}}(\texttt{B}, \texttt{false})$. We can then apply the induction hypothesis and we are done.

Case ($\mathcal{N}$(`A || B`, `true`)) We know that $\mathcal{N}(\texttt{A || B}, \texttt{true}) = \mathcal{N}(\texttt{A}, \texttt{true}) \cap \mathcal{N}(\texttt{B}, \texttt{true})$. We also know that the expression $A || B$ evaluates to `true` if either A or B is `true`. This means that $\mathcal{N}_{\text{all}}(\texttt{A || B}, \texttt{true})$ must contain at least the elements that are *both* in $\mathcal{N}_{\text{all}}(\texttt{A}, \texttt{true})$ and $\mathcal{N}_{\text{all}}(\texttt{B}, \texttt{true})$. The result follows by the induction hypothesis.

Case ($\mathcal{N}$(`A || B`, `false`)) We have $\mathcal{N}(\texttt{A || B}, \texttt{false}) = \mathcal{N}(\texttt{A}, \texttt{false}) \cup \mathcal{N}(\texttt{B}, \texttt{false})$. Observe that $A || B$ is false if *both* A and B are false. This means that $\mathcal{N}_{\text{all}}(\texttt{A || B}, \texttt{false})$ must contain at least the elements in both $\mathcal{N}_{\text{all}}(\texttt{A}, \texttt{false})$ and $\mathcal{N}_{\text{all}}(\texttt{B}, \texttt{false})$. The result then follows from the induction hypothesis.

Case ($\mathcal{N}$(`!A, true`)). This follows from the induction hypothesis and the fact that $!A$ evaluates to `true` iff A evaluates to `false`.

Case ($\mathcal{N}$(`!A, true`)) Similar to the case above.

Case ($\mathcal{N}$(`{s1; ..., sn, cond}, b`)) This follows from the induction hypothesis and the fact that a block evaluates to its last expression. □

Using Flow Facts

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

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

2.5 Asserting Non-Nullability

For cases where flow typing is not powerful enough to infer non-nullability and, more generally, as an "escape hatch" from the explicit nulls type system, we added a `.nn` ("assert non-nullable") "extension method" to cast away nullability from any term.

```
var s: String|Null = ???
if (s != null) {
  val l = s.nn.length // ok: .nn method "casts away" nullability
}
```

In general, if `e` is an expression with type `T|Null` (or `T|JavaNull`), then `e.nn` has type `T`. The implementation of `.nn` is interesting, because it is done purely as part of the Scala standard library, and does not touch compiler internals.

```
def (x: T|Null) nn[T]: T =
  if (x == null) throw new NullPointerException("tried to cast away nullability , but value is null")
  else x.asInstanceOf[T]
```

The `nn` method is defined as an *extension method*. This is a kind of implicit definition that makes `nn` available for any receiver of type `T|Null`. We see that `nn` fails if the value being asserted is equal to `null`.

2.6 Dropped Features

The features described below were either considered but not implemented as part of the explicit nulls project, or implemented and then dropped for different reasons. They are documented here as a kind of "negative result", because I think there is value in describing everything we tried during the project.

2.6.1 Arrays

Arrays are problematic for explicit nulls, because constructing an array through the constructor allocates the array, but the elements are initialized to `null`. This means that accessing an uninitialized array slot is both easy and unsound.

```
// Scala
val a = new Array[String](10)
a(0).length // throws NullPointerException
```

To fix the unsoundness, we considered forcing arrays to have nullable element types.

```
// Not implemented: force arrays to have a nullable element type
val a = new Array[String](10)       // type error: 'String' is not nullable
val a = new Array[String|Null](10) // ok: element type is nullable
```

We decided against this after looking at the implementation of the Scala standard library. The standard library makes widespread use of arrays, for efficiency. In particular, it uses arrays of value types. Unfortunately, if we enforce the restriction above about the element type being nullable, arrays of value types are no longer possible:

```
val a: Array[Boolean](10) // not allowed if the restriction above is implemented
val a: Array[Boolean|Null](10) // ok
```

This is problematic, because e.g. `Boolean|Null` is erased to `Object`, meaning that it is no longer possible to have arrays with unboxed elements. Even if we are able to special case e.g. `Array[Boolean]`, there are more complicated cases. Consider

```
def foo[T](): Array[T] = ???
```

In the current version of Scala, a call `foo[Boolean]` returns an `Array[Boolean]`, which can be efficiently represented in the JVM as an array of a primitive (unboxed) type. However, if we force the element type to be nullable, we could only call `foo[Boolean|Null]` and get back an `Array[Boolean|Null]`. Such an array cannot be represented as a primitive array of booleans, because `null` might very well be an element of it.

So it seems that we can either handle arrays unsoundly, or give up our ability to have arrays of primitive types. We choose the former. As future work, we are considering issuing warnings for uses of the array constructor with a reference type (e.g. `new Array[String](10)`). In those cases, we would suggest one of two safe alternatives:

- If the user knows the array's elements at creation time, they can use an (existing) helper method that specifies them: `Array[String]("hello", "world")`.
- Alternatively, if the size of the array is determined at runtime, the user can call any of several (existing) helper methods like `fill` or `iterate`, which require an initializer: `val names = Array.fill(10)("")` (creates an array of ten elements, all of which the empty string).

2.6.2 Override Checks

If we have a Scala class that extends a Java class and *overrides* one of its methods, then after switching to explicit nulls the Scala class will need to adjust the argument and return types in the override.

```
// Java
class Animal {
  String getName(String language)
}

// Scala
class Dog extends Animal {
  override def getName(language: String): String
}
```

After compiling `Dog` with explicit nulls, we will get a type error because the override is no longer valid. We need to adjust `getName` to reflect the fact that the argument and return type can be nullable.

```
// Scala
class Dog extends Animal {
  override def getName(language: String|Null): String|Null
}
```

From the correctness perspective, this is the right thing to do, but it affects usability because the required changes are *non-local*: if another Scala class depends on `Dog`, it too needs to be modified.

We initially implemented the following relaxation of the override check: if the base and derived types *differ only in their nullability, report a warning instead of an error*. Eventually, we decided to disable this relaxed override check, because we did not encounter many cases where a Scala class overrides a method coming from a Java class, and so the usability gains did not seem to outweigh the added unsoundness. We might revisit this decision in the future if this type of override is common.

2.6.3 Binary Compatibility

Our strategy for binary compatibility with Scala binaries that predate explicit nulls is to leave the types unchanged and be compatible but unsound. The problem is how to interpret the return type of foo below

```
// As compiled by e.g. Scala 2.12, pre explicit nulls
class Old {
  def foo(): String = ???
}
```

There are two options:

- `def foo(): String`
- `def foo(): String|Null`

The first option is unsound. The second option matches how we handle Java methods. However, this approach is too conservative in the presence of generics:

```
// Scala pre explicit nulls
class Old[T] {
  def id(x: T): T = x
}

// The same class, seen by Scala with explicit nulls
class Old[T] {
  def id(x: T|Null): T|Null = x
}
```

If we instantiate `Old[T]` with a value type, then `id` now returns a nullable value, even though it should not.

```
val o: Old[Boolean] = ???
val b = o.id(true) // b: Boolean|Null
```

So really the options are between being unsound and being too conservative. The unsoundness only kicks in if the Scala code being used returns a `null` value. We hypothesize that null is used infrequently in Scala libraries, so we go with the first option.

If using an unported Scala library that *produces* `null`, the user can wrap the (hopefully rare) API in a type-safe wrapper:

```
// Scala pre explicit nulls
class Old {
  def foo(): String = null
}

// User code in explicit -nulls world
def fooWrapper(o: Old): String|Null = o.foo() // ok: String <: String|Null

val o: Old = ???
val s = fooWrapper(o) // s: String|Null
```

If the offending API *consumes* `null`, then the user can cast the null literal to the right type (the cast will succeed, since at runtime `Null` is a subtype of any reference type).

```
// Scala pre explicit nulls
class Old() {
  // s should be a String, or null to signal a special case
  def foo(s: String): Unit = ???
}
```

```
// User code in explicit-null world
val o: Old = ???
o.foo(null.asInstanceOf[String]) // ok: cast will succeed at runtime
```

Another reason for choosing the unsound approach is that it eases migration. Imagine two Scala libraries, `A` and `B`, where `B` has a dependency on `A`. Suppose that we treat Scala libraries conservatively, and that `B` is migrated *before* `A`. Then `B` will need to be modified to handle `A`'s nullable methods and fields. However, if `A` is later migrated to explicit nulls, then `B` would need to be modified *a second time*. If we treat Scala libraries unsoundly, then `B` only needs to be migrated *once*, after `A` is migrated. This higlights a key difference between Scala and Java libraries, which is that we expect Scala libraries to eventually be migrated to explicit nulls, while Java libraries will remain implicitly nullable.

2.7 Evaluation

There are at least two evaluative questions we could ask about our design for explicit nulls:

1. Are explicit nulls *useful*? That is, how effective are explicit nulls in reducing the number of runtime errors in Scala programs?
2. Are explicit nulls a *reasonable* extension to the type system? Specifically, given that existing Scala code would need to be modified to compile under explicit nulls, *how much effort* would it be to undertake such a migration?

Our evaluation addresses the second question. I explain why the first question is interesting, but hard to measure, in Section 2.7.2.

As mentioned earlier, we implemented our design for explicit nulls as a modification of the Dotty (Scala 3) compiler. We evaluated our design by migrating (most of) the libraries in the Dotty "community build" [2] to compile under explicit nulls. The community build is a set of open-source libraries that are used to test the Dotty compiler (additionally, Dotty has its own internal tests). These libraries are shown in Table 2.1. We have not migrated the three libraries that appear greyed-out in the table. We focused on these libraries because Dotty is not backwards-compatible with Scala 2. The libraries in the community build had previously been modified to compile under a version of Dotty without our changes.

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

Name	Description	Size (LOC)	Files
`scala-pb`	Scala protocols buffer compiler	37,029	275
`squants`	DSL for quantities	14,367	222
`fastparse`	Parser combinators	13,701	80
`effpi`	Verified message passing	5,760	60
`betterfiles`	IO library	3,321	29
`algebra`	Algebraic type classes	3,032	75
`scopt`	Command-line options parsing	3,445	28
`shapeless`	Type-level generic programming	2,328	18
`scalap`	Class file decoder	2,210	22
`semanticdb`	Data model for semantic information	2,154	49
`intent`	Test framework	1,866	48
`minitest`	Test framework	1,171	32
`xml-interpolator`	XML string interpolator	993	20
Subtotal		91,377	958
`stdlib123`	Scala standard library	31,723	588
`scala-xml`	XML support	6,989	115
`scalactic`	Utility library	3,952	53
Total		134,041	1,714

Table 2.1: Community build libraries. We have not migrated the greyed-out libraries to explicit nulls yet. The size of each library is given in lines of code (LOC).

2.7.1 Assessing Migration Effort

We estimated the effort required to migrate the community build libraries in two ways:

1. We ran our version of Dotty with explicit nulls under different configurations, by turning on and off different features of our system. For each configuration, we recorded the number of errors reported by the compiler when building each library. Our results show that the vast majority of null-related type errors happen because of interaction with Java. Specifically, most errors happen because of the conservative assumptions we make about the return types of Java fields and methods being nullable. Additionally, the results suggest that neither `JavaNull` nor flow typing are effective in reducing the number of nullability errors (but see the caveats about flow typing in Section 2.7.1).

2. Using a compiler configuration that makes "optimistic" assumptions (see below), we manually migrated the libraries, while classifying the remaining null-related type errors. Our results confirm the intuition that uses of `null` in Scala code (without interoperation with Java) is relatively rare, generating 5.34 errors per thousand lines of Scala code.

Later in this section I present some takeaways about how we could modify our design to address the results of the evaluation. I now describe the evaluation process and results in more detail.

Estimating Error Counts

In the first experiment, we modified Dotty so that we could independently turn on and off different features of our implementation. Specifically, we added the following flags:

Flag	Purpose
`-Yexplicit-nulls`	Turns on *just* the new type hierarchy, where `nulls` are explicit.
`-Yjava-null`	If set, `JavaNull` allows field selections.
`-Yflow-typing`	Controls flow typing.
`-Yjava-interop-checker-framework`	Enables use of Checker Framework annotations.
`-Yjava-interop-optimistic`	If set, assume that all Java fields are non-null, and that the return types of Java methods are non-nullable.

Name	explicit nulls	JavaNull	flow typing	nullability annotations	optimistic interop
naive	✓				
java-null	✓	✓			
flow	✓	✓	✓		
checker	✓	✓	✓	✓	
optimistic	✓	✓	✓		✓

Table 2.2: Run configurations for migrating community build libraries

Here is an example for the use of `-Yjava-interop-optimistic`. Suppose we have a Java method with signature

```
Array[String] => Array[String]
```

Under `-Yjava-interop-optimistic`, this method will be interpreted as having signature `Array[String|Null]|Null => Array[String|Null]`.

In particular, notice how the return type of the method is non-nullable (even though the inner element type remains nullable).

Using these flags, we defined four *run configurations* (flag states) that we used when evaluating the libraries. The configurations are shown in Table 2.2. They start with the `naive` configuration, which has explicit nulls, but where all other features are turned off, all the way to the `optimistic` configuration, where all feature flags are turned on (`-Yjava-interop-optimistic` subsumes `-Yjava-interop-checker-framework`). We then ran the Dotty compiler on each library/configuration pair. The results are shown in Table 2.3, and allow us to make a few observations:

- There is significant variance among libraries in how common nullability errors are. The number of errors per library (reported in Table 2.3 as errors per thousand lines of Scala code) can serve as a proxy for how much work it would be to migrate each of them. The number of errors ranged from zero (for `squants`), to very few errors (`algebra` and `effpi`), to a relatively large number of errors (`betterfiles`, `scala-pb`, and `stdlib123`).

- "Higher level" or more "abstract" libraries like `effpi`, `algebra`, and `shapeless`, had fewer errors. These are libraries that focus on sophisticated uses of the Scala type system, and have fewer Java dependencies. "Lower level" libraries, like `scala-pb` (which deals with serialization and de-serialization of objects) and `betterfiles` (needs to interact with the operating system) had more errors. I speculate that lower level libraries have more errors because they interact with Java APIs more often, as well as use `null` for efficiency reasons (for caching, or to avoid the performance penalty of using `Option` types).

- The Scala collections library, `stdlib123`, had a relatively large number of errors (37.61 errors per thousand LOC). Since this library is also large (31,723 LOC), it will be challenging to migrate it, if and when explicit nulls is adopted upstream by Scala.

- The `java-null` and `flow` configurations were ineffective in reducing the number of errors, when compared to `naive`. In particular, flow typing only made a difference in two cases, `stdlib123` and `scala-xml`. I offer some possible explanations for this later in the section.

- The use of annotation information generated by the Checker Framework reduced the number of errors in all cases, and sometimes significantly, such as in `scalap` and `semanticdb`. Since this is a *sound* optimization (assuming that the annotations are well-placed), this confirms the value of annotations in a type system for explicit nulls.

- The `optimistic` configuration drastically reduced the number of errors. Looking at the totals row, we can see that using `checker` the number of errors is 18.52 per thousand LOC, while `optimistic` brings that number down to 5.34. This is evidence in favour of the "common wisdom" within the Scala community that Scala programs rarely use `null`, and that most `null` values flow in from Java.

Classifying Nullability Errors

In the second experiment, we first set the compiler to the `optimistic` configuration. We then manually migrated most libraries, except for `stdlib123`, `scala-xml`, and `scalactic`, to explicit nulls. By "migrated", I mean that we fixed type errors in the libraries until they compiled. We also categorized the errors as we fixed them. The results are shown in Table 2.4.

Name	naive	java-null	flow	checker	optimistic
scalactic	72.37	71.86	71.86	57.19	3.04
betterfiles	38.54	37.04	37.04	32.22	1.51
stdlib123	37.61	36.88	36.54	33.54	17.24
scala-pb	24.71	24.66	24.66	24.49	1.11
minitest	18.79	17.93	17.93	12.81	6.83
scalap	15.84	14.93	14.93	7.24	1.81
scala-xml	13.88	13.59	13.31	11.3	9.3
semanticdb	10.68	9.29	9.29	6.04	1.39
intent	8.57	8.04	8.04	6.97	0.54
scopt	5.22	4.64	4.64	4.64	2.32
xml-interpolator	2.01	2.01	2.01	2.01	2.01
shapeless	1.72	1.29	1.29	0	0
fastparse	1.61	1.61	1.61	1.53	1.46
effpi	0.35	0.35	0.35	0.35	0
algebra	0.33	0.33	0.33	0.33	0
squants	0	0	0	0	0
Total (weighted mean)	20.62	20.29	20.2	18.52	5.34
Mean	15.76	15.28	15.24	12.54	3.04
Standard deviation	19.65	19.46	19.44	16.28	4.57

Table 2.3: Error frequency by run configuration. The unit is number of (type) errors per thousand LOC.

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

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

Table 2.4: Error classification. Libraries were migrated under `optimistic` configuration. Normalized count is in errors per thousand LOC.

The use of the `optimistic` configuration has advantages and disadvantages. On the one hand, it makes it easier to do the migration, since many of the errors are no longer present. On the other hand, the results become more illustrative of a "best case" scenario for migration than a "worst case" scenario.

The different classes of observed errors are described below:

- *Declaration of nullable field or local symbol.* These are cases where the Scala code declares a `var` or `val` (as a field, or locally within a method) that is provably nullable. For example, in `scala-pb` we have the code

```
def messageCompanionForFieldNumber(__number: _root_.scala.Int): ... = {
  var __out: _root_.scalapb.GeneratedMessageCompanion[_] = null
  ...
  __out
}
```

  The `__out` variable is set to `null` when defined, so its type must be changed to `GeneratedMessageCompanion[_]|Null`.

- *Use of nullable field or local symbol (`.nn`).* This is the dual of the case above, and happens when a variable like `__out` (in the example above) is later used. At that point, we need to cast away the nullability using `__out.nn`.

- *Overriding error due to nullability.* This error happens when a Scala class overrides a Java-defined method that takes a reference type as an argument. Because arguments must be overriden *contravariantly*, the Scala method must then be updated to take a nullable argument as well.

- *Generic* received *from Java with nullable inner type.* Sometimes we encounter a Java method that returns a generic with a nullified inner type. The common example are Java methods returning arrays of reference types. For example, `String`'s `split` method

  ```
  // Splits this string around matches of the given regular expression.
  public String[] split(String regex)
  ```

  Looking at `split`'s documentation, it is clear that the elements of the returned array should not be `null`. However, since we only detect annotations at the outermost level of a type, nullification turns `split`'s return type into `Array[String|JavaNull]|JavaNull`.

  This creates problems for the client Java code. For example, `scopt` does

  ```
  s.split(sep).toList.map(implicitly[Read[A]].reads)
  ```

  The method selection `.toList` works because of the outer `JavaNull`. The `map`, however, is operating over a list of nullable elements (mistakenly marked as such), which later causes an error when searching for the implicit. The fix is to eta-expand the argument to `map` and add a `.nn`.

  ```
  s.split(sep).toList.map(x => implicitly[Read[A]].reads(x.nn))
  ```

- *Generic* passed *to Java requires nullable inner type.* This happens when a Java method expects as argument a generic of some reference type. After nullification, the array is sometimes assigned a nullable element type. Again, arrays are the common case. For example, `java.nio.file.Path`[3] defines a method

  ```
  // Registers the file located by this path with a watch service.
  WatchKey register(WatchService watcher, WatchEvent.Kind<?>... events)
  ```

  After nullification, the method signature changes to

  ```
  def register[T](watcher: WatchService|JavaNull,
                  events: Array[WatchEvent.Kind[_]|JavaNull]|JavaNull)
  ```

[3] https://docs.oracle.com/javase/7/docs/api/java/nio/file/Path.html

In `betterfiles`, we try to call `register`

```
path.register(service, events.toArray)
```

The problem is that `events` has type `Seq[WatchEvent.Kind[_]]`, and `toArray` produces a `Array[WatchEven.Kind[_]]`. The expected type is

`Array[WatchEvent.Kind[_]|JavaNull]|JavaNull`.

The *outer* `JavaNull` is not a problem, because `T <: T|JavaNull` for all `T`. The inner `JavaNull` causes a type error, because arrays are *invariant* in Scala. One possible fix is to use a cast (`asInstanceOf`) to convince the compiler that the array elements might be nullable

```
path.register(service, events.toArray.map(_.asInstanceOf[WatchEvent.Kind[_] | Null]))
```

Notice that this is sound, because even though the original list contained non-nullable elements, what we pass to `register` is a *copy* of the list (in the form of an array).

Another, better, solution, would be to modify Scala's type inference to handle this case. We start by observing `toArray`'s signature[4]:

```
def toArray[B >: A](implicit arg0: ClassTag[B]): Array[B]
```

Right now Scala infers `B = A = WatchEvent.Kind[_]`, but in fact we could infer `B = WatchEvent.Kind[_]|Null`. This is because `WatchEvent.Kind[_] = A <: B = WatchEvent.Kind[_]|Null`. With this fix, no changes would be required to the code.

- *Incorrect Scala standard library definition.* This class contains type errors that could be prevented by changing a definition in the Scala standard library (it is hard to actually make the required changes without first migrating the standard library, which is why we fixed these errors in a roundabout way). In practice, every single error we recorded for this class was due to a use of `Option` type to wrap nullable values. The `Option` *companion object* (a singleton used in Scala to define a class' static methods) defines an `apply`[5] method as follows:

  ```
  // An Option factory which creates Some(x) if the argument is not null, and None if it is null.
  def apply[A](x: A): Option[A]
  ```

  Scala treats `apply` specially: it uses it to define function application. This provides a convenient idiom for working with nullable values: wrap them in an `apply` and then work within the `Option` monad. For example (in Scala with implicit nulls)

[4] https://www.scala-lang.org/api/current/scala/collection/Seq.html
[5] https://www.scala-lang.org/api/current/scala/Option.html

```
val x: String = null
// Option(x) is desugared by Scala to Option.apply[String](x)
val y = Option(x).map(s => s.length) // y: Option[Int] inferred, and y = None
```

The problem is again with the interaction between explicit nulls and type inference:

```
val x: String|Null = null
val y = Option(x).map(s => s.length) // error: s is inferred to have type String|Null,
                                     // which does not have a 'length' field
```

Specifically, type inference leads to desugaring `Option(x)` to

```
Option.apply[String|Null](x)
```

Instead, we want `Option.apply[String](x)`. That is, the type argument to `apply` should be a non-nullable type. We could accomplish this by changing `Option.apply` to

```
// An Option factory which creates Some(x) if the argument is not null, and None if it is null.
def apply[A <: NotNull](x: A|Null): Option[A]
```

where `NotNull` contains all non-nullable types. For example, we could set `NotNull = AnyVal|AnyRef`.

- *Limitation of flow typing.* The idea of this class was to capture situations where our implementation of flow typing is too primitive. In practice, we only found one error in this class, which is due to a bug in our implementation that is not yet fixed. When migrating some of the libraries under configurations that are more error-prone than `optimistic`, we saw additional errors that fall within this class. The more-common one was due to flow typing not having support for local `vars`.

Flow Typing

I would like to offer a few hypotheses for the disappointing (lack of) results of flow typing:

- First, it seems plausible that flow typing would be more useful for `vars`, which we do not support, than for `vals`, which we support. This is because state (local variables, fields) that starts out as `null` does not remain so for the duration of the program: it is eventually mutated to be non-null. In order to handle these cases, we would need support for `vars`.

- Some of the instances where flow typing might be useful are probably "hidden" by the use of the `optimistic` configuration. For example, in the snippet

  ```
  val s = javaMethodReturnsString()
  if (s != null) {
    val l = s.length
  }
  ```

  With `optimistic` or even `JavaNull`, the assignment `val l = s.length` would typecheck, even in the absence of flow typing.

- Null checks à la `if (x != null)` might not be very common in Scala. Instead, it seems that Scala programs often wrap nullable values flowing from Java using the `Option` idiom previously described.

Takeaways

Below I sketch some changes we could make to our design that are motivated by the evaluation results.

- **Adding an `optimistic` mode**. Table 2.3 shows a large variability in the number of errors per library. If we take the number of errors as a proxy for the effort required to migrate a library, then some libraries like `squants` will be easy to migrate (no changes required), while others like `stdlib123` will take substantial effort (because it is both a large library and `null` errors are relatively frequent). One way to make the migration of a library like `stdlib123` easier would be to have a "loose" or "optimistic" configuration as part of our design. Such a configuration would behave like `optimisic`, but could possibly be even more permissive.

  For example, we could make `T|JavaNull` a subtype of `T`. From a theoretic point of view, this change would be problematic, because it goes against our intuition of types as sets of values. Informally, this rule would express the idea that when we see a Java method

  ```
  String foo(String s)
  ```

  unless we have annotations that make the nullability of `foo`'s argument and return type clear, we really *do not know* whether e.g. the return type is nullable or not. The type `T|JavaNull` would express this idea: `T` might or might not be nullable. We would then have three kinds of nullability: `T` (non-nullable type), `T|Null` (nullable type), and `T|JavaNull` (nullability unknown).

Assigning a `T|JavaNull` to a `T` would typecheck, but perhaps generate a warning and a runtime check. Assigning a `T` to a `T|JavaNull` would always be sound, so no warnings would be needed. This is the idea behind *platform types* in Kotlin (see Section 2.8.1).

- **Supporting annotations within Java generics**. Table 2.4 shows that Java generics with nullable inner types are not common, but do occur sometimes. *Type annotations* [Oracle], introduced in Java 8, allow us to mark a generic's inner type as non-nullable. We should add support for type annotations.
- **Relaxing override checks**. Table 2.4 shows that instances of Scala code overriding Java methods do occur. We should revisit our decision to drop relaxed override checks, from Section 2.6.2.

Threats to Validity

It is difficult to accurately estimate the effort that a code migration will take without carrying out the migration. Table 2.3 shows error counts for the different configurations, but the results are only a lower bound. If we tried to carry through the migration, we would find additional errors due to different reasons:

- Some of the libraries consist of multiple modules, some of which depend on others as dependencies. If a module A depends on B, and B fails to compile because of nullability errors, the compiler will sometimes not report the errors in A (because it did not even try to build A).
- Some nullability errors only appear in later phases of the compiler. For example, overriding-related errors are only reported if the program typechecks.
- Even reported errors might be masking other would-be errors in the same phase. For example, suppose we have a symbol definition `val s: String` whose type needs to be changed to `String|Null`. Any other symbol that s is assigned to, e.g. `val s2: String = s`, will need to have its type adjusted as well (generated another error).

Reproducibility

The way in which I generated the tables in this chapter is described in Appendix A.

2.7.2 An Interesting But Hard Question

Recall the first question at the beginning of Section 2.7, which was about how effective explicit nulls are in preventing runtime errors. This first question is perhaps the more interesting one, if only because it would be useful to know how beneficial explicit nulls are *before* deciding whether the pain of a migration is justified. We know that explicit nulls are useful, because they close a soundness hole in the Scala type system, but determining just *how* useful they are is trickier.

One can imagine a possible experiment: pick a few Scala libraries, migrate them to explicit nulls, and then look for uses of nullable variables. Every use of a nullable variable not preceded by a null check is a potential `NullPointerException`-in-waiting. Unfortunately, this analysis is complicated by two factors:

- Our type system conservatively assumes that the return types of Java methods are nullable. This leads to false positives. We can eliminate some of the false positives by looking at nullability annotations (as in Section 2.3.4), or by consulting the documentation of the Java method in question. However, sometimes the documentation is not explicit about whether the return value can indeed be `null`. For example, the Java class `java.net.URL`[6] defines two `getContent` methods with the following signatures:

  ```
  public final Object getContent() throws IOException
  public final Object getContent(Class[] classes) throws IOException
  ```

  The documentation for the *second* method specifies that the returned value can be `null`. However, the first method's documentation just says

  ```
  Returns:
    the contents of this URL.
  ```

  It is unclear, without looking at `getContent`'s implementation, whether we should interpret its return type as `Object` or `Object|Null`.

- Our type system does not enforce sound initialization patterns. That is, even when e.g. a class field starts out as `null`, it might become non-null, and therefore safe to access later on. For example, the following fragment from Dotty shows a nullable field `reductionContext` that starts out as `null`, is initialized in a method `updateReductionContext` and accessed in another method `isUpToDate`:

[6] https://docs.oracle.com/javase/7/docs/api/java/net/URL.html

```
private var reductionContext: mutable.Map[Type, Type] = null
...
def updateReductionContext(): Unit = {
  reductionContext = new mutable.HashMap
  ...
}
...
def isUpToDate: Boolean =
  reductionContext. keysIterator . forall  {...}
```

As long as `updateReductionContext` is always called by the time `isUpToDate` is called, all is well. However, this is hard to manually check.

Both of the problems above can be sometimes solved by manual inspection, which does not scale. Designing an experiment to evaluate the effectiveness of explicit nulls remains future work.

2.8 Related Work

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

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

2.8.1 Nullability in the Mainstream

Kotlin

Kotlin is an object-oriented, statically-typed programming language for the JVM [Kotlin Foundation, a]. Because Kotlin, like Scala, targets the JVM, it needs to address many of

the issues surrounding Java interoperability and nullability that I have described in this chapter. Indeed, the design of Kotlin was a source of inspiration for us as we set up to retrofit Scala with a type system for nullability.

Table 2.5 compares the support for explicit nulls in Scala and Kotlin. In summary, Kotlin has more mature support; specifically, its flow typing handles more cases, and Kotlin can recognize more kinds of non-null annotations. I highlight some of the differences below.

Nullable types Like in Scala with explicit nulls, reference types are non-nullable by default in Kotlin. Nullability can be recovered with special syntax: if `T` is a non-nullable type, then `T?` is `T`'s nullable counterpart. By contrast, in Scala, nullability is expressed using union types (which Kotlin does not have).

Flow typing Kotlin's flow typing can handle not only `vals`, but also local `vars`. In order to handle `vars`, Kotlin needs to "kill" (in the sense of dataflow analysis) flow facts about `vars` that are mutated. For example

```
// Kotlin
var b: String? = null

if (b != null) {
  val len = b.length // ok: `b` is non-null
  b = null
  val len2 = b.length // type error: flow fact about `b` was killed
}
```

Similarly, the Kotlin compiler needs to detect `vars` that have been captured by closures and not do flow typing on them (since such `vars` could be unexpectedly mutated by the closure).

```
// Kotlin
var b: String? = null

if (b != null) {
  val f = { x: Int -> b = null } // closure that captures `b`
  val len = b.length // type error: Smart cast to 'String' is impossible,
  // because 'b' is a local variable that is captured by a changing closure
}
```

Detecting when local variables are captured by closures is hard in Scala, because Scala allows forward references to `defs`:

```
var a: String|Null = ???
if (a != null) {
  mutateA() // forward reference is allowed
  a.length  // should issue a type error
}
def mutateA(): Unit = { a = null }
```

In the example above, `mutateA` sets the local variable to `null` *after* the test in the `if` expression. The problem is that call to `mutateA` is a forward reference, and the body of `mutateA` is not typed until *after* we have processed the `if`. This means that by the time when we have to decide whether to allow `a.length`, we do not yet know whether `a` is captured or not.

Platform types Kotlin handles Java interoperability via *platform types*. A platform type, written `T!`, is a type with *unknown* nullability. Kotlin turns all Java-originated types into platform types. For example, if a Java method returns a `String`, then Kotlin will interpret the method as returning a `String!`. Given a type `T!`, Kotlin allows turning it (automatically) into both a `T?` and a `T`. The cast from `T!` to `T?` always succeeds, but the cast from `T!` to `T` might fail at runtime, because the Kotlin compiler automatically inserts a runtime assertion that the value being cast is non-null. Informally, the Kotlin type system includes the following two subtyping rules: `T! <: T` and `T! <: T?`. Additionally, Kotlin allows member selections on platform types, just like we do in Scala via `JavaNull`. In fact, we can think of platform types as a generalization of `JavaNull` that allows not only member selections, but also subtyping with respect to non-nullable types.

Swift

Swift is a statically-typed programming language, originally designed for the iOS and macOS ecosystems [Apple Inc, a]. Swift has a `nil` reference, which is similar to `null` in Scala [Apple Inc, b]. Types in Swift are non-nullable by default, but one can use *optionals* to get back nullability. For example, the type `Int?`, an optional, is either an integer or `nil`. Optionals can be *force unwrapped* using the `!` operator, which potentially raises a runtime error if the underlying optional is `nil`. Swift also has a notion of *optional binding*, which is similar to the monadic `bind` operator in Haskell [Wadler, 1995], but specialized for optionals. Additionally, Swift has *implicitly unwrapped optionals*, which are similar to

	Scala with explicit nulls	Kotlin
types are non-nullable by default	yes	yes
can express nullable types	yes, using union types (e.g. `String|Null`)	yes, with special syntax (e.g. `String?`)
special nullability operators	yes, through library support: assert non-null (`.nn`)	yes, through compiler internals: assert non-null (`!!`), safe-call (`??`), and "Elvis operator" (`?:`)
flow typing	only for `vals`	supports both `vals` and `vars` not captured by closures
java interoperability	nullification using `JavaNull`	through *platform types*
arrays with non-nullable elements	not enforced by the type system, since arrays can be created with uninitialized elements	enforced by the type system, because non-nullable arrays must be initialized when the array is created
nullability annotations	recognized only at the top-level: e.g. `Array[String] @NonNull` is handled, but not `Array[String @NonNull] @NonNull`	recognized everywhere, including in type arguments

Table 2.5: Comparison of explicit nulls in Scala and Kotlin

Kotlin's platform types. That is, the type `Int!`, an implicitly unwrapped optional, need not be forced unwrapped explicitly before a value can be retrieved, but if the underlying value is `nil`, it will produce a runtime error.

C#

Versions of C# prior to 8.0 supported only *nullable value types* [Microsoft, b]: these are types of the form `Nullable<T>`, which could contain either a `T` or `null`. However, `T` could not be a reference type, only a value type. There is flow typing and a *null coalescing* operator `??`, such that `a ?? b` evaluates to `a` unless the left-hand side is `null`, in which case the entire expression evaluates to `b`. This is the same as Kotlin's "Elvis" operator (shown in Table 2.5).

Version 8.0 extends nullable types so that they can be used with references as well [Microsoft, a]. The syntax is familiar: `string?` denotes a nullable `string`. Interestingly, the compiler issues *warnings* as opposed to errors if a nullable reference is dereferenced, unless flow typing can show that the underlying value is non-null.

This last version of the language *changes* the semantics of *regular* reference types. While in previous versions reference types (e.g. `string`) were nullable, they are now *non-nullable*. Swift and Kotlin are relatively recent languages, and were designed from the start with non-nullable references. By contrast, C# needs a mechanism to migrate *existing* codebases into the world of nullable references, because of the new semantics. The migration is managed through two different nullable *contexts*. We can think of each context as an (independent) flag that can be turned on or off at the project, file, or line range level:

- Nullable *annotation* context: if this context is *enabled*, then regular reference types are non-nullable, and the user can write nullable references using the `?` syntax (e.g. `string?`). If the annotation context is *disabled*, then C# uses the pre version 8.0 semantics where nullable types are nullable, and the `?` syntax is *not* available.
- Nullable *warning* context: if this context is *enabled*, then the compiler will issue a warning if we try to dereference a reference that is classified as "maybe nullable", among other cases. Notice that this flag can be enabled even when the annotation context is *disabled*. In that case, the user cannot write a type like `string?`, but the compiler will still issue warnings if a reference of type `string` is accessed, but cannot be proven to be non-null. If the warning context is disabled, then compiler will not issue warnings in the presence of unsafe accesses.

Summarizing, the *annotation* context controls the type system, and the *warning* context turns the static analysis on and off.

Compared to our implementation of explicit nulls in Scala, C# offers more fine-grained control over where the type system is enabled. In our system, explicit nulls can only be enabled or disabled at the project level. In C#, the user can additionally opt-in to explicit nulls via "pragmas" (program metadata). For example, the code below enables *both* the annotation and warning contexts for the code contained within the `#nullable` pragmas, and then *restores* the previous state of both contexts upon exiting the delimited range:

```
#nullable enable
// Both contexts enabled here
...
#nullable restore
```

2.8.2 Functional Programming

In functional programming languages such as Haskell [O'Sullivan et al., 2008] and OCaml [Minsky et al., 2013], the absence of values is usually handled using an *abstract data type* (ADT) with two constructors, one of which (typically named `None` or `Nothing`) corresponds to `null`. For example, in OCaml this ADT is called `option`, and is thus defined

```
type 'a option =
  Some of 'a
| None
```

Notice that `option` takes a type argument `'a`, and so `None` has type `'a option` *for any* `'a`. In particular, `None` does *not* have type `'a`. That is, the only terms that can evaluate to `None` are those with type `'a option`. The OCaml compiler can then make sure that all such terms are deconstructed via *pattern matching* before the underlying value can be examined. During pattern matching, *both* the `Some` and `None` cases need to be handled, so there is no possibility of runtime errors. For example, in OCaml the following expression evaluates to `42`.

```
match (None: int option) with
| Some num -> num
| None -> 42
```

A common pattern is to have a sequence of computations, each producing an optional value and depending on the previous ones. The naive approach for sequencing these computations leads to deeply nested pattern matching expressions, which is hard to understand.

For such cases, some functional languages like Haskell, Agda [Bove et al., 2009], and Idris [Brady, 2013] provide *syntactic sugar* in the form of *do notation* [Wadler, 1995], to make the sequencing more succint and understandable. For example, in Haskell we would write

```
f :: int -> Maybe int
g :: int -> Maybe int
h :: int -> Maybe int

do x <- f 42
   y <- g x
   z <- h y
   return z
```

Here we have sequenced three functions, `f`, `g`, and `h`, each producing an optional int value (`Maybe int` in Haskell). If any of the functions fails to produce a value (returns `Nothing`), then the result of the `do` expression is also `Nothing`. Otherwise, the returned values are threaded as `x, y`, and `z` for the subsequent function to consume. The type of the entire expression is then `Maybe int`.

Scala, being a functional language, *also* has an `Option[T]` type. It even has a form of do notation via *for-comprehensions* [Odersky et al., 2004]. Unfortunately, because of the JVM baggage, reference types used to be nullable and so a value of type `Option[T]` can in fact have three forms: `Some x`, `None`, or `null`!

2.8.3 Sound Initialization

As mentioned in Section 1.3, even with a type system that has non-nullable types, there is a possibility of unsoundness because of *incomplete initialization*. Specifically, in mainstream object-oriented languages, it is possible to manipulate an object *before* it is fully-constructed. This can happen, for example, due to dynamic dispatch, or *leaking* of the `this` reference from the constructor to helper methods. The problem is that in an unitialized (or *partially* uninitialized) object, the *invariants* enforced by the type system need not hold yet. Specifically, fields that are marked as non-null might *nevertheless* be `null` (or contain nonsensical data) because they have not yet been initialized.

Over the years, many solutions have been proposed to this initialization problem, usually involving a combination of type system features and static analyses. These prior designs include *raw* types [Fähndrich and Leino, 2003], *masked* types [Qi and Myers, 2009], *delayed* types [Fähndrich and Xia, 2007], the "Freedom Before Commitment" (FBC) scheme [Summers and Müller, 2011], and X10's "hardhat" design [Zibin et al., 2012].

Even though this chapter is not about sound initialization, I would like to describe in some detail the FBC and X10 approaches. These two schemes have been identified in Liu et al. [2018] as the bases for a sound initialization scheme for Scala.

Freedom Before Commitment

Freedom Before Commitment is presented in Summers and Müller [2011]. FBC is a sound initialization scheme for object-oriented languages (like Java or C#), that strikes a balance between expressivity (it can correctly handle realistic initialization patterns, like cyclic data structures) and ease-of-use (it requires fewer annotations than e.g. delayed types). The initialization state of expressions is tracked via their types. For example a non-nullable string that is fully initialized will have type `committed String!`, while a non-nullable string that that is potentially uninitialized has type `free String!`.

FBC can be combined not only with a type system for nullability, but in general can be used to track any invariant that is *monotonic*: that is, any invariant that is established during object construction, and continues to hold thereafter (another such invariant would be *immutability*). Initialization with respect to *null* is monotonic because once a (non-null) field has been assigned a value, the type system guarantees that its value is non-null (because only assignments to non-null values are allowed).

In FBC, objects are in one of two (logical) initialization states: *under initialization*, or *initialized*. The program point when an object changes from under initialization to initialized is called the *commitment point*. Commitment points happen when certain *new* expressions terminate.

Initialized objects must have all their fields initialized in a *recursive* manner. That is, if we start at an initialized object, all other objects that we encounter while traversing the graph of object references must *also* be initialized. This they call the *deep initialization guarantee*.

FBC tracks both nullability and initialization state of expressions via types. As usual, types are either *non-nullable* or *nullable*, denoted with a `!` or `?` suffix, respectively. For example, nullable strings are denoted by `String?`. Initialization state is conveyed via *initialization modifiers*, which are also part of types. There are three modifiers:

- *committed*: the relevant expression evaluates to an initialized object.
- *free*: the expression evaluates to an object under initialization.

- *unclassified*: the expression can evaluate to an object that is either initialized, or under initialization. This is a "super type" of the previous two (e.g. `committed C! <: unclassified C!`).

The initialization state is independent from the nullability of the expression. This means we have $2 \times 3 = 6$ possible "variants" of the same underlying class type `C`.

Their system limits the number of annotations required by specifying defaults. For example, most expressions are committed, non-null by default. The `this` reference is implicitly free within the constructor, but committed within methods. Methods must indicate initialization state of their parameters.

Field updates `x.f = y` are allowed only if `x` is free, or `y` is committed. For example, the assignment is not allowed if `x` is committed and `y` is free, because this would violate the deep initialization guarantee (we would have an initialized object that points to an uninitialized one).

On field reads, initialization interacts with nullability. For example, if we have a read `x.f` where `x` is free and `f` has type `C!` (non-null), then the resulting type is `unclassified C?`, because in a free object we have no guarantees that the field invariants hold.

Commitment points happen when we have a `new` expression where *all arguments to the constructor call* are already committed. Specifically, *all objects* created during that constructor call move to the committed state after the `new` terminates. The typing rule is simple: given a constructor call `new` $C(a_1, \ldots, a_n)$, we have two options:

- if all the a_i are *committed*, the type of the call is `committed C!`
- otherwise, the type is `free C!`

Intuitively, this rule is sound because if a constructor's arguments are all `committed`, then the constructed object is unable to reference any outside uninitialized objects.

Finally, the authors tested FBC by implementing their type system on top of the Spec# compiler [Barnett et al., 2004], a version of C# enhanced to allow program verification (and now superseded by the Dafny [Leino, 2010] language). The authors then migrated two large Spec# projects to the FBC scheme, as well as several smaller examples. In total, they migrated around 60K LOC, which required 121 initialization annotations when ported to the new system.

Object Initialization in X10

X10 is an object-oriented programming language designed for parallel computing, developed at IBM [Charles et al., 2005]. Instead of carefully tracking the initialization state of expressions via the type system, like in FBC, X10 takes a different approach. X10 has a "hardhat" design that *prohibits* leaking the `this` reference or doing dynamic dispatch during object construction [Zibin et al., 2012]. As a result, the language is safe from initialization errors, but *cannot* create cyclic immutable data structures.

Additionally, the language guarantees that the user will never be able to read *uninitialized fields*. Because of this, the runtime does not need to zero-initialize data structures. This is in contrast with FBC, where the user can read uninitialized fields, which results in a type with a `free` tag (but does require the runtime to e.g. set all fields of an object to `null` before the constructor runs). Interestingly, the cost of zero-initialization has been reported to be non-negligible in some applications [Yang et al., 2011].

As mentioned, the paper identifies two important sources of initialization errors: leaking `this` and dynamic dispatch. The hardhat design addresses these issues with two annotations on methods:

- `NonEscaping`: the method is not allowed to leak `this`.
- `NoThisAccess`: the method cannot *access* `this`. This is more restrictive than `NonEscaping`.

The main restrictions in the system are as follows:

- Constructors and non-escaping methods can only call other non-escaping methods. Non-escaping methods must be `private` or `final`, so they cannot be overridden.
- Only methods marked with `NoThisAccess` can be overridden, but dynamic dispatch is not a problem in this case, because the methods cannot leak `this`.

Additionally, they implement an inter-procedural, intra-class static analysis for ensuring that a field is read only after it has been written to, and that final fields are assigned exactly once. For final fields, they compute for each non-leaking method which final fields the method (transitively) reads. The non-leaking method can only be called from the constructor if the right final fields have been set. A similar process happens for non-final fields, except that this one requires a fixpoint computation.

Their main result is migrating 200K LOC (the X10 compiler) to the new system. By setting the right default annotations on methods, the migration required adding only 104 annotations.

2.8.4 Pluggable Type Checkers

Another line of work that is relevant to nullability is *pluggable* type checkers. A pluggable type checker is a custom-built typechecker that *refines* the typing rules of a host system [Papi et al., 2008]. That is, the pluggable type checker *rules out* programs that the "built-in" type system considers valid, but that for some reason need to be classified as type-incorrect. Nullability is a fitting example of a property that a built-in type system, such as Java's, might not take into consideration, but that a user might want to check using a pluggable type checker.

A typical workflow for a pluggable type system might look as follows:

- The programmer identifies a property of interest that they would like to enforce via a type system (for example, that non-nullable fields are never assigned a `null` value).
- Using one of the frameworks mentioned below, the programmer then writes a pluggable type system. Because of facilities provided by the framework (e.g. ability to write a typechecker using a declarative specification), this step is usually significantly less involved than writing a full-blown type system (say, the type system of the host language).
- Once the custom type checker is written, the user adds metadata to the source code. Typically, this metadata comes in the form of domain or problem-specific *annotations*. For example, in order to use the type checker for nullability, the user might annotate certain fields or method arguments as `@NonNull`, using for example Java annotations.
- The pluggable type checker can then be run on the annotated code, and reports warnings or errors. False positives or negatives are possible, so manual inspection of the results might be required.

Checker Framework

The Checker Framework [Papi et al., 2008] is a framework for building pluggable type checkers for Java. Users have the option of writing their typecheckers in a *declarative* style, which requires less work (they do not need to write Java code) but is less expressive, or in a *procedural* style, where the checker can have arbitrarily complex logic, but is therefore harder to implement. Many checkers have been built using the Checker Framework; see Papi et al. [2008], Dietl et al. [2011], Brotherston et al. [2017], among others.

One of the checkers that comes "pre-packaged" with the framework is the Nullness Checker. In fact, "the Nullness Checker is the largest checker by far that has been built with the Checker Framework" [Dietl et al., 2011]. As of 2017, the Nullness Checker implemented a variant of the Freedom Before Commitment scheme, as well as support for flow typing and multiple heuristics to improve the accuracy of its static analysis [Brotherston et al., 2017, Dietl et al., 2011].

Dietl et al. [2011] conducted an extensive evaluation of the Nullness Checker (among others) in production code, finding multiple errors in the Google Collections library for Java.

Granullar

The Granullar project [Brotherston et al., 2017] combines the null checker from the Checker Framework with techniques from *gradual typing* [Siek and Taha, 2006]. Granullar allows the user to migrate only part of a project to use null checks. To that effect, the code under consideration is divided into *checked* and *unchecked* regions.

Nullability checks are done statically within the checked region, using the Freedom Before Commitment scheme implemented by the Checker Framework. No checks are done for the unchecked portion of the code.

The contribution in Granullar is *insulating* the checked region from unsafe interactions with the unchecked region, by inserting *runtime* non-null checks at the boundary. To this effect, they define *three* nullability annotations: the usual `NonNull` and `Nullable`, and an additional `Dynamic` annotation that means "unknown nullability". The subtyping is `NonNull <: Dynamic <: Nullable`.

Values that flow from unchecked code to checked code are automatically tagged as `Dynamic`. The tool then detects conversions from `Dynamic` to `NonNull` values (generated, for example, from assignments and calls into unchecked code), and allows them, but also generates runtime non-null checks. The treatment of `Dynamic` in Granullar is similar to Kotlin's platform types. Additionally, Granullar ensures that calls from unchecked code to checked code are similarly checked at the boundary at runtime.

NullAway

NullAway is a nullness checker for Android applications, developed at Uber [Banerjee et al., 2019]. NullAway is implemented as a pluggable type system on top of the Error Prone

framework [Aftandilian et al., 2012]. In contrast with the Nullness Checker described above, NullAway trades away soundness for efficiency. Specifically, the tool is *unsound* in multiple ways:

- NullAway's custom initialization scheme ignores the problem of leaking the `this` reference.
- In interactions with unchecked code (for example, unannotated libraries), the tool makes *optimistic* assumptions. That is, all unchecked methods are assumed to return *non-null* values, and be able to handle nullable arguments.
- NullAway's flow typing assumes that all methods are *pure* (free of side effects) and *deterministic* (e.g. a getter method will always return the same value).

In exchange for the unsoundness, NullAway has a lower build-time (2.5×) and annotation overheads than similar tools (2.8 − 5.1×) [Banerjee et al., 2019]. Perhaps more surprisingly, the authors note that after extensive use of the tool in both open-source code and Uber's internal codebase (including integration of the tool in their continuous integration pipeline), none of the detected `NullPointerExceptions` were due to the unsound assumptions made by NullAway. Instead, the exceptions happened "due to interactions with unchecked code, suppression of NullAway warnings, or post-checking code modification [reflection]" [Banerjee et al., 2019].

2.9 Conclusions

In this chapter, I described a modification to the Scala type system that makes nullability explicit in the types. Reference types are no longer nullable, and nullability can be recovered using type unions. Because interoperability with Java is important, a type nullification phase translates Java types into Scala types. A simple form of flow typing allows for more idiomatic handling of nullable values. I implemented the design as a modification to the Dotty (Scala 3) compiler. To evaluate the implementation of explicit nulls, I migrated several Scala libraries to use the new type system. The results of the evaluation suggest some improvements to the current implementation, including a mechanism to ease the migration of Scala libraries that have many Java dependencies.

Chapter 3

Denotational Semantics of Nullification

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

3.1 Reasoning about Nullification with Sets

As we set out to design a version of Scala with explicit nulls, there was nothing I wished for more than to be able to completely disregard the question of what to do about Java code. Alas, this is a question that must be answered, because one of Scala's big selling points is its ability to use Java code, and null remains implicit in Java. To be concrete, suppose we have a Java class representing movies

```
class Movie { public String title ; }
```

We then want to write Scala code for testing whether a movie is good.

```
def isGood(m: Movie): Bool = m.title.length > 15
```

What should the type of `m.title` be? In other terms, if we call our type nullification function F_{null}, what is $F_{\text{null}}(\texttt{String}_j)$[1]. We can try a few alternatives:

- The lazy approach would be to make F_{null} the identity, and say that $F_{\text{null}}(\texttt{String}_j) = \texttt{String}_s$. However, this potentially leads to runtime errors because if `m.title` is indeed `null`, then `m.title.length` fails with an exception. That is, forgetting to include `null` in the result set leads to *unsoundness*.

- Another option is to say that $F_{\text{null}}(\texttt{String}_j) = \texttt{Any}$, a Scala type that is a super type of all other types. This is safe (will not lead to runtime exceptions), but `m.name.length` will no longer typecheck, because `Any` has no field named `length`. Making the result set too large leads to *loss of precision*.

- Both underapproximating and overapproximating the set of values contained in the type leads to problems, so what if we designed F_{null} so that the representable values remain unchanged? We can do this using union types by setting $F_{\text{null}}(\texttt{String}_j) = \texttt{String}_s|\texttt{Null}$. This is the case because anything typed as $\texttt{String}_j$ by the Java type system is either a string literal *or* `null`. If we think of types as sets, the type `String|Null` contains exactly the set of values $\{\texttt{null}\} \cup \texttt{strings}$, where `strings` stands for the set of all string literals.

  Once again, the method above will no longer typecheck, because `String|Null` has no field named `length`. However, this time we can fix things by handling the `null` case explicitly:

  ```
  def isGood(m: Movie): Bool = m.title != null && m.title.length > 15
  ```

  This works because if we know that $m \in \{\texttt{null}\} \cup \texttt{strings}$ but $m \neq \texttt{null}$, then we must have $m \in \texttt{strings}$, and it is safe to access `m.title.length`.

Now that we have established that $F_{\text{null}}(\texttt{String}_j) = \texttt{String}_s|\texttt{Null}$, what should we do about other types? As I have shown in Section 2.3.2, we can carry out a similar analysis for each of the "kinds" of Java types (value types, methods, generics, etc.). The important property is that *for every type T, we want the values contained in T and $F_{null}(T)$ to be the same.*

[1] We will differentiate between Java and Scala types with the same name by subscripting them with $_{\text{j}}$ and $_{\text{s}}$, respectively

$s, t ::=$	**Terms**	$S, T ::=$	**Types**
$\lambda(x\colon T).s$	abstraction	$S \to T$	function
$s\ t$	application	$\Pi X.S$	universal
$\Lambda\alpha.s$	type abstraction	X	type variable
$s\ [T]$	type application		
x	variable		

Figure 3.1: Terms and types of System F

It is this last claim that we set out to prove in this chapter, for a particular choice of nullification function F_{null}, and with the usual caveat that the theorems that follow apply not to Java and Scala, but to simplified type systems that model some of the features of their real-world counterparts. Beyond the specifics, however, I want to show how we can use the language of denotational semantics to put the correctness arguments we saw before onto solid formal footing. The first step is to define these simplified type systems.

3.2 System F, λ_j, and λ_s

We will model the Java and Scala type systems as variants of System F [Reynolds, 1974], the polymorphic lambda calculus. Figure 3.1 shows the terms and types of System F. The defining property of this calculus is its ability to abstract not only over terms but also over types. The precise mechanism is described by the typing rules below for type abstraction and type application.

$$\frac{\Gamma, X \vdash s : S}{\Gamma \vdash \Lambda X.s : \Pi X.S}\ \text{(SF-TAbs)} \qquad \frac{\Gamma \vdash s : \Pi X.S}{\Gamma \vdash s\ [T] : [X/T]S}\ \text{(SF-TApp)}$$

The specific variant of System F that we use is *predicative*, which in this case means that in the universal type $\Pi X.S$, X ranges over all types that are *not themselves universal*. For example, given a term s of type $\Pi X.S$, $s\ [Y]$ and $s\ [Y \to Z]$ would be valid type applications, but $s\ [\Pi U.U]$ would not, because the latter uses a type argument ($\Pi U.U$) that is itself a universal type. By contrast, in the (more common) *impredicative* version of the calculus, X ranges over all possible types, so applications like $s\ [\Pi U.U]$ are perfectly valid. By using the predicative variant, we incur a loss of expressivity: notably, we can no longer typecheck recursive data structures (which are ubiquitous in both Java and Scala). On the other hand, giving a denotational semantics for the predicative variant is much

easier, because one can use a semantic model purely based on sets. I will say more about this in Section 3.3.

Even though the predicative variant of System F is simpler than either Java's or Scala's type system, it is nevertheless a useful starting point for modelling both of them *for the purposes of nullification*. There are at least two reasons for this:

- Nullification turns Java types into Scala types, so when defining nullification we mostly need to worry about Java types, which are simpler. For example, we do not need to define F_{null} on the type `class Monad[M[_]]`, because Java does not support higher-kinded types.
- The main difficulty in designing nullification was handling Java generics. As it turns out, given a generic such as `class List<T>`, Java only allows instantiations of `List` with a *reference* type *that is not itself generic*. For example, `List<String>` is a valid type application, but `List<List>` is not. This is precisely the kind of restriction imposed by predicative System F!

That said, System F is too spartan: it lacks a distinction between value and reference types, does not have records (present in both Java and Scala), and does not have union types (needed for explicit nulls). To remedy this we can come up with slight variations of System F that have the above-mentioned features. I call these λ_j ("lambda j") and λ_s ("lambda s"), and they are intended to stand for the Java and Scala type systems, respectively. Figures 3.2 and 3.3 show the types of these two calculi. From now on we will focus solely on the types and will forget about terms: this is because nullification is a function from types to types.

3.2.1 λ_j Type System

As previously mentioned, the λ_j type system models elements of the Java type system, notably generics and implicit nullability. λ_j extends System F with several types. These are informally described below:

- The $\texttt{int}_j$ type contains all integers; that is, if $n \in \mathbb{Z}$, then $\Gamma \vdash n : \texttt{int}_j$. $\texttt{int}_j$ is a *value type*, so it is non-nullable, meaning that the judgment $\Gamma \vdash \texttt{null} : \texttt{int}_j$ does not hold.
- The $\texttt{String}_j$ type contains all string literals (like `""` and `"hello world"`), *plus* the value `null`, so it is implicitly nullable.

$$
\begin{array}{rl}
S, T ::= & \textbf{Types} \\
 & \texttt{int}_j \quad \text{int} \\
 & \texttt{String}_j \quad \text{string} \\
 & S \times_j T \quad \text{product} \\
 & S \to_j T \quad \text{function} \\
 & \Pi_j(X :: *_n).S \quad \text{generic} \\
 & \text{App}_j(S, T) \quad \text{type application} \\
 & X \quad \text{type variable}
\end{array}
$$

$$
\begin{array}{rl}
K ::= & \textbf{Kinds} \\
 & *_n \quad \text{kind of nullable types} \\
 & *_v \quad \text{kind of non-nullable types} \\
 & * \quad \text{kind of proper types} \\
 & *_n \Rightarrow K \quad \text{kind of type operators}
\end{array}
$$

$$
\begin{array}{rl}
\Gamma ::= & \textbf{Contexts} \\
 & \varnothing \quad \text{empty context} \\
 & \Gamma, X :: *_n \quad \text{nullable type binding}
\end{array}
$$

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

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

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

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

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

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

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

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

Figure 3.2: Types, kinds, and kinding rules of λ_j. Differences with System F are highlighted.

- A *product* type $S \times_j T$ contains all pairs where the first component has type S, and the second has type T. It is also implicitly nullable, so that $\Gamma \vdash \texttt{null} : S \times_j T$ holds for all S and T that are well-kinded. Product types are implicitly nullable because they emulate Java classes, which are also implicitly nullable: e.g. if we have a variable `Movie m`, where `Movie` is a class, then `m` can be null.

- A *generic* type $\Pi_j(X :: *_n).S$ is conceptually a function that maps types to types. This kind of type represents generic Java classes. What is different from System F here is that X can only be instantiated with *nullable* types (where nullability is determined by the kind system, as described below). This matches Java's behaviour: in Java, if `class List<T>` is a generic class, we can instantiate the generic with `List<String>` and `List<Person>`, but not with `List<int>`, because `int` is non-nullable.

- Finally, $\text{App}_j(S, T)$ is a *type application*. Following the intuition about generics being functions from types to types, type applications represent the application of the function associated with S to the type argument T. System F doesn't have type applications at the type level; instead, the user can write the type application as a *term* $s\ [T]$. Java however does have explicit type applications: e.g. `List<String>`, so we will find it useful to have explicit type applications at the type level in order to model Java more closely.

Figure 3.1 shows some Java types and their corresponding representation in λ_j. Some Java types (notably recursive classes) do not have counterparts in λ_j.

Kinding Rules

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

The *kinding* rules in Figure 3.2 fulfill precisely this purpose. The judgment $\Gamma \vdash_j T :: K$ establishes that type T has kind K under context Γ, and is thus well-kinded. Further, we can look at K to learn something about T:

- $K = *_n$. In this case, we say that T is a *nullable* type. If $[\![T]\!]_j$ is the set of values that have type T, we expect $\texttt{null} \in [\![T]\!]_j$ (we will make this statement precise when

Java type	λ_j type	Comments
`int`	$\texttt{int}_j$	non-nullable
`Integer`	—	no nullable integers in λ_j
`String`	$\texttt{String}_j$	

class Movie {
String director;
int releaseYear;
}
	$\texttt{String}_j \times_j \texttt{int}_j$	λ_j, like System F, is a *structural* type system, as opposed to Java, which is *nominal*.

class Pair<A, B> {
A first;
B secondl
}
	$\Pi_j(A :: *_n).\Pi_j(B :: *_n).A \times_j B$	

`Pair<String, Movie>`	$\star$ `PairApp` (see below)	because λ_j is structural, we need to inline all type definitions every time

class List<T> {
T first;
List<T> next;
}
	—	λ_j does not have recursive types

class Box< ? **extends** Movie> {}	—	λ_j does not have wildcards or subtyping

$\star$ `PairApp` $\equiv \text{App}_j(\text{App}_j(\Pi_j(A :: *_n).\Pi_j(B :: *_n).A \times_j B, \texttt{String}_j), \texttt{String}_j \times_j \texttt{int}_j)$

Table 3.1: Java types and their corresponding λ_j types

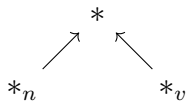
we give denotations to types in Section 3.3.3). Inspecting the kinding rules, we can notice that KJ-String marks $\texttt{String}_j$ as nullable, and similarly KJ-Prod marks all product types as nullable. Notice that all type variables in a context are nullable as well.

- $K = *_v$. This means that T is *non-nullable*, which will later turn out to mean that $\texttt{null} \notin [\![T]\!]_j$. Only $\texttt{int}_j$ and function types are non-nullable. The idea behind making function types non-nullable is to model the fact that in Java methods cannot be null: e.g. in

  ```
  class Movie { String getTitle(String language) {...} }
  Movie m = new Movie();
  m.getTitle("Spanish")
  ```

  there cannot be a null pointer exception due to `getTitle` being null.

- $K = *$. T is a *proper* type in this case. These are non-generic types. Syntactically, $*$ is a "superkind" (think "supertype", but for kinds) of $*_n$ and $*_v$. We could draw an "inheritance" diagram that looks like

  The KJ-Null and KJ-NonNull rules ensure the above. We need $*$ so we can form products and functions of both nullable and non-nullable types.

  Semantically, we will show that the set of values in $*$ is exactly the union of those in $*_n$ and $*_v$: $[\![*]\!]_j = [\![*_n]\!]_j \cup [\![*_v]\!]_j$.

- $K = *_n \Rightarrow K'$. In this case, we say T is a *generic* type. Two points of note: in the premise of KJ-Pi, we make sure that the context is extended with a nullable type variable; correspondingly, in KJ-App, the type argument must have a nullable kind.

Definition 3.2.1 (Base Kinds)**.** *We say K is a* base *kind if $K \in \{*, *_n, *_v\}$.*

3.2.2 λ_s Type System

We can now take a look at the λ_s type system, which is our stand-in for Scala. The types and kinds of λ_s are shown in Figure 3.3. λ_s differs from λ_j by adding a `Null` type and type unions ($\sigma + \tau$). Additionally, λ_s types are *explicitly* nullable, which can be seen by contrasting KS-String with the corresponding KJ-String rule in λ_j:

$$\Gamma \vdash_j \texttt{String}_j :: *_n \text{ (KJ-String)} \qquad \Gamma \vdash_s \texttt{String}_s :: *_v \text{ (KS-String)}$$

If we want to say that a type is nullable in λ_s, we need to use type unions:

$$\text{KS-Union}\ \frac{\text{KS-String}\ \dfrac{}{\Gamma \vdash_s \texttt{String}_s :: *_v} \quad \text{KS-Null}\ \dfrac{}{\Gamma \vdash_s \texttt{Null} :: *_n}}{\Gamma \vdash_s \texttt{String}_s + \texttt{Null} :: *_v \sqcup *_n}$$

The kinding derivation above uses the least upper bound operator $\sqcup$ on kinds. In this case, we reason that since $\texttt{String}_s$ is non-nullable and `Null` is nullable, then their union is nullable and $*_v \sqcup *_n = *_n$.

In general, to track nullability via kinds in the presence of type unions, we use the KS-Union rule from Figure 3.3 and its associated *nullability lattice*:

$$*_v \longrightarrow * \longrightarrow *_n$$

For example, suppose that $\vdash_s \sigma :: *_v$ and $\vdash_s \tau :: *$. If we view σ and τ as sets (which we will do in Section 3.3), then we know that $\texttt{null} \notin [\![\sigma]\!]_s$ and `null` might or might not be in $[\![\tau]\!]_s$. This, in turn, means that we cannot know (statically) whether $\texttt{null} \in [\![\sigma + \tau]\!]_s$, so $\vdash_s \sigma + \tau :: * = *_v \sqcup *$ as per KS-Union. However, if either σ or τ have kind $*_n$, then we must have $\texttt{null} \in [\![\sigma + \tau]\!]_s$ (this is why $*_n$ is at the top of the lattice).

One additional difference between λ_j and λ_s is that in λ_s, generics can take as arguments any type of kind $*$ (as opposed to $*_n$ in λ_j). This reflects the fact that in Scala, generic classes can be instantiated with any type as argument, and not just with reference types. For example, if `List[T]` is a generic Scala list, then the type applications `List[int]` and `List[String]` are both well-kinded, even though the argument in the former is non-nullable (`List<int>` would not be valid in Java).

3.3 Denotational Semantics

Now that we have a formal type system with implicit nullability, as well as one with explicit nullability, one can imagine how to formalize the nullification function. Nullification will need to turn λ_j types into λ_s types. However, before we can prove properties of nullification, we need a *semantics* for our types and kinds. That is, so far types and kinds are just

$$
\begin{array}{rl}
\sigma, \tau ::= & \textbf{Types} \\
\texttt{Null} & \text{null} \\
\texttt{int}_s & \text{int} \\
\texttt{String}_s & \text{string} \\
\sigma + \tau & \text{union} \\
\sigma \times_s \tau & \text{product} \\
\sigma \to_s \tau & \text{function} \\
\Pi_s(X :: *).\sigma & \text{generic} \\
\text{App}_s(\sigma, \tau) & \text{type application} \\
X & \text{type variable}
\end{array}
$$

$$
\begin{array}{rl}
K ::= & \textbf{Kinds} \\
*_n & \text{kind of nullable types} \\
*_v & \text{kind of non-nullable types} \\
* & \text{kind of proper types} \\
* \Rightarrow K & \text{kind of type operators}
\end{array}
$$

$$
\begin{array}{rl}
\Gamma ::= & \textbf{Contexts} \\
\varnothing & \text{empty context} \\
\Gamma, X :: * & \text{nullable type binding}
\end{array}
$$

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

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

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

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

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

$\sqcup$ is least-upper bound operation on the lattice $*_v \longrightarrow * \longrightarrow *_n$

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

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

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

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

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

.

Figure 3.3: Types, kinds, and kinding rules of λ_s. Differences with λ_j are highlighted.

syntactic objects, and kinding rules are syntactic rules devoid of meaning. For this task of assigning meaning we turn to the machinery of *denotational semantics*.

The basic idea of denotational semantics is to map the syntactic objects under consideration (procedures, types, kinds) to mathematical objects (functions, sets, families of sets) [Scott and Strachey, 1971]. Once this is done, we can prove interesting properties about programs using standard mathematical tools. The mapping is done via a *denotation* function, typically written $[\![]\!]$.

Example 3.3.1 (Constant Folding)**.** *To argue the correctness of a constant folding pass in a compiler, we might define the denotation of binary addition thus*

$$[\![E_1 + E_2]\!] = [\![E_1]\!] + [\![E_2]\!]$$

This might look like mere symbol shuffling: the two sides look pretty similar! The key difference is that the plus on the left is a syntactic *symbol without meaning, whereas the one on the right is the addition operator in the group of integers.*

If we think of constant folding as rewriting $E_1 + E_2 \to E_3$, where E_3 is the result of adding E_1 and E_2 if they are both constants, then correctness of constant folding amounts to showing that $[\![E_1 + E_2]\!] = [\![E_3]\!]$.

Roadmap

Since the rest of Section 3.3 is rather technical and dry, here is a summary of what will happen:

- First, I construct a set-theoretic model $\mathcal{J}$ for λ_j (Section 3.3.1). This is just a family of sets[2] that contains denotations of λ_j types and kinds.
- I then show how to map λ_j kinds (Section 3.3.2) and types (Section 3.3.3) to their denotations in the model. The mapping is roughly as follows:

[2]Models in denotational semantics can get rather complicated, involving algebraic structures with fancy-sounding names like *complete partial orders* and *categories*. For our type systems based on predicative System F, however, regular sets will suffice.

$$\begin{aligned} \text{kinds} &\longrightarrow \text{families of sets} \\ \text{proper types} &\longrightarrow \text{sets} \\ \text{generic types} &\longrightarrow \text{functions from sets to sets} \end{aligned}$$

- I then prove a *soundness* lemma for λ_j kinding rules that says that if a type is well-kinded, then its denotation is defined and, further, it is contained in the denotation of the corresponding kind: $\Gamma \vdash_j T :: K \implies [\![T]\!]_j \eta \in [\![K]\!]_j$ (Lemma 3.3.3).
- Finally, sections 3.3.4 to 3.3.6 and Lemma 3.3.6 repeat the work above for λ_s.

The technical presentation is based on the treatment of predicative System F in Mitchell [1996].

3.3.1 λ_j Semantic Model

Definition 3.3.1 (String Literals)**.** *`strings` denotes the set of finite-length strings.*

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

- U_1 is the universe of proper types. It is the least set closed under the following inference rules[3]:

$$\mathbb{Z} \in U_1 \quad \text{(U1-Int)}$$

$$\texttt{strings} \in U_1 \quad \text{(U1-String)}$$

$$\{\texttt{null}\} \in U_1 \quad \text{(U1-Null)}$$

$$\frac{u \in U_1 \qquad v \in U_1}{u \times v \in U_1} \quad \text{(U1-Prod)}$$

$$\frac{u \in U_1 \qquad v \in U_1}{u \cup v \in U_1} \quad \text{(U1-Union)}$$

$$\frac{u \in U_1 \qquad v \in U_1}{u^v \in U_1} \quad \text{(U1-Fun)}$$

[3] For sets u and v, u^v denotes the set of functions with domain u and range v.

Additionally, we define two families of sets that contain nullable and non-nullable types, respectively:

$$U_1^{null} = \{u | u \in U_1, \texttt{null} \in u\}$$
$$U_1^{val} = \{u | u \in U_1, \texttt{null} \notin u\}$$

Notice that both U_1^{null} and U_1^{val} are subsets of U_1, and that $U_1 = U_1^{null} \cup U_1^{val}$.

- The universe U_2 is a superset of U_1 that, additionally, contains all generic types.

  First, we define a family of sets $\{U_2^i\}$, for $i \geq 0$.

$$U_2^0 = U_1 \tag{3.1}$$
$$U_2^{i+1} = U_2^i \cup \{f : U_1^{null} \to U_2^i\} \tag{3.2}$$

  Then we set $U_2 = \bigcup_{i \geq 0} U_2^i$.

  Equation 3.2 is where the fact that we are working with a predicative type system comes into play. In an impredicative system, we would want to define

$$U_2 = U_1 \cup \{f : U_2 \to U_2\}$$

  but such a self-reference leads to inconsistencies in the style of Russell's paradox. In fact, we know that no set-theoretic model exists for the impredicative variant of System F [Reynolds, 1984].

3.3.2 Meaning of λ_j Kinds

Definition 3.3.2 (Number of Arrows in a Kind)**.** *Let K be a kind. Then $arr(K)$ denotes the number of arrows ($\Rightarrow$) in K.*

Example 3.3.2.

$$arr(*_v) = 0$$
$$arr(*_n \Rightarrow *_n \Rightarrow *_v) = 2$$

Definition 3.3.3 (Meaning of λ_j Kinds)**.** *We give meaning to λ_j kinds via a function $[\![]\!]_j$ inductively defined on the structure of a kind K.*

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

We now show that the meaning of kinds is contained within the model $\mathcal{J}$.

Lemma 3.3.1. *Let K be a kind. Then $[\![K]\!]_j \subseteq U_2^{arr(K)}$.*

Proof. By induction on the number of arrows in K.

If K has no arrows, then we must have $K \in \{*_n, *_v, *\}$.

- If $K = *_n$, then $[\![K]\!]_j = U_1^{null} \subseteq U_1 = U_2^0$, as needed.
- If $K = *_v$, then $[\![K]\!]_j = U_1^{val} \subseteq U_1$.
- If $K = *$, then $[\![K]\!]_j = U_1$.

In general, suppose K has $m \geq 1$ arrows. Then K can only be of the form $*_n \Rightarrow K'$, where K' has $m - 1$ arrows. We have

$$[\![*_n \Rightarrow K']\!]_j = \{f : U_1^{null} \to [\![K']\!]_j\}$$

Now consider an arbitrary function $f : U_1^{null} \to [\![K']\!]_j$. By the induction hypothesis, we know that $[\![K']\!]_j \subseteq U_2^{m-1}$. This means we can think of f as a function with range U_2^{m-1}, which means that $f : U_1^{null} \to U_2^{m-1}$ as well. But then by Definition 3.2 we have $f \in U_2^m$. This means that $[\![*_n \Rightarrow K']\!]_j \subseteq U_2^m$, as required. □

Corollary 3.3.2 (λ_j Kinds are Well-Defined)**.** *Let K be a kind. Then $[\![K]\!]_j \subseteq U_2$.*

Proof. Follows directly from Lemma 3.3.1 and the definition of U_2. □

3.3.3 Meaning of λ_j Types

We would now like to repeat our exercise from the previous section (assigning meaning to kinds), but for types. Right away, we face two complications:

1. How do we assign a meaning to types that are not closed (e.g. $\Pi_j(X :: *_n).Y$)?.
2. How do we deal with variable shadowing within types (e.g. $\Pi_j(X :: *_n).\Pi_j(X :: *_n).X$)?

The first complication is solved by making the denotation function take *two* arguments: the type whose meaning is being computed and an *environment* that gives meaning to the free variables ($[\![]\!]_j : Types \to Env \to U_2$).

The second problem is solved by making the simplifying assumption that types have been alpha-renamed so that there are no name collisions. I believe that the results in this chapter would still hold without alpha-renaming, but the proofs are simpler if we make this assumption.

Definition 3.3.4 (λ_j Environments)**.** *A λ_j environment $\eta : Var \to U_1^{null}$ is a map from variables to elements of U_1^{null}. The empty environment is denoted by $\emptyset$. An environment can be extended with the notation $\eta[X \to a]$, provided that X was not already in the domain.*

Definition 3.3.5 (Environment Conformance)**.** *An environment η* conforms *to a context Γ, written $\eta \vDash \Gamma$, if $dom(\eta) = dom(\Gamma)$.*

Notice that contexts in λ_j map type variables to *kinds*, but only variables of kind $*_n$ are allowed in contexts. We could have elided the kind and make contexts just be sets of variables. When we look at λ_s, however, we will see that contexts store variables with kind $*$, so I have left the kind in the context as a disambiguator.

Definition 3.3.6 (Meaning of λ_j Types)**.** *We define the meaning of types by induction on the structure of the type:*

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

Example 3.3.3.

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

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

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

Lemma 3.3.3 (Soundness of λ_j Kinding Rules)**.** *Let* $\Gamma \vdash_j T :: K$ *and* $\eta \vDash \Gamma$*. Then* $[\![T]\!]_j\eta$ *is well-defined and, further,* $[\![T]\!]_j\eta \in [\![K]\!]_j$*.*

Proof. By induction on a derivation of $\Gamma \vdash_j T :: K$.

Case (KJ-Int) $[\![\texttt{int}_j]\!]_j\eta = \mathbb{Z}$ and $[\![K]\!]_j = [\![*_v]\!]_j = U_1^{val}$. Since $\texttt{null} \notin \mathbb{Z}$ and $\mathbb{Z} \in U_1$, we have $\mathbb{Z} \in U_1^{val}$ as needed.

Case (KJ-String) $[\![\texttt{String}_j]\!]_j\eta = \{\texttt{null}\} \cup \texttt{strings}$ and $[\![*_n]\!]_j = U_1^{null}$. Since $\texttt{strings} \in U_1$, then $\{\texttt{null}\} \cup \texttt{strings} \in U_1^{null}$.

Case (KJ-Prod) We have $T = S_1 \times S_2$ and $K = *_n$. Then $[\![T]\!]_j\eta = \{\texttt{null}\} \cup ([\![S_1]\!]_j\eta \times [\![S_2]\!]_j\eta)$ and $[\![*_n]\!]_j = U_1^{null}$. By the induction hypothesis, $[\![S_1]\!]_j\eta \in U_1$ and $[\![S_2]\!]_j\eta \in U_1$. Then $[\![S_1]\!]_j\eta \times [\![S_2]\!]_j\eta \in U_1$, and so the result follows.

Case (KJ-Fun) We have $T = S_1 \to S_2$ and $K = *_v$. Then $[\![T]\!]_j\eta = [\![S_1]\!]_j\eta^{[\![S_2]\!]_j\eta}$ and $[\![*_v]\!]_j = U_1^{val}$. By the induction hypothesis, $[\![S_1]\!]_j\eta \in U_1$ and $[\![S_2]\!]_j\eta \in U_1$. Then $[\![S_1]\!]_j\eta^{[\![S_2]\!]_j\eta} \in U_1$, and so the result follows (because $\texttt{null}$ is not a function).

Case (KJ-Pi) We have $T = \Pi_j(X :: *_n).S$ and $K = *_n \Rightarrow K'$. $[\![\Pi_j(X :: *_n).S]\!]_j\eta = \lambda(a \in U_1^{null}).[\![S]\!]_j(\eta[X \to a])$ and $[\![*_n \Rightarrow K']\!]_j = \{f : U_1^{null} \to [\![K']\!]_j\}$. Now take an arbitrary $a \in U_1^{null}$. Notice that $[\![\Pi_j(X :: *_n).S]\!]_j\eta(a) = [\![S]\!]_j\eta[X \to a]$. From the typing derivation, we know that $\Gamma, X :: *_n \vdash_j S :: K'$. Also $\eta[X \to a] \vDash \Gamma, X :: *_n$. We can then use the induction hypothesis to conclude that $[\![S]\!]_j\eta[X \to a] \in [\![K']\!]_j$. This means that $[\![\Pi_j(X :: *_n).S]\!]_j\eta$ is a function from U_1^{null} to $[\![K']\!]_j$, which implies that $[\![\Pi_j(X :: *_n).S]\!]_j\eta \in [\![*_n \Rightarrow K']\!]_j$ as needed.

Case (KJ-App) We have $T = \mathrm{App}_j(S_1, S_2)$ and K can be an arbitrary kind.

$$[\![\mathrm{App}_j(S_1, S_2)]\!]_j\eta = [\![S_1]\!]_j\eta([\![S_2]\!]_j\eta)$$

From the typing rule, we know that $\Gamma \vdash_j S_1 :: *_n \Rightarrow K$ and $\Gamma \vdash_j T :: *_n$. Then by the induction hypothesis, we get two facts: $[\![S]\!]_j\eta \in [\![*_n \Rightarrow K]\!]_j$ and $[\![S_2]\!]_j\eta \in [\![*_n]\!]_j = U_1^{null}$. This means that $[\![S]\!]_j\eta$ must be a function from U_1^{null} to $[\![K]\!]_j$. But then the result of the function application $[\![S_1]\!]_j\eta([\![S_2]\!]_j\eta)$ must be in $[\![K]\!]_j$, as needed.

Case (KJ-Var) We have $T = X$ (or some other type variable) and $K = *_n$. From the premise of the typing rule we know that $\Gamma(X) = *_n$ and, in particular, X is in the domain of Γ. Since $\eta \vDash \Gamma$, then X must be in the domain of η as well. Since η maps variables to elements of U_1^{null}, then $[\![X]\!]_j\eta = \eta(X) \in U_1^{null}$, as needed.

Case (KJ-Null) We have $K = *$. From the premise of the typing rule, we know that $\Gamma \vdash_j T :: *_n$. By the induction hypothesis, we have $[\![S]\!]_j\eta \in U_1^{null} \subseteq U_1$.

Case (KJ-NonNull) We have $K = *$. From the premise of the typing rule, we know that $\Gamma \vdash_j T :: *_v$. By the induction hypothesis, we have $[\![S]\!]_j\eta \in U_1^{val} \subseteq U_1$. □

3.3.4 λ_s Semantic Model

The model for λ_s is very similar to that from Section 3.3.1, except for how we represent generics. Once again, the model is a pair $\mathcal{S} = (U_1, U_2')$, where U_1 is as defined in Section 3.3.1 and U_2' is defined below.

First, we define a family of sets $\{U_2'^i\}$, for $i \geq 0$.

$$U_2'^0 = U_1 \tag{3.3}$$

$$U_2'^{i+1} = U_2'^i \cup \{f : U_1 \to U_2'^i\} \tag{3.4}$$

Highlighted is the fact that generics in λ_s take arguments from U_1, as opposed to U_1^{null}. Then we set $U_2' = \bigcup_{i \geq 0} U_2'^i$.

3.3.5 Meaning of λ_s Kinds

Definition 3.3.7 (Meaning of λ_s Kinds)**.** *We give meaning to λ_s kinds via a function $[\![]\!]_s$ inductively defined on the structure of a kind K.*

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

We now show that the meaning of kinds is contained within the model $\mathcal{S}$.

Lemma 3.3.4. *Let K be a kind. Then $[\![K]\!]_s \subseteq U_2'^{arr(K)}$.*

Proof. By induction on the number of arrows in K.

If K has no arrows, then we must have $K \in \{*_n, *_v, *\}$.

- If $K = *_n$, then $[\![K]\!]_s = U_1^{null} \subseteq U_1 = U_2'^0$, as needed.
- If $K = *_v$, then $[\![K]\!]_s = U_1^{val} \subseteq U_1$.
- If $K = *$, then $[\![K]\!]_s = U_1$.

In general, suppose K has $m \geq 1$ arrows. Then K can only be of the form $* \Rightarrow K'$, where K' has $m - 1$ arrows. We have

$$[\![* \Rightarrow K']\!]_s = \{f : U_1 \to [\![K']\!]_s\}$$

Now consider an arbitrary function $f : U_1 \to [\![K']\!]_s$. By the induction hypothesis, we know that $[\![K']\!]_s \subseteq U_2'^{m-1}$. This means we can think of f as a function with range $U_2'^{m-1}$, which means that $f : U_1 \to U_2^{m-1}$ as well. But then by definition 3.4 we have $f \in U_2'^m$. This means that $[\![*_n \Rightarrow K']\!]_s \subseteq U_2'^m$, as required. □

Corollary 3.3.5 (λ_s Kinds are Well-Defined)**.** *Let K be a kind. Then $[\![K]\!]_s \subseteq U_2'$.*

Proof. Follows directly from Lemma 3.3.4 and the definition of U_2'. □

3.3.6 Meaning of λ_s Types

Definition 3.3.8 (λ_s Environment)**.** *A λ_s environment $\eta : Var \to U_1$ is a map from variables to elements of U_1.*

Definition 3.3.9 (Environment Conformance)**.** *An environment η* conforms *to a context Γ, written $\eta \vDash \Gamma$, if $dom(\eta) = dom(\Gamma)$.*

Definition 3.3.10 (Meaning of λ_s Types)**.** *We define the meaning of types by induction on the structure of the type:*

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

Lemma 3.3.6 (Soundness of λ_s Kinding Rules)**.** *Let $\Gamma \vdash_s T :: K$ and $\eta \vDash \Gamma$. Then $[\![T]\!]_s\eta$ is well-defined and, further, $[\![T]\!]_s\eta \in [\![K]\!]_s$.*

Proof. By induction on a derivation of $\Gamma \vdash_s T :: K$.

Case (KS-Int) $[\![\texttt{int}_s]\!]_s\eta = \mathbb{Z}$ and $[\![K]\!]_s = [\![*_v]\!]_s = U_1^{val}$. Since $\texttt{null} \notin \mathbb{Z}$, we have $\mathbb{Z} \in U_1^{val}$ as needed.

Case (KS-String) $[\![\texttt{String}_s]\!]_s\eta = \texttt{strings}$ and $[\![*_v]\!]_s = U_1^{val}$. Since $\texttt{null} \notin \texttt{strings}$, then $\texttt{strings} \in U_1^{null}$.

Case (KS-NullType) We have $T = \texttt{Null}$ and $K = *_n$. $[\![\texttt{Null}]\!]_s\eta = \{\texttt{null}\}$, and the result follows immediately.

Case (KS-Union) We have $T = T_1 + T_2, K = K_1 \sqcup K_2$, and $[\![T_1 + T_2]\!]_s\eta = [\![T_1]\!]_s\eta \cup [\![T_2]\!]_s\eta$. Notice that $\sqcup$ is symmetric. Then we consider the following cases:

- $K_1 = *_n, K_2 = *_n, K = *_n$. By the induction hypothesis, $[\![T_1]\!]_s\eta \in U_1^{null}$ and $[\![T_2]\!]_s\eta \in U_1^{null}$. Because U_1 is closed under set union, we know that $[\![T_1]\!]_s\eta \cup [\![T_2]\!]_s\eta \in U_1$. Further, since $\texttt{null} \in [\![T_1]\!]_s\eta$, we have $\texttt{null} \in [\![T_1]\!]_s\eta \cup [\![T_2]\!]_s\eta$, so $[\![T_1]\!]_s\eta \cup [\![T_2]\!]_s\eta \in U_1^{null}$ as needed.
- $K_1 = *_n, K_2 = *_v, K = *_n$. Similar to the previous case.
- $K_1 = *_n, K_2 = *, K = *_n$. Similar to the previous case.
- $K_1 = *, K_2 = *, K = *$. Follow directly from the induction hypothesis and the fact that U_1 is closed under unions.

- $K_1 = *, K_2 = *_v, K = *$. Similar to the previous case.
- $K_1 = *_v, K_2 = *_v, K = *_v$. Follows from the fact that $\texttt{null} \notin [\![T_1]\!]_s\eta$ and $\texttt{null} \notin [\![T_2]\!]_s\eta$ implies $\texttt{null} \notin [\![T_1]\!]_s\eta \cup [\![T_2]\!]_s\eta$.

Case (KS-Prod) We have $T = S_1 \times S_2$ and $K = *_v$. Then $[\![T]\!]_s\eta = [\![S_1]\!]_s\eta \times [\![S_2]\!]_s\eta$ and $[\![*_v]\!]_s = U_1^{val}$. By the induction hypothesis, $[\![S_1]\!]_s\eta \in U_1$ and $[\![S_2]\!]_s\eta \in U_1$. Then $[\![S_1]\!]_s\eta \times [\![S_2]\!]_s\eta \in U_1^{val}$, and so the result follows.

Case (KS-Fun) We have $T = S_1 \to S_2$ and $K = *_v$. Then $[\![T]\!]_s\eta = [\![S_1]\!]_s\eta^{[\![S_2]\!]_s\eta}$ and $[\![*_v]\!]_s = U_1^{val}$. By the induction hypothesis, $[\![S_1]\!]_s\eta \in U_1$ and $[\![S_2]\!]_s\eta \in U_1$. Then $[\![S_1]\!]_s\eta^{[\![S_2]\!]_s\eta} \in U_1$, and so the result follows (because `null` is not a function).

Case (KS-Pi) We have $T = \Pi_s(X :: *).S$ and $K = * \Rightarrow K'$. $[\![\Pi_s(X :: *).S]\!]_s\eta = \lambda(a \in U_1).[\![S]\!]_s(\eta[X \to a])$ and $[\![*_n \Rightarrow K']\!]_s = \{f : U_1 \to [\![K']\!]_s\}$. Now take an arbitrary $a \in U_1$. Notice that $[\![\Pi_s(X :: *).S]\!]_s\eta(a) = [\![S]\!]_s\eta[X \to a]$. From the typing derivation, we know that $\Gamma, X :: * \vdash_s S :: K'$. Also $\eta[X \to a] \vDash \Gamma, X :: *$. We can then use the induction hypothesis to conclude that $[\![S]\!]_s\eta[X \to a] \in [\![K']\!]_s$. This means that $[\![\Pi_s(X :: *).S]\!]_s\eta$ is a function from U_1 to $[\![K']\!]_s$, which implies that $[\![\Pi_s(X :: *).S]\!]_s\eta \in [\![*_n \Rightarrow K']\!]_s$ as needed.

Case (KS-App) We have $T = \mathrm{App}_s(S_1, S_2)$ and K can be an arbitrary kind.

$$[\![\mathrm{App}_s(S_1, S_2)]\!]_s\eta = [\![S_1]\!]_s\eta([\![S_2]\!]_s\eta)$$

From the typing rule, we know that $\Gamma \vdash_s S_1 :: * \Rightarrow K$ and $\Gamma \vdash_s T :: *$. Then by the induction hypothesis, we get two facts: $[\![S]\!]_s\eta \in [\![* \Rightarrow K]\!]_s$ and $[\![S_2]\!]_s\eta \in [\![*]\!]_s = U_1$. This means that $[\![S]\!]_s\eta$ must be a function from U_1 to $[\![K]\!]_s$. But then the result of the function application $[\![S_1]\!]_s\eta([\![S_2]\!]_s\eta)$ must be in $[\![K]\!]_s$, as needed.

Case (KS-Var) We have $T = X$ (or some other type variable) and $K = *$. From the premise of the typing rule we know that $\Gamma(X) = *$ and, in particular, X is in the domain of Γ. Since $\eta \vDash \Gamma$, then X must be in the domain of η as well. Since η maps variables to elements of U_1, then $[\![X]\!]_s\eta = \eta(X) \in U_1^{null}$, as needed.

Case (KS-Null) We have $K = *$. From the premise of the typing rule, we know that $\Gamma \vdash_s T :: *_n$. By the induction hypothesis, we have $[\![S]\!]_s\eta \in U_1^{null} \subseteq U_1$.

Case (KS-NonNull) We have $K = *$. From the premise of the typing rule, we know that $\Gamma \vdash_s T :: *_v$. By the induction hypothesis, we have $[\![S]\!]_s\eta \in U_1^{val} \subseteq U_1$. □

3.4 Type Nullification

Now that we have formal definitions for both λ_j and λ_s, we can also formally define type nullification. Recall that type nullification makes nullability *explicit* as we go from a type system where `null` is implicit (Java's) to one where `null` is explicit (Scala's). For example,

```
class Movie { // Java
  int views;
  String getTitle(String language) {...}
}
```

becomes

```
class Movie { // Scala
  var views: Int;
  def getTitle(language: String|Null): String|Null {...}
}
```

Or, using our newly-introduced type systems:

$$(\texttt{int}_j) \times_j (\texttt{String}_j \to_j \texttt{String}_j)$$

becomes

$$(\texttt{int}_s) \times_s (\texttt{String}_s + \texttt{Null} \to_s \texttt{String}_s + \texttt{Null})$$

That is, *type nullification is a function that turns λ_j types into λ_s types.* One wrinkle remains, which is that in the implementation, we decided *not to nullify* arguments in type applications. That is, given a Java class `List<T>`, type applications such as `List<String>` are translated as `List<String>`, and not `List<String|Null>`. Section 2.3.2 gives an informal argument for why this is correct. The motivation for special casing type arguments is maximizing backwards-compatibility.

Because of the different treatment for types based on whether they are in an argument position or not, we will model nullification as a *pair* of functions $(F_{\text{null}}, A_{\text{null}})$. These are defined below.

Definition 3.4.1 (Type Nullification)**.**

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

$$
\begin{array}{ll}
A_{null}(\texttt{int}_j) & = \texttt{int}_s \\
A_{null}(\texttt{String}_j) & = \texttt{String}_s \\
A_{null}(S \times_j T) & = F_{null}(S) \times_s F_{null}(T) \\
A_{null}(S \to_j T) & = F_{null}(S) \to_s F_{null}(T) \\
A_{null}(App_j(S,T)) & = App_s(F_{null}(S), A_{null}(T)) \\
A_{null}(X) & = X
\end{array}
$$

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

The following example shows that type nullification faithfully models our implementation.

Example 3.4.1 (Pairs)**.** *Let* $\texttt{Pair} \equiv \Pi_j(A :: *_n).\Pi_j(B :: *_n).A \times_j B$*. Then*

$$
\begin{aligned}
F_{null}(\texttt{Pair}) &= F_{null}(\Pi_j(A :: *_n).\Pi_j(B :: *_n).A \times_j B) \\
&= \Pi_s(A :: *).F_{null}(\Pi_j(B :: *_n).A \times_j B) \\
&= \Pi_s(A :: *).\Pi_s(B :: *).F_{null}(A \times_j B) \\
&= \Pi_s(A :: *).\Pi_s(B :: *).(F_{null}(A) \times_s F_{null}(B)) + \texttt{Null} \\
&= \Pi_s(A :: *).\Pi_s(B :: *).(A + \texttt{Null} \times_s A + \texttt{Null}) + \texttt{Null} \qquad (3.5)
\end{aligned}
$$

Nullifying the type application "`Pair<String, Pair<int, int>>`*" expands to*

$$
\begin{aligned}
&F_{null}(App_j(App_j(\texttt{Pair}, \texttt{String}_j), App_j(App_j(\texttt{Pair}, \texttt{int}_j), \texttt{int}_j))) \\
&= App_s(F_{null}(App_j(\texttt{Pair}, \texttt{String}_j)), A_{null}(App_j(App_j(\texttt{Pair}, \texttt{int}_j), \texttt{int}_j))) \\
&= App_s(F_{null}(App_j(\texttt{Pair}, \texttt{String}_j)), A_{null}(App_j(App_j(\texttt{Pair}, \texttt{int}_j), \texttt{int}_j))) \\
&= App_s(F_{null}(App_j(\texttt{Pair}, \texttt{String}_j)), A_{null}(App_j(App_j(\texttt{Pair}, \texttt{int}_j), \texttt{int}_j))) \\
&= App_s(App_s(F_{null}(\texttt{Pair}), A_{null}(\texttt{String}_j)), A_{null}(App_j(App_j(\texttt{Pair}, \texttt{int}_j), \texttt{int}_j))) \\
&= App_s(App_s(F_{null}(\texttt{Pair}), \texttt{String}_s), A_{null}(App_j(App_j(\texttt{Pair}, \texttt{int}_j), \texttt{int}_j))) \\
&= App_s(App_s(F_{null}(\texttt{Pair}), \texttt{String}_s), App_s(F_{null}(App_j(\texttt{Pair}, \texttt{int}_j)), A_{null}(\texttt{int}_j))) \\
&= App_s(App_s(F_{null}(\texttt{Pair}), \texttt{String}_s), App_s(App_s(F_{null}(\texttt{Pair}), A_{null}(\texttt{int}_j)), \texttt{int}_s)) \\
&= App_s(App_s(F_{null}(\texttt{Pair}), \underline{\texttt{String}_s}), App_s(App_s(F_{null}(\texttt{Pair}), \underline{\texttt{int}_s}), \underline{\texttt{int}_s})) \qquad (3.6)
\end{aligned}
$$

The corresponding Java and nullified Scala code are shown below.

Java	$\xrightarrow{\textit{type nullification}}$	*Scala*
`class Pair<A, B> {`		`class Pair[A, B] {`
`  A a;`		`  var a: A|Null;`
`  B b;`		`  var b: B|Null;`
`}`		`}`
`Pair<String,Pair<int,int>> p`		`var p:Pair[String,Pair[int,int]]|Null`

This is all verbose and hard to follow, but the important part is that F_{null} matches the behaviour of type nullification in the implementation. In particular, notice how all of a, b and p were nullified in equation 3.5, but none *of the type arguments were nullified in type application 3.6.*

Definition 3.4.2 (Context Nullification)**.** *We lift nullification to work on contexts.*

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

Notice how nullification turns λ_j contexts into (syntactic) λ_s contexts.

Definition 3.4.3 (Kind Nullification)**.** *We also lift nullification to work on kinds.*

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

That is, nullification turns λ_j kinds into λ_s kinds.

3.4.1 Soundness

Summary: the main results of this section are Lemma 3.4.2 and Theorem 3.4.3.

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

Conjecture 3.4.1 (Soundness — Incorrect)**.** *Let $\Gamma \vdash_j T :: K$, and let η be an environment such that $\eta \vDash \Gamma$. Then $[\![T]\!]_j\eta = [\![F_{null}(T)]\!]_s\eta$.*

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

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

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

What are we to do then? Is nullification wrong as presented in Definition 3.4.1. Luckily, the answer is no! We can arrive at an alternative, and this time correct, definition by observing the following:

- Consider a generic Java class `List<T>` and its nullified Scala counterpart `List[T]`:

```
class List<T> { // Java
  T head;}
```

```
class List[T] { // Scala
  var head: T|Null}
```

- As we previously saw, their denotations are different because the Java generic can only take nullable type arguments. However, consider a Java type application and its nullification:

```
List<Integer> =
class List {
  Integer head;
}
```

```
List[Integer] =
class List {
  var head: Integer|Null
}
```

- What we see here is that the denotations on both sides actually match! That is, accessing the `head` field on both sides gives us back an element from $\mathbb{Z} \cup \{\texttt{null}\}$.
- In general, if `G` is a generic, even though $[\![G]\!]_j$ and $[\![G]\!]_s$ are not equal, for *any* valid type application `G<T>` in Java, $[\![G < T >]\!]_j = [\![F_{\text{null}}(G < T >)]\!]_s$. That is, our soundness theorem will say that nullification leaves *fully-applied* generic types unchanged. This is just as well because users can only manipulate values of type `G<T>` and never values of type `G` directly.

Before we state the soundness theorem we need a few ancillary definitions. These are needed because our nullification function is complicated by the presence of A_{null}, which *does* change the meaning of types (albeit only slightly).

Definition 3.4.4 (Similar Types)**.** *Let $S, T \in U_1$. Then we say S is similar to T, written $S \sim T$, if $S \cup \{\texttt{null}\} = T \cup \{\texttt{null}\}$.*

Note that $\sim$ is symmetric.

Definition 3.4.5 (Similar Type Vectors)**.** *Let $\vec{S} = (S_1, \ldots, S_n)$ and $\vec{T} = (T_1, \ldots, T_n)$ be vector of types, where $\vec{S}$ and $\vec{T}$ have the same number of elements. Then $\vec{S} \sim \vec{T}$ if they are similar at every component.*

Definition 3.4.6 (Similar Environments)**.** *Let η, η' be environments (either from λ_j or λ_s). Then η' is similar to η', written $\eta \sim \eta'$, if $dom(\eta') = dom(\eta)$ and for all type variables X in the domain, we have $\eta'(X) \sim \eta(X)$.*

Note this relation is also symmetric.

Definition 3.4.7 (Similar Kinds)**.** *Let K_1 and K_2 be two base kinds. Then $K_1 \rightsquigarrow K_2$ is defined by case analysis.*

$$* \rightsquigarrow * \qquad \textit{(SK-Prop)}$$

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

$$*_n \rightsquigarrow *_n \qquad \textit{(SK-Null1)}$$

$$*_n \rightsquigarrow *_v \qquad \textit{(SK-Null2)}$$

$$*_n \rightsquigarrow * \qquad \textit{(SK-Null3)}$$

Note this relation is not symmetric.

Lemma 3.4.1. *If K is a base kind, then $K \rightsquigarrow K$.*

Proof. Immediate from Definition 3.4.7. □

The rules in Definition 3.4.7 capture what happens to the kind of a type after being transformed by A_{null}. For example, $\texttt{String}_j$ has kind $*_n$ in λ_j, but $A_{\text{null}}(\texttt{String}_j) = \texttt{String}_s$ has kind $*_v$ in λ_s. This is described by rule (SK-Null2).

Before proving soundness, we can prove a weaker lemma that says that nullification preserves well-kindedness. Proving this lemma is a useful exercise, because if T is well-kinded and nullification turns T into $F_{\text{null}}(T)$, the latter had better be well-kinded as well.

Lemma 3.4.2 (Nullification Preserves Well-Kindedness)**.** *Let $\Gamma \vdash_j T :: K$ and $\Gamma' = F_{null}(\Gamma)$. Then*

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

Proof. By induction on a derivation of $\Gamma \vdash_j T :: K$. For some cases, we need to prove *both* claims, and for some just claim 1.

Case (KJ-Int) Claim 1. $T = \texttt{int}_j$, $K = *_v$, and $F_{\text{null}}(*_v) = *_v$. We have $F_{\text{null}}(T) = \texttt{int}_s$, and $\Gamma' \vdash_s \texttt{int}_s :: *_v$ as needed.

Claim 2. $A_{\text{null}}(\texttt{int}_j) = \texttt{int}_s$. We can then take $K' = *_v$.

Case (KJ-String) Claim 1. $T = \texttt{String}_j$, $K = *_n$, and $F_{\text{null}}(*_n) = *_n$. We have $F_{\text{null}}(T) = \texttt{String}_s + \texttt{Null}$, and $\Gamma' \vdash_s \texttt{String}_s + \texttt{Null} :: *_n$.

Claim 2. $A_{\text{null}}(\texttt{String}_j) = \texttt{String}_s$. We can take $K' = *_v$ because $*_n \rightsquigarrow *_v$.

Case (KJ-Prod) Claim 1. $T = T_1 \times_j T_2$, $K = *_n$, and $F_{\text{null}}(*_n) = *_n$. We have $F_{\text{null}}(T) = (F_{\text{null}}(T_1) \times F_{\text{null}}(T_2)) + \texttt{Null}$. From the antecedent of the rule, we get $\Gamma \vdash_j T_1 :: *$ and $\Gamma \vdash_j T_2 :: *$. By the induction hypothesis, we then get $\Gamma' \vdash_s F_{\text{null}}(T_1) :: *$ and $\Gamma' \vdash_s F_{\text{null}}(T_2) :: *$. But then using KS-Prod we can put them back together in $\Gamma' \vdash_s F_{\text{null}}(T_1) \times F_{\text{null}}(T_2) :: *$. Then using KS-Union we get $\Gamma' \vdash_s (F_{\text{null}}(T_1) \times F_{\text{null}}(T_2)) + \texttt{Null} :: *_n$, as needed.

Claim 2. $A_{\text{null}}(T_1 \times_j T_2) = F_{\text{null}}(T_1) \times_s F_{\text{null}}(T_2)$. By the induction hypothesis, we have $\Gamma' \vdash_s F_{\text{null}}(T_1) :: *$ and $\Gamma' \vdash_s F_{\text{null}}(T_2) :: *$. Using KS-Prod we get $\Gamma' \vdash_s F_{\text{null}}(T_1) \times_s F_{\text{null}}(T_2) :: *_v$. We can take $K' = *_v$ since $*_n \rightsquigarrow *_v$.

Case (KJ-Fun) Claim 1. $T = T_1 \to_j T_2$ and $K = *_v$. $F_{\text{null}}(T_1 \to_j T_2) = F_{\text{null}}(T_1) \to_s F_{\text{null}}(T_2)$. From the antecedent of the kinding rule, we know that $\Gamma \vdash_j T_1 :: *$ and $\Gamma \vdash_j T_2 :: *$. By the induction hypothesis, we then have $\Gamma' \vdash_s F_{\text{null}}(T_1) :: *$ and $\Gamma' \vdash_s F_{\text{null}}(T_2) :: *$. Using KS-Fun, we get $\Gamma' \vdash_s F_{\text{null}}(T_1) \to_s F_{\text{null}}(T_2) :: *_v$, as needed.

Claim 2. Notice that $A_{\text{null}}(T) = F_{\text{null}}(T)$. Then using the argument above, we can then take $K' = *_v$ and $K \rightsquigarrow K'$.

Case (KJ-Pi) Claim 1. $T = \Pi_j(X :: *_n).S$, $K = *_n \Rightarrow K_1$ for some K_1, and $F_{\text{null}}(*_n \Rightarrow K_1) = * \Rightarrow F_{\text{null}}(K_1)$. We have $F_{\text{null}}(T) = \Pi_s(X :: *).F_{\text{null}}(S)$. From the antecedent of the rule, we know that $\Gamma, X :: *_n \vdash_j S :: K_1$. By the induction hypothesis, this means that $F_{null}(\Gamma, X :: *_n) \vdash_s F_{\text{null}}(S) :: F_{\text{null}}(K_1)$. Notice that $F_{\text{null}}(\Gamma, X :: *_n) = \Gamma', X :: *$. Then using KS-Pi we get $\Gamma', X :: * \vdash_s F_{\text{null}}(S) :: * \Rightarrow F_{\text{null}}(K_1)$, as needed.

Claim 2. Does not apply because K is not a base kind.

Case (KJ-App) Claim 1 $T = \text{App}_j(T_1, T_2)$, K is an arbitrary kind, and $F_{\text{null}}(T) = \text{App}_s(F_{\text{null}}(T_1), A_{\text{null}}(T_2))$. From the antecedent of the rule, we know that $\Gamma \vdash_j T_1 :: *_n \Rightarrow K$ and $\Gamma \vdash_j T_2 :: *_n$. By the induction hypothesis, $\Gamma' \vdash_s F_{\text{null}}(S) :: F_{\text{null}}(*_n \Rightarrow K)$. Simplifying, we get $\Gamma' \vdash_s F_{\text{null}}(S) :: * \Rightarrow F_{\text{null}}(K)$. Using the induction hypothesis for claim 2 we get that there exists a kind K' with $*_n \rightsquigarrow K_2$ such that $\Gamma' \vdash_s A_{\text{null}}(T_2) :: K'$. By inversion of the rules for similar kinds, we get that $K' \in \{*_n, *_v, *\}$. Notice that if $K' = *_n$, $\Gamma' \vdash_s A_{\text{null}}(T_2) :: *_n$ implies $\Gamma' \vdash_s A_{\text{null}}(T_2) :: *$ by KS-Null. A similar argument applies if $K' = *_v$ using KS-NonNull. So no matter the value of K', we know that $\Gamma' \vdash_s A_{\text{null}}(T_2) :: *$. Then we can use KS-App to conclude $\Gamma' \vdash_s \text{App}_s(F_{\text{null}}(T_1), A_{\text{null}}(T_2)) :: F_{\text{null}}(K')$, as needed.

Claim 2. If K is a base kind, we need to show there exists some kind K' with $K \rightsquigarrow K'$ such that $\Gamma' \vdash_s A_{\text{null}}(T) :: K'$. $A_{\text{null}}(T) = \text{App}_s(F_{\text{null}}(T_1), A_{\text{null}}(T_2))$. Notice that $F_{\text{null}}(T) =$

$A_{\text{null}}(T)$. Also, since K is a base kind, $F_{\text{null}}(K) = K$. Then from the result for claim 1 above we get $\Gamma' \vdash_s A_{\text{null}}(T) :: K$. But $K \rightsquigarrow K$ for all base kinds, so we can take $K' = K$, as needed.

Case (KJ-Var) Claim 1. $T = X$ and $K = *_n$. $F_{\text{null}}(T) = X + \texttt{Null}$. Since $\Gamma' = F_{\text{null}}(\Gamma)$ and $X \in \text{dom}(\Gamma)$, we know that $X \in \text{dom}(\Gamma')$. This means that $\Gamma'(X) = *$. We then have $\Gamma' \vdash_s X :: *$ and $\Gamma' \vdash_s \texttt{Null} :: *_n$. Using KS-union, we get $\Gamma' \vdash_s X + \texttt{Null} :: *_n$ as needed.

Claim 2. $A_{\text{null}}(X) = X$ and as we saw $\Gamma' \vdash_s X :: *$. We can then take $K' = *$, because $*_n \rightsquigarrow *$.

Case (KJ-Null) Claim 1. T is an arbitrary type and $K = *$. From the antecedent, we have $\Gamma \vdash_j T :: *_n$. By the induction hypothesis, $\Gamma' \vdash_s F_{\text{null}}(T) :: *_n$. But then using KS-Null we get $\Gamma' \vdash_s F_{\text{null}}(T) :: *$ as needed.

Claim 2. By the induction hypothesis, there exists a kind K'' such that $*_n \rightsquigarrow K''$ with $\Gamma' \vdash_s A_{\text{null}}(T) :: K''$. By inversion, we get $K'' \in \{*, *_n, *_v\}$. If $K'' = *$, we're done. Otherwise, we can use either SK-Null or SK-NonNull to get $\Gamma' \vdash_s A_{\text{null}}(T) :: *$ and then we take $K' = *$ as needed.

Case (KJ-NonNull)

Claim 1. T is an arbitrary type and $K = *$. From the antecedent, we have $\Gamma \vdash_j T :: *_v$. By the induction hypothesis, $\Gamma' \vdash_s F_{\text{null}}(T) :: *_v$. But then using KS-NonNull we get $\Gamma' \vdash_s F_{\text{null}}(T) :: *$ as needed.

Claim 2. By the induction hypothesis, there exists a kind K'' such that $*_v \rightsquigarrow K''$ with $\Gamma' \vdash_s A_{\text{null}}(T) :: K''$. By inversion, we get $K'' *_v$. We can then use either SK-NonNull to get $\Gamma' \vdash_s A_{\text{null}}(T) :: *$ and then we take $K' = *$ as needed.

□

Definition 3.4.8 (Curried Type Application)**.** *If f is a function of m arguments and $\vec{x} = (x_1, \ldots, x_m)$, we use the notation $f(\vec{x})$ to mean the curried function application $f(x_1)(x_2)\ldots(x_m)$. In the degenerate case where f is not a function (i.e. $m = 0$), we set $f(\vec{x}) = f$.*

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

Theorem 3.4.3 (Soundness of Type Nullification)**.** *Let $\Gamma \vdash_j T :: K$. Let η, η' be environments such that $\eta \vDash \Gamma$ and $\eta \sim \eta'$. Then the following two hold:*

1. *If K is a base kind, then*

(a) $[\![T]\!]_j\eta = [\![F_{null}(T)]\!]_s\eta'$ *and*

(b) $[\![T]\!]_j\eta \sim [\![A_{null}(T)]\!]_s\eta'$.

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

Proof. By induction on a derivation of $\Gamma \vdash S :: K$.

Case (KJ-Int) $K = *_v, T = \texttt{int}_j$, $F_{\text{null}}(T) = \texttt{int}_s = A_{\text{null}}(\texttt{int}_j)$.

Claim 1a.

$[\![\texttt{int}_j]\!]_j\eta = \mathbb{Z} = [\![\texttt{int}_s]\!]_s\eta$

Claim 1b. Identical.

Claim 2. Does not apply.

Case (KJ-String) $K = *_n, T = \texttt{String}_j$, $F_{\text{null}}(T) = \texttt{String}_s + \texttt{Null}$, and $A_{\text{null}}(T) = \texttt{String}_s$.

Claim 1a.

$$\begin{aligned}[\![\texttt{String}_j]\!]\eta &= \{\texttt{null}\} \cup \texttt{strings} \\ &= [\![\texttt{Null}]\!]_s\eta' \cup [\![\texttt{String}_s]\!]_s\eta' \\ &= [\![\texttt{Null} + \texttt{String}_s]\!]_s\eta'\end{aligned}$$

Claim 1b.

$[\![\texttt{String}_s]\!]_s\eta' = \texttt{strings}$, and $\texttt{strings} \sim \{\texttt{null}\} \cup \texttt{strings}$.

Claim 2. Does not apply.

Case (KJ-Prod) $K = *_n, T = T_1 \times_j T_2$, $F_{\text{null}}(T) = (F_{\text{null}}(T_1) \times_s F_{\text{null}}(T_2)) + \texttt{Null}$ and $A_{\text{null}}(T) = F_{\text{null}}(T_1) \times_s F_{\text{null}}(T_2)$.

Claim 1a.

$$\begin{aligned}[\![T_1 \times_j T_2]\!]_j\eta &= \{\texttt{null}\} \cup ([\![T_1]\!]_j\eta \times [\![T_2]\!]_j\eta) \\ &= \{\texttt{null}\} \cup ([\![F_{\text{null}}(T_1)]\!]_s\eta' \times [\![T_2]\!]_j\eta) && \text{by the I.H. using } \Gamma \vdash_j T_1 :: * \\ &= \{\texttt{null}\} \cup ([\![F_{\text{null}}(T_1)]\!]_s\eta' \times [\![F_{\text{null}}(T_2)]\!]_s\eta') && \text{by the I.H. using } \Gamma \vdash_j T_2 :: * \\ &= [\![(F_{\text{null}}(T_1) \times_s F_{\text{null}}(T_2)) + \texttt{Null}]\!]_s\eta'\end{aligned}$$

Claim 1b. Notice that $[\![F_{\text{null}}(T)]\!]_s\eta' = [\![A_{\text{null}}(T)]\!]_s\eta' \cup \{\texttt{null}\}$. This implies that $[\![A_{\text{null}}(T)]\!]_s\eta' \sim [\![F_{\text{null}}(T)]\!]_s\eta' = [\![T]\!]_j\eta$, as needed.

Claim 2. Does not apply.

Case (KJ-Fun)

$K = *_v, T = T_1 \to_j T_2$, $F_{\text{null}}(T) = (F_{\text{null}}(T_1) \to_s F_{\text{null}}(T_2)) = A_{\text{null}}(T)$.

Claim 1a.

$$\begin{aligned}
[\![T_1 \to_j T_2]\!]_j\eta &= [\![T_1]\!]_j\eta^{[\![T_2]\!]_j\eta} \\
&= [\![F_{\text{null}}(T_1)]\!]_s\eta'^{[\![T_2]\!]_j\eta} && \text{by the I.H. using } \Gamma \vdash_j T_1 :: * \\
&= [\![F_{\text{null}}(T_1)]\!]_s\eta'^{[\![F_{\text{null}}(T_2)]\!]_s\eta'} && \text{by the I.H. using } \Gamma \vdash_j T_2 :: * \\
&= [\![F_{\text{null}}(T_1) \to_s F_{\text{null}}(T_2)]\!]_s\eta'
\end{aligned}$$

Claim 1b. Follow from the previous claim and $F_{\text{null}}(T) = A_{\text{null}}(T)$.

Claim 2. Does not apply.

Case (KJ-Pi) $K = *_n \Rightarrow K', T = \Pi_j(X :: *_n).T', F_{\text{null}}(T) = \Pi_s(X :: *).F_{\text{null}}(T)$. $A_{\text{null}}(T)$ is not defined in T because K is not a base kind.

Claim 1a. Does not apply.

Claim 1b. Does not apply.

Claim 2.

Let $\text{arr}(K) = m$. Let $\vec{x} = (x_1, \ldots, x_m)$ where the x_i are in U_1^{null}. Similarly, let $\vec{y} = (y_1, \ldots, y_m)$ where the y_i are in U_1. We can also assume that $\vec{x} \sim \vec{y}$. We need to show that $[\![T]\!]_j\eta(\vec{x}) = [\![F_{\text{null}}(T)]\!]_s\eta'(\vec{y})$.

We can calculate denotations on both sides of the equation:

- $$\begin{aligned}
  [\![T]\!]_j\eta(\vec{x}) &= [\![\Pi_j(X :: *_n).T']\!]_j\eta(\vec{x}) \\
  &= (\prod_{a \in U_1^{null}} [\![T']\!]_j(\eta[X \to a]))(\vec{x}) \\
  &= ([\![T']\!]_j\eta[X \to x_1])(x_2, \ldots, x_m) \quad \text{since } x_1 \in U_1^{null}
  \end{aligned}$$
- $$\begin{aligned}
  [\![F_{\text{null}}(T)]\!]_s\eta'(\vec{y}) &= [\![F_{\text{null}}(\Pi_j(X :: *_n).T')]\!]_s\eta'(\vec{y}) \\
  &= [\![\Pi_s(X :: *).F_{\text{null}}(T')]\!]_s\eta'(\vec{y}) \\
  &= (\prod_{a \in U_1} [\![F_{\text{null}}(T')]\!]_s(\eta'[X \to a]))(\vec{y}) \\
  &= ([\![F_{\text{null}}(T')]\!]_s\eta'[X \to y_1])(y_2, \ldots, y_m) \quad \text{since } y_1 \in U_1
  \end{aligned}$$

So we need to show that

$$([\![T']\!]_j\eta[X \to x_1])(x_2, \ldots, x_m) = ([\![F_{\text{null}}(T')]\!]_s\eta'[X \to y_1])(y_2, \ldots, y_m)$$

Now recall the form of (KJ-Pi)

$$\frac{\Gamma, X :: *_n \vdash T' :: K'}{\Gamma \vdash \Pi_j(X :: *_n).T' :: *_n \Rightarrow K'} \qquad \text{(KJ-Pi)}$$

There are two cases, depending on whether K' is a base kind or not.

- If K' is a base kind, then $m = 1$ and we need to show $[\![T']\!]_j\eta[X \to x_1] = [\![F_{\text{null}}(T')]\!]_s\eta'[X \to x_1]$. From $\Gamma, X :: *_n \vdash_j T' :: K'$ and the induction hypothesis we get that $[\![T']\!]_j\eta'' = [\![F_{\text{null}}(T')]\!]_s\eta'''$ for all environments η'' and η''', where $\eta'' \vDash \Gamma, X :: *_n$ and $\eta'' \sim \eta'''$. Notice that we can set $\eta'' = \eta[X \to x_1]$ and $\eta''' = \eta'[X \to y_1]$. This works because $x_1 \in U_1^{null}$ and $x_1 \sim y_1$.
- If K' is a type application, we must have $\text{arr}(K') = m - 1 >= 1$. We need to show that $([\![T']\!]_j\eta[X \to x_1])(x_2, \ldots, x_m) = ([\![F_{\text{null}}(T')]\!]_s\eta'[X \to y_1])(y_2, \ldots, y_m)$. Again, from the induction hypothesis we get that $[\![T']\!]_j\eta''(\vec{x'}) = [\![F_{\text{null}}(T')]\!]_s\eta'''(\vec{y'})$, where $\eta'' \vDash \Gamma, X :: *_n$, $\eta'' \sim \eta'''$, $\vec{x'}$ is a vector of elements of U_1^{null} of length $m - 1$, $\vec{y'}$ is similarly a vector of elements of U_1 of length $m - 1$, and $\vec{x'} \sim \vec{y'}$. All of $\eta'', \eta''', \vec{x'}, \vec{y'}$ can be chosen arbitrarily as long as they satisfy the conditions above. We claim that we can choose $\eta'' = \eta[X \to x_1], \eta''' = \eta'[X \to y_1], \vec{x'} = (x_2, \ldots, x_m)$ and $\vec{y'} = (y_2, \ldots, y_m)$. First notice that $x_1 \in U_1^{null}$ and $y_1 \in U_1$ with $x_1 \sim y_1$, so $\eta'' \vDash \Gamma[X \to *_n]$ and $\eta'' \sim \eta'''$. Finally, $\vec{x} \sim \vec{y}$ implies $\vec{x'} \sim \vec{y'}$.

Case (KJ-App) K is an arbitrary kind.

$$T = \text{App}_j(T_1, T_2), F_{\text{null}}(T) = \text{App}_s(F_{\text{null}}(T_1), A_{\text{null}}(T_2))$$

Recall the rule in question

$$\frac{\Gamma \vdash_j T_1 :: *_n \Rightarrow K \qquad \Gamma \vdash_j T_2 :: *_n}{\Gamma \vdash_j \text{App}_j(T_1, T_2) :: K} \qquad \text{(KJ-App)}$$

There are two cases:

- If K is a base kind, then we need to prove claims 1a and 1*b*.

  Claim 1a. We need to show that $[\![\mathrm{App}_j(T_1, T_2)]\!]_j\eta = [\![\mathrm{App}_s(F_{\mathrm{null}}(T_1), A_{\mathrm{null}}(T_2))]\!]_s\eta'$.

  We can compute the denotations on both sides:

  - $[\![\mathrm{App}_j(T_1, T_2)]\!]_j\eta = [\![T_1]\!]_j\eta([\![T_2]\!]_j\eta)$
  - $[\![\mathrm{App}_s(F_{\mathrm{null}}(T_1), A_{\mathrm{null}}(T_2))]\!]_s\eta' = [\![F_{\mathrm{null}}(T_1)]\!]_s\eta'([\![A_{\mathrm{null}}(T_2)]\!]_s\eta')$
    By the induction hypothesis, we know that for arbitrary types $x \in U_1^{null}$ and $y \in U_1$ with $x \sim y$, we have $[\![T_1]\!]_j\eta(x) = [\![F_{\mathrm{null}}(T_1)]\!]_s\eta'(y)$. Again by the induction hypothesis, we have $[\![T_2]\!]_j\eta \sim [\![A_{\mathrm{null}}(T_2)]\!]_s\eta'$. Additionally, by Lemma 3.3.3, we know that $\Gamma \vdash_j T :: *_n$ implies $[\![T_2]\!]_j\eta \in U_1^{null}$. This means that we can choose $x = [\![T_2]\!]_j\eta$ and $y = [\![T_2]\!]_s\eta'$ as needed.

  Claim 1b. Notice that $A_{\mathrm{null}}(T) = \mathrm{App}_s(F_{\mathrm{null}}(T_1), A_{\mathrm{null}}(T_2)) = F_{\mathrm{null}}(T)$. The claim then follows from 1a above.

- If K is a type application, we need to prove claim 2. Let $\mathrm{arr}(K) = m \geq 1$. Then $\mathrm{arr}(*_n \Rightarrow K) = m + 1$. Let $\vec{x}$ and $\vec{y}$ be arbitrary m-vectors as in the statement of the theorem, such that $\vec{x} \sim \vec{y}$. We need to show that $[\![\mathrm{App}_j(T_1, T_2)]\!]_j\eta(\vec{x}) = [\![\mathrm{App}_s(F_{\mathrm{null}}(T_1), A_{\mathrm{null}}(T_2))]\!]_s\eta'(\vec{y})$. Now build the vectors $\vec{x'} = ([\![T_2]\!]_j\eta, x_1, \ldots, x_m)$ and $\vec{y'} = ([\![A_{\mathrm{null}}(T_2)]\!]_s\eta', y_2, \ldots, y_m)$. Notice that $[\![T_2]\!]_j\eta \sim [\![A_{\mathrm{null}}(T_2)]\!]_s\eta'$ by the induction hypothesis, so $\vec{x'} \sim \vec{y'}$. Further, both new vectors have $m + 1$ elements. By the induction hypothesis applied to $\Gamma \vdash_j T_1 :: *_n \Rightarrow K$ we can conclude that $[\![T_1]\!]_j\eta(\vec{x'}) = [\![F_{\mathrm{null}}(T_1)]\!]_s\eta'(\vec{y'})$. But

  $$\begin{aligned}[\![\mathrm{App}_j(T_1, T_2)]\!]_j\eta(\vec{x}) &= ([\![T_1]\!]_j\eta([\![T_2]\!]_j\eta))(\vec{x}) \\ &= [\![T_1]\!]_j\eta(\vec{x'})\end{aligned}$$

  And similarly,

  $$\begin{aligned}[\![\mathrm{App}_s(F_{\mathrm{null}}(T_1), A_{\mathrm{null}}(T_2))]\!]_s\eta'(\vec{y}) &= ([\![F_{\mathrm{null}}(T_1)]\!]_s\eta'([\![A_{\mathrm{null}}(T_2)]\!]_s\eta'))(\vec{y}) \\ &= [\![F_{\mathrm{null}}(T_1)]\!]_s\eta'(\vec{y'})\end{aligned}$$

  And we are done.

Case (KJ-Var) $K = *_n, T = X, F_{\mathrm{null}}(T) = X + \texttt{Null}$, and $A_{\mathrm{null}}(T) = X$.

Claim 1a.

$[\![X]\!]_j\eta = \eta(X)$ and $[\![X + \texttt{Null}]\!]_s\eta' = \eta'(X) \cup \{\texttt{null}\}$. Notice that $\eta(X) = \eta(X) \cup \{\texttt{null}\}$ since $\texttt{null} \in \eta(X)$, because $\eta \vDash \Gamma$. But then $\eta \sim \eta'$ implies $\eta(X) \cup \{\texttt{null}\} = \eta'(X) \cup \{\texttt{null}\}$ as needed.

Claim 1b. We need to show that $\eta(X) \sim \eta'(X)$. This follows directly from $\eta \sim \eta'$.

Claim 2. Does not apply.

Case (KJ-Null) $K = *$ and T is an arbitrary type.

Claims 1a and 1b.

From the antecedent of the rule, we know that $\Gamma \vdash_j T :: *_n$. By the induction hypothesis, we have $[\![T]\!]_j\eta = [\![F_{\text{null}}(T)]\!]_s\eta'$ and $[\![T]\!]_j\eta \sim [\![A_{\text{null}}(T)]\!]_s\eta'$. This is exactly what we need to show, so we are done.

Claim 2. Does not apply.

Case (KJ-NonNull)

Claims 1a and 1b.

From the antecedent of the rule, we know that $\Gamma \vdash_j T :: *_v$. By the induction hypothesis, we have $[\![T]\!]_j\eta = [\![F_{\text{null}}(T)]\!]_s\eta'$ and $[\![T]\!]_j\eta \sim [\![A_{\text{null}}(T)]\!]_s\eta'$. This is exactly what we need to show, so we are done.

Claim 2. Does not apply.

□

3.4.2 Discussion

Theorem 3.4.3 (soundness) has a couple of interesting implications.

- First, the theorem shows that our intuitive criteria for correctness of type nullification, "nullification does not change the meaning of types", was not far off the mark. The formal version that we showed correct could be summarized as "nullification does not change the meaning of types that have base kinds".

- Second, we were able to prove correct a slightly more sophisticated version of nullification where the type arguments are not nullified. This version is more useful than naive nullification (where type arguments are nullified), because it minimizes changes required to "migrate" Scala code to the world of explicit nulls. For example, suppose we are working with a Java library `ListOps` that provides operations on Java lists of the form `List<T>`. Further, suppose that `List<T>` is an *invariant* generic, meaning that `List<A>` is a subtype of `List<B>` *only if* `A` is equal to `B`. Now imagine using the Java library from Scala:

```
val lst: List[Int] = ...
val gt3 = ListOps.filter(lst, (x: Int) => x > 3)
```

After naive nullification, the definition of `gt3` will no longer typecheck. This is because the signature of `filter` (which is Java-defined), changes from

```
List<T> filter( List<T> lst, ...)
```

to

```
List<T|Null>|Null filter(List<T|Null>|Null lst, ...)
```

Since `List<T>` is invariant, we need to update the type of `lst` from `List[Int]` to `List[Int|Null]`.

By contrast, with our version of nullification, `filter` becomes

```
List<T>|Null filter(List<T>|Null lst, ...)
```

and now no changes are required.

- Additionally, the theorem implies that it does not matter whether we use nullable or non-nullable type arguments with a generic that has been nullified. For example (still thinking about `List<T>`), on the Java side, `List<T>` can only be instantiated with a nullable type: `List<Integer>` is valid, but `List<int>` is not. From Scala, however, both `List[Int|Null]` (nullable argument) `List[Int]` (non-nullable) are valid. Theorem 3.4.3 says that the sets denoting both of these last two types are equal, because `Int|Null` $\sim$ `Int`.

3.5 Related Work

The model of predicative System F that I used in this chapter is based on the one given in Mitchell [1996] (which, in turn, is based on Bruce et al. [1990]). The denotations for sums and product types are standard in the literature.

There is one deviation from Mitchell [1996] in how I construct denotations for generics. The standard way is to say that the denotation of a generic type is an (infinite) Cartesian product, whereas I use a simple function on types. That is, instead of saying

$$[\![\Pi_s(X :: *).S]\!]_s\eta = \prod_{a \in U_1} [\![S]\!]_j(\eta[X \to a])$$

I define

$$[\![\Pi_s(X :: *).S]\!]_s\eta = \lambda(a \in U_1).[\![S]\!]_s(\eta[X \to a])$$

Given a family of sets $\{X_i\}$ indexable by a set I, the infinite Cartesian product $\prod_{i\in I} X_i$ is defined as

$$\prod_{i\in I} X_i = \left\{ f : I \to \bigcup_{i\in I} X_i | \forall i. f(i) \in X_i \right\}$$

The reason for the discrepancy is that λ_j and λ_s have type applications at the type level (e.g. $\mathrm{App}_j(S, T)$), whereas in System F, type applications are terms ($t\ [T]$). If we use the variant with the Cartesian product, then $[\![\mathrm{App}_s(\Pi_s(X :: *).X, \mathtt{int}_s)]\!]_s\emptyset$ would be an *element* of $[\![\mathtt{int}_s]\!]_s$ (an element of $\mathbb{Z}$). However, what we need for the soundness theorem is that $[\![\mathrm{App}_s(\Pi_s(X :: *).X, \mathtt{int}_s)]\!]_s\emptyset$ be *equal* to $[\![\mathtt{int}_s]\!]_s$, hence the second definition.

The novelty in this chapter is the use of denotational semantics for reasoning specifically about nullification. I am not aware of any related work that formalizes and proves soundness of nullification.

3.6 Conclusions

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

That the meaning of types with base kinds remains unchanged is important, because program values always have base kinds. If nullification *underapproximated* type denotations (e.g. by mapping Java's `String` to Scala's `String`), then we would see unexpected values during execution, leading to unsoundness. On the other hand, if nullification *overapproximated* type denotations (e.g. by mapping Java's `String` to Scala's `Any`), then

usability would suffer. Preservation of type denotations then gives us the most usable interoperability that is still sound.

The meaning of generics *is* changed by nullification. This reflects the fact that, in λ_s and Scala, type arguments can be either value or reference types, while in λ_j and Java only reference types can be used. The soundness theorem (Theorem 3.4.3) in this chapter shows that *fully-applied* generics (which have base kinds) remain unchanged. This means that Java types corresponding fully-applied generics (e.g. `ArrayList<String>`), can be represented *exactly* in Scala. The other direction does not hold; e.g. the Scala type `List[Int]` cannot be represented directly in Java (because `Int` is a value type). Instead, `List[Int]` must be translated as `List<Integer>` or `List<Object>`, where `Integer` is the Java type for boxed integers. The type translation from Scala to Java is not modelled in this chapter and remains as future work.

Chapter 4

Blame for Null

Even though type nullification preserves the meaning of types, when we run Scala code that depends on Java code, nullability errors can still occur. In this chapter, I show how to reason about the presence and provenance of such errors using the concept of *blame* from gradual typing [Wadler and Findler, 2009]. Specifically, I introduce a calculus, $\lambda_{\texttt{null}}$, where some terms are typed as *implicitly nullable* and others as *explicitly nullable*. Just like in the original blame calculus, interactions between both kinds of terms are mediated by *casts* with attached *blame labels*, which indicate the origin of errors. On top of $\lambda_{\texttt{null}}$, I then create a second calculus, $\lambda^{s}_{\texttt{null}}$, which closely models the interoperation between Java and Scala code as it relates to nullability. The main result of this chapter is a theorem that says that if an interop cast in $\lambda^{s}_{\texttt{null}}$ fails, an implicitly nullable term is to blame. Most of the results in this chapter have been formalized in the Coq theorem prover.

4.1 Nullability Errors

The soundness theorem (Theorem 3.4.3) from Chapter 3 showed that type nullification leaves the meaning of types (almost) unchanged. Since we already have a "soundness" theorem, are we done? Unfortunately, the answer is *no*, because Theorem 3.4.3 talks only about *types* and says nothing about *terms*. In particular, we do not know whether in a world where Scala has explicit nulls, there can still be nullability (runtime) errors. As it turns out, such errors are still possible. We are then in need of new theorems that characterize whether errors can occur in specific programs and, when they *do* occur, are able to trace back the cause of the errors.

Here are two hypotheses about nullability errors, both of which will turn out to be incorrect:

1. Explicit nulls statically rule out nullability errors.

   This seems plausible (and is what we would like to be true): after all, the type system *will* disallow some programs that would lead to runtime errors, such as

   ```
   val s: String = null // error: expected a value of type 'String',
                        // but got 'Null'
   val len = s.length
   ```

   or its dual

   ```
   val s: String|Null = null // ok, nullable string
   val len = s.length // error: 'String|Null' has no 'length' field
   ```

   However, Scala programs can use Java-defined methods, and Java code is still typed with implicit nulls. This can lead to errors while executing Java-defined code:

   ```
   // Java-defined
   int javaGetLength(String s) {
     return s.length()
   }
   ```

   ```
   // Scala-defined
   def scalaGetLength(s: String|Null): Int = {
     return javaGetLength(s) // Defer to Java helper.
   }

   scalaGetLength(null) // throws NullPointerException
   ```

   The problem here is that type nullification changes `javaGetLength`'s argument type from `String` to `String|Null`. That is, *the Java-defined method advertises itself as being able to handle all strings and* ***`null`****. However, it violates its own contract by not performing any null checks*, leading to an error if the argument `s` is `null`.

   This leads us to our second incorrect attempt at characterizing when nullability errors occur.

2. Nullability errors *do* happen, but *only while running Java code*.

   The first step is to clarify what we mean by "running". Looking at the example above, we could refine the hypothesis to be

   > If an error (null pointer exception) occurs, the instruction currently being executed must be "Java-generated".

   Even though this definition is still informal, it seems to hold for the example we saw above. Unfortunately, this will not do either; consider

```
1 // S.scala
2 class StringOps {
3   def len(s: String): Int = s.length
4 }
5
6 def main() = {
7   val dec = new Decorator(new StringOps)
8   val len2x = dec.twiceLen(null) // throws NullPointerException
9 }
```

```
1 // J.java
2 class Decorator {
3   StringOps ops;
4
5   Decorator(StringOps ops) {
6     this.ops = ops
7   }
8
9   int twiceLen(String s) {
10    return ops.len(s) * 2;
11  }
12 }
```

   When the exception is thrown, the call stack looks as follows (assuming the stack grows upwards):

```
StringOps.len (S.scala:3)
```

```
Decorator.twiceLen (J.java:10)
StringOps.main (S.scala:8)
```

That is, the exception occurred while executing Scala code! This invalidates our hypothesis. The problem is that *Java code can access Scala code, but the Java type system does not track nullability.* In this case, `Decorator::twiceLen`, which is Java-defined, calls `StringOps::len`, which is Scala-defined. Crucially, because of erasure, Java "casts"[1] `StringOps::len`'s original Scala type

$$\texttt{String}_{\texttt{Scala}} \to \texttt{Int}$$

to

$$\texttt{String}_{\texttt{Java}} \to \texttt{Int}$$

Informally, for such a cast to be sound, we need the first type to be a subtype of the second:

$$\texttt{String}_{\texttt{Scala}} \to \texttt{Int} <: \texttt{String}_{\texttt{Java}} \to \texttt{Int}$$

The subtyping rule for functions is contravariant in the argument type, so we need $\texttt{String}_{\texttt{Java}} <: \texttt{String}_{\texttt{Scala}}$. However, when viewed as sets, $\texttt{String}_{\texttt{Java}} = \texttt{String}_{\texttt{Scala}} \cup \{\texttt{null}\}$, so $\texttt{String}_{\texttt{Java}}$ is *not* a subtype of $\texttt{String}_{\texttt{Scala}}$. This makes the cast unsound, as witnessed by the exception.

To summarize, we have seen that nullability errors do happen even in the presence of explicit nulls. Further, they can happen while executing not only Java code, but also Scala code. This is disappointing, because the motivation for making `nulls` explicit in Scala was precisely to avoid runtime errors. However, if we take the presence of errors as given, what is the next best thing? That is, what can we say or guarantee about nullability errors in these dual Java-Scala programs?

Intuitively, the reason there are errors is that the Java type system is "leaking" the unsoundness that comes from being implicitly nullable into the Scala world. If we had a way of assigning responsibility to Java or Scala every time a runtime error occurs, we would then expect that Java could always be *blamed* for such errors. In fact, we can re-purpose techniques from *gradual typing* to make this kind of blame assignment.

[1] This is a conceptual cast; there are no runtime checks.

Roadmap

Here is an outline of the rest of the chapter:

- Section 4.2 briefly reviews the *blame calculus* of Wadler and Findler [2009].
- Section 4.3 adapts the ideas of the blame calculus to the nullability setting. We will introduce a new core calculus, $\lambda_{\texttt{null}}$, that can express both implicit and explicit nullability in its types. The main results of this section are theorems 4.3.10 and 4.3.11. Taken together, they constrain how $\lambda_{\texttt{null}}$ terms can fail during evaluation.
- Section 4.5 builds on $\lambda_{\texttt{null}}$ to construct a higher-level calculus, $\lambda^s_{\texttt{null}}$. $\lambda^s_{\texttt{null}}$ models language interoperability more closely, because terms are stratified into an "explicit `nulls`" sublanguage and an "implicit `nulls`" one. The main result in this section is Theorem 4.5.4, which implies that if interop casts in $\lambda^s_{\texttt{null}}$ fail, then the implicit sublanguage can be blamed.
- Finally, Section 4.4 informally explains how we can use the concept of blame to assign responsibility for nullability errors in the Java/Scala setting.

4.2 Blame Calculus

The blame calculus of Wadler and Findler [2009] models the interactions between less-precisely and more-precisely typed code. For example, the less-precisely typed code could come from a dynamically-typed (or unityped [Harper, 2016]) language, and the more-precisely typed code could come from a statically-typed language like Scala. The goal of the calculus is twofold:

- To characterize situations where errors can or cannot occur as a result of the interaction between both languages: e.g. "there will not be runtime errors, unless the typed code calls the untyped code".
- If runtime errors do occur, to assign *blame* (responsibility) for the error to some term present in the evaluation.

To do the above, the blame calculus extends the simply-typed lambda calculus with *casts* that contain *blame labels*[2]. The notation for casting a term s from a type S to another

[2]The original presentation in Wadler and Findler [2009] also adds *refinement types*, but we will not need them here.

type T with blame label p is[3]

$$s : S \Longrightarrow^{p} T$$

During evaluation, a cast might succeed, fail, or be *be broken up into further casts*. For example, suppose that we have a cast that turns 4 from an integer into a natural number. Such a cast would naturally succeed, and one step of evaluation then makes the cast disappear:

$$4 : \texttt{Int} \Longrightarrow^{p} \texttt{Nat} \longmapsto 4$$

A cast can also fail. This is when we use the blame label. For example, if we try to turn an integer into a string using a cast with blame label p, then we fail and blame p, written $\Uparrow p$:

$$4 : \texttt{Int} \Longrightarrow^{p} \texttt{String} \longmapsto \Uparrow p$$

If the cast is *higher-order*, however, things get tricky. How are we to determine whether a function of type $\texttt{Int} \to \texttt{Int}$ also has type $\texttt{Nat} \to \texttt{Nat}$?

$$(\lambda(x\colon \texttt{Int}).x - 2) : \texttt{Int} \to \texttt{Int} \Longrightarrow^{p} \texttt{Nat} \to \texttt{Nat}$$

Informally, the cast above is saying: "if you provide as input a `Nat` that is *also* an `Int`, the function will return an `Int` that is *also* a `Nat`". Intuitively, the cast is incorrect, because the function can return negative numbers. In general, however, we cannot hope to statically ascertain the validity of a higher-order cast. The insight about what to do here comes from work on higher-order contracts [Findler and Felleisen, 2002]. The key idea is to *delay* the evaluation of the cast *until the function is applied*. That is, we consider the entire term above, the lambda plus its cast, a *value*. Then, if we need to apply the lambda wrapped in a cast, we use the following rule:

$$((v : (A \to B) \Longrightarrow^{p} (A' \to B'))\ w) \longmapsto (v\ (w : A' \Longrightarrow^{\overline{p}} A)) : B' \Longrightarrow^{p} B$$

Notice how the original cast was decomposed into two separate casts on subterms. This rule says that applying a lambda wrapped in a cast involves three steps:

- First, we cast the argument w, which is expected to have type A', to type A.
- Then we apply the function v to its argument, as usual.
- Finally, we cast the result of the application from B' back to the expected type B.

[3]The notation for casts used in this chapter comes from [Ahmed et al., 2011].

Also notice how the blame label in the cast $w : A' \Longrightarrow^{\overline{p}} A'$ changed from p to its *complement* $\overline{p}$. We can think of blame labels as strings (or, more abstractly, arbitrary sentinel values). We assume the existence of a *complement* function with type `Blame label` $\to$ `Blame label`, and write $\overline{p}$ for the label that is the complement of blame label p. The complement operation is *involutive*, meaning that it is its own inverse: $\overline{\overline{p}} = p$.

When a runtime error happens, complementing blame labels leads to *two* kinds of blame: *positive* and *negative*:

Positive blame Given a cast with blame label p, positive blame happens when the term *inside* the cast is responsible for the failure. In this case, the (failed) term will evaluate to $\Uparrow p$. For example, recall our example with the faulty function that subtracts two from its argument:

$$
\begin{aligned}
& ((\lambda(x\colon \texttt{Int}).x - 2) : \texttt{Int} \to \texttt{Int} \Longrightarrow^{p} \texttt{Nat} \to \texttt{Nat})\ 1 \\
\longmapsto\ & ((\lambda(x\colon \texttt{Int}).x - 2)\ (1 : \texttt{Nat} \Longrightarrow^{\overline{p}} \texttt{Int})) : \texttt{Int} \Longrightarrow^{p} \texttt{Nat} \\
\longmapsto\ & ((\lambda(x\colon \texttt{Int}).x - 2)\ 1) : \texttt{Int} \Longrightarrow^{p} \texttt{Nat} \\
\longmapsto\ & (1 - 2) : \texttt{Int} \Longrightarrow^{p} \texttt{Nat} \\
\longmapsto\ & -1 : \texttt{Int} \Longrightarrow^{p} \texttt{Nat} \\
\longmapsto\ & \Uparrow p
\end{aligned}
$$

The term being cast (the lambda) is reponsible for the failure, because it promised to return a `Nat`, which -1 is not.

Negative blame If the cast fails because it is provided an argument of an incorrect type by its *context* (surrounding code), then we will say the failure has negative blame. In this case, the term will evaluate to $\Uparrow \overline{p}$. For example, suppose our example function is used in a unityped context, where the only type is $\star$. Without help from its type system, the context might try to pass in a `String` as argument:

$$
\begin{aligned}
& ((\lambda(x\colon \texttt{Int}).x - 2) : \texttt{Int} \to \texttt{Int} \Longrightarrow^{p} \star \to \star)\ \texttt{"one"} \\
\longmapsto\ & ((\lambda(x\colon \texttt{Int}).x - 2)\ (\texttt{"one"} : \star \Longrightarrow^{\overline{p}} \texttt{Int})) : \texttt{Int} \Longrightarrow^{p} \star \\
\longmapsto\ & \Uparrow \overline{p}
\end{aligned}
$$

Because the context tried to pass an argument that is not an `Int`, we blame the failure on the context.

4.2.1 Well-typed Programs Can't Be Blamed

The central result in Wadler and Findler [2009] is a *blame theorem* that provides two guarantees:

- Casts from less-precise to more-precise types, like $v : \texttt{Int} \to \texttt{Int} \Longrightarrow^{p} \texttt{Nat} \to \texttt{Nat}$, only fail with *positive* blame.
- Casts from more-precise to less-precise types, like $v : \texttt{Int} \to \texttt{Int} \Longrightarrow^{p} \star \to \star$, only fail with *negative* blame.

In both cases, the less precisely typed code is assigned responsibility for the failure. Wadler and Findler [2009] summarize this result with the slogan "well-typed programs can't be blamed", itself a riff on an earlier catchphrase, "well-typed programs cannot go wrong", due to Milner [Milner, 1978].

In the next section, I will show how we can adapt ideas from the blame calculus to reason about nullability errors.

4.3 A Calculus with Implicit and Explicit Nulls

In this section, I introduce the $\lambda_{\texttt{null}}$ ("lambda null") calculus. $\lambda_{\texttt{null}}$ is based on the blame calculus of Wadler and Findler [2009], and the presentation in Wadler [2015].

$\lambda_{\texttt{null}}$ contains the two key ingredients we need to model interoperation between Java and Scala, as it relates to `null`:

- Types that are *implicitly* nullable and types that are *explicitly* nullable.
- Casts that mediate the interaction between the types above, along with blame labels to track responsibility for failures, should they occur.

The terms and types of $\lambda_{\texttt{null}}$ are shown in Figure 4.1, and are explained below. Section 4.5 shows how to use $\lambda_{\texttt{null}}$ to model the interaction between two languages, each treating nullability differently (like Java and Scala). This section focuses on $\lambda_{\texttt{null}}$ and its metatheory.

x, y, z	**Variables**
p, q	**Blame labels**
$\overline{p}$	**Blame complement**
$f, s, t ::=$	**Terms**
x	variable
null	null literal
$\lambda(x\colon T).s$	abstraction
$s\ t$	application
$\mathrm{app}(f, s, t)$	safe application
$s : S \Longrightarrow^p T$	cast
$u, v ::=$	**Values**
$\lambda(x\colon T).s$	abstraction
null	null literal
$v : S \Longrightarrow^p T$	cast
$r ::=$	**Results**
t	term
$\Uparrow p$	blame
$S, T, U ::=$	**Types**
Null	null
$\#(S \to T)$	presumed non-nullable function
$?(S \to T)$	safe nullable function
$!(S \to T)$	unsafe nullable function

Figure 4.1: Terms and types of $\lambda_{\texttt{null}}$

4.3.1 Values of $\lambda_{\texttt{null}}$

A value in $\lambda_{\texttt{null}}$ is any of the following: an *abstraction* $\lambda(x\colon T).s$, the *null literal* `null`, or another value v wrapped in a cast, $v : S \Longrightarrow^p T$.

The motivation for classifying certain casts as values is as follows. Consider the cast $\texttt{null} : \texttt{Null} \Longrightarrow^p !(S \to T)$. As we will see later, $!(S \to T)$ is an *unsafe nullable* function type, so the cast can fail. However, the cast does not fail immediately; instead, the cast only fails if *we try to apply* the (null) function to an argument, like so $(\texttt{null} : \texttt{Null} \Longrightarrow^p !(S \to T))\ w$. This matches Java's behaviour, where passing a `null` when an object is expected only triggers an exception if we try to select a field or method from the `null` object:

```
String s = null; // no exception is raised here
s.length()       // an exception is raised only when we try to select a method or field
```

4.3.2 Terms of $\lambda_{\texttt{null}}$

A term of $\lambda_{\texttt{null}}$ is either a variable x, the null literal `null`, an abstraction $\lambda(x\colon T).s$, an application $s\ t$, a *safe application* $\text{app}(s, t, u)$, or a cast $s : S \Longrightarrow^p T$. The meaning of most terms is standard; the interesting ones are explained below:

- The `null` literal is useful for modelling null pointer exceptions. Specifically, an application $s\ t$, where s reduces to `null`, results in a failure (I will say more about this later on).
- A safe application $\text{app}(s, t, u)$ is a regular application that can also handle the case where s is `null`. If s is non-null, then the safe application behaves like the regular application $s\ t$. However, if s is `null` then the entire safe application reduces to u.

  Safe applications could be desugared into a combination of if-expressions and flow typing, if we had the latter:

  $$\text{app}(s, t, u) \equiv \texttt{if (s != null) then s t else u}$$

  For this desugaring we would need flow typing, because within the `then` branch we need to be able to assume that s is non-null. Safe applications allow us to work with nullable values without introducing flow typing.

  Safe applications are similar to Kotlin's "Elvis" operator [Kotlin Foundation, b].
- The cast $s : S \Longrightarrow^p T$ is used to change the type of s from S to T. The blame label p will be used to assign blame for a failure that happens due to a cast.

Finally, the result of evaluating a $\lambda_{\texttt{null}}$ term is either a value v or an error with blame p, denoted by $\Uparrow p$.

4.3.3 Types of $\lambda_{\texttt{null}}$

The types of $\lambda_{\texttt{null}}$ are also shown in Figure 4.1. There are four kinds of types:

- The `Null` type contains a single element: `null`.
- The *presumed non-nullable* function type $\#(S \to T)$, as the name indicates, contains values that *should not* be `null`. However, the value might *still* end up being `null`, through casts. This corresponds to non-nullable types like $\texttt{String}_{\texttt{Scala}}$. To be more concise, from now on I will refer to these types simply as *non-nullable* function types.

$$\boxed{\Gamma \vdash t : T}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \text{ (T-Var)}$$

$$\Gamma \vdash \texttt{null} : \texttt{Null} \quad \text{(T-Null)}$$

$$\frac{\Gamma, x : S \vdash s : T}{\Gamma \vdash \lambda(x\colon S).s : \#(S \to T)} \text{ (T-Abs)}$$

$$\frac{\Gamma \vdash s : \diamond_{\#,!}(S \to T) \qquad \Gamma \vdash t : S}{\Gamma \vdash s\ t : T} \text{ (T-App)}$$

$$\frac{\Gamma \vdash f : \diamond_{?,!}(S \to T) \qquad \Gamma \vdash s : S \qquad \Gamma \vdash t : T}{\Gamma \vdash \text{app}(f, s, t) : T} \text{ (T-SafeApp)}$$

$$\frac{\Gamma \vdash s : S \qquad S \rightsquigarrow T}{\Gamma \vdash (s : S \Longrightarrow^{p} T) : T} \text{ (T-Cast)}$$

$$\boxed{S \rightsquigarrow T}$$

$$\texttt{Null} \rightsquigarrow \texttt{Null} \text{ (C-NullRefl)}$$

$$\texttt{Null} \rightsquigarrow \diamond_{?,!}(S \to T) \text{ (C-Null)}$$

$$\frac{S' \rightsquigarrow S \qquad T \rightsquigarrow T'}{\diamond_{\#,?,!}(S \to T) \rightsquigarrow \diamond_{\#,?,!}(S' \to T')} \text{ (C-Arrow)}$$

Figure 4.2: Typing and compatibility rules of $\lambda_{\texttt{null}}$

- A value with *safe nullable* function type $?(S \to T)$ is allowed to be `null`. The type system will ensure that any such functions are applied using safe applications. This corresponds to nullable union types like $\texttt{String}_{\texttt{Scala}}\texttt{|Null}$.
- By contrast, a value with *unsafe nullable* function type $!(S \to T)$ is also allowed to be `null`, but the type system does not enforce a null check before an application. That is, if s has type $!(S \to T)$, the type system will allow both $s\ t$ and $\text{app}(s, t, u)$, even though the former might fail. This corresponds to types in Java, which are implicitly nullable.

4.3.4 Typing $\lambda_{\texttt{null}}$

The typing rules for $\lambda_{\texttt{null}}$ are shown in Figure 4.2. The three interesting rules are T-App, T-SafeApp, and T-Cast:

- **(T-App)** The rule for a type application $s\ t$ is *almost* standard, except that s can not only have type $\#(S \to T)$, but *also* the *unsafe nullable* function type $!(S \to T)$. This represents the fact that in a type system with implicit nullability (like Java's), the type system allows operations (in this case, function applications) that can lead to null-related errors. We use the syntax $\Gamma \vdash s : \diamond_{\#,!}(S \to T)$ to indicate that either $\Gamma \vdash s : \#(S \to T)$ *or* $\Gamma \vdash s : !(S \to T)$. This kind of judgment, where the type is ambiguous, is purely a convenience to simplify the presentation: we could instead have *two* rules, T-App1 and T-App2, each using a different function type.

- **(T-SafeApp)** To type a safe application $\mathrm{app}(f, s, t)$, we check that f is a nullable function type; that is, it must have type $?(S \to T)$ *or* $!(S \to T)$ (if f had type $\#(S \to T)$ we would use T-App). Notice that the type of s must be S (the argument type), but t must have type T (the return type). This is because t is the "default" value that we return if f is null.

- **(T-Cast)** To type a cast $s : S \Longrightarrow^p T$ we check that s indeed has the source type S. The entire cast then has type T. Additionally, we make sure that S and T are *compatible*, written $S \rightsquigarrow T$. Type compatibility is described below.

Notice that the type of `null` is always `Null`, so in order to get a nullable function we need to use casts[4]. For instance,

$$\textsc{T-Cast}\ \frac{\textsc{T-Null}\ \dfrac{}{\vdash \texttt{null} : \texttt{Null}} \qquad \textsc{C-Null}\ \dfrac{}{\texttt{Null} \rightsquigarrow ?(\texttt{Null} \to \texttt{Null})}}{\vdash \texttt{null} : \texttt{Null} \Longrightarrow^p ?(\texttt{Null} \to \texttt{Null}) : ?(\texttt{Null} \to \texttt{Null})}$$

Compatibility

Compatibility is a binary relation on types that is used to limit (albeit only slightly) which casts are valid. Given types S and T, we can cast S to T only if $S \rightsquigarrow T$. The compatibility rules are shown in Figure 4.2.

Lemma 4.3.1. *Compatibility is reflexive, but is neither symmetric nor transitive.*

Proof. The reflexive case follows by a simple induction.

[4] In an implementation of $\lambda_{\texttt{null}}$, some of the verbosity in this example could be avoided via desugaring, and the casts might be ellided.

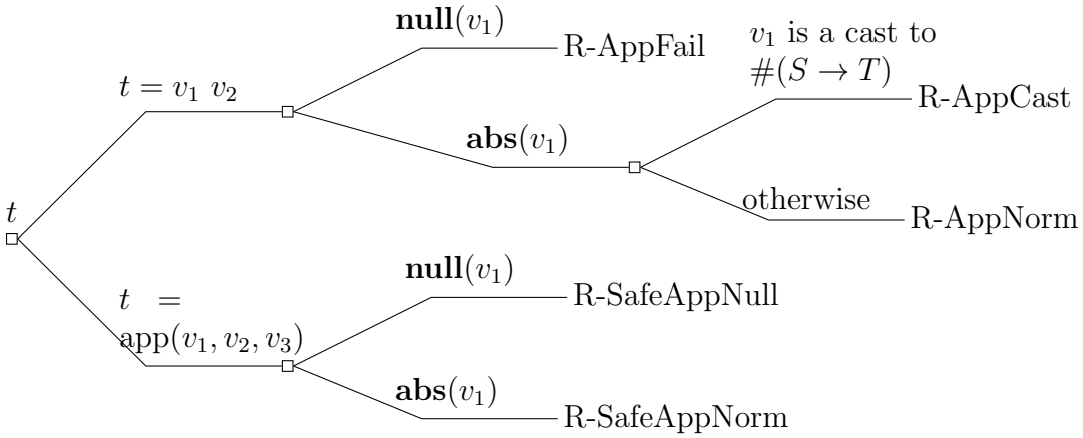
A counter-example to symmetry is that $\texttt{Null} \rightsquigarrow \#(\texttt{Null} \to \texttt{Null})$, but the latter is not compatible with the former.

A counter-example to transitivity is that $\texttt{Null} \rightsquigarrow ?(\texttt{Null} \to \texttt{Null})$ and $?(\texttt{Null} \to \texttt{Null}) \rightsquigarrow \#(\texttt{Null} \to \texttt{Null})$, but $\texttt{Null}$ is not compatible with $\#(\texttt{Null} \to \texttt{Null})$. □

4.3.5 Evaluation of $\lambda_{\texttt{null}}$

The evaluation rules for $\lambda_{\texttt{null}}$ are given in Figure 4.3. $\lambda_{\texttt{null}}$ has a small-step operational semantics that uses evaluation contexts. Notice that the result r of an evaluation step can be a term *or* an error, denoted by $\Uparrow p$.

The decision tree below shows a simplified view of the evaluation rules:



The unary predicates on types **null** and **abs** test whether a value v is equal to `null` or to a lambda abstraction, respectively. Additionally, these predicates are able to "see through" casts.

Example 4.3.1. *The following hold:*

- $\textbf{\textit{null}}(\texttt{null})$
- $\textbf{\textit{null}}(\texttt{null} : \texttt{Null} \Longrightarrow^p \#(\texttt{Null} \to \texttt{Null}))$
- $\textbf{\textit{abs}}(\lambda(x\colon \texttt{Null}).x)$
- $\textbf{\textit{abs}}(\lambda(x\colon \texttt{Null}).x : \#(\texttt{Null} \to \texttt{Null}) \Longrightarrow^p ?(\texttt{Null} \to \texttt{Null}))$

$$\boxed{s \longmapsto r}$$

$$E[(\lambda(x\colon T).s)\ v] \longmapsto E[[v/x]s]\ \text{(R-App)}$$

$$\frac{\mathbf{abs}(v) \qquad v \gg v'}{E[v\ u] \longmapsto E[v'\ u]}\ \text{(R-AppNorm)}$$

$$\frac{\mathbf{null}(v)}{E[(v : S \Longrightarrow^{p} !(T \to U)\ )u] \longmapsto \Uparrow \overline{p}}\ \text{(R-AppFailExt)}$$

$$\frac{\mathbf{null}(v)}{E[\mathrm{app}(v, u, u')] \longmapsto E[u']}\ \text{(R-SafeAppNull)}$$

$$\frac{\mathbf{null}(v)}{E[(v : S \Longrightarrow^{p} \#(T \to U)\ )u] \longmapsto \Uparrow p}\ \text{(R-AppFailSelf)}$$

$$\frac{\mathbf{abs}(v) \qquad v \gg v'}{E[\mathrm{app}(v, u, u')] \longmapsto E[v'\ u]}\ \text{(R-SafeAppNorm)}$$

$$\frac{\mathbf{abs}(v)}{E[(v : \#(S_1 \to S_2) \Longrightarrow^{p} \#(T_1 \to T_2))\ u] \longmapsto E[(v\ (u : T_1 \Longrightarrow^{\overline{p}} S_1)) : S_2 \Longrightarrow^{p} T_2]}\ \text{(R-AppCast)}$$

$E ::=$ **Evaluation contexts**

$[]$	$v\ E$	$\mathrm{app}(E, s, s)$	$\mathrm{app}(v, v, E)$
$E\ s$	$E : S \Longrightarrow^{p} T$	$\mathrm{app}(v, E, s)$	

Auxiliary predicates

$$\boxed{\mathbf{null}(v)} \qquad\qquad \boxed{\mathbf{abs}(v)}$$

$$\mathbf{null}(\mathrm{null})\ \text{(N-Null)} \qquad\qquad \mathbf{abs}(\lambda(x\colon T).s)\ \text{(A-Abs)}$$

$$\frac{\mathbf{null}(v)}{\mathbf{null}(v : S \Longrightarrow^{p} T)}\ \text{(N-Cast)} \qquad\qquad \frac{\mathbf{abs}(v)}{\mathbf{abs}(v : S \Longrightarrow^{p} T)}\ \text{(A-Cast)}$$

Normalization

$$\boxed{v \gg u}$$

$$\lambda(x\colon T).s \gg \lambda(x\colon T).s\ \text{(Norm-Abs)}$$

$$\frac{v \gg u}{v : \diamond_{\#,?,!}(S_1 \to S_2) \Longrightarrow^{p} \diamond_{\#,?,!}(T_1 \to T_2) \gg u : \#(S_1 \to S_2) \Longrightarrow^{p} \#(T_1 \to T_2)}\ \text{(Norm-Cast)}$$

Figure 4.3: Evaluation rules of $\lambda_{\texttt{null}}$, along with auxiliary predicates and the normalization relation

The evaluation rules can be divided into two groups: rules for applications (R-App, R-AppFail, R-AppCast, and R-AppNorm), which might fail, and rules for safe applications (R-SafeAppNull, R-SafeAppNorm), which never fail. The rules are described below:

- R-App is standard beta reduction.
- R-AppFailExt handles the case where we have a function application and the value in the function position is in fact `null`. This last fact is checked via the auxiliary predicate **null**(v). In this case, the entire term (and not just the subterm within the context) evaluates to $\overline{p}$. We complement the blame label because the rule requires that the value in the function position be a cast to an unsafe nullable function type. As previously mentioned, the code *surrounding* the cast $v : S \Longrightarrow^{p} !(T \to U)$ is responsible for the application failing, since it should be "aware" that the underlying value v might be `null`. Notice that there is no corresponding rule where the cast uses a *safe* nullable function type (i.e. $v : S \Longrightarrow^{p} ?(T \to U)$). This is because the type system ensures that such functions can only be applied via safe applications.
- R-AppFailSelf is like R-AppFailExt, but this time the `null` term is cast to a non-nullable function type $\#(T \to U)$. We fail with label p (as opposed to $\overline{p}$), because it is the term *within* the cast that is responsible for the failure. Conversely, the code surrounding cast bears no responsibility, because it was "promised" a non-nullable function.
- R-AppCast handles the case where the value v' in the function position is a cast involving only non-nullable function types; i.e. $v' = v : \#(S_1 \to S_2) \Longrightarrow^{p} \#(T_1 \to T_2)$. In this case, the application $v'\ u$ reduces to

  $$(v\ (u : T_1 \Longrightarrow^{\overline{p}} S_1)) : S_2 \Longrightarrow^{p} T_2$$

  This is the classic behaviour of blame in a function application, and comes from Findler and Felleisen [2002]. The type system guarantees that the argument u is typed as a T_1, but the function v expects it to have type S_1. We then need the cast $u : T_1 \Longrightarrow^{\overline{p}} S_1$ before passing the argument to function. Notice that the blame label has been negated ($\overline{p}$), because it is the *context* (the code calling the function v) that is responsible for passing an argument of the right type. Conversely, when the function v returns, its return value will have type S_2, but the surrounding code is expecting a value of type T_2. We then need to cast the entire application from S_2 to T_2; this time, the blame label is p. As Findler and Felleisen [2002] remark, the handling of the blame label matches the rule for function subtyping present in

other systems, where the argument and return type must be contra and covariant, respectively.

- R-AppNorm handles the case where we have an application $v\ u$, and v is a cast to a nullable function type (either a ? function or a ! function). Additionally, we know that $\mathbf{abs}(v)$ holds. In this case, what we would want to do is "translate" the nullable function type into a *non-nullable* function type. This is fine because $\mathbf{abs}(v)$ implies that the underlying function is non-null. The *normalization* relation $v \gg v'$ (also shown in Figure 4.3) achieves this translation of casts.

  Example 4.3.2. *Let* $t = \lambda(x\colon \mathtt{Null}).x$. *Suppose we are evaluating the application*

  $$(t : \#(\mathtt{Null} \to \mathtt{Null}) \Longrightarrow^{p} ?(\mathtt{Null} \to \mathtt{Null}))\ \mathtt{null}$$

  We proceed by first noticing that

  $$\mathbf{abs}(t : \#(\mathtt{Null} \to \mathtt{Null}) \Longrightarrow^{p} ?(\mathtt{Null} \to \mathtt{Null}))$$

  Then we normalize the value in the function position

  $$\frac{\dfrac{}{t \gg t}\ \textsc{Norm-Abs}}{t : \#(\mathtt{Null} \to \mathtt{Null}) \Longrightarrow^{p} ?(\mathtt{Null} \to \mathtt{Null}) \gg t : \#(\mathtt{Null} \to \mathtt{Null}) \Longrightarrow^{p} \#(\mathtt{Null} \to \mathtt{Null})}\ \textsc{Norm-Cast}$$

  Now we can use R-AppNorm to turn the original application into

  $$(t : \#(\mathtt{Null} \to \mathtt{Null}) \Longrightarrow^{p} \#(\mathtt{Null} \to \mathtt{Null}))\ \mathtt{null}$$

  We can then proceed with the evaluation using R-AppCast.

- R-SafeAppNull is simple: if we are evaluating a safe application $\mathrm{app}(v, u, u')$ and the underlying function v is `null`, then the entire term reduces to u' (the "default" value).

- Finally, R-SafeAppNorm handles the remaining case. We have a safe application $\mathrm{app}(v, u, u')$ like before, but this time we know that v is an abstraction (via $\mathbf{abs}(v)$). What we would like to do is to turn the safe application into a regular one:

  $$\mathrm{app}(v, u, u') \longmapsto v\ u$$

However, this can lead to the term getting stuck, if v is a cast to a safe nullable function (a ? function). The problem is that safe nullable functions are not supposed to appear in regular applications.

The solution is to normalize v to v'. Since v' is guaranteed to have a regular function type after normalization, we can take the step

$$\mathrm{app}(v, u, u') \longmapsto v'\ u$$

and then follow up with R-AppCast or R-App.

This concludes the description of $\lambda_{\texttt{null}}$. Let us now prove some theorems about it.

4.3.6 Metatheory of $\lambda_{\texttt{null}}$

In developing the metatheory, I followed the approach taken in Wadler and Findler [2009] closely. All the results in this section have been verified using the Coq proof assistant. The Coq code is available at https://github.com/abeln/null-calculus.

Safety Lemmas

The first step is establishing that evaluation of well-typed $\lambda_{\texttt{null}}$ terms does not get stuck. We do this by proving the classic progress and preservation lemmas due to Wright and Felleisen [1994].

First, we need an auxiliary lemma that says that normalization preserves well-typedness.

Lemma 4.3.2 (Soundness of normalization)**.** *Let* $\Gamma \vdash v : \diamond_{\#,?,!}(S \to T)$ *and let* $v \gg v'$. *Then* $\Gamma \vdash v' : \#(S \to T)$.

Then we can prove preservation.

Lemma 4.3.3 (Preservation)**.** *Let* $\Gamma \vdash t : T$ *and suppose that* $t \longmapsto r$. *Then either*

- $r = \Uparrow p$, *for some blame label* p, *or*
- $r = t'$ *for some term* t', *and* $\Gamma \vdash t' : T$

Notice that, because of unsafe casts like $\texttt{null} : \texttt{Null} \Longrightarrow^{p} !(S \to T)$, taking an evaluation step might lead to an error $\Uparrow p$.

Before showing progress, we need a lemma that says that non-nullable values typed with a function type can be normalized.

Lemma 4.3.4 (Completeness of normalization)**.** *Let* $\Gamma \vdash v : \diamond_{\#,?,!}(S \to T)$ *and suppose that* $\textbf{\textit{abs}}(v)$ *holds. Then there exists a value* v' *such that* $v \gg v'$*.*

This lemma is necessary because if we are ever evaluating a well-typed safe application (e.g. $\text{app}(v, u, u')$) where the function value (v) is known to be non-nullable, then we need to be able to turn the safe application into a regular application ($v\ u$) using R-SafeAppNorm.

We also need a weakening lemma.

Lemma 4.3.5 (Weakening)**.** *Let* $\Gamma \vdash t : T$ *and* $x \notin dom(\Gamma)$*. Then* $\Gamma[x \to U] \vdash t : T$ *for any type* U*.*

We can then show progress.

Lemma 4.3.6 (Progress)**.** *Let* $\vdash t : T$*. Then either*

- t *is a value*
- $t \longmapsto \Uparrow p$*, for some blame label* p
- $t \longmapsto t'$*, for some term* t'

Blame Lemmas

The progress and preservation lemmas do not tell us as much as they usually do, because of the possibility of errors. It would then be nice to rule out errors in some cases. Examining the evaluation rules, we can notice that errors occur due to casts: specifically, because we sometimes cast a `null` value to a function type, which we later try to apply.

R-AppFailExt and R-AppFailSelf show that casts to $!(T \to U)$ can lead to *negative* blame, and casts to $\#(T \to U)$ can lead to *positive* blame. We can then define two relations: *positive subtyping* ($T <:^{+} U$) and *negative subtyping* ($T <:^{-} U$), that identify which casts *cannot* lead to positive and negative blame, respectively. The subtyping rules are shown in Figure 4.4.

$\boxed{S \;<:^{+}\; T}$

$$\text{Null} \;<:^{+}\; \text{Null}\;(\text{PS-NullRefl})$$

$$\text{Null} \;<:^{+}\; \diamond_{?,!}\,(S \to T)\;(\text{PS-Null})$$

$$\frac{S' \;<:^{-}\; S \qquad T \;<:^{+}\; T'}{\#(S \to T) \;<:^{+}\; \diamond_{?,!}\,(S' \to T')}\;(\text{PS-Arrow}\#)$$

$$\frac{S' \;<:^{-}\; S \qquad T \;<:^{+}\; T'}{\diamond_{?,!}(S \to T) \;<:^{+}\; \diamond_{?,!}\,(S' \to T')}\;(\text{PS-ArrowNullable})$$

$\boxed{S \;<:^{-}\; T}$

$$\text{Null} \;<:^{-}\; \text{Null}\;(\text{NS-NullRefl})$$

$$\text{Null} \;<:^{-}\; ?(S \to T)\;(\text{NS-Null})$$

$$\frac{S' \;<:^{+}\; S \qquad T \;<:^{-}\; T'}{\#(S \to T) \;<:^{-}\; ?(S' \to T')}\;(\text{NS-Arrow}\#)$$

$$\frac{S' \;<:^{+}\; S \qquad T \;<:^{-}\; T'}{!(S \to T) \;<:^{-}\; \diamond_{\#,?}\,(S' \to T')}\;(\text{NS-ArrowNullable})$$

Figure 4.4: Positive and negative subtyping

Example 4.3.3. *Since the type system ensures that $?(S \to T)$ functions are only ever applied through safe applications, we would hope that the cast $\texttt{null} : \texttt{Null} \Longrightarrow^{p} ?(S \to T)$ will not fail with* either *blame $\Uparrow p$ or $\Uparrow \overline{p}$. Therefore we have both $\texttt{Null} \;<:^{+}\; ?(S \to T)$ and $\texttt{Null} \;<:^{-}\; ?(S \to T)$.*

Example 4.3.4. *Since a cast $\texttt{null} : \texttt{Null} \Longrightarrow^{p} !(S \to T)$ can fail with blame $\overline{p}$, we have $\texttt{Null} \;<:^{+}\; !(S \to T)$, but not $\texttt{Null} \;<:^{-}\; !(S \to T)$.*

Remark 4.3.1. *Unfortunately, as currently defined, neither positive nor negative subtyping are reflexive. For example, $\#(\texttt{Null} \to \texttt{Null}) \not<:^{+} \#(\texttt{Null} \to \texttt{Null})$. The reason is that in the PS-Arrow# and NS-Arrow# rules we are missing a case that relates e.g. $\#(S \to T)$ with $\#(S' \to T')$. If we re-add the missing case, then define the following term:*

$$t = (\texttt{null} : \texttt{Null} \Longrightarrow^{q} !(\texttt{Null} \to \texttt{Null})) : \; !(\texttt{Null} \to \texttt{Null}) \Longrightarrow^{p} \#(\texttt{Null} \to \texttt{Null})$$

We managed to construct a well-typed term such that $\textbf{\textit{null}}(t)$, yet t has type $\#(\texttt{Null} \to \texttt{Null})$. Then we could cast t again and try to apply it as a function.

$$(t : \#(\texttt{Null} \to \texttt{Null}) \Longrightarrow^{z} \#(\texttt{Null} \to \texttt{Null}))\;\texttt{null}$$

$\boxed{t \textbf{ safe for } p}$

$$x \textbf{ safe for } p \quad \text{(SF-Var)}$$

$$\text{null } \textbf{safe for } p \quad \text{(SF-Null)}$$

$$\frac{s \textbf{ safe for } p}{\lambda(x\colon T).s \textbf{ safe for } p} \text{ (SF-Abs)}$$

$$\frac{s \textbf{ safe for } p \qquad t \textbf{ safe for } p}{s\ t \textbf{ safe for } p} \text{ (SF-App)}$$

$$\frac{s \textbf{ safe for } p \qquad t \textbf{ safe for } p \qquad u \textbf{ safe for } p}{\text{app}(s,t,u) \textbf{ safe for } p} \text{ (SF-SafeApp)}$$

$$\frac{S <:^+ T \qquad s \textbf{ safe for } p}{s : S \Longrightarrow^p T \textbf{ safe for } p} \text{ (SF-CastPos)}$$

$$\frac{S <:^- T \qquad s \textbf{ safe for } p}{s : S \Longrightarrow^{\overline{p}} T \textbf{ safe for } p} \text{ (SF-CastNeg)}$$

$$\frac{\textbf{abs}(s) \qquad s \textbf{ safe for } p \qquad T_1 <:^- S_1 \qquad S_2 <:^+ T_2}{s : \#(S_1 \to S_2) \Longrightarrow^p \#(T_1 \to T_2) \textbf{ safe for } p} \text{ (SF-CastAbsPos)}$$

$$\frac{\textbf{abs}(s) \qquad s \textbf{ safe for } p \qquad T_1 <:^+ S_1 \qquad S_2 <:^- T_2}{s : \#(S_1 \to S_2) \Longrightarrow^{\overline{p}} \#(T_1 \to T_2) \textbf{ safe for } p} \text{ (SF-CastAbsNeg)}$$

$$\frac{s \textbf{ safe for } p \qquad q \neq p \qquad q \neq \overline{p}}{s : S \Longrightarrow^q T \textbf{ safe for } p} \text{ (SF-CastDiff)}$$

Figure 4.5: Safe for relation

The application above fails with blame label z, so we cannot have $\#(\texttt{Null} \to \texttt{Null}) <:^+ \#(\texttt{Null} \to \texttt{Null})$.

The following lemma shows that subtyping refines compatibility.

Lemma 4.3.7. *Let S and T be types. Then*

- $S <:^+ T \implies S \rightsquigarrow T$
- $S <:^- T \implies S \rightsquigarrow T$

The next step is to lift positive and negative subtyping to work on terms. The "safe for" relation, shown in Figure 4.5, accomplishes this. We say that a term t is *safe for* a blame label p, written t **safe for** p, if evaluating t cannot lead to an error with blame p. That is, evaluating t either diverges, results in a value, or results in an error with blame different from p. I formalize this fact as a theorem below.

Most of the rules in the **safe for** relation just involve structural recursion on the subterms of a term. The connection with subtyping appears in SF-CastPos and SF-CastNeg. For example, to conclude that $(s : S \Longrightarrow^p T)$ **safe for** p, we require that s **safe for** p and $S <:^+ T$. The SF-CastAbsPos and SF-CastAbsNeg rules deal with the case where we cast a function known to be non-null. For example, we can conclude $s : \#(S_1 \to S_2) \Longrightarrow^p \#(T_1 \to T_2)$ **safe for** p if we know that $\mathbf{abs}(s)$ (plus a few other conditions); that is, the cast will not fail because the underlying value is non-null. SF-CastAbsPos and SF-CastAbsNeg cannot be handled purely via subtyping because in general a cast $s : S \Longrightarrow^p \#(T_1 \to T_2)$ can fail with positive blame.

Positive and negative subtyping, and **safe for**, are all adapted from Wadler and Findler [2009].

The following lemmas say that **safe for** is preserved by normalization and substitution.

Lemma 4.3.8 (Normalization preserves **safe for**)**.** *Let v be a value such that v **safe for** p and suppose that $v \gg v'$. Then v' **safe for** p.*

Lemma 4.3.9 (Substitution preserves **safe for**)**.** *Let t and t' be terms such that t **safe for** p and t' **safe for** p. Then $[t'/x]t$ **safe for** p.*

We now arrive at the main results in this section, the progress and preservation theorems for safe terms.

Theorem 4.3.10 (Preservation of safe terms)**.** *Let $\Gamma \vdash t : T$ and t **safe for** p. Now suppose that t steps to a term t' (that is, taking an evaluation step from t is possible and does not result in an error). Then t' **safe for** p.*

Proof. By induction on a derivation of $t \longmapsto t'$. □

Theorem 4.3.11 (Progress of safe terms)**.** *Let $\Gamma \vdash t : T$ and t **safe for** p. Suppose that $t \longmapsto \Uparrow p'$. Then $p' \neq p$.*

Proof. By induction on a derivation of $t \longmapsto \Uparrow p$. □

Notice that the progress theorem does not preclude the term from stepping to an error, but it does say that the error will not have blame label p.

Here are a few implications of the theorems above. The following claims have *not* been mechanized in Coq:

- Every application that fails does so with blame. This is a consequence of blame being the only way for a term to fail.
- A term without casts cannot fail. This is because a term can only fail with some blame label p, and a term without casts is necessarily **safe for** p.
- Casts that turn `Null` into a safe nullable function type $?(S \to T)$ cannot fail either. This is because $\texttt{Null} <:^{+} ?(S \to T)$ *and* $\texttt{Null} <:^{-} ?(S \to T)$.
- Casts that turn a "Java" type like $!(!(\texttt{Null} \to \texttt{Null}) \to \texttt{Null})$ into the corresponding "Scala" type $?(?(\texttt{Null} \to \texttt{Null}) \to \texttt{Null})$ via "nullification" can only fail with positive blame, because of negative subtyping.
- Conversely, casts that turn a "Scala" type like $\#(\#(\texttt{Null} \to \texttt{Null}) \to \texttt{Null})$ into the corresponding "Java" type $!(!(\texttt{Null} \to \texttt{Null}) \to \texttt{Null})$ via erasure can only fail with negative blame, because of positive subtyping.

The last two claims form the bases for my model of language interoperability, described in Section 4.5.

4.3.7 Coq Mechanization

As previously mentioned, the results in Section 4.3.6 have been verified using the Coq theorem prover. The code is available at https://github.com/abeln/null-calculus. The two main differences between the presentation of $\lambda_{\texttt{null}}$ in this chapter and in the Coq proofs are:

- The definition of evaluation in the Coq code does not use evaluation contexts, unlike Figure 4.3. Instead, we have explicit rules for propagating errors.
- The definition of terms in the Coq code uses a locally-nameless representation of terms [Charguéraud, 2012].

In the mechanization of the proofs, I used the Ott [Sewell et al., 2010] and LNgen [Aydemir and Weirich, 2010] tools, which automate the generation of some useful auxiliary lemmas from a description of the language grammar. In total, the Coq code has 3627 lines of code, of which 1341 are manually-written proofs, while the rest are either library code or automatically-generated by Ott and LNgen.

4.4 Who is to Blame?

Now that we have seen how $\lambda_{\texttt{null}}$ works, I would like to show how the ideas in $\lambda_{\texttt{null}}$ can be used reason about Scala/Java interoperability. Specifically, we will use blame to assign responsibility in the failure scenarios described in Section 4.1.

Example 4.4.1. *Consider the term*

$$(\texttt{null} : \texttt{String}_{\texttt{Scala}} / \texttt{Null} \Longrightarrow^{\texttt{Scala}} \texttt{String}_{\texttt{Java}}).\mathit{length}()$$

This is precisely the kind of term that would be generated by the call `scalaGetLength(null)` *in the first incorrect hypothesis from Section 4.1. A null value is typed as a nullable union in Scala, and then passed as a regular Java string to Java code. The call to* `length` *fails because the underlying receiver is* `null`*. The label attached to the cast is* `Scala` *because the term that is being cast (the* `null` *literal) is Scala-generated.*

The cast should fail with negative *blame; i.e.* $\overline{\texttt{Scala}}$*. This is because it* is *valid for Java strings to be* `null`*, so it is the responsibility of the context (the surrounding Java code) to do a* `null` *check before calling the* `length` *method. If we define the blame label* $\texttt{Java} \equiv \overline{\texttt{Scala}}$*, then Java is to blame in this case.*

Example 4.4.2. *Recall the second failure scenario from Section 4.1. There, the Scala-defined* `len` *method is accessed from Java code. Erasure changes* `len`*'s type from* $\texttt{String}_{\texttt{Scala}} \to \texttt{Int}$ *to* $\texttt{String}_{\texttt{Java}} \to \texttt{Int}$*. If* `twiceLen` *is called with a* `null` *argument (possible because* `twiceLen` *is Java-defined), then the following (conceptual) cast happens:*

$$(\texttt{null} : \texttt{String}_{\texttt{Java}} \Longrightarrow^{\texttt{Java}} \texttt{String}_{\texttt{Scala}}).\mathit{length}()$$

The cast's label is `Java` *because the term being cast is Java-generated. This cast should fail with* positive *blame. This is because* `null` *is* not *an element of the* $\texttt{String}_{\texttt{Scala}}$ *type. Once again, Java is to blame for the failure.*

Remark 4.4.1. *In both of the examples above, the casts are conceptual, in that there is no explicit "cast with labels" instruction in the Java Virtual Machine. However, the null pointer exceptions produced by trying to select a field or method on a* `null` *receiver are very much real. We can think of null pointer exceptions as resulting from casts (derefs) without blame labels; adding the latter allows us to assign responsibility for the exception.*

In summary, by carefully reifying type casts that take place while Java and Scala interoperate (Figure 4.6), and tagging the casts with blame labels, we are able to assign responsibility for null-related failures.

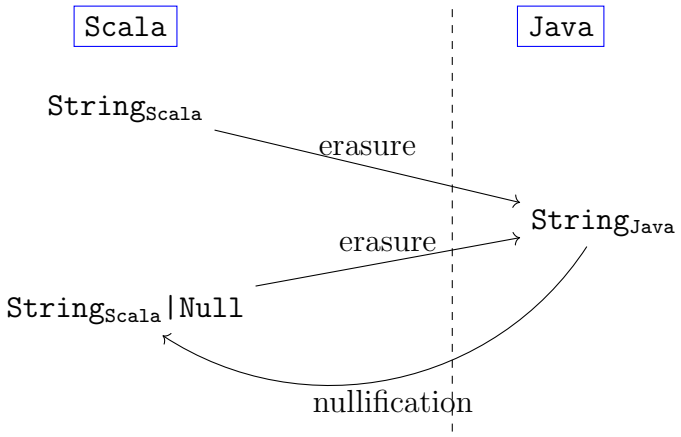


Figure 4.6: Type casts between Scala and Java

4.5 A Calculus for Null Interoperability

The $\lambda_{\texttt{null}}$ calculus is very flexible in that it allows us to freely mix in implicitly nullable terms with explicitly nullable terms. On the other hand, it is perhaps *too flexible*. In the real world, when a language where `null` is explicit interoperates with a language where `null` is implicit, the separation between terms from both languages is very clear (it is usually enforced at a file or module boundary). For example, in the Java and Scala case, the Scala typechecker will *only* allow *explicit* nulls, while the Java typechecker *only* allows *implicit* nulls. To more faithfully model this kind of language interoperability, this section introduces a slight modification of $\lambda_{\texttt{null}}$ that called $\lambda^s_{\texttt{null}}$ ("stratified lambda null").

4.5.1 Terms and Types of $\lambda^s_{\texttt{null}}$

The terms and types of $\lambda^s_{\texttt{null}}$ are shown in Figure 4.7. The main difference with respect to $\lambda_{\texttt{null}}$ is that terms and types are stratified into the world of explicit nulls (subscript $_{\texttt{e}}$) and the world of implicit nulls (subscript $_{\texttt{i}}$). Notice that the grammar for types in the "explicit" sublanguage only allows for non-nullable functions ($\#(S \to T)$) and *safe* nullable functions ($?(S \to T)$). Similarly, the implicit sublanguage only has unsafe nullable functions ($!(S \to T)$). The only new terms are "imports", which in the explicit sublanguage have syntax

$$\text{import}_{\texttt{e}}\ x : T_e = (t_i : T_i)\ \text{in}\ t_e$$

$t ::=$	**Terms**
t_e	terms with *explicit* `nulls`
t_i	terms with *implicit* `nulls`

$f_e, s_e, t_e ::=$	**Explicit terms**	$f_i, s_i, t_i ::=$	**Implicit terms**
x	variable	x	variable
$\mathtt{null}$	null literal	$\mathtt{null}$	null literal
$\lambda(x\colon T_e).s_e$	abstraction	$\lambda(x\colon T_i).(s_i : S_i)$	abstraction
$s_e\ t_e$	application	$s_i\ t_i$	application
$\mathrm{app}(f_e, s_e, t_e)$	safe application	$\mathrm{app}(f_i, s_i, t_i)$	safe application
$s_e : S_e \Longrightarrow^p T_e$	cast	$s_i : S_i \Longrightarrow^p T_i$	cast
$\mathrm{import_e}\ x : T_e = (t_i : T_i)\ \mathrm{in}\ t_e$	import	$\mathrm{import_i}\ x : T_i = (t_e : T_e)\ \mathrm{in}\ t_i$	import

$S_e, T_e ::=$	**Explicit types**	$S_i, T_i ::=$	**Implicit types**
$\mathtt{Null}$	null	$\mathtt{Null}$	null
$\#(S_e \to T_e)$	presumed non-nullable function	$!(S_i \to T_i)$	unsafe nullable function
$?(S_e \to T_e)$	safe nullable function		

Figure 4.7: Terms and types of $\lambda^s_{\mathtt{null}}$

Informally, an import term is similar to a let-binding: it binds x as having type T_e in the body t_e. *However*, the term that x is bound *to*, t_i, comes from the *implicit* sublanguage: it is a t_i and not a t_e. Furthermore, t_i is expected to have type T_i. Dually, the implicit sublanguage has an import term that binds x to an element of t_e, as opposed to a t_i:

$$\mathrm{import_i}\ x : T_i = (t_e : T_e)\ \mathrm{in}\ t_i$$

Imports allow us to link the world of explicit nulls with the world of implicit nulls, in much the same way as Scala's "import" statements allow us to use Java libraries from Scala code (similarly, Java's import statements allow us to use Scala libraries from Java code).

4.5.2 Typing $\lambda^s_{\mathtt{null}}$

The typing rules for $\lambda^s_{\mathtt{null}}$ are shown in Figure 4.8. These rules are almost verbatim copies of the typing rules for $\lambda_{\mathtt{null}}$ (and the compatibility relation is reused from Figure 4.2). The two new rules handle imports:

- TE-Import handles the case where an implicitly nullable term is used from the world of explicit nulls. To type $\text{import}_\text{e}\ x : T_e = (t_i : T_i)\ \text{in}\ t_e$, we first type t_e in the context $\Gamma, x :: T_e$, obtaining a type S_e. This will be the type of the entire term. The interesting twist comes next: the term t_i is typed with the $\vdash_i$ relation in an *empty* context, so that $\varnothing \vdash_\text{i} t_i : T_i$. Finally, we need to somehow check that the type T_i determined by the $\vdash_i$ relation and the type T_e expected by the $\vdash_e$ relation are "in agreement". This is done by the *nullification* relation, whose judgment is written $T_i \hookrightarrow_\text{N} T_e$, and is shown in Figure 4.9.

- TI-Import handles the opposite case, where a term from the world of explicit nulls is used in an implicitly nullable term. Here we use the "dual" of nullification: the *erasure* relation, written $T_e \hookrightarrow_\text{E} T_i$. Erasure is also shown in Figure 4.9.

Remark 4.5.1. *In designing TE-Import and TI-import, we have to decide under* which context *we will type the "embedded" term that comes from the foreign sublanguage. For simplicity, I have chosen to do the typechecking under the* empty *context. This prevents* $\lambda^s_{\texttt{null}}$ *from modelling circular dependencies between terms of different languages, but otherwise seems not unduly restrictive.*

Nullification and erasure, shown in Figure 4.9, are binary relations on types. They are inspired by how Java and Scala interoperate; specifically, the types of Java terms are "nullified" before being used by Scala code, and the types of Scala terms are "erased" before being used by Java code. Of course, the real-world nullification and erasure are more complicated than the simple relations presented here, but I believe the formalization in this section does capture the essence of how these relations affect nullability of types; namely, nullification conservatively assumes that every component of a Java type is nullable, while erasure eliminates the distinction between nullable and non-nullable types in the $\vdash_\text{e}$ type system.

4.5.3 Desugaring $\lambda^s_{\texttt{null}}$ to $\lambda_{\texttt{null}}$

The last step is to give meaning to $\lambda^s_{\texttt{null}}$ terms. We could repeat the approach followed for $\lambda_{\texttt{null}}$ using operational semantics, but instead we will do something different. We will *desugar* $\lambda^s_{\texttt{null}}$ terms and types to $\lambda_{\texttt{null}}$ terms and types, respectively. This is useful, because in Section 4.3.6 we proved many results about $\lambda_{\texttt{null}}$ terms, and we would like to re-use these results to reason about $\lambda^s_{\texttt{null}}$ as well.

$\boxed{\Gamma \vdash_{\mathrm{e}} t_e : T_e}$ $\qquad$ $\boxed{\Gamma \vdash_{\mathrm{i}} t_i : T_i}$

$$\frac{\Gamma(x) = T_e}{\Gamma \vdash_{\mathrm{e}} x : T_e}\ (\text{TE-Var}) \qquad \frac{\Gamma(x) = T_i}{\Gamma \vdash_{\mathrm{i}} x : T_i}\ (\text{TI-Var})$$

$$\Gamma \vdash_{\mathrm{e}} \texttt{null} : \texttt{Null}\ (\text{TE-Null}) \qquad \Gamma \vdash \texttt{null} : \texttt{Null}\ (\text{TI-Null})$$

$$\frac{\Gamma, x : S_e \vdash_{\mathrm{e}} s_e : T_e}{\Gamma \vdash_{\mathrm{e}} \lambda(x\colon S_e).s_e : \#(S_e \to T_e)}\ (\text{TE-Abs}) \qquad \frac{\Gamma, x : S_i \vdash_{\mathrm{e}} s_i : T_i}{\Gamma \vdash_{\mathrm{e}} \lambda(x\colon S_i).(s_i : T_i) :!(S_i \to T_i)}\ (\text{TI-Abs})$$

$$\frac{\Gamma \vdash_{\mathrm{e}} s_e : \#(S_e \to T_e) \qquad \Gamma \vdash_{\mathrm{e}} t_e : S_e}{\Gamma \vdash_{\mathrm{e}} s_e\ t_e : T_e}\ (\text{TE-App}) \qquad \frac{\Gamma \vdash_{\mathrm{e}} s_i :!(S_i \to T_i) \qquad \Gamma \vdash_{\mathrm{e}} t_i : S_i}{\Gamma \vdash_{\mathrm{e}} s_i\ t_i : T_i}\ (\text{TI-App})$$

$$\frac{\Gamma \vdash_{\mathrm{e}} f_e :?(S_e \to T_e) \qquad \Gamma \vdash_{\mathrm{e}} s_e : S \qquad \Gamma \vdash_{\mathrm{e}} t_e : T_e}{\Gamma \vdash_{\mathrm{e}} \mathrm{app}(f_e, s_e, t_e) : T_e}\ (\text{TE-SafeApp})$$

$$\frac{\Gamma \vdash_{\mathrm{i}} f_i :!(S_i \to T_i) \qquad \Gamma \vdash_{\mathrm{i}} s_i : S_i \qquad \Gamma \vdash_{\mathrm{i}} t_i : T_i}{\Gamma \vdash_{\mathrm{i}} \mathrm{app}(f_i, s_i, t_i) : T_i}\ (\text{TI-SafeApp})$$

$$\frac{\Gamma \vdash_{\mathrm{e}} s_e : S_e \qquad S_e \rightsquigarrow T_e}{\Gamma \vdash_{\mathrm{e}} (s_e : S_e \Longrightarrow^{p} T_e) : T_e}\ (\text{TE-Cast}) \qquad \frac{\Gamma \vdash_{\mathrm{i}} s : S_i \qquad S_i \rightsquigarrow T_i}{\Gamma \vdash_{\mathrm{i}} (s_i : S_i \Longrightarrow^{p} T_i) : T_i}\ (\text{TI-Cast})$$

$$\frac{\Gamma, x :: T_e \vdash_{\mathrm{e}} t_e : S_e \qquad \varnothing \vdash_{\mathrm{i}} t_i : T_i \qquad T_i \hookrightarrow_{\mathrm{N}} T_e}{\Gamma \vdash_{\mathrm{e}} \mathrm{import}_{\mathrm{e}}\ x : T_e = (t_i : T_i)\ \mathrm{in}\ t_e : S_e}\ (\text{TE-Import})$$

$$\frac{\Gamma, x :: T_i \vdash_{\mathrm{i}} t_i : S_i \qquad \varnothing \vdash_{\mathrm{e}} t_e : T_e \qquad T_e \hookrightarrow_{\mathrm{E}} T_i}{\Gamma \vdash_{\mathrm{i}} \mathrm{import}_{\mathrm{i}}\ x : T_i = (t_e : T_e)\ \mathrm{in}\ t_i : S_i}\ (\text{TI-Import})$$

Figure 4.8: Typing rules of $\lambda^{s}_{\texttt{null}}$

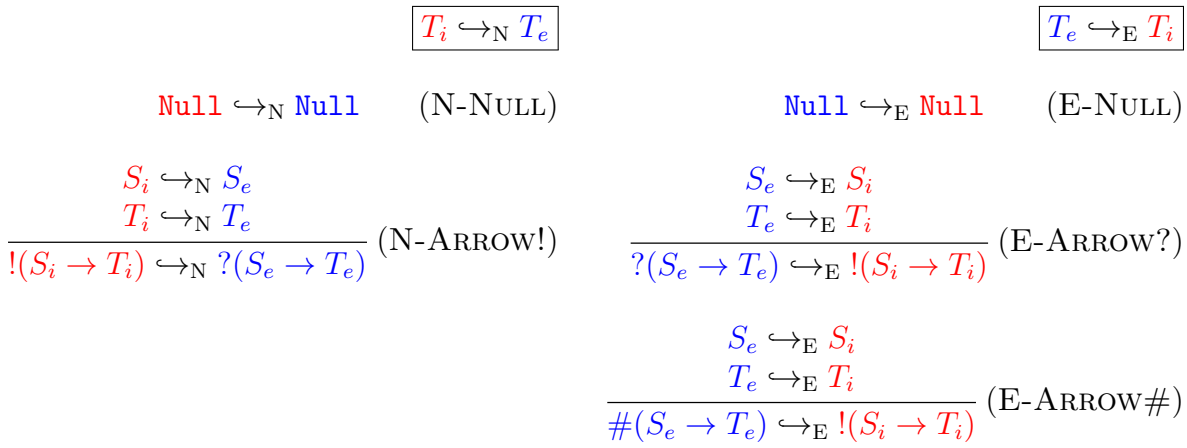


Figure 4.9: Nullification and erasure relations

We will do the desugaring using a pair of functions $(\mathcal{D}_{\mathrm{e}}, \mathcal{D}_{\mathrm{i}})$. $\mathcal{D}_{\mathrm{e}}$ is a function that sends $\lambda^s_{\texttt{null}}$ terms from the explicit sublanguage to $\lambda_{\texttt{null}}$ terms. Similarly, $\mathcal{D}_{\mathrm{i}}$ is a function that maps $\lambda^s_{\texttt{null}}$ terms from the implicit sublanguage to $\lambda_{\texttt{null}}$ terms. Both functions are shown in Figure 4.10.

The first thing to notice is that we do not actually need to desugar *types*. This is because $\lambda^s_{\texttt{null}}$ types (from both sublanguages) are *also* $\lambda_{\texttt{null}}$ types.

When it comes to terms, most cases in Figure 4.10 are handled by straightforward structural recursion on the term. There are only three interesting cases:

- (DI-Abs) An abstraction $\lambda(x\colon S_i).(s_i : T_i)$ from the implicit sublanguage is typed as $!(S_i \to T_i)$ (Figure 4.8). However, the corresponding lambda in $\lambda_{\texttt{null}}$, $\lambda(x\colon S_i).\mathcal{D}_{\mathrm{i}}(s_i)$, will have type $\#(S_i \to T_i)$. So that the metatheory in Section 4.5.4 works out, we need the types to match; hence the cast. This kind of "automatically inserted" cast will have blame label $\mathcal{I}_{\mathrm{int}}$: the $\mathcal{I}$ stands for "implicit", indicating that the term being cast is from the implicit sublanguage. The $_{int}$ subscript indicates that it is an "internal" cast; that is, it does not occur at the boundary between the implicit and explicit sublanguages. To do the cast, we need the return type T_i of the function: this is why abstractions in the implicit sublanguage contain type annotations for the return type.

- (DE-Import) This handles the case where we import a term from the implicit world into the explicit world. There are two desugarings that happen in this rule. The

$\boxed{\mathcal{D}_{\mathrm{e}} : s_e \longrightarrow s}$

$$\mathcal{D}_{\mathrm{e}}(x) = x \qquad \text{(DE-Var)}$$
$$\mathcal{D}_{\mathrm{e}}(\texttt{null}) = \texttt{null} \qquad \text{(DE-Null)}$$
$$\mathcal{D}_{\mathrm{e}}(\lambda(x\colon T_e).s_e) = \lambda(x\colon T_e).\mathcal{D}_{\mathrm{e}}(s_e) \qquad \text{(DE-Abs)}$$
$$\mathcal{D}_{\mathrm{e}}(s_e\ t_e) = \mathcal{D}_{\mathrm{e}}(s_e)\ \mathcal{D}_{\mathrm{e}}(t_e) \qquad \text{(DE-App)}$$
$$\mathcal{D}_{\mathrm{e}}(\mathrm{app}(f_e, s_e, t_e)) = \mathrm{app}(\mathcal{D}_{\mathrm{e}}(f_e), \mathcal{D}_{\mathrm{e}}(s_e), \mathcal{D}_{\mathrm{e}}(t_e)) \qquad \text{(DE-SafeApp)}$$
$$\mathcal{D}_{\mathrm{e}}(s_e : S_e \Longrightarrow^{p} T_e) = \mathcal{D}_{\mathrm{e}}(s_e) : S_e \Longrightarrow^{p} T_e \qquad \text{(DE-Cast)}$$
$$\mathcal{D}_{\mathrm{e}}(\mathrm{import}_{\mathrm{e}}\ x_e : T_e = (t_i : T_i)\ \mathrm{in}\ t_e) = (\lambda(x\colon T_e).\mathcal{D}_{\mathrm{e}}(t_e))\ (\mathcal{D}_{\mathrm{i}}(t_i) : T_i \Longrightarrow^{\mathcal{I}} T_e) \qquad \text{(DE-Import)}$$

$\boxed{\mathcal{D}_{\mathrm{i}} : s_i \longrightarrow s}$

$$\mathcal{D}_{\mathrm{i}}(x) = x \qquad \text{(DI-Var)}$$
$$\mathcal{D}_{\mathrm{i}}(\texttt{null}) = \texttt{null} \qquad \text{(DI-Null)}$$
$$\mathcal{D}_{\mathrm{i}}(\lambda(x\colon S_i).(s_i : T_i)) = (\lambda(x\colon S_i).\mathcal{D}_{\mathrm{i}}(s_i)) : \#(S_i \to T_i) \Longrightarrow^{\mathcal{I}_{\mathrm{int}}} !(S_i \to T_i) \qquad \text{(DI-Abs)}$$
$$\mathcal{D}_{\mathrm{i}}(s_i\ t_i) = \mathcal{D}_{\mathrm{i}}(s_i)\ \mathcal{D}_{\mathrm{i}}(t_i) \qquad \text{(DI-App)}$$
$$\mathcal{D}_{\mathrm{i}}(\mathrm{app}(f_i, s_i, t_i)) = \mathrm{app}(\mathcal{D}_{\mathrm{i}}(f_i), \mathcal{D}_{\mathrm{i}}(s_i), \mathcal{D}_{\mathrm{i}}(t_i)) \qquad \text{(DI-SafeApp)}$$
$$\mathcal{D}_{\mathrm{i}}(s_i : S_i \Longrightarrow^{p} T_i) = \mathcal{D}_{\mathrm{i}}(s_i) : S_i \Longrightarrow^{p} T_i \qquad \text{(DI-Cast)}$$
$$\mathcal{D}_{\mathrm{i}}(\mathrm{import}_{\mathrm{i}}\ x_i : T_i = (t_e : T_e)\ \mathrm{in}\ t_i) = (\lambda(x\colon T_i).\mathcal{D}_{\mathrm{i}}(t_i))\ (\mathcal{D}_{\mathrm{e}}(t_e) : T_e \Longrightarrow^{\mathcal{E}} T_i) \qquad \text{(DI-Import)}$$

Figure 4.10: Desugaring $\lambda^{s}_{\texttt{null}}$ terms to $\lambda_{\texttt{null}}$ terms

first is a standard desugaring that turns the import (effectively, a let binding) into a lambda abstraction that is immediately applied. In this way, we do not need to add let bindings to $\lambda_{\texttt{null}}$. The second desugaring is the insertion of a cast that "guards" the transformation of the original implicit type T_i into the explicit type T_e. The cast has blame label $\mathcal{I}$ to indicate that the term being cast is from the implicit world (conversely, we could say that the *context* using the term is from the explicit world).

- (DI-Import) We also need a dual rule for importing a term from the *explicit* world into the implicit world. This rule does the same as (DE-Import), except that the cast now goes in the opposite direction: from T_e to T_i. The cast is labelled with blame $\mathcal{E}$, indicating that the term being cast comes from the explicit sublanguage.

4.5.4 Metatheory of $\lambda^{s}_{\texttt{null}}$

The following two lemmas are key. They show that nullification implies negative subtyping, and erasure implies positive subtyping.

Lemma 4.5.1. *Let S and T be two types, such that $S \hookrightarrow_N T$. Then $S <:^{-} T$ and $T <:^{+} S$.*

Lemma 4.5.2. *Let S and T be two types, such that $S \hookrightarrow_E T$. Then $S <:^{+} T$ and $T <:^{-} S$.*

This is important because nullification is used to import implicit terms into the explicit world. The lemma shows that nullification implies negative subtyping, and casts where the arguments are negative subtypes never fail with *negative* blame. This means that if nullification-related casts fail, they do so by blaming the *term* being cast (which belongs to the implicit world), and never the context (which belongs to the explicit world). That is, the code with implicit nulls is at fault!

Dually, erasure is used to import explicit terms into the implicit world. Since erasure implies positive subtyping, then erasure-related casts can only fail with negative blame. That is, the *context* (which belongs to the implicit world) is at fault for erasure-related failures. Again, implicit nulls are to blame!

The remaining results in this section have not been mechanized in Coq.

Theorem 4.5.3 (Desugaring preserves typing)**.** *Let t_e and t_i be explicit and implicit terms from λ^{s}_{null}, respectively. Then*

- $\Gamma \vdash_e t_e : T_e \implies \Gamma \vdash \mathcal{D}_e(t_e) : T_e$, *and*
- $\Gamma \vdash_i t_i : T_i \implies \Gamma \vdash \mathcal{D}_i(t_i) : T_i$

Proof. By induction on a derivation of $\Gamma \vdash_\mathrm{e} t_e : T_e$ or $\Gamma \vdash_\mathrm{i} t_i : T_i$. In the proof below, "IH" stands for "induction hypothesis". The interesting cases are TE-Import, TI-Import, and TI-Abs.

$\boxed{\Gamma \vdash_\mathrm{e} t_e : T_e}$

Case (TE-Var) $t_e = x$ and $\mathcal{D}_\mathrm{e}(t_e) = x$. We know that $\Gamma(x) = T_e$, so $\Gamma \vdash x : T_e$ by T-Var.

Case (TE-Null) $t_e = \texttt{null}$ and $\mathcal{D}_\mathrm{e}(t_e) = \texttt{null}$. Then $\Gamma \vdash \texttt{null} : \texttt{Null}$ by T-Null.

Case (TE-Abs) $t_e = \lambda(x\colon S_e).s_e$, $T_e = \#(S_e \to U_e)$, and $\mathcal{D}_\mathrm{e}(t_e) = \lambda(x\colon S_e).\mathcal{D}_\mathrm{e}(s_e)$. From the premises, we know that $\Gamma, x :: S_e \vdash_\mathrm{e} s_e : U_e$. By the IH we then get $\Gamma, x :: S_e \vdash \mathcal{D}_\mathrm{e}(s_e) : T_e$. We can then use T-Abs to get $\Gamma \vdash \lambda(x\colon S_e).\mathcal{D}_\mathrm{e}(s_e) : \#(S_e \to U_e)$ as needed.

Case (TE-App) $t_e = s_e\ s'_e$, $\Gamma \vdash_\mathrm{e} s_e : \#(S_e \to T_e)$, and $\Gamma \vdash_\mathrm{e} s'_e : S_e$. Additionally, $\mathcal{D}_\mathrm{e}(t_e) = \mathcal{D}_\mathrm{e}(s_e)\ \mathcal{D}_\mathrm{e}(s'_e)$. By the IH we get $\Gamma \vdash s_e : \#(S_e \to T_e)$ and $\Gamma \vdash s'_e : S_e$. We can then use T-App to get what we need.

Case (TE-SafeApp) $t_e = \mathrm{app}(f_e, s_e, s'_e)$, $\Gamma \vdash_\mathrm{e} f_e :?(S_e \to T_e)$, $\Gamma \vdash_\mathrm{e} s_e : S_e$, and $\Gamma \vdash_\mathrm{e} s'_e : T_e$. Also, $\mathcal{D}_\mathrm{e}(t_e) = \mathrm{app}(\mathcal{D}_\mathrm{e}(f_e), \mathcal{D}_\mathrm{e}(s_e), \mathcal{D}_\mathrm{e}(s'_e))$. We can then use the IH thrice and T-SafeApp to get what we need.

Case (TE-Cast) $t_e = s_e : S_e \Longrightarrow^p T_e$, $\Gamma \vdash_\mathrm{e} s_e : S_e$, and $S_e \rightsquigarrow T_e$. Also, $\mathcal{D}_\mathrm{e}(t_e) = \mathcal{D}_\mathrm{e}(s_e) : S_e \Longrightarrow^p T_e$. By the IH we get $\Gamma \vdash \mathcal{D}_\mathrm{e}(s_e) : S_e$. Notice that the compatibility relation is shared by $\lambda_{\texttt{null}}$ and $\lambda^s_{\texttt{null}}$, and the types do not are not changed by desugaring. Therefore, we can use T-Cast to get $\Gamma \vdash \mathcal{D}_\mathrm{e}(s_e) : S_e \Longrightarrow^p T_e : T_e$ as needed.

Case (TE-Import) $t_e = \mathrm{import}_\mathrm{e}\ x : S_e = (t_i : S_i)\ \mathrm{in}\ t'_e$, $\Gamma, x :: S_e \vdash_\mathrm{e} t_e : T_e$, $\varnothing \vdash_\mathrm{i} t_i : S_i$, and $S_i \hookrightarrow_\mathrm{N} S_e$. Also, $\mathcal{D}_\mathrm{e}(t_e) = (\lambda(x\colon S_e).\mathcal{D}_\mathrm{e}(t'_e))\ (\mathcal{D}_\mathrm{i}(t_i) : S_i \Longrightarrow^{\mathcal{I}} S_e)$.

What do we need for $\mathcal{D}_\mathrm{e}(t'_e)$ to be type-correct?

- $\mathcal{D}_\mathrm{e}(t'_e)$ is an application: the lhs needs to have type $\#(S_e \to T_e)$, and the rhs needs to have type S_e.
- Let us look at the rhs first. By the IH, $\varnothing \vdash \mathcal{D}_\mathrm{i}(t_i) : S_i$. Weakening (Lemma 4.3.5) gives us $\Gamma \vdash \mathcal{D}_\mathrm{i}(t_i) : S_i$. In order to use T-Cast to type the cast, we only additionally need that $S_i \rightsquigarrow S_e$. Lemma 4.5.1 tells us that $S_i \hookrightarrow_\mathrm{N} S_e$ implies $S_i \rightsquigarrow S_e$. This gives us $\Gamma \vdash (\mathcal{D}_\mathrm{i}(t_i) : S_i \Longrightarrow^{\mathcal{I}} S_e) : S_e$ as needed.

- To type the LHS, notice that the IH tells us that $\Gamma, x :: S_e \vdash \mathcal{D}_{\mathrm{e}}(t'_e) : T_e$. Applying T-Abs we get $\Gamma \vdash \lambda(x\colon S_e).\mathcal{D}_{\mathrm{e}}(t'_e) : \#(S_e \to T_e)$.
- Finally, we use T-App to give $\mathcal{D}_{\mathrm{e}}(t_e)$ type T_e, as needed.

$\boxed{\Gamma \vdash_{\mathrm{i}} t_i : T_i}$

Case (TI-Var) Similar to TE-Var.

Case (TI-Null) Similar to TE-Null.

Case (TI-Abs) $t_i = \lambda(x\colon S_i).(s_i : T'_i)$, $T_i = !(S_i \to T'_i)$, and $\Gamma, x :: S_i \vdash_{\mathrm{i}} s_i : T'_i$. Also, $\mathcal{D}_{\mathrm{i}}(t_i) = (\lambda(x\colon S_i).\mathcal{D}_{\mathrm{i}}(s_i)) : \#(S_i \to T'_i) \Longrightarrow^{\mathcal{I}_{\mathrm{int}}} !(S_i \to T'_i)$. We need two things to hold here:

- By the IH, we get that $\Gamma, x :: S_i \vdash \mathcal{D}_{\mathrm{i}}(s_i) : T'_i$. Applying T-Abs we then get $\Gamma \vdash \lambda(x\colon S_i).\mathcal{D}_{\mathrm{i}}(T'_i) : \#(S_i \to T'_i)$.
- For the cast to be valid, we need $\#(S_i \to T'_i) \rightsquigarrow !(S_i \to T'_i)$. Lemma 4.3.1 says that compatibility is reflexive, so we know that $S_i \rightsquigarrow S_i$ and $T'_i \rightsquigarrow T'_i$. Using C-Arrow we then get $\#(S_i \to T_i) \rightsquigarrow !(S_i \to T'_i)$ as needed.

We can then use T-Cast to type $\mathcal{D}_{\mathrm{i}}(t_i)$ with type $!(S_i \to T'_i)$ as needed.

Case (TI-App) Similar to TE-App.

Case (TI-SafeApp) Similar to TE-SafeApp.

Case (TI-Cast) Similar to TE-Cast.

Case (TI-Import) This is similar to TE-Import, except that instead of Lemma 4.5.1 we use Lemma 4.5.2 to derive compatibility, so that the cast on the argument is well-formed. □

Definition 4.5.1 (Set of blame labels in a term)**.** *We will denote the set of blame labels in a term t of λ^s_{null} by* ***labels***(t)*. We do not give an explicit definition here, but* ***labels***(t) *can be defined inductively on the structure of terms.*

Theorem 4.5.4 (Explicit terms cannot be blamed for interop failures)**.** *Let t be a term of λ^s_{null}. Suppose that $\{\mathcal{I}, \overline{\mathcal{I}}, \mathcal{E}, \overline{\mathcal{E}}\} \cap$* ***labels***$(t) = \emptyset$*, $\mathcal{I} \notin \{\mathcal{E}, \overline{\mathcal{E}}\}$, and $\mathcal{I}_{int} \notin \{\mathcal{I}, \overline{\mathcal{I}}, \mathcal{E}, \overline{\mathcal{E}}\}$. Further, suppose that t is well-typed under $\vdash_e$ or $\vdash_i$ and a context Γ. Then*

- *If $t = t_e$, then $\mathcal{D}_e(t_e)$* ***safe for*** *$\overline{\mathcal{I}}$ and $\mathcal{D}_e(t_e)$* ***safe for*** *$\mathcal{E}$.*

- *If* $t = t_i$, *then* $\mathcal{D}_i(t_i)$ ***safe for*** $\overline{\mathcal{I}}$ *and* $\mathcal{D}_i(t_e)$ ***safe for*** $\mathcal{E}$.

Proof. By induction on a derivation of $\Gamma \vdash_{\mathrm{e}} t_e : T_e$ or $\Gamma \vdash_{\mathrm{i}} t_i : T_i$.

$\boxed{\Gamma \vdash_{\mathrm{e}} t_e : T_e}$

Case (TE-Var) $t_e = x$ and $\mathcal{D}_{\mathrm{e}}(t_e) = x$. Use SF-Var.

Case (TE-Null) $t_e = \texttt{null}$ and $\mathcal{D}_{\mathrm{e}}(t_e) = \texttt{null}$. Use SF-Null.

Case (TE-Abs) $t_e = \lambda(x\colon S_e).t_e$ and $\mathcal{D}_{\mathrm{e}}(t_e) = \lambda(x\colon S_e).\mathcal{D}_{\mathrm{e}}(t_e)$. Use the IH and SF-Abs.

Case (TE-App) Use the IH twice and SF-App.

Case (TE-SafeApp) Use the IH thrice and SF-SafeApp.

Case (TE-Cast) $t_e = s_e : S_e \Longrightarrow^{p} T_e$. Use the IH and the fact that $\{\mathcal{I}, \overline{\mathcal{I}}, \mathcal{E}, \overline{\mathcal{E}}\} \cap \mathbf{labels}(t) = \emptyset$ implies $\overline{\mathcal{I}} \neq p$ and $\overline{\mathcal{I}} \neq \overline{p}$. Similarly, we get $\mathcal{E} \neq p$ and $\mathcal{E} \neq \overline{p}$. Then we can use SF-CastDiff.

Case (TE-Import) $t_e = \mathrm{import}_{\mathrm{e}}\ x : S_e = (t_i : S_i)\ \mathrm{in}\ t'_e$, and

$$\mathcal{D}_{\mathrm{e}}(t_e) = (\lambda(x\colon S_e).\mathcal{D}_{\mathrm{e}}(t'_e))\ (\mathcal{D}_{\mathrm{i}}(t_i) : S_i \Longrightarrow^{\mathcal{I}} S_e)$$

On the LHS, we can use the IH and SF-Abs. On the RHS, we need to show

$$\mathcal{D}_{\mathrm{i}}(t_i) : S_i \Longrightarrow^{\mathcal{I}} S_e \ \textbf{safe for}\ \overline{\mathcal{I}}$$

By the IH, we get that t_i **safe for** $\overline{I}$, so it only remains to check that $S_i <:^{-} S_e$. Since t_e is well-typed, then we know $S_i \hookrightarrow_{\mathrm{N}} S_e$. Lemma 4.5.1 then gives us $S_i <:^{-} S_e$, as needed. We can then apply SF-CastNeg to get that the RHS is **safe for** $\overline{\mathcal{I}}$.

Finally, we apply SF-App.

To prove $\mathcal{D}_{\mathrm{e}}(t_e)$ **safe for** $\mathcal{E}$, use the IH hypothesis, the fact that $\mathcal{E} \notin \{\mathcal{I}, \overline{\mathcal{I}}\}$, and SF-CastDiff.

$\boxed{\Gamma \vdash_{\mathrm{i}} t_i : T_i}$

Case (TI-Var) Similar to TE-Var.

Case (TI-Null) Similar to TE-Null.

Case (TI-Abs) Similar to TE-Abs, using the fact that $\mathcal{I}_{\mathrm{int}} \notin \{\mathcal{I}, \overline{\mathcal{I}}, \mathcal{E}, \overline{\mathcal{E}}\}$.

Case (TI-App) Similar to TE-App.

Case (TI-SafeApp) Similar to TE-SafeApp.

Case (TI-Cast) Similar to TE-Cast.

Case (TI-Import) $t_i = \text{import}_\text{i}\ x : S_i = (t_e : S_e)\ \text{in}\ t_i'$. Similar to TE-Import, except that we use Lemma 4.5.2 to prove

$$\mathcal{D}_\text{e}(t_e) : S_e \Longrightarrow^{\mathcal{E}} S_i \textbf{ safe for } \mathcal{E}$$

by first noticing that $S_e \hookrightarrow_\text{E} S_i$ implies $S_e <:^+ S_i$, and then using SF-CastPos.

□

Discussion

Taken together, theorems 4.5.3 and 4.5.4 imply the following. Start with a well-typed $\lambda^s_{\texttt{null}}$ term, desugar it to $\lambda_{\texttt{null}}$, and then evaluate it. If evaluation results in an error because of a cast that was done "at the boundary", that is, a cast generated when desugaring an `import` term, then the blame is either $\mathcal{I}$ or $\overline{\mathcal{E}}$. If the blame is $\mathcal{I}$, then this means the term being cast, which originated in the implicit sublanguage, is at fault. If the blame is $\overline{\mathcal{E}}$, then the *context* surrounding the term being cast is at fault; in this case, the term being cast comes from the explicit sublanguage, so the context is in the implicit sublanguage. In both cases, the implicit term is at fault!

Just like a central result in gradual typing is that "well-typed programs can't be blamed" Wadler and Findler [2009], we can summarize our main result in this section as "explicit terms can't be blamed for null interop errors".

The following examples illustrate how blame is assigned in $\lambda^s_{\texttt{null}}$. They have been verified in Coq.

Example 4.5.1. *This first example is about importing an explicit term into the implicit world. After desugaring, the program below evaluates to* $\Uparrow \overline{\mathcal{E}}$.

```
import_i f: !(Null -> Null) =
  ((null: Null ==> E_int ?(Null -> Null)) : ?(Null -> Null))
in
  f null
```

The problem here is with the application f `null`. The type of f is a nullable type, so the code using f should have used a safe application. Instead, since a regular application was used and f is indeed `null`, then the blame is assigned to the context using the interop cast; hence the $\Uparrow \overline{\mathcal{E}}$ result.

Example 4.5.2. *This example shows how erasure is unsound, because it "widens" the domain of functions. The explicit world declares a function with type $\#(\#(\texttt{Null} \to \texttt{Null}) \to \texttt{Null})$, which is turned by erasure into $!(!(\texttt{Null} \to \texttt{Null}) \to \texttt{Null})$. In effect, what this says is that the function can accept `null` as an argument, but the function cannot in fact handle this case.*

```
import_i f: !(!(Null -> Null) -> Null) =
  ((lam (f': #(Null -> Null)). f' null): #(#(Null -> Null) -> Null))
in
  f (null: Null ==> I_int !(Null -> Null))
```

After desugaring the program above and performing a few evaluation steps, we end up with an application

$$((\texttt{null} : \texttt{Null} \Longrightarrow^{\mathcal{I}_{int}} !(\texttt{Null} \to \texttt{Null})) :! (\texttt{Null} \to \texttt{Null}) \Longrightarrow^{\overline{\mathcal{E}}} \#(\texttt{Null} \to \texttt{Null}))\ \texttt{null}$$

The blame label $\mathcal{E}$ corresponds to the cast generated for the import. The label appears negated because it is passed as an argument to a function (recall that arguments behave contravariantly in the rule for applying a cast, in Figure 4.3). Finally, because the cast is to a non-nullable function type, the context is not at fault (it is ok to apply a non-nullable function type without checks); instead, the term being cast is blamed. This gives us a result of $\Uparrow \overline{\mathcal{E}}$, as expected.

Example 4.5.3. *This example shows an implicit term being imported into the explicit world. In this case, the term being imported is `null`, but it is masked with a function type $!(\texttt{Null} \to \texttt{Null})$. Nullification turns this type into $?(\texttt{Null} \to \texttt{Null})$, so the type system for the explicit sublanguage ensures that a null check happens before the imported term is used. As a result, no errors occur. The result of evaluation is the `null` sentinel value passed to the safe application.*

```
import_e f: ?(Null -> Null) =
  ((null: Null ==> I_int !(Null -> Null)): !(Null -> Null))
in
  app f null null
```

Example 4.5.4. *The last example illustrates a limitation of our current strategy for blame assignment. Consider the code*

```
import_e f: ?(?(Null -> Null) -> Null) =
  ((lam (f': !(Null -> Null)). f' null)
    : !(!(Null -> Null) -> Null))
in
  app f (null: Null ==> E_int ?(Null -> Null)) null
```

The idea of this code is that the implicit term takes an argument f' of type $!(\mathtt{Null} \to \mathtt{Null})$ and unsafely calls it without a null check. Nullification turns the type of term being imported from $!(!(\mathtt{Null} \to \mathtt{Null}) \to \mathtt{Null})$ into $?(?(\mathtt{Null} \to \mathtt{Null}) \to \mathtt{Null})$. This means that the explicit world will be able to pass $\mathtt{null}$ as an argument to the function f. The problem is that, as we saw, f cannot handle a null argument gracefully, causing an error.

Intuitively, this failure should be blamed on the implicit term, and indeed that happens, but the blame label for the result is $\mathcal{I}_{int}$ and not $\mathcal{I}$. The problem is that the abstraction being imported

$$\lambda(f' : !(\mathtt{Null} \to \mathtt{Null})).f'\ \mathtt{null}$$

is desugared by $\mathcal{D}_i$ into

$$(\lambda(f' : !(\mathtt{Null} \to \mathtt{Null})).f'\ \mathtt{null}) : \#(!(\mathtt{Null} \to \mathtt{Null}) \to \mathtt{Null}) \Longrightarrow^{\mathcal{I}_{int}} !(!(\mathtt{Null} \to \mathtt{Null}) \to \mathtt{Null})$$

Now consider what happens when we apply the function above to the argument

$$a \equiv (\mathtt{null} : \mathtt{Null} \Longrightarrow^{\mathcal{E}_{int}} ?(\mathtt{Null} \to \mathtt{Null})) :?(\mathtt{Null} \to \mathtt{Null}) \Longrightarrow^{\overline{\mathcal{I}}} !(\mathtt{Null} \to \mathtt{Null})$$

We first apply R-AppNorm, then R-AppCast from Figure 4.3, and wrap the cast above in an extra cast

$$a :!(\mathtt{Null} \to \mathtt{Null}) \Longrightarrow^{\overline{\mathcal{I}_{int}}} !(\mathtt{Null} \to \mathtt{Null})$$

Finally, we apply the above to $\mathtt{null}$, and the application fails with blame $\overline{\overline{\mathcal{I}_{int}}} = \mathcal{I}_{int}$. That is, in this case the blame is assigned to the internal cast within the implicit code, and not to the interop cast $\mathcal{I}$ as we would perhaps expect. The reason is that the cast with label $\overline{\mathcal{I}}$ is nested *within the outer cast with label $\overline{\mathcal{I}_{int}}$, and our current definition of the operational semantics always assigns blame based* only *on the outermost case.*

Limitations

There are at least two limitations in our model for language interop as presented in this section:

- We assign blame *solely* based on the outermost cast. This can lead to loss of precision when casts are nested, as shown by Example 4.5.4. For example, suppose we have the term (where the `Null` type is abbreviated as $\mathcal{N}$)

  $$((\texttt{null} : \mathcal{N} \Longrightarrow^{z} ?(\mathcal{N} \to \mathcal{N})) : ?(\mathcal{N} \to \mathcal{N}) \Longrightarrow^{q} \#(\mathcal{N} \to \mathcal{N})) : \#(\mathcal{N} \to \mathcal{N}) \Longrightarrow^{p} \#(\mathcal{N} \to \mathcal{N})$$

  If we try to apply such a cast, we currently fail with blame p. The interpretation of this blame is that there is *something* wrong with the term being cast. Or to put it differently, that the surrounding code is not at fault. While this is true, an arguably more accurate blame assignment would identify the cast with blame q as responsible for the failure, since it is an unsound "downcast" that turns a nullable function type into its non-nullable variant.

  More accurate blame assignment would reflect this intuition better.

- Theorem 4.5.4 constrains how interop casts can fail, but it says nothing about "internal" casts within the implicit or explicit sublanguages. Thinking back to Java and Scala, we would expect that internal Scala casts cannot fail (for example, the implicit casts that originate from subtyping judgments should not fail), but internal Java casts can.

  A better model of language interop would be able to reason about internal casts.

4.6 Related Work

The concept of blame comes from work on higher-order contracts by Findler and Felleisen [2002]. The application of blame to gradual typing was pioneered by Wadler and Findler [2009], which I followed closely when developing the operational semantics and safety proofs for $\lambda_{\texttt{null}}$. My syntax for casts comes from Ahmed et al. [2011]. Wadler [2015] provided additional context on the use of blame for gradual typing. The design of $\lambda^{s}_{\texttt{null}}$ was inspired by our ongoing work in designing a version of Scala where `nulls` are explicit. The interoperability between Scala and Java uses real world versions of the erasure and nullification relations.

4.7 Conclusions

In this chapter, I looked at the problem of characterizing the nullability errors that occur from two interoperating languages: one with explicit `nulls`, the other with implicit `nulls`. I showed how the concept of *blame* from gradual typing can be co-opted to provide such a characterization. Specifically, by making type casts explicit and labelling casts with blame labels, we are able to assign responsibility for runtime failures. To formally study the use of blame for tracking nullability errors, I introduced $\lambda_{\texttt{null}}$, a calculus where terms can be explicitly nullable or implicitly nullable. I showed that, even though evaluation of $\lambda_{\texttt{null}}$ terms can fail, such failures can be constrained if we restrict casts using positive and negative subtyping. Finally, I used $\lambda_{\texttt{null}}$ as the basis for a higher-level calculus, $\lambda^{s}_{\texttt{null}}$, which more closely models language interoperability. The main result of the chapter is a theorem that says that the explicit sublanguage of $\lambda^{s}_{\texttt{null}}$ is never at fault for failures of casts that mediate the interop.

Chapter 5

Conclusions

Recall the thesis this dissertation argues for:

> We can retrofit the Scala language with explicit nulls. The resulting type system can be formally studied and understood.

I have shown that Scala can be retrofitted with explicit nulls by implementing a type system for nullability as a modification of the Dotty (Scala 3) compiler. The new type system makes reference types non-nullable, but nullability can be recovered using union types. My design for explicit nulls tries to balance soundness and usability, with a special focus on making Java interoperability practical. I evaluated the design by migrating 90,000 lines of real-world Scala code. The results of the evaluation yielded valuable insights for future improvements.

To show that a type system with explicit nulls can be well-understood, I presented theoretical foundations for two key features of the explicit nulls design: type nullification and language interoperability. I gave a denotational semantics for nullification, and showed that nullification is an element-preserving transformation. Regarding interoperability, I introduced a core calculus for modelling the interoperability between languages with explicit nulls (like Scala) and languages with implicit nulls (like Java). Finally, I showed that, using the concept of blame from gradual typing, we can characterize and rule out certain kinds of nullability errors. Specifically, one can show that in well-typed programs, implicitly-typed subterms can be blamed for interoperability errors.

Bibliography

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

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for All. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 201–214. ACM, 2011. doi: 10.1145/1926385.1926409. URL https://doi.org/10.1145/1926385.1926409.

Nada Amin and Tiark Rompf. Type Soundness Proofs with Definitional Interpreters. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 666–679. ACM, 2017. URL http://dl.acm.org/citation.cfm?id=3009866.

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

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World*, pages 249–272. Springer, 2016.

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

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

Brian Aydemir and Stephanie Weirich. LNgen: Tool Support for Locally Nameless Representations. 2010.

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

Mike Barnett, K Rustan M Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer, 2004.

Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The Coq proof assistant reference manual: Version 6.1. 1997.

Ana Bove, Peter Dybjer, and Ulf Norell. A Brief Overview of Agda–a Functional Language with Dependent Types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

Edwin Brady. Idris, a General-Purpose Dependently Typed Programming Panguage: Design and Implementation. *Journal of functional programming*, 23(5):552–593, 2013.

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

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

Arthur Charguéraud. The Locally Nameless Representation. *Journal of automated reasoning*, 49(3):363–408, 2012.

Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an Object-Oriented Approach to Non-Uniform Cluster Computing. In Ralph E. Johnson and

Richard P. Gabriel, editors, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 519–538. ACM, 2005. doi: 10.1145/1094811.1094852. URL https://doi.org/10.1145/1094811.1094852.

Bruno C. d. S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. The Implicit Calculus: A New Foundation for Generic Programming. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 35–44. ACM, 2012. doi: 10.1145/2254064.2254070. URL https://doi.org/10.1145/2254064.2254070.

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

Dotty Team. Dotty, a next-generation compiler for Scala. URL dotty.epfl.ch. [Online; accessed 7-November-2019].

EPFL. The Scala Programming Language. URL https://www.scala-lang.org/. [Online; accessed 17-November-2019].

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

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

Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In Mitchell Wand and Simon L. Peyton Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh,*

Pennsylvania, USA, October 4-6, 2002, pages 48–59. ACM, 2002. doi: 10.1145/581478.581484. URL https://doi.org/10.1145/581478.581484.

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

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016.

Jason Hu and Ondřej Lhoták. Undecidability of $D_{-}\{<:\}$ and Its Decidable Fragments. *arXiv preprint arXiv:1908.05294*, 2019.

Ifaz Kabir and Ondřej Lhoták. κDOT: Scaling DOT with Mutation and Constructors. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*, pages 40–50. ACM, 2018.

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

Kotlin Foundation. Null safety, b. URL https://kotlinlang.org/docs/reference/null-safety.html. [Online; accessed 5-November-2019].

K Rustan M Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

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

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

Microsoft. Nullable reference types, a. URL https://docs.microsoft.com/en-us/dotnet/csharp/nullable-references. [Online; accessed 5-November-2019].

Microsoft. Nullable value types, b. URL https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/nullable-types/index. [Online; accessed 5-November-2019].

Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional Programming for the Masses*. " O'Reilly Media, Inc.", 2013.

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

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

Abel Nieto. Towards algorithmic typing for dot. *arXiv preprint arXiv:1708.05437*, 2017.

Martin Odersky and Matthias Zenger. Scalable Component Abstractions. In Ralph E. Johnson and Richard P. Gabriel, editors, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 41–57. ACM, 2005. doi: 10.1145/1094811.1094815. URL https://doi.org/10.1145/1094811.1094815.

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language. Technical report, 2004.

Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. Simplicitly: Foundations and Applications of Implicit Function Types. In *45th ACM SIGPLAN Symposium on Principles of Programming Languages*, number CONF, 2017.

Oracle. Type annotations and pluggable type systems. URL https://docs.oracle.com/javase/tutorial/java/annotations/type_annotations.html. [Online; accessed 16-November-2019].

Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart. *Real World Haskell: Code You Can Believe In*. " O'Reilly Media, Inc.", 2008.

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

Dmitry Petrashko, Ondrej Lhoták, and Martin Odersky. Miniphases: Compilation Using Modular and Efficient Tree Transformations. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 201–216. ACM, 2017. doi: 10.1145/3062341.3062346. URL https://doi.org/10.1145/3062341.3062346.

Benjamin C Pierce. Programming with Intersection Types, Union Types, and Polymorphism. 2002.

Wolfgang Pree and Erich Gamma. *Design Patterns for Object-Oriented Software Development*, volume 183. Addison-wesley Reading, MA, 1995.

PYPL. PYPL PopularitY of Programming Language. URL http://pypl.github.io/PYPL.html. [Online; accessed 17-November-2019].

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

Marianna Rapoport and Ondrej Lhoták. A path to DOT: Formalizing Fully Path-Dependent Types. *PACMPL*, 3(OOPSLA):145:1–145:29, 2019. doi: 10.1145/3360571. URL https://doi.org/10.1145/3360571.

Marianna Rapoport, Ifaz Kabir, Paul He, and Ondřej Lhoták. A Simple Soundness Proof for Dependent Object Types. *Proceedings of the ACM on Programming Languages*, 1 (OOPSLA):46, 2017.

RedMonk. The RedMonk Programming Language Rankings: June 2019. URL https://redmonk.com/sogrady/2019/07/18/language-rankings-6-19/. [Online; accessed 17-November-2019].

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

John C Reynolds. Polymorphism is not set-theoretic. In *International Symposium on Semantics of Data Types*, pages 145–156. Springer, 1984.

ScalaDays. About ScalaDays. URL https://scaladays.org/2019/lausanne. [Online; accessed 17-November-2019].

Dana S Scott and Christopher Strachey. *Toward a Mathematical Semantics for Computer Languages*, volume 1. Oxford University Computing Laboratory, Programming Research Group Oxford, 1971.

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective Tool Support for the Working Semanticist. *J. Funct. Program.*, 20(1):71–122, 2010. doi: 10.1017/S0956796809990293. URL https://doi.org/10.1017/S0956796809990293.

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

Richard Statman. A local translation of untyped λ calculus into simply typed λ calculus. 1991.

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

TIOBE. TIOBE Index for November 2019. URL https://www.tiobe.com/tiobe-index/. [Online; accessed 17-November-2019].

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

Philip Wadler. A Complement to Blame. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

Philip Wadler and Robert Bruce Findler. Well-Typed Programs Can't Be Blamed. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2009. doi: 10.1007/978-3-642-00590-9_1. URL https://doi.org/10.1007/978-3-642-00590-9_1.

Fei Wang and Tiark Rompf. Towards Strong Normalization for Dependent Object Types (DOT). In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

Xi Yang, Stephen M. Blackburn, Daniel Frampton, Jennifer B. Sartor, and Kathryn S. McKinley. Why Nothing Matters: The Impact of Zeroing. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 307–324. ACM, 2011. doi: 10.1145/2048066.2048092. URL https://doi.org/10.1145/2048066.2048092.

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

APPENDICES

Appendix A

Evaluating Explicit Nulls

This appendix describes the procedure I followed for generating the data in Section 2.7.

1. Fetch the community build libraries from

Library	URL	Commit ID
`betterfiles`	https://github.com/dotty-staging/better-files	49b55d6
`scala-pb`	https://github.com/dotty-staging/ScalaPB	329e6d0
`minitest`	https://github.com/dotty-staging/minitest	9e5d9b8
`scalap`	https://github.com/dotty-staging/scala	7ecfce1
`semanticdb`	https://github.com/dotty-staging/dotty-semanticdb	cddb67d
`intent`	https://github.com/dotty-staging/intent	580f6cb
`scopt`	https://github.com/dotty-staging/scopt	9155bdc
`xml-interpolator`	https://github.com/dotty-staging/xml-interpolator	7232e3d
`shapeless`	https://github.com/dotty-staging/shapeless	000131d
`fastparse`	https://github.com/dotty-staging/fastparse	79431b0
`effpi`	https://github.com/dotty-staging/effpi	f6b0b3f
`algebra`	https://github.com/dotty-staging/algebra	5dda5f9
`squants`	https://github.com/dotty-staging/squants	c178ff0
`scala-xml`	https://github.com/dotty-staging/scala-xml	0daee4a
`stdlib123`	https://github.com/dotty-staging/scala	7ecfce1
`scalactic`	https://github.com/dotty-staging/scalatest	0cf1ffa

2. To estimate the size of each library, use the `cloc` utility available at https://github.com/AlDanial/cloc. For most libraries, except for `scalactic` and `stdlib123`, `cloc` can be run from the root directory of the library. For `scalactic`, `cloc` must be run

from `scalatest/scalactic/src/main/`. For `stdlib123`, `cloc` should be run from `scala/src/library/`.

3. Fetch the version of Dotty used in the evaluation, from https://github.com/abeln/dotty/tree/dotty-explicit-nulls-evaluation, with commit id `382da84`.

4. Modify the `build.sbt` file for each library to include the desired flags. See Section 2.7 for a description of the flags used during the evaluation.

5. From the Dotty root directory, run `sbt community-build/test`. `sbt` will then report the error counts for each library.

The second part of the evaluation involved classifying the errors remaining under an `optimistic` run. The patches that migrate the libraries were written by Angela Chang and Justin Pu, and can be found at

Library	**Migration Patch**
`betterfiles`	https://github.com/changangela/better-files/pull/1.diff
`scala-pb`	https://gist.github.com/abeln/02686005f88f8120a0f1a59e241548a6
`minitest`	https://gist.github.com/abeln/8295eeb6d0980c545b9c65c8217fe751
`scalap`	https://gist.github.com/abeln/8b38d6a9b7ec588c743a2008cd492aac
`semanticdb`	https://gist.github.com/abeln/dc90d8a1aeeceb5638adbf5207e2c024
`intent`	https://github.com/changangela/intent/pull/1.diff
`scopt`	https://gist.github.com/abeln/17f622591da2bdb39575a1081e85f798
`xml-interpolator`	https://github.com/changangela/xml-interpolator/pull/1.diff
`fastpartse`	https://gist.github.com/abeln/9a769d9150521e0d0d5c3114915421d7

The remaining libraries were either not migrated, or did not require additional fixes when compiled under `optimistic`. To classify the remaining errors, one can apply the corresponding patch, verify that the library compiles, and then inspect the patch. Each error can then be assigned to a class described in Table 2.4.