Blame for Null

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

Marianna Rapoport
University of Waterloo, Canada
mrapoport@uwaterloo.ca

Gregor Richards
University of Waterloo, Canada
gregor.richards@uwaterloo.ca

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

Abstract

Multiple modern programming languages, including Kotlin, Scala, Swift, and C#, have type systems where nullability is *explicitly* specified in the types. All of the above also need to interoperate with languages where types remain *implicitly nullable*, like Java. This leads to runtime errors that can manifest in subtle ways. In this paper, we show how to reason about the presence and provenance of such nullability errors using the concept of *blame* from gradual typing. Specifically, we 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 of Wadler and Findler, interactions between both kinds of terms are mediated by *casts* with attached *blame labels*, which indicate the origin of errors. On top of $\lambda_{\texttt{null}}$, we then create a second calculus, $\lambda^{s}_{\texttt{null}}$, which closely models the interoperability between languages with implicit nullability and languages with explicit nullability, such as Java and Scala. Our main result is a theorem that states that nullability errors in $\lambda^{s}_{\texttt{null}}$ can always be blamed on terms with less-precise typing; that is, terms typed as implicitly nullable. By analogy, this would mean that `NullPointerExceptions` in combined Java/Scala programs are always the result of unsoundness in the Java type system. We summarize our result with the slogan *explicitly nullable programs can't be blamed.* All our results are formalized in the Coq proof assistant.

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

Keywords and phrases nullability, type systems, blame calculus, gradual typing

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

Acknowledgements We would like to thank the anonymous reviewers for their valuable feedback.

1 Introduction

The problem of null pointers has plagued programming languages since 1965 [28]. In languages with null pointers, references may be to valid values, or may be null, which cannot be dereferenced. Attempting to dereference a null reference typically raises a runtime exception in modern, garbage-collected programming languages. This presents a problem for type soundness and for program maintainability: null is considered a subtype of all reference types, and yet has the interface of none. A number of solutions have been created to address this problem, ranging from type-based solutions [4, 7, 9, 10, 20] to static analyses [24, 30], and from statically sound [10] to heuristic [3].

One type-based solution is to liberate null from its special status as subtype of all reference types. In a language with a null isolated as such, references which are nullable must be explicitly specified as such: the type `T` cannot reference null, but a type such as `T?` ("nullable T", in Kotlin) or `T|Null` ("`T` or `Null`", in Scala) can. These *explicitly nullable* types must be explicitly verified not to be null before being dereferenced. This adds an extra burden on the programmer to perform such checks, but eliminates *all* null dereference errors if used consistently[1].

Unfortunately, modern programming languages with null often inherit it from connected languages, and this inheritance restricts the scope of nullability. Kotlin, C#, and Swift, for example, all have explicitly nullable types, but due to their interactions with Java, other .NET languages, and Objective-C respectively, may still encounter null dereference errors. For instance, Kotlin [15] has explicitly nullable types, but is designed to be fully compatible with Java. But, Java has *implicitly nullable* types—that is, variables and fields of all reference types may refer to null, unsoundly. As a consequence, even if Kotlin's own type system perfectly prevents all null dereferences, its interactions with Java will lead to problems.

Luckily, the interaction between languages with differing levels of type soundness has been studied, in the field of gradual typing [22]. In this paper, we apply the principles of gradual typing—and, in particular, the core result that unsoundness can always be correctly blamed on the unsound language—to the problem of interfacing languages with explicit nullability and languages with implicit nullability. We use the context of Scala, which has implemented explicitly nullable types as an optional feature of its in-development next compiler[2], and Java, which has implicitly nullable reference types.

A sophisticated infrastructure, such as gradual typing's blame, is needed, because there are several ways that nulls can cause problems. Consider the following snippets of Scala and Java code:

```
1  // Scala
2  class ScalaStringOps {
3    def len(s: String): Int = s.length
4  }
5
6  def main() = {
7    val jso = new JavaStringOps()
8    jso.len(null)
9    jso.nlen()
10 }
```

```
1  // Java
2  class JavaStringOps {
3    int len(String s) {
4      return s.length;
5    }
6
7    int nlen() {
8      return new ScalaStringOps().len(null);
9    }
10 }
```

Scala's line 8 calls the `len` method of Java's `JavaStringOps`. When importing Java code into Scala, Scala must choose how to represent Java's implicitly nullable types. Naturally, the Java code might—and in this context, will—fail: Java's line 4 is unsafe. It's reasonable to instead try to guarantee that the execution of Scala code will never dereference null. A natural assumption is that Scala can assure this by importing all reference types as nullable types. For instance, Java's `String` is reinterpreted as `String|Null`. This option could be cumbersome for users, but may prevent Scala from raising null errors, as all values from Java must be checked. For practical reasons, most implementations choose instead to unsoundly

[1] Care must be taken to handle the related problem of *uninitialized* or *partially-initialized* objects, which can lead to subtle nullability errors [24, 30].

[2] `https://dotty.epfl.ch/`

import `String` as `String`, allowing null dereferences in the "safe" language, but as we will see in the next paragraph, plugging this hole is insufficient to solve the soundness problem anyway. A further problem arises because the interaction between these languages is not one-directional.

Consider Java's line 8. In this context, Scala's `ScalaStringOps` is imported into Java, and we have no choice: Its `String` can only reasonably be a `String`, even though Scala `String`s are not nullable, and Java `String`s are. With this forced unsound type conversion, Java is free to call `len` with null, causing *Scala* to raise a null dereference error on line 3. But, while the error was raised in Scala code, the *cause* for the problem is Java: Java put a null where it was not suitable. We aim to prove that even when errors occur in Scala code, it is the Java code's fault.

In gradual typing, "well-typed programs can't be blamed" [27]. In this work, *explicitly nullable programs can't be blamed.*

This paper's contributions are:

- A core calculus, $\lambda_{\texttt{null}}$ ("lambda null"), that formalizes the essence of type systems with implicit and explicit nullability, like those of Kotlin and Scala. $\lambda_{\texttt{null}}$ is based on the blame calculus of Wadler and Findler [27].
- A higher-level calculus, $\lambda^s_{\texttt{null}}$ ("stratified lambda null"), that models the interoperability between languages with implicit nullability and languages with explicit nullability. We can think of $\lambda^s_{\texttt{null}}$ as a stratified version of $\lambda_{\texttt{null}}$, where the implicit and explicit terms are kept separate, but can depend on each other, much like Scala code, which can depend on Java code.
- A metatheory for $\lambda_{\texttt{null}}$, consisting of the standard progress and preservation lemmas (Lemmas 5 and 8), as well as well as blame theorems that characterize how nullability errors can occur in $\lambda_{\texttt{null}}$ (Theorems 15 and 16).
- A metatheory for $\lambda^s_{\texttt{null}}$ with two main components. First, a semantics of $\lambda^s_{\texttt{null}}$ that desugars $\lambda^s_{\texttt{null}}$ terms as $\lambda_{\texttt{null}}$ terms. Second, our main result, Theorem 22, which states that nullability errors can always be blamed on terms with less-precise typing; that is, terms typed as implicitly nullable. By analogy, this would mean that `NullPointerExceptions` in combined Java/Scala programs are always the result of unsoundness in the Java type system, which treats reference types as implicitly nullable. In the style of Wadler and Findler [27], we summarize our result with the slogan *explicitly nullable programs can't be blamed.*
- A Coq mechanization of all our results.

2 Blame Calculus

The blame calculus of Wadler and Findler [27] models the interactions between less-precisely and more-precisely typed code. For example, the less-precisely typed code could come from a dynamically-typed 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*[3]. The notation [4] for casting a term s from a type S to another type T with blame label p is $s : S \Longrightarrow^p T$.

During evaluation, a cast might succeed, fail, or be *be broken up into further casts*. For example, suppose that we cast the value 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: $4 : \texttt{Int} \Longrightarrow^p \texttt{String} \longmapsto \Uparrow p$.

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 [11]. 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.

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 opaque identifiers. We assume the existence of a *complement* function on blame labels, 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:

$$\begin{aligned}
&((\lambda(x\colon \texttt{Int}).x - 2) : \texttt{Int} \to \texttt{Int} \Longrightarrow^p \texttt{Nat} \to \texttt{Nat})\ 1\\
\longmapsto\ &((\lambda(x\colon \texttt{Int}).x - 2)\ (1 : \texttt{Nat} \Longrightarrow^{\overline{p}} \texttt{Int})) : \texttt{Int} \Longrightarrow^p \texttt{Nat}\\
\longmapsto\ &((\lambda(x\colon \texttt{Int}).x - 2)\ 1) : \texttt{Int} \Longrightarrow^p \texttt{Nat}\\
\longmapsto\ &(1 - 2) : \texttt{Int} \Longrightarrow^p \texttt{Nat}\\
\longmapsto\ &-1 : \texttt{Int} \Longrightarrow^p \texttt{Nat}\\
\longmapsto\ &\Uparrow p
\end{aligned}$$

[3] The original presentation in Wadler and Findler [27] also adds *refinement types*, but we will not need them here.
[4] The notation for casts we use comes from Ahmed et al. [1].

The term being cast (the lambda) is responsible 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 an untyped 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.

2.1 Well-typed Programs Can't Be Blamed

The central result in Wadler and Findler [27] is a *blame theorem* that provides two guarantees:

- Casts from less-precise[5] 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. The authors 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", by Milner [18]. In the next section, we will show how we can adapt ideas from the blame calculus to reason about nullability errors.

3 Main Ideas

This section offers a bird's-eye view of the rest of the paper. The main idea is to cast (no pun intended) the null interoperability problem as a gradual typing problem. Then, using casts with blame, we show that the implicit language can always be blamed for interoperability errors. That is, *explicitly nullable programs can't be blamed.*

3.1 $\lambda_{\texttt{null}}$

The first step is to formalize null pointer exceptions. We start with a calculus $\lambda_{\texttt{null}}$ ("lambda null"), based on the blame calculus of Wadler and Findler [27], to which we add a `null` literal with type `Null`. We keep the casts with blame: $s : S \Longrightarrow^{p} T$. Additionally, we distinguish between three kinds of function types:

- $\#(S \to T)$ is a *presumed non-nullable* function, meaning that values of this type are expected to be *non-null*, but *could* be `null` if a downcast was involved (see Section 4). That these functions should be non-null is relevant to how we assign blame.
- $?(S \to T)$ is a *safe nullable* function, meaning that values of this type *can* be `null`, but the type system makes sure that they are safely used.

[5] The formal definition of "less-precise" is given by a *naive subtyping* relation in Wadler and Findler [27].

- $!(S \to T)$ is an *unsafe nullable* function, meaning that values of this type *can* be `null`, but the type system does not protect against unsafe uses of them.

The table below shows the three function types in $\lambda_{\texttt{null}}$ and the kinds of Java and Scala types they model[6]:

$\lambda^{s}_{\texttt{null}}$	Scala	Java
$\#(S \to T)$	$\texttt{String}_{\texttt{Scala}}$	
$?(S \to T)$	`String|Null`	
$!(S \to T)$		$\texttt{String}_{\texttt{Java}}$

Nullability errors happen when we have a function application $u\ v$, but the value u in the function position is in fact `null`. This corresponds closely to what happens in real languages, where null pointer exceptions occur when we select a field or method on a `null` receiver: e.g. we evaluate `s.length()` and `s` is `null`. In fact, u will be "disguised" inside one or more casts, so the type system is fooled into thinking u is a function. For example, taking one step of evaluation on the following term leads to an error $\Uparrow \overline{p}$, where the label in the error comes from the cast: $(\texttt{null} : \texttt{Null} \Longrightarrow^{p} ?(\texttt{Null} \to \texttt{Null}))\ \texttt{null} \longmapsto \Uparrow \overline{p}$.

If one wants to be safe from nullability errors, then instead of a regular application $s\ t$, we can use a *safe application* $\text{app}(s, t, r)$, which conceptually desugars into `if (s != null) then (s t) else r`.

3.2 Blame Assignment

In the example above, $(\texttt{null} : \texttt{Null} \Longrightarrow^{p} ?(\texttt{Null} \to \texttt{Null}))\ \texttt{null} \longmapsto \Uparrow \overline{p}$, how did we decide to blame $\overline{p}$? The basic rules for assigning blame are as follows:

- If the cast that causes the failure casts *to a presumed non-nullable* function, e.g. $v :?(S \to T) \Longrightarrow^{p} \#(S \to T)$, then we blame the *cast*: i.e. $\Uparrow p$. This is because the context (the surrounding code) was promised a value that should *not* be `null`, yet the cast delivered `null`.
- On the other hand, if the cast is *to an unsafe nullable function*, e.g. $v : \#(S \to T) \Longrightarrow^{p} !(S \to T)$, then we blame the context, because the context should know that the presumptive function value could in fact be `null`, but nevertheless chose to use a regular application, instead of a safe application.
- Casts to a *safe nullable function*, e.g. $v : \#(S \to T) \Longrightarrow^{p} ?(S \to T)$, will never fail, because the type system ensures that such functions are *always* applied through safe applications.

In addition to the rules above, our blame assignment needs to support *nested casts*. For example, suppose we have a `null` value that passes through the following casts, $\texttt{Null} \Rightarrow^{p}\ ? \Rightarrow^{q}\ \# \Rightarrow^{r}\ !$[7]. If the resulting cast is used in the function position of an application, it will lead to a failure, but which cast should we blame? We could blame $\overline{r}$, as per the second blame assignment rule above. However, something feels off, because intuitively a cast $\# \Rightarrow^{r}\ !$ should never be blamed for a failure. Indeed, the cast was promised a non-null value, which it should be safe to consider as a !. Instead, we identify $? \Rightarrow^{q}\ \#$ as the problem, and blame q, as per the first rule above.

To summarize, blame assignment is a two-part process: we first identify the cast responsible for the error using a *blame assignment* relation $\uparrow$ (this might involve skipping over one or

[6] Since $\lambda_{\texttt{null}}$ is a core calculus, it does not have objects or classes, but only functions. In $\lambda_{\texttt{null}}$ it is function types that are nullable or non-nullable.

[7] Here we are using a shorthand syntax for casts, where we only show the top-level function type. For example, we abbreviate a cast $s : \#(S \to T) \Longrightarrow^{p} !(S \to T)$ as $\# \Rightarrow^{p}\ !$

more nested casts), and then we blame the relevant label, or its complement, depending on whether the destination type is # or !.

3.3 $\lambda^s_{\texttt{null}}$

With $\lambda_{\texttt{null}}$ sketched, we then define a *second, higher-level* calculus $\lambda^s_{\texttt{null}}$ ("stratified lambda null"). Whereas in $\lambda_{\texttt{null}}$ the three function types can be mixed freely, $\lambda^s_{\texttt{null}}$ stratifies terms into *implicit* and *explicit* sublanguages. Within the implicit sublanguage, we can only use unsafe nullable functions (e.g. $!(S \to T)$), while in the explicit sublanguage we can use both non-nullable ($\#(S \to T)$) and safe nullable functions ($?(S \to T)$). The implicit sublanguage models languages where `null` is a subtype of any other (reference) type, like Java. The explicit sublanguage models languages where the user can choose whether a type is nullable or not, like Kotlin and Scala.

The last step is to model the interoperability between the implicit and explicit worlds. To do that, we add to $\lambda^s_{\texttt{null}}$ an `import` term that makes an implicit term available to the explicit world and vice versa. Imports look very similar to let-bindings: $\text{import}_{\text{e}}\ x : T_e = (t_i : T_i)\ \text{in}\ t_e$. This says that we evaluate the implicit term t_i and assign it to x, which is then available in the body t_e (implicit and explicit terms and types are written in red and blue, respectively). Additionally, the implicit type of t_i is T_i, but to the explicit world the type is translated as T_e. This kind of view shift in the type closely models what happens in real-world languages that support explicit nulls, but need to operate with another language where `null` is implicit. For example, the Java type `String` is translated as `String|Null` in Scala.

3.3.1 Semantics

We give type systems for $\lambda_{\texttt{null}}$ and $\lambda^s_{\texttt{null}}$, and an operational semantics for $\lambda_{\texttt{null}}$. The semantics of $\lambda^s_{\texttt{null}}$ are given via a desugaring to $\lambda_{\texttt{null}}$. The desugaring is straightforward, but it allows us to identify the three kinds of casts that can make a program fail:

- *Internal* casts within the *implicit* world.
- *Internal* casts within the *explicit* world.
- *Interoperability* casts that result from desugaring `imports`. For example, the `import` term above generates the cast $t_i : T_i \overset{\mathcal{I}}{\Longrightarrow} T_e$. Similarly, an import of an explicit term into the implicit world would generate a cast $t_e : T_e \overset{\mathcal{E}}{\Longrightarrow} T_i$. Here, $\mathcal{I}$ and $\mathcal{E}$ are labels that interoperability casts based on the cast's "direction".

3.3.2 Metatheory

We show that if we start with a well-typed term from $\lambda^s_{\texttt{null}}$, desugar it, and evaluate it using the $\lambda_{\texttt{null}}$ operational semantics, then the term's normal form (if it exists) is either a value, or an error with blame. In fact, we are able to characterize this behaviour more precisely. By reasoning about which casts are safe using *positive* and *negative* subtyping, which are standard tools from gradual typing, we are able to show our main result:

- Internal casts within the explicit world can *never* be blamed for failures.
- Interoperatibility casts *can* be blamed, but we always blame the implicit world in such cases. That is, the blame always goes to $\mathcal{I}$ or $\overline{\mathcal{E}}$.

This main result formalizes our intuition that *explicitly nullable programs can't be blamed.* It is also evidence that gradual typing can accurately model the null interoperability problem. All our results have been verified in Coq.

x, y, z	**Variables**	$r ::=$	**Results**
		t	term
p, q	**Blame labels**	$\Uparrow p$	blame
$\overline{p}$	**Label complement**		
		$S, T, U ::=$	**Types**
$f, s, t ::=$	**Terms**	`Null`	`null` type
x	variable	$\alpha\ (S \to T)$	function type with modality
`null`	`null` literal		
$\lambda(x\colon T).s$	abstraction	$\alpha, \beta ::=$	**Function Type Modality**
$s\ t$	application	$\#$	presumed non-nullable
$\mathrm{app}(f, s, t)$	safe application	$?$	safe nullable
$s : S \Longrightarrow^p T$	cast	$!$	unsafe nullable
$u, v ::=$	**Values**		
$\lambda(x\colon T).s$	abstraction		
`null`	`null` literal		
$v : S \Longrightarrow^p T$	cast		

Figure 1 Terms and types of $\lambda_{\texttt{null}}$

4 A Calculus with Implicit and Explicit Nulls

In this section, we describe the $\lambda_{\texttt{null}}$ calculus in full. $\lambda_{\texttt{null}}$ is based on the blame calculus of Wadler et al. [27, 26]. $\lambda_{\texttt{null}}$ contains the two key ingredients we need to model language interoperability with respect 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 1, and are explained below. Section 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.

4.1 Values of $\lambda_{\texttt{null}}$

A value in $\lambda_{\texttt{null}}$ can be any of the following: an abstraction $\lambda(x\colon T).s$, the `null` literal, 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 e.g. 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
```

4.2 Terms of $\lambda_{\texttt{null}}$

A term of $\lambda_{\texttt{null}}$ is either a variable x, the literal `null`, an abstraction $\lambda(x\colon T).s$, an application $s\ t$, a *safe application* $\mathrm{app}(s,t,u)$, 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.
- A safe application $\mathrm{app}(s,t,u)$ 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 [12]:

  $\mathrm{app}(s,t,u) \equiv$ `if (s != null) then s t else u`

  In particular, this means safe applications are "lazy": they do not initially evaluate either the argument t or sentinel value u. Instead, we only evaluate the expression s in function position, and then proceed depending on whether s is `null` or not.
  For the desugaring above to work 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 closely model Kotlin's "Elvis" operator [16], written `?:`. In Kotlin, the expression `a ?: b` evaluates to `a`, unless the left-hand side is `null`, in which case the entire expression evaluates to `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 should the cast cause a failure.

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 Types of $\lambda_{\texttt{null}}$

The types of $\lambda_{\texttt{null}}$ are also shown in Figure 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 $\texttt{String}_{\texttt{Scala}}$. For conciseness, we will refer to these types simply as *non-nullable* function types.
- 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 $\mathrm{app}(s,t,u)$, even though the former might fail. This corresponds to types in Java, which are implicitly nullable.

As we will see below, some typing rules apply to more than one function type. For example, when typing an application $s\ t$, we will require that s have a type of the form $\#(S \to T)$ or $!(S \to T)$. Instead of duplicating the relevant inference rule, the syntax for function types $\alpha\ (S \to T)$ includes a *modality* α. In the application case, we can then say that s must have type $\alpha\ (S \to T)$ with $\alpha \in \{\#, !\}$.

$$\boxed{\Gamma \vdash t : T}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\Gamma \vdash \texttt{null} : \texttt{Null} \quad \text{(T-NULL)}$$

$$\frac{\Gamma, x : S \vdash s : T}{\Gamma \vdash \lambda(x\colon S).s : \#(S \to T)} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash s : \alpha\ (S \to T) \quad \alpha \in \{\#, !\} \quad \Gamma \vdash t : S}{\Gamma \vdash s\ t : T} \quad \text{(T-APP)}$$

$$\frac{\Gamma \vdash f : \alpha\ (S \to T) \quad \alpha \in \{?, !\} \quad \Gamma \vdash s : S \quad \Gamma \vdash t : T}{\Gamma \vdash \mathrm{app}(f, s, t) : T} \quad \text{(T-SAFEAPP)}$$

$$\frac{\Gamma \vdash s : S \quad S \rightsquigarrow T}{\Gamma \vdash (s : S \Longrightarrow^p T) : T} \quad \text{(T-CAST)}$$

$$\boxed{S \rightsquigarrow T}$$

$$\texttt{Null} \rightsquigarrow \texttt{Null} \quad \text{(C-NULLREFL)}$$

$$\frac{\alpha \in \{?, !\}}{\texttt{Null} \rightsquigarrow \alpha\ (S \to T)} \quad \text{(C-NULL)}$$

$$\frac{S' \rightsquigarrow S \quad T \rightsquigarrow T' \quad \alpha, \beta \in \{\#, ?, !\}}{\alpha\ (S \to T) \rightsquigarrow \beta\ (S' \to T')} \quad \text{(C-ARROW)}$$

Figure 2 Typing and compatibility rules of $\lambda_{\texttt{null}}$

Keeping $\lambda_{\texttt{null}}$ simple. We could reduce the number of function types and avoid the need for safe applications through a combination of sum types and case analysis. For example, in Scala nullable values are represented with sum types (e.g. a nullable string has type `String | Null`). The case analysis in turn requires support for flow-typing:

```
val s: String | Null = ...
// s inferred to have type String in the 'then' branch, so s.length is type-correct
val len: Int = if (s != null) s.length else 0
```

Since λ_{null} is a core calculus, we focus on modelling the assignment of blame for nullability errors, which revolves around blaming casts or their client code, at function application time. This is why $\lambda_{\texttt{null}}$ eschews sum types and flow typing in favour of primitives for nullable function types and safe applications. Additionally, both of these primitives appear in modern programming languages (e.g. in Kotlin).

4.4 Typing $\lambda_{\texttt{null}}$

The typing rules for $\lambda_{\texttt{null}}$ are shown in Figure 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 models languages with implicit nullability (like Java), where the type system allows operations that can lead to null-related errors.
- **(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. For instance,

$$\dfrac{\dfrac{}{\vdash \texttt{null} : \texttt{Null}}\,\textsc{T-Null} \qquad \dfrac{}{\texttt{Null} \rightsquigarrow ?(\texttt{Null} \rightarrow \texttt{Null})}\,\textsc{C-Null}}{\vdash \texttt{null} : \texttt{Null} \Longrightarrow^p ?(\texttt{Null} \rightarrow \texttt{Null}) : \ ?(\texttt{Null} \rightarrow \texttt{Null})}\,\textsc{T-Cast}$$

4.4.1 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 2.

▶ **Lemma 1.** *Compatibility is reflexive, but is neither symmetric nor transitive.*

A counter-example to symmetry is that $\texttt{Null} \rightsquigarrow ?(\texttt{Null} \rightarrow \texttt{Null})$, but the latter is not compatible with the former. A counter-example to transitivity is that $\texttt{Null} \rightsquigarrow ?(\texttt{Null} \rightarrow \texttt{Null})$ and $?(\texttt{Null} \rightarrow \texttt{Null}) \rightsquigarrow \#(\texttt{Null} \rightarrow \texttt{Null})$, but `Null` is not compatible with $\#(\texttt{Null} \rightarrow \texttt{Null})$.

4.5 Semantics of $\lambda_{\texttt{null}}$

We give a small-step operational semantics for $\lambda_{\texttt{null}}$, using evaluation contexts. The rules are shown in Figure 5. Notice that the result r of an evaluation step can be a term *or* an error, denoted by $\Uparrow p$.

4.5.1 Auxiliary Predicates

The unary predicates on types **null** and **abs**, shown in Figure 3, test whether a value v is equal to `null` or to a lambda abstraction, respectively. These predicates are able to “see through” casts.

▶ **Example 2.** The following hold:

- $\textbf{null}(\texttt{null})$, $\textbf{null}(\texttt{null} : \texttt{Null} \Longrightarrow^p \#(\texttt{Null} \rightarrow \texttt{Null}))$
- $\textbf{abs}(\lambda(x\colon \texttt{Null}).x)$, $\textbf{abs}(\lambda(x\colon \texttt{Null}).x : \#(\texttt{Null} \rightarrow \texttt{Null}) \Longrightarrow^p ?(\texttt{Null} \rightarrow \texttt{Null}))$

4.5.2 Reduction Relation

The decision tree in Figure 4 shows a simplified view of the reduction rules. The rules are described in detail below.

- **R-App** is standard beta reduction.

$\boxed{\mathbf{null}(v)}$ $\boxed{\mathbf{abs}(v)}$

$$\mathbf{null}(\mathrm{null}) \quad (\text{N-NULL}) \qquad\qquad \mathbf{abs}(\lambda(x\colon T).s) \quad (\text{A-ABS})$$

$$\frac{\mathbf{null}(v)}{\mathbf{null}(v : S \Longrightarrow^{p} T)} \quad (\text{N-CAST}) \qquad\qquad \frac{\mathbf{abs}(v)}{\mathbf{abs}(v : S \Longrightarrow^{p} T)} \quad (\text{A-CAST})$$

Figure 3 **abs** and **null** predicates

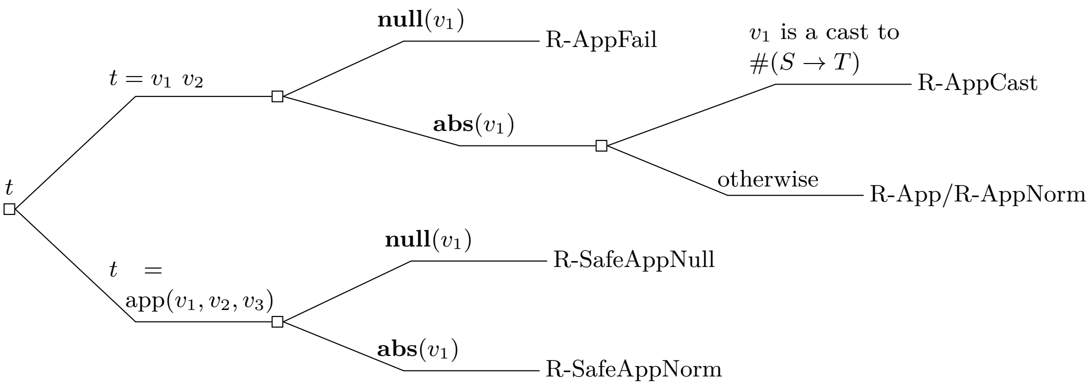

Figure 4 Simplified decision tree for $\lambda_{\texttt{null}}$ reduction rules

- **R-AppFail** 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 $\mathbf{null}(v)$. In this case, the entire term (and not just the subterm within the evaluation context) evaluates to an error. What remains is to determine the blame label that we will use. This we do using the *blame assignment* relation (also shown in Figure 5): we write $v \uparrow p$ to indicate that the blame should go to a label p. As we will see in Section 4.5.3, v will contain one or more casts, and the label p is obtained from one of the casts. Here is a sample application of R-AppFail, where $v = \texttt{null} : \texttt{Null} \Longrightarrow^{p} !(\texttt{Null} \to \texttt{Null})$:

  $$(\texttt{null} : \texttt{Null} \Longrightarrow^{p} !(\texttt{Null} \to \texttt{Null}))\ \texttt{null} \longmapsto \Uparrow \overline{p}$$

  In this case, the only cast in v is selected as the source of the blame (in general, v could contain multiple casts). We blame $\overline{p}$ because the surrounding code (the code doing the application v `null`), should have used a safe application, based on v's nullable type.
- **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 [11]. 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 complemented ($\overline{p}$), because it is the *context* (the code calling the function v) who 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 [11] remark, the handling of the blame label matches the rule for function subtyping present in other system, 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 5) achieves this translation of casts.

  ▶ **Example 3.** Let $t = \lambda(x\colon \texttt{Null}).x$. Suppose we are evaluating the application

  $$(t : \#(\texttt{Null} \to \texttt{Null}) \Longrightarrow^p ?(\texttt{Null} \to \texttt{Null}))\ \texttt{null}$$

  We proceed by first noticing that $\mathbf{abs}(t : \#(\texttt{Null} \to \texttt{Null}) \Longrightarrow^p ?(\texttt{Null} \to \texttt{Null}))$. Then we normalize the value in the function position

  $$\dfrac{\dfrac{}{t \gg t}\ \textsc{Norm-Abs}}{t : \#(\texttt{Null} \to \texttt{Null}) \Longrightarrow^p ?(\texttt{Null} \to \texttt{Null}) \gg t : \#(\texttt{Null} \to \texttt{Null}) \Longrightarrow^p \#(\texttt{Null} \to \texttt{Null})}\ \textsc{Norm-Cast}$$

  Now we can use R-AppNorm to turn the origin application into

  $$(t : \#(\texttt{Null} \to \texttt{Null}) \Longrightarrow^p \#(\texttt{Null} \to \texttt{Null}))\ \texttt{null}$$

  We can then proceed 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.

4.5.3 Blame Assignment

The blame assignment relation is responsible for determining which cast in a value is responsible for a nullability error. Once the responsible cast has been identified, blame assignment also determines whether the blame is *positive* (blame the *cast*) or *negative* (blame the context). The notation for blame assignment is $v \uparrow p$, and indicates that if the value v, containing one or more casts, leads to a failure (because $\mathbf{null}(v)$ holds and v was used in the function position of an application), then we will blame label p.

The rules for blame assignment are shown in Figure 5. There are two kinds of rules, based on what they do with the outermost cast: those that *discard* the outermost cast, and those that use the outermost cast to assign blame. Both kinds are described below.

Reduction

$$\boxed{s \longmapsto r}$$

$$E[(\lambda(x\colon T).s)\ v] \longmapsto E[[v/x]s] \quad \text{(R-App)}$$

$$\frac{\mathbf{null}(v) \qquad v \uparrow p}{E[v\ u] \longmapsto \Uparrow p} \quad \text{(R-AppFail)}$$

$$\frac{\mathbf{abs}(v) \qquad v \gg v'}{E[v\ u] \longmapsto E[v'\ u]} \quad \text{(R-AppNorm)}$$

$$\frac{\mathbf{null}(v)}{E[\mathrm{app}(v, s, t)] \longmapsto E[t]} \quad \text{(R-SafeAppNull)}$$

$$\frac{\mathbf{abs}(v) \qquad v \gg v'}{E[\mathrm{app}(v, s, t)] \longmapsto E[v'\ s]} \quad \text{(R-SafeAppNorm)}$$

$$\frac{\mathbf{abs}(v)}{E[(v : \#(S_1 \to S_2) \Longrightarrow^p \#(T_1 \to T_2))\ u] \longmapsto E[(v\ (u : T_1 \Longrightarrow^{\overline{p}} S_1)) : S_2 \Longrightarrow^p T_2]} \quad \text{(R-AppCast)}$$

Evaluation contexts

$E ::=$

$[]$
$E\ s$
$v\ E$
$\mathrm{app}(E, s, t)$
$E : S \Longrightarrow^p T$

Blame assignment

$$\boxed{v \uparrow p}$$

$$(v : \mathtt{Null} \Longrightarrow^p !(S \to T)) \uparrow \overline{p} \quad \text{(B-Null)}$$

$$\frac{v \uparrow p'}{(v : \#(S \to T) \Longrightarrow^p U) \uparrow p'} \quad \text{(B-NonNullable)}$$

$$\frac{v \uparrow p'}{(v :!(S \to T) \Longrightarrow^p !(S' \to T')) \uparrow p'} \quad \text{(B-Unsafe!)}$$

$$(v :?(S \to T) \Longrightarrow^p !(S' \to T')) \uparrow \overline{p} \quad \text{(B-Safe!)}$$

$$\frac{\alpha \in \{?, !\}}{(v : \alpha\ (S \to T) \Longrightarrow^p \#(S' \to T')) \uparrow p} \quad \text{(B-Nullable\#)}$$

Normalization

$$\boxed{v \gg u}$$

$$\lambda(x\colon T).s \gg \lambda(x\colon T).s \quad \text{(Norm-Abs)}$$

$$\frac{v \gg u \qquad \alpha, \beta \in \{\#, ?, !\}}{v : \alpha\ (S_1 \to S_2) \Longrightarrow^p \beta\ (T_1 \to T_2) \gg u : \#(S_1 \to S_2) \Longrightarrow^p \#(T_1 \to T_2)} \quad \text{(Norm-Cast)}$$

Figure 5 Reduction rules of $\lambda_{\mathtt{null}}$, along with blame assignment and normalization relations

Rules that discard the outermost cast:

- **B-NonNullable** handles the cast where the outermost cast has the form $v' : \#(S \to T) \Longrightarrow^p U$; that is, the source type is a *non-nullable* function type. Intuitively, we do not want to assign blame to either p or $\overline{p}$, because the *source* type in the cast promised that the underlying value is *non-null*, but the value being cast *is* in fact `null`. That is, there must be another "risky" cast that is part of v' that should be blamed. For example, consider the cast $((\texttt{Null} \Rightarrow^r\ ?) \Rightarrow^q\ \#) \Rightarrow^p\ !$, where we have written only the top level "modalities" of the function types. In this cast, a `null` value that starts as having type `Null` is cast first to a safe nullable function, then to a non-nullable function, and finally to an unsafe nullable function. Blame assignment models the intuition that the second cast (from ? to #) is the unsafe one, and so should be blamed. Because the destination type in that second cast is a #, we blame the term (i.e. blame q).
- **B-Unsafe!** is similar to the previous case: when confronted with a cast $v' : S \Longrightarrow^p T$ where *both* S and T are ! types, then we "recurse" on v' to find the guilty cast. The reason is that the last cast did not change the kind of function type, so whatever went wrong must have happened earlier. For example, suppose the outermost cast is $!(\texttt{Null} \to \texttt{Null}) \Rightarrow^p\ !(\texttt{Null} \to \texttt{Null})$. This cast leaves the type unchanged, so it should never be blamed for a failure.
  Notice that the equivalent rule for # types is subsumed by B-NonNullable. ? types do not need an equivalent rule, because a cast of the form $v : S \Longrightarrow^p ?$ cannot fail.

Rules that assign blame based on the outermost cast:

- **B-Null** handles the case where we cast `Null` to an unsafe function type. In this case, we blame the context, because the target type is a !.
- **B-Nullable#** casts some kind of nullable function (either a ? or a !) to a non-nullable function. In this case, we want to blame the term, because the context was promised a non-nullable value that nevertheless ended up being `null`.
- **B-Unsafe!** handles casts of the form $? \Rightarrow^p\ !$. In this case, we blame $\overline{p}$, because the context should know that the value is potentially `null`.

4.6 Metatheory of $\lambda_{\texttt{null}}$

In developing the metatheory, we closely followed the syntactic approach taken in Wadler and Findler [27]. All the results in this section have been verified using the Coq proof assistant.

4.6.1 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 [29]. First, we need an auxiliary lemma that says that normalization preserves well-typedness.

▶ **Lemma 4** (Soundness of normalization). *Let $\alpha \in \{\#, ?, !\}$, $\Gamma \vdash v : \alpha\ (S \to T)$ and $v \gg v'$. Then $\Gamma \vdash v' : \#(S \to T)$.*

Then we can prove preservation.

▶ **Lemma 5** (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 6** (Completeness of normalization). *Let $\alpha \in \{\#, ?, !\}$, $\Gamma \vdash v : \alpha\ (S \to T)$ and suppose that $\boldsymbol{abs}(v)$ 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. $\mathrm{app}(v, u, u')$) 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 7** (Weakening). *Let $\Gamma \vdash t : T$ and $x \notin dom(\Gamma)$. Then $\Gamma, x : U \vdash t : T$ for any type U.*

We can then show progress.

▶ **Lemma 8** (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'*

4.6.2 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.

Inspecting the rules for blame assignment shows 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, adapted from Wadler and Findler [27], are shown in Figure 6.

▶ **Example 9.** Since the type system ensures that $?(S \to T)$ functions are only ever applied through safe casts, 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 10.** 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 $\texttt{Null} \ <:^- \ !(S \to T)$.

▶ **Lemma 11** (Positive and negative subtyping are reflexive). *Let T be an arbitrary type. Then $T \ <:^+ \ T$ and $T \ <:^- \ T$.*

▶ **Lemma 12** (Subtyping implies compatibility). *Let S and T be types. Then*
- $S \ <:^+ \ T \implies S \rightsquigarrow T$
- $S \ <:^- \ T \implies S \rightsquigarrow T$

Lemma 12 implies that if S is a (positive or negative) subtype of T, then we can cast S to T (which requires compatibility).

$$\boxed{S \ <:^+ \ T}$$

$$\text{Null} \ <:^+ \ \text{Null} \ (\text{PS-NullRefl})$$

$$\frac{\alpha \in \{?, !\}}{\text{Null} \ <:^+ \ \alpha \ (S \to T)} \ (\text{PS-Null})$$

$$\frac{S' \ <:^- \ S \qquad T \ <:^+ \ T' \qquad \alpha \in \{\#, ?, !\}}{\#(S \to T) \ <:^+ \ \alpha \ (S' \to T')} \ (\text{PS-Arrow\#})$$

$$\frac{S' \ <:^- \ S \qquad T \ <:^+ \ T' \qquad \alpha, \beta \in \{?, !\}}{\alpha \ (S \to T) \ <:^+ \ \beta \ (S' \to T')} \ (\text{PS-ArrowNullable})$$

$$\boxed{S \ <:^- \ T}$$

$$\text{Null} \ <:^- \ \text{Null} \ (\text{NS-NullRefl})$$

$$\text{Null} \ <:^- \ ?(S \to T) \ (\text{NS-Null})$$

$$\frac{S' \ <:^+ \ S \qquad T \ <:^- \ T' \qquad \alpha \in \{\#, ?, !\}}{\#(S \to T) \ <:^- \ \alpha \ (S' \to T')} \ (\text{NS-Arrow\#})$$

$$\frac{S' \ <:^+ \ S \qquad T \ <:^- \ T' \qquad \alpha \in \{\#, ?, !\}}{!(S \to T) \ <:^- \ \alpha \ (S' \to T')} \ (\text{NS-Arrow!})$$

$$\frac{S' \ <:^+ \ S \qquad T \ <:^- \ T' \qquad \alpha \in \{\#, ?\}}{?(S \to T) \ <:^- \ \alpha \ (S' \to T')} \ (\text{NS-Arrow?})$$

Figure 6 Positive and negative subtyping

The next step is to lift positive and negative subtyping to work on terms. The *safe for* relation, again adapted from Wadler and Findler [27] and shown in Figure 7, 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. We 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 following lemmas say that **safe for** is preserved by normalization and substitution.

▶ **Lemma 13** (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 14** (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 15** (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.*

▶ **Theorem 16** (Progress of safe terms). *Let $\vdash t : T$ and t **safe for** p. Then either*

- *t is a value*
- *$t \longmapsto \Uparrow p'$, for some blame label $p' \neq p$.*
- *$t \longmapsto t'$, for some term t'*

$$\boxed{t \textbf{ safe for } p}$$

$$x \textbf{ safe for } p \quad \text{(SF-Var)}$$

$$\text{null} \textbf{ safe for } p \quad \text{(SF-Null)}$$

$$\frac{s \textbf{ safe for } p}{\lambda(x\colon T).s \textbf{ safe for } p} \quad \text{(SF-Abs)}$$

$$\frac{s \textbf{ safe for } p \qquad t \textbf{ safe for } p}{s\ t \textbf{ safe for } p} \quad \text{(SF-App)}$$

$$\frac{f \textbf{ safe for } p \qquad s \textbf{ safe for } p \qquad t \textbf{ safe for } p}{\text{app}(f, s, t) \textbf{ safe for } p} \quad \text{(SF-SafeApp)}$$

$$\frac{S \;<:^{+}\; T \qquad s \textbf{ safe for } p}{s : S \Longrightarrow^{p} T \textbf{ safe for } p} \quad \text{(SF-CastPos)}$$

$$\frac{S \;<:^{-}\; T \qquad s \textbf{ safe for } p}{s : S \Longrightarrow^{\overline{p}} T \textbf{ safe for } p} \quad \text{(SF-CastNeg)}$$

$$\frac{s \textbf{ safe for } p \qquad q \neq p \qquad q \neq \overline{p}}{s : S \Longrightarrow^{q} T \textbf{ safe for } p} \quad \text{(SF-CastDiff)}$$

Figure 7 Safe for relation

Notice that this theorem does not preclude the term from stepping to an error, but it does say that the error *will not have blame label* p. This is a stronger guarantee than what we get from Lemma 8 (Progress), which placed no restrictions on the blame label p' when $t \longmapsto \Uparrow p'$.

Here are a few implications of the theorems above:

- 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 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 our model of language interoperability, described in the next section.

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 $\lambda_{\texttt{null}}$ called $\lambda^{s}_{\texttt{null}}$ ("stratified lambda null").

$t ::=$	**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
null	null literal	null	null literal
$\lambda(x\colon T_e).s_e$	abstraction	$\lambda(x\colon T_i).(s_i : \boxed{S_i})$	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
$s_e : S_e \Longrightarrow T_e$	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**
`Null`	null	`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 8 Terms and types of $\lambda^s_{\texttt{null}}$. Differences with $\lambda_{\texttt{null}}$ are highlighted.

5.1 Terms and Types of $\lambda^s_{\texttt{null}}$

The terms and types of $\lambda^s_{\texttt{null}}$ are shown in Figure 8. The main difference with respect to $\lambda_{\texttt{null}}$ is that terms and types are stratified into the world of explicit nulls (subscript $_\mathrm{e}$) and the world of implicit nulls (subscript $_\mathrm{i}$). 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

$$\mathrm{import_e}\ x : T_e = (t_i : T_i)\ \mathrm{in}\ t_e$$

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).

Casts in the explicit sublanguage do not have blame labels. This is because the type system will force all such casts to be *upcasts*: i.e. casts that respect subtyping. We will see that this means that "internal" casts within the explicit sublanguage will never be blamed for failures. Relatedly, notice that λ^s_{null}, unlike e.g. Scala, has no subsumption rule. We opted for casts instead of subsumption to keep λ^s_{null} close to λ_{null}. Subsumptions and casts are similarly expressive: one can think of subsumption as casts automatically introduced by the type checker.

Finally, abstractions in the implicit sublanguage, written $\lambda(x\colon T_i).(s : S_i)$, are annotated with their return type S_i. This is not strictly necessary, but it simplifies the presentation of desugaring in Section 5.3.

5.2 Typing $\lambda^s_{\texttt{null}}$

The typing rules for $\lambda^s_{\texttt{null}}$ are shown in Figure 9. These rules are almost verbatim copies of the typing rules for $\lambda_{\texttt{null}}$ (and the compatibility relation is reused from Figure 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 $\text{import}_e\ x : T_e = (t_i : T_i)\ \text{in}\ t_e$, 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 $\varnothing \vdash_{\mathrm{i}} t_i : T_i$. 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 $T_i \hookrightarrow_{\mathrm{N}} T_e$, and is shown in Figure 10.
- 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 $T_e \hookrightarrow_{\mathrm{E}} T_i$. Erasure is also shown in Figure 10.

▶ Remark 17. 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, we 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 10, 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 we 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 $\vdash_{\mathrm{e}}$ type system.

Notice that the typing rules for casts are now different in the explicit and implicit sublanguages. In the implicit sublanguage, like in $\lambda_{\texttt{null}}$, to type the cast $s_i : S_i \Longrightarrow^p T_i$, we require that S_i be *compatible* with T_i ($S_i \rightsquigarrow T_i$). By contrast, when typing casts in the explicit sublanguage, e.g. $s_e : S_e \Longrightarrow T_e$, we check that S_e can be *upcasted* to T_e, written $S_e <:_e T_e$. The upcasting is defined by the *explicit subtyping* relation, given in Figure 11. Explicit subtyping is defined just like we would define a regular subtyping relation, that is, it implies *substitutability* [17]. For example, we have the judgment $\#(S \to T) <:_e ?(S \to T)$, which is akin to the Scala judgment `String <: String|Null`.

Crucially, we can show that explicit subtyping implies *both* positive and negative subtyping.

▶ **Lemma 18.** *$S <:_e T$ implies $S <:^+ T$ and $S <:^- T$.*

This is useful, because it hints that casts that rely on explicit subtyping will never be blamed for failures.

$\boxed{\Gamma \vdash_{\mathrm{e}} t_e : T_e}$

$$\frac{\Gamma(x) = T_e}{\Gamma \vdash_{\mathrm{e}} x : T_e} \quad (\text{TE-Var})$$

$$\Gamma \vdash_{\mathrm{e}} \texttt{null} : \texttt{Null} \quad (\text{TE-Null})$$

$$\frac{\Gamma, x : S_e \vdash_{\mathrm{e}} s_e : T_e}{\Gamma \vdash_{\mathrm{e}} \lambda(x{:}\ S_e).s_e : \#(S_e \to T_e)} \quad (\text{TE-Abs})$$

$$\frac{\Gamma \vdash_{\mathrm{e}} s_e : \#(S_e \to T_e) \qquad \Gamma \vdash_{\mathrm{e}} t_e : S_e}{\Gamma \vdash_{\mathrm{e}} s_e\ t_e : T_e} \quad (\text{TE-App})$$

$$\frac{\Gamma \vdash_{\mathrm{e}} f_e :?(S_e \to T_e) \qquad \Gamma \vdash_{\mathrm{e}} s_e : S \qquad \Gamma \vdash_{\mathrm{e}} t_e : T_e}{\Gamma \vdash_{\mathrm{e}} \mathrm{app}(f_e, s_e, t_e) : T_e} \quad (\text{TE-SafeApp})$$

$$\frac{\Gamma \vdash_{\mathrm{e}} s_e : S_e \qquad S_e <:_{\mathrm{e}} T_e}{\Gamma \vdash_{\mathrm{e}} (s_e : S_e \Longrightarrow T_e) : T_e} \quad (\text{TE-Cast})$$

$$\frac{\Gamma, x : T_e \vdash_{\mathrm{e}} t_e : S_e \qquad \varnothing \vdash_{\mathrm{i}} t_i : T_i \qquad T_i \hookrightarrow_{\mathrm{N}} T_e}{\Gamma \vdash_{\mathrm{e}} \mathrm{import}_{\mathrm{e}}\ x : T_e = (t_i : T_i)\ \mathrm{in}\ t_e : S_e} \quad (\text{TE-Import})$$

$\boxed{\Gamma \vdash_{\mathrm{i}} t_i : T_i}$

$$\frac{\Gamma(x) = T_i}{\Gamma \vdash_{\mathrm{i}} x : T_i} \quad (\text{TI-Var})$$

$$\Gamma \vdash \texttt{null} : \texttt{Null} \quad (\text{TI-Null})$$

$$\frac{\Gamma, x : S_i \vdash_{\mathrm{i}} s_i : T_i}{\Gamma \vdash_{\mathrm{i}} \lambda(x{:}\ S_i).(s_i : T_i) :!(S_i \to T_i)} \quad (\text{TI-Abs})$$

$$\frac{\Gamma \vdash_{\mathrm{i}} s_i :!(S_i \to T_i) \qquad \Gamma \vdash_{\mathrm{i}} t_i : S_i}{\Gamma \vdash_{\mathrm{i}} s_i\ t_i : T_i} \quad (\text{TI-App})$$

$$\frac{\Gamma \vdash_{\mathrm{i}} f_i :!(S_i \to T_i) \qquad \Gamma \vdash_{\mathrm{i}} s_i : S_i \qquad \Gamma \vdash_{\mathrm{i}} t_i : T_i}{\Gamma \vdash_{\mathrm{i}} \mathrm{app}(f_i, s_i, t_i) : T_i} \quad (\text{TI-SafeApp})$$

$$\frac{\Gamma \vdash_{\mathrm{i}} s : S_i \qquad S_i \rightsquigarrow T_i}{\Gamma \vdash_{\mathrm{i}} (s_i : S_i \Longrightarrow^p T_i) : T_i} \quad (\text{TI-Cast})$$

$$\frac{\Gamma, x : T_i \vdash_{\mathrm{i}} t_i : S_i \qquad \varnothing \vdash_{\mathrm{e}} t_e : T_e \qquad T_e \hookrightarrow_{\mathrm{E}} T_i}{\Gamma \vdash_{\mathrm{i}} \mathrm{import}_{\mathrm{i}}\ x : T_i = (t_e : T_e)\ \mathrm{in}\ t_i : S_i} \quad (\text{TI-Import})$$

Figure 9 Typing rules of $\lambda^s_{\texttt{null}}$

$$\boxed{T_i \hookrightarrow_{\mathrm{N}} T_e} \qquad \boxed{T_e \hookrightarrow_{\mathrm{E}} T_i}$$

$$\mathtt{Null} \hookrightarrow_{\mathrm{N}} \mathtt{Null} \quad (\textsc{N-Null}) \qquad \mathtt{Null} \hookrightarrow_{\mathrm{E}} \mathtt{Null} \quad (\textsc{E-Null})$$

$$\frac{S_i \hookrightarrow_{\mathrm{N}} S_e \qquad T_i \hookrightarrow_{\mathrm{N}} T_e}{!(S_i \to T_i) \hookrightarrow_{\mathrm{N}} ?(S_e \to T_e)}\ (\textsc{N-Arrow!}) \qquad \frac{S_e \hookrightarrow_{\mathrm{E}} S_i \qquad T_e \hookrightarrow_{\mathrm{E}} T_i}{?(S_e \to T_e) \hookrightarrow_{\mathrm{E}} !(S_i \to T_i)}\ (\textsc{E-Arrow?})$$

$$\frac{S_e \hookrightarrow_{\mathrm{E}} S_i \qquad T_e \hookrightarrow_{\mathrm{E}} T_i}{\#(S_e \to T_e) \hookrightarrow_{\mathrm{E}} !(S_i \to T_i)}\ (\textsc{E-Arrow\#})$$

Figure 10 Nullification and erasure relations

$$\mathtt{Null} <:_e \mathtt{Null} \quad (\textsc{ES-NullRefl})$$

$$\mathtt{Null} <:_e ?(S \to T) \quad (\textsc{ES-Null?})$$

$$\frac{S' <:_e S \qquad T <:_e T'}{\#(S \to T) <:_e \#(S' \to T')} \quad (\textsc{ES-Arrow\#})$$

$$\frac{S' <:_e S \qquad T <:_e T'}{\#(S \to T) <:_e ?(S' \to T')} \quad (\textsc{ES-Arrow?})$$

$$\frac{S' <:_e S \qquad T <:_e T'}{?(S \to T) <:_e ?(S' \to T')} \quad (\textsc{ES-Safe})$$

Figure 11 Explicit subtyping (upcast) relation

5.3 Desugaring $\lambda^s_{\mathtt{null}}$ to $\lambda_{\mathtt{null}}$

The last step is to give meaning to $\lambda^s_{\mathtt{null}}$ terms. We could repeat the approach followed for $\lambda_{\mathtt{null}}$ using operational semantics, but instead we will do something different. We will *desugar* $\lambda^s_{\mathtt{null}}$ terms and types to $\lambda_{\mathtt{null}}$ terms and types, respectively. This is useful, because in Section 4.6 we proved many results about $\lambda_{\mathtt{null}}$ terms, and we would like to re-use these results to reason about $\lambda^s_{\mathtt{null}}$ as well.

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 $\lambda^s_{\mathtt{null}}$ terms from the explicit sublanguage to $\lambda_{\mathtt{null}}$ terms. Similarly, $\mathcal{D}_{\mathrm{i}}$ is a function that maps $\lambda^s_{\mathtt{null}}$ terms from the implicit sublanguage to $\lambda_{\mathtt{null}}$ terms. Both functions are shown in Figure 12.

The first thing to notice is that we do not actually need to desugar *types*. This is because $\lambda^s_{\mathtt{null}}$ types (from both sublanguages) are *also* $\lambda_{\mathtt{null}}$ types.

When it comes to terms, most cases in Figure 12 are handled by straightforward structural recursion on the term. There are only four interesting cases:

- **(DE-Cast)** Casts in the explicit sublanguage do not have blame labels, but casts in $\lambda_{\mathtt{null}}$ must always have labels. When we desugar explicit casts, we tag them with the same ("compiler-generated") label $\mathcal{E}_{\mathrm{int}}$. Later, we show that these casts are never blamed for failures (neither positively nor negatively).
- **(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 9). However, the corresponding lambda in $\lambda_{\mathtt{null}}$, $\lambda(x\colon S_i).\mathcal{D}_{\mathrm{i}}(s_i)$, will have type $\#(S_i \to T_i)$. So that the metatheory in Section 5.4 works out, we need

$$\boxed{\mathcal{D}_\mathrm{e} : s_e \longrightarrow s}$$

$$\mathcal{D}_\mathrm{e}(x) = x \tag{DE-Var}$$
$$\mathcal{D}_\mathrm{e}(\texttt{null}) = \texttt{null} \tag{DE-Null}$$
$$\mathcal{D}_\mathrm{e}(\lambda(x\colon T_e).s_e) = \lambda(x\colon T_e).\mathcal{D}_\mathrm{e}(s_e) \tag{DE-Abs}$$
$$\mathcal{D}_\mathrm{e}(s_e\ t_e) = \mathcal{D}_\mathrm{e}(s_e)\ \mathcal{D}_\mathrm{e}(t_e) \tag{DE-App}$$
$$\mathcal{D}_\mathrm{e}(\mathrm{app}(f_e, s_e, t_e)) = \mathrm{app}(\mathcal{D}_\mathrm{e}(f_e), \mathcal{D}_\mathrm{e}(s_e), \mathcal{D}_\mathrm{e}(t_e)) \tag{DE-SafeApp}$$
$$\mathcal{D}_\mathrm{e}(s_e : S_e \Longrightarrow T_e) = \mathcal{D}_\mathrm{e}(s_e) : S_e \Longrightarrow^{\mathcal{E}_\mathrm{int}} T_e \tag{DE-Cast}$$
$$\mathcal{D}_\mathrm{e}(\mathrm{import}_\mathrm{e}\ x_e : T_e = (t_i : T_i)\ \mathrm{in}\ t_e) = (\lambda(x\colon T_e).\mathcal{D}_\mathrm{e}(t_e))\ (\mathcal{D}_\mathrm{i}(t_i) : T_i \Longrightarrow^{\mathcal{I}} T_e) \tag{DE-Import}$$

$$\boxed{\mathcal{D}_\mathrm{i} : s_i \longrightarrow s}$$

$$\mathcal{D}_\mathrm{i}(x) = x \tag{DI-Var}$$
$$\mathcal{D}_\mathrm{i}(\texttt{null}) = \texttt{null} \tag{DI-Null}$$
$$\mathcal{D}_\mathrm{i}(\lambda(x\colon S_i).(s_i : T_i)) = (\lambda(x\colon S_i).\mathcal{D}_\mathrm{i}(s_i)) : \#(S_i \to T_i) \Longrightarrow^{\mathcal{I}_\mathrm{int}} !(S_i \to T_i) \tag{DI-Abs}$$
$$\mathcal{D}_\mathrm{i}(s_i\ t_i) = \mathcal{D}_\mathrm{i}(s_i)\ \mathcal{D}_\mathrm{i}(t_i) \tag{DI-App}$$
$$\mathcal{D}_\mathrm{i}(\mathrm{app}(f_i, s_i, t_i)) = \mathrm{app}(\mathcal{D}_\mathrm{i}(f_i), \mathcal{D}_\mathrm{i}(s_i), \mathcal{D}_\mathrm{i}(t_i)) \tag{DI-SafeApp}$$
$$\mathcal{D}_\mathrm{i}(s_i : S_i \Longrightarrow^{p} T_i) = \mathcal{D}_\mathrm{i}(s_i) : S_i \Longrightarrow^{p} T_i \tag{DI-Cast}$$
$$\mathcal{D}_\mathrm{i}(\mathrm{import}_\mathrm{i}\ x_i : T_i = (t_e : T_e)\ \mathrm{in}\ t_i) = (\lambda(x\colon T_i).\mathcal{D}_\mathrm{i}(t_i))\ (\mathcal{D}_\mathrm{e}(t_e) : T_e \Longrightarrow^{\mathcal{E}} T_i) \tag{DI-Import}$$

Figure 12 Desugaring $\lambda^s_{\texttt{null}}$ terms to $\lambda_{\texttt{null}}$ terms

the types to match; hence the cast. This is another instance of a blame label being automatically inserted by desugaring. We will use the blame label $\mathcal{I}_{\mathrm{int}}$: the $\mathcal{I}$ stands for *implicit*, indicating that the term being cast is from the implicit sublanguage. The $_{\mathrm{int}}$ 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 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.

5.4 Metatheory of $\lambda^s_{\texttt{null}}$

The following lemma shows that nullification implies negative subtyping, and erasure implies positive subtyping.

▶ **Lemma 19.** *Let S and T be types. Then $S \hookrightarrow_N T$ implies $S <:^- T$ and $T <:^+ S$, and $S \hookrightarrow_E T$ implies $S <:^+ T$ 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!

▶ **Theorem 20** (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*
- $\Gamma \vdash_i t_i : T_i \implies \Gamma \vdash \mathcal{D}_i(t_i) : T_i$

▶ **Definition 21** (Set of user-written blame labels in a term). *We will denote the set of user-written 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. Notice that user-written blame labels can only come from implicit casts $s_i : S_i \Longrightarrow^p T_i$.*

The next theorem is our main result: it characterizes the failures that can occur while evaluating a (desugared) $\lambda^s_{\texttt{null}}$ term. Specifically, it says that:

- Upcasts within the explicit world, which have blame $\mathcal{E}_{\text{int}}$, are *never blamed* for failures, neither positively nor negatively.
- Interop casts that result from importing an implicit term into an explicit term can only fail with *positive* blame, that is, they blame $\mathcal{I}$. This means the term being cast, which originated in the implicit sublanguage, is at fault.
- Interop casts that result from importing an explicit term into an implicit term can only fail with *negative* blame, that is, they blame $\overline{\mathcal{E}}$. 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.
- Internal casts tagged with $\mathcal{I}_{\text{int}}$, which result from desugaring $\lambda(x\colon S_i).(s_i : T_i)$ expressions, are *never blamed* for failures, neither positively nor negatively. That is, the desugaring does not introduce faulty casts.
- User-written casts ($s_i : S_i \Longrightarrow^p T_i$) within the implicit sublanguage can still be blamed, but that is expected because some of those casts are indeed unsafe.

▶ **Theorem 22** (Explicitly nullable programs can't be blamed). *Let t be a term of λ^s_{null}. Suppose that $\{\mathcal{I}, \overline{\mathcal{I}}, \mathcal{I}_{int}, \overline{\mathcal{I}_{int}}, \mathcal{E}, \overline{\mathcal{E}}, \mathcal{E}_{int}, \overline{\mathcal{E}_{int}}\} \cap \textbf{labels}(t) = \emptyset$. 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** $\{\mathcal{E}_{int}, \overline{\mathcal{E}_{int}}, \overline{\mathcal{I}}, \mathcal{E}, \mathcal{I}_{int}, \overline{\mathcal{I}_{int}}\}$.*[8]
- *If $t = t_i$, then $\mathcal{D}_i(t_i)$ **safe for** $\{\mathcal{E}_{int}, \overline{\mathcal{E}_{int}}, \overline{\mathcal{I}}, \mathcal{E}, \mathcal{I}_{int}, \overline{\mathcal{I}_{int}}\}$.*

Just like a central result in gradual typing is that "well-typed programs can't be blamed" [27], we can summarize our main result as *explicitly nullable programs can't be blamed.*

6 Coq Mechanization

All our results have been verified using the Coq theorem prover. The two main differences between the presentation of $\lambda_{\texttt{null}}$ in this paper and in the Coq proofs are:

- The definition of evaluation in the Coq code does not use evaluation contexts, unlike Figure 5. Instead, we have explicit rules for propagating errors.
- The definition of terms in the Coq code uses a locally-nameless representation of terms [5].

In the mechanization of the proofs, we used the Ott [21] and LNgen [2] tools, which automate the generation of some useful auxiliary lemmas from a description of the language grammar. In total, the Coq code has 4657 lines of code, of which 1423 are manually-written proofs, while the rest are either library code or automatically-generated by Ott and LNgen.

7 Related Work

The concept of blame comes from work on higher-order contracts by Findler and Felleisen [11]. The application of blame to gradual typing was pioneered by Tobin-Hochstadt and Felleisen [25], and Wadler and Findler [27]. We followed the latter closely when developing the operational semantics and safety proofs for $\lambda_{\texttt{null}}$. Our syntax for casts comes from Ahmed et al. [1]. Wadler [26] provides additional context on the use of blame for gradual typing.

The *gradual guarantee*, introduced by Siek et al. [23], is a property of gradually-typed languages that characterizes the behaviour of terms as type annotations are added or removed from a program. Roughly speaking, removing type annotations preserves program behaviour, while adding type annotations can lead only to certain classes of errors. In this way, languages that satisfy the gradual guarantee allow well-behaved migrations of untyped code into the typed world. Determining whether $\lambda^s_{\texttt{null}}$ satisfies a property analogous to the gradual guarantee remains future work.

Linking types [19] solve the related (and more general) problem of ensuring that typing guarantees that hold in one or more source languages (e.g. Java and Scala) continue to hold, after compilation, in a target language (e.g. JVM bytecode), even in the presence of linking. However, linking types require that the source languages be augmented with additional types (the linking types), and that the target language be sufficiently expressive. In the case of `null` interoperability for Java and Scala, for example, this would mean adding a notion of nullable types both to Java (the source language) and JVM bytecode (the target language). This makes the `null` interoperability problem trivial, but would require considerable additional effort, when compared to our approach.

[8] The notation t **safe for** L, where L is a set of blame labels, indicates that t **safe for** l for every $l \in L$.

Multiple modern programming languages have types that are non-nullable by default. Examples include Kotlin [16], Swift [13], C# [6], and (recently) Scala [8]. In all of these, it is possible to recover nullability at the type level. For example, in Kotlin the type `String` is non-nullable, but `String?` is nullable. In Scala, nullability is expressed as a special case of *type unions*: `String|Null` represents nullable strings. Additionally, all of these languages also need to support some form of interoperability with a "less-precisely typed" language, where nullability remains implicit and is not tracked in the types. In the Kotlin and Scala case, the less-precisely typed language is Java; for Swift, it is Objective-C; and for C#, it is any language that compiles to the .NET runtime.

All of the languages above make pragmatic design decisions in their `null` interoperability. Specifically, their versions of type nullification trade off soundness for usability. For example, in Kotlin, a `String` type flowing from Java is translated as the *platform type* [14] `String!`, as opposed to `String?`. Platform types allow different kinds of unsound, yet convenient, behaviour. For example, we can select fields and methods on a platform type, or assign a platform type to the corresponding non-nullable type (e.g. assign a `String!` to a `String`). Naturally, these unsafe operations might fail at runtime. Similarly to platform types in Kotlin, Swift has *implicitly unwrapped optionals* and Scala has an `UncheckedNull` type (which has fewer soundness holes, but does not help as much with usability).

The design of $\lambda^s_{\texttt{null}}$ was inspired by `null` interoperability in Scala and Kotlin. The main difference is that type nullification is "sound" in $\lambda^s_{\texttt{null}}$: that is, the unsafe nullable type $!(S \to T)$ is translated into the *safe* nullable type $?(S \to T)$. However, as we have seen, nullability errors remain, which motivates the use of blame to assign responsibility.

8 Conclusions

In this paper, we looked at the problem of characterizing the nullability errors that occur from two interoperating languages: one with explicit `null`s, the other with implicit `null`s. We 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, we introduced $\lambda_{\texttt{null}}$, a calculus where terms can be explicitly nullable or implicitly nullable. We 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, we used $\lambda_{\texttt{null}}$ as the basis for a higher-level calculus, $\lambda^s_{\texttt{null}}$, which more closely models language interoperability. Our main result is a theorem that says that *explicitly nullable programs can't be blamed* for null interoperability errors in $\lambda^s_{\texttt{null}}$.

References

1 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.

2 Brian Aydemir and Stephanie Weirich. LNgen: Tool support for locally nameless representations. Technical Report MS-CIS-10-24, Computer and Information Science, University of Pennsylvania, 2010.

3 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.

4 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.

5 Arthur Charguéraud. The locally nameless representation. *Journal of automated reasoning*, 49(3):363–408, 2012.

6 Microsoft Corporation. Nullable reference types. [Online; accessed 5-November-2019]. URL: `https://docs.microsoft.com/en-us/dotnet/csharp/nullable-references`.

7 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.

8 Dotty Team. Explicit nulls. [Online; accessed 9-January-2020]. URL: `https://dotty.epfl.ch/docs/reference/other-new-features/explicit-nulls.html`.

9 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.

10 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.

11 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.

12 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.

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

14 JetBrains. Calling Java code from Kotlin. [Online; accessed 9-January-2020]. URL: `https://kotlinlang.org/docs/reference/java-interop.html`.

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

16 JetBrains. Null safety. [Online; accessed 5-November-2019]. URL: `https://kotlinlang.org/docs/reference/null-safety.html`.

17 Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

18 Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

19 Daniel Patterson and Amal Ahmed. Linking types for multi-language software: Have your cake and eat it too. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, volume 71 of *LIPIcs*, pages 12:1–12:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

20 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.

21 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.

22 Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

23 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, volume 32 of *LIPIcs*, pages 274–293. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

24 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.

25 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *OOPSLA Companion*, pages 964–974. ACM, 2006.

26 Philip Wadler. A complement to blame. In Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, volume 32 of *LIPIcs*, pages 309–320. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

27 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.

28 Niklaus Wirth and C. A. R. Hoare. A contribution to the development of ALGOL. *Commun. ACM*, 9(6):413–432, 1966.

29 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.

30 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.