A Simpler Blame Calculus for Explicit Nulls

ANONYMOUS AUTHOR(S)

New programming languages that track the possibility of null references explicitly in their type system often interact with older languages that do not. Previous work defined complicated calculi that used gradual typing to formally reason about such interactions and assign blame for any cast failures to the less precisely typed language. We define a pair of considerably simpler languages that more closely follow standard formulations of the blame calculus while enjoying the same blame assignment properties as previous work. Our system, mechanized in Coq, can serve as a simpler, more canonical foundation for formal models of the interaction between languages with explicit and implicit nullability.

1 INTRODUCTION

Null pointers are infamous for causing software errors. **?** characterised them as "The Billion Dollar Bug". TODO: Find citation. The only one I know of is https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/ This is a presentation, not a paper.

One way to tame the danger of nulls is via types. Whereas older languages, such as Pascal and Java, permit nulls at any reference type, more recent designs, including Kotlin, Scala, C#, and Swift, adopt type systems that track whether a reference may be null. How do we permit code in older and newer languages to interact while preserving the type guarantees of the newer languages?

Gradual typing provides a sound theoretical basis for answering such questions, where a legacy language with a less precise type system (such as Java) interacts with a newer language with a more precise type system (such as Kotlin or Scala). Important early systems include those by Findler and Felleisen [2002] and Siek and Taha [2006]. They introduce casts to model monitoring the barrier between the two languages. Each cast checks at runtime whether values passed from the less-precisely typed language violate guarantees expected by the more-precisely typed language.

A key innovation, introduced by Findler and Felleisen [2002] is that when a cast fails blame is attributed to either the source or the target of the cast. **?** [TODO: check citation!] TODO: Perhaps Tobin-Hochstadt and Felleisen (2006) or Matthews and Findler (2007). https://dl.acm.org/doi/10.1145/1176617.1176755 https://dl.acm.org/doi/10.1145/1190215.1190220 exploit this innovation to prove that when a cast fails, blame always lies with the less-precisely typed side of the cast. Though the fact is obvious, their proof is not, depending on observational equivalence. Wadler and Findler [2009] introduced the *blame calculus* as an abstraction of the earlier models, and offered a simpler proof of the obvious fact based on a simple syntactic notion of *blame safety* and a straightforward proof based on progress and preservation.

Nieto et al. [2020a] applied gradual typing and blame to the case of type systems that track null references. Their $\lambda_{\texttt{null}}$ calculus supports three function types:

- $\#(S \to T)$ is a non-nullable function, corresponding to a non-nullable type such as String in Scala or Kotlin. Values of this type cannot be null. (There are technical exceptions where the value can be null, as explained in that paper.)
- $?(S \to T)$ is a safe nullable function, corresponding to a nullable type such as String|Null in Scala or String? in Kotlin. Values of this type can be null, and the type system ensures nulls are properly handled.
- $!(S \to T)$ is an unsafe nullable function, corresponding to a type such as String in Java. Values of this type can be null, but the type system guarantees nothing about proper handling of such nulls.

PL'18, January 01–03, 2018, New York, NY, USA
2018.

Their system also supports two forms of application, normal application $s\ t$ and safe application $app(s, t, u)$. Both apply s to t when the function is not null, but when the function is null the former gets stuck while the latter returns u. The two forms of application align with the three function types as follows. Consider the type of a function term s.

- $\#(S \to T)$ can be applied using standard application $s\ t$.
- $?(S \to T)$ can be applied using safe application $app(s, t, u)$.
- $!(S \to T)$ can be applied using either standard application $s\ t$ or safe application $app(s, t, u)$.

Casts may be used to convert the types of terms, and in particular to convert functions between these various types. At runtime, if a cast attempts to convert null from one of the latter two types to the first type the cast will fail, assigning blame appropriately to one side or the other of the cast.

On top of $\lambda_{\texttt{null}}$, that paper also defines $\lambda^s_{\texttt{null}}$, a calculus representing two languages, one with nulls reflected explicitly in its types (like Scala or Kotlin) and one where nulls are implicitly permitted everywhere (like Java). The syntax of the two languages is mutually recursive with an import construct that makes it possible to embed a term of one of the languages within a term of the other, modeling that it is possible to call either language from the other. The typing rules require each such embedded term to be closed, so it cannot have free variables bound in the other language. Thus it is not possible to construct heterogeneous data structures, such as a closure that closes over bindings from the other language. [TODO: Check!] The semantics of $\lambda^s_{\texttt{null}}$ is defined by translation to $\lambda_{\texttt{null}}$, with import constructs translated to corresponding casts. The key result is that if any of these casts fails, the blame is always assigned to code from the less-precisely typed implicit language.

This paper reiterates the development of the earlier paper, but using a simpler system and one that is closer to the standard development of blame calculus.

- Instead of three variants of function types, our design is more orthogonal. There is a function type $A \to B$, and there is a nullable type $D?$, which adds nulls to an existing type D. Here A and B range over all types, while D is restricted to *definite* types that do not already admit nulls. (This syntax rules out potentially confusing types such as $D??$.) The values of type $D?$ are either null or of the form $[V]$, where V is a value of type D.
- Instead of two forms of application, one safe and one unsafe, our orthogonal system of types leads to a corresponding orthogonal system of terms, based on standard forms of application for functions and case analysis for nullable values.
- Instead of a high-level language $\lambda^s_{\texttt{null}}$ with explicit and implicit sublanguages that translates into a core language $\lambda_{\texttt{null}}$, we use a simpler framework. We define an *explicit* language that fills the roll of both $\lambda_{\texttt{null}}$ and the explicit half of $\lambda^s_{\texttt{null}}$ and we define an *implicit* language that is given a semantics by translation into the explicit language.
- Because the implicit language is given a semantics by translation into the explicit language, it is easy to assign a semantics to arbitrary nesting of explicit and implicit terms. There is no longer a requirement that nested terms be closed; free variables of a term in one language can be bound in the other language.
- The resulting development is simpler and more standard than the previous development. In particular, we adapt the Tangram Lemma of Wadler and Findler [2009] to prove that blame is always assigned to the less-precisely typed language. The previous development never mentioned the Tangram Lemma, relying instead on a more convoluted argument.

Thus, our system can serve as a simpler and more canonical foundation for formal models of the interaction between languages with explicit and implicit nulls.

The paper is organised as follows. Section 2 defines the explicit language. Section 3 proves its key properties: type safety, blame safety, and the Tangram Lemma. Section 4 defines the implicit language and its translation to the explicit language, and proves that the translation preserves types. Section 5 explores interoperability of the two languages: we show how terms of each language can be embedded in the other, define the casts needed to mediate between the two, and prove that any failure of these casts always blames the implicit language. Section 6 surveys related work. Section 7 concludes.

We have formalized all of our lemmas and theorems in Coq. We will submit our Coq formalization to the OOPSLA Artifact Evaluation process.

2 THE EXPLICIT LANGUAGE

In this section, we introduce a language that tracks the possibility of null references explicitly in types. We will call it the explicit language for short. We define the syntax, typing and subtyping rules, and a reduction relation. In Section 3, we will prove standard type safety and blame safety properties.

The syntax of the explicit language is shown in Figure 1. The basic values are constants c of a base type ι, function abstractions $\lambda x{:}A.N$ of a function type $A \to B$, and the null constant null. In addition to the two *definite types* ι and $A \to B$, the type system includes *nullable types* $D?$, where D is any definite type. The constructors of $D?$ are the null constant null and the lift operation $[N]$, where N is a term of type D. When V is a value, $[V]$ is also considered a value. A cast $V : A \to B \Longrightarrow^p A' \to B'$ of a function value V is also a value. When such a cast-function value is applied to an argument, the argument will first be cast from A' to A, then the function V will be applied to it, and finally the result will be cast from B to B'.

In addition to values, the calculus includes terms for function application $L\ M$, general casts $M : A \Longrightarrow^p B$, a pattern matching construct case L of $\{\mathsf{null} \mapsto M; [x] \mapsto N\}$ that destructs terms of nullable types $D?$, and a failure result blame p. As is standard in gradual type systems, each cast has a blame label p so that the result of a failing computation can be traced to the cast that failed. A blame label can be positive p, indicating that the term inside the cast caused the cast to fail, or negative $\overline{p}$, indicating that the context in which the cast appears caused the cast to fail.

The typing rules of the explicit language are shown in Figure 2. The rules for variables, base type constants and operations, and function abstraction and application are standard. The Null and Lift rules identify the null constant null and the lift operation $[M]$ as the constructors of a nullable type $D?$. The Blame rule specifies that a failure result blame p is possible at any type A. The Cast rule allows casts from type A to type B as long as A and B are *compatible*, written $A \sim B$. Informally, two types are compatible if they have the same structure, but differ only in the nullability of their components. Finally, the Case rule specifies that the case construct destructs terms of a nullable type $D?$.

The operational semantics of the explicit language is shown in Figure 3. The App rule is standard β-reduction. The Cast-App rule defines β-reduction for a function wrapped in a cast, ensuring that that the argument W and the final result of the function application are cast accordingly. There are four rules for reducing casts from a nullable type $D?$. A cast of null to another nullable type $E?$ reduces to just null (Cast-Null-Nullable). A cast of null to a non-nullable type E reduces to blame p (Cast-Null-NonNull). A cast of a lifted value $[V]$ from type $D?$ to a non-nullable type E evaluates to V wrapped in a cast from D to E (Cast-Lift-NonNull). When such a lifted value is cast to a *nullable* type $E?$, this result is additionally lifted: $[V : D \Longrightarrow^p E]$ (Cast-Lift-Nullable). A cast from a base type can only be back to the base type; it reduces to the value V inside the cast (Cast-Base). Two rules reduce the pattern-matching case construct. When the scrutinee is null, the case reduces to the term in the null branch (Case-Null). When the scrutinee is a lifted value $[V]$, the case reduces to the term in the non-null branch, with V substituted for the parameter x (Case-NonNull). A grammar of evaluation contexts $\mathcal{E}$ ensures call-by-value reduction in function

Labels and Variables

x	Variables
$p, q, \overline{p}, \overline{q}$	Blame Labels

Terms

$L, M, N ::= x$	Variable
$\mid c$	Base Constant
$\mid M \oplus N$	Base Operation
$\mid \lambda x{:}A.N$	Function Abstraction
$\mid L\ M$	Function Application
$\mid \mathsf{null}$	Null Constant
$\mid [M]$	Lift
$\mid \mathsf{case}\ L\ \mathsf{of}\ \{\mathsf{null} \mapsto M; [x] \mapsto N\}$	Case
$\mid M : A \Longrightarrow^p B$	Cast
$\mid \mathsf{blame}\ p$	Blame

Values

$V, W ::= c$	Base Constant
$\mid V \oplus W$	Base Operation on Values
$\mid \lambda x{:}A.N$	Function Abstraction
$\mid \mathsf{null}$	Null Constant
$\mid [V]$	Lift of a Value
$\mid V : A \to B \Longrightarrow^p A' \to B'$	Function-typed Cast of a Value

Types

$A, B, C ::= D$	Definite Type
$\mid D?$	Nullable Type
$D, E ::= \iota$	Base Type
$\mid A \to B$	Function Type

Fig. 1. Syntax of the explicit language

Typing

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash x : A} \quad \text{(Var)}$$

$$\Gamma \vdash c : \iota \quad \text{(Constant)}$$

$$\frac{\Gamma \vdash N : \iota \qquad \Gamma \vdash M : \iota}{\Gamma \vdash N \oplus M : \iota} \quad \text{(Binop)}$$

$$\frac{\Gamma, x{:}A \vdash N : B}{\Gamma \vdash \lambda x{:}A.N : A \to B} \quad \text{(Abs)}$$

$$\frac{\Gamma \vdash N : A \to B \qquad \Gamma \vdash M : A}{\Gamma \vdash N\ M : B} \quad \text{(App)}$$

$$\Gamma \vdash \mathsf{null} : D? \quad \text{(Null)}$$

$$\frac{\Gamma \vdash N : D}{\Gamma \vdash [N] : D?} \quad \text{(Lift)}$$

$$\frac{\Gamma \vdash N : D? \qquad \Gamma \vdash M : A \qquad \Gamma, x{:}D \vdash L : A}{\Gamma \vdash \mathsf{case}\ N\ \mathsf{of}\ \{\mathsf{null} \mapsto M; [x] \mapsto L\} : A} \quad \text{(Case)}$$

$$\frac{\Gamma \vdash M : A \qquad A \sim B}{\Gamma \vdash (M : A \Longrightarrow^{p} B) : B} \quad \text{(Cast)}$$

$$\Gamma \vdash \mathsf{blame}\ p : A \quad \text{(Blame)}$$

Compatibility

$$\iota \sim \iota \quad \text{(Compat-Base)}$$

$$\frac{A \sim D}{A \sim D?} \quad \text{(Compat-Null-R)}$$

$$\frac{D \sim A}{D? \sim A} \quad \text{(Compat-Null-L)}$$

$$\frac{A \sim A' \qquad B \sim B'}{A \to B \sim A' \to B'} \quad \text{(Compat-Arrow)}$$

Fig. 2. Typing rules of the explicit language

$$(\lambda x{:}A.N)\ V \longrightarrow N[x \mapsto V] \quad (\textsc{App})$$

$$(V : A \to B \Longrightarrow^{p} A' \to B')\ W \longrightarrow (V\ (W : A' \Longrightarrow^{\bar{p}} A)) : B \Longrightarrow^{p} B' \quad (\textsc{Cast-App})$$

$$\mathsf{null} : D? \Longrightarrow^{p} E? \longrightarrow \mathsf{null} \quad (\textsc{Cast-Null-Nullable})$$

$$[V] : D? \Longrightarrow^{p} E? \longrightarrow [V : D \Longrightarrow^{p} E] \quad (\textsc{Cast-Lift-Nullable})$$

$$\mathsf{null} : D? \Longrightarrow^{p} E \longrightarrow \mathsf{blame}\ p \quad (\textsc{Cast-Null-NonNull})$$

$$[V] : D? \Longrightarrow^{p} E \longrightarrow V : D \Longrightarrow^{p} E \quad (\textsc{Cast-Lift-NonNull})$$

$$V : \iota \Longrightarrow^{p} \iota \longrightarrow V \quad (\textsc{Cast-Base})$$

$$\mathsf{case\ null\ of}\ \{\mathsf{null} \mapsto M; [x] \mapsto L\} \longrightarrow M \quad (\textsc{Case-Null})$$

$$\mathsf{case}\ [V]\ \mathsf{of}\ \{\mathsf{null} \mapsto M; [x] \mapsto L\} \longrightarrow L[x \mapsto V] \quad (\textsc{Case-NonNull})$$

$$\frac{N \longrightarrow M}{\mathcal{E}[N] \longrightarrow \mathcal{E}[M]} \quad (\textsc{Ctx})$$

$$\mathcal{E}[\mathsf{blame}\ p] \longrightarrow \mathsf{blame}\ p \quad (\textsc{Err})$$

$$\begin{aligned}\mathcal{E} ::= \quad & [\,] \quad |\ \mathcal{E}\ N \quad |\ V\ \mathcal{E} \quad |\ \mathcal{E} : A \Longrightarrow^{p} B \quad |\ \mathsf{case}\ \mathcal{E}\ \mathsf{of}\ \{\mathsf{null} \mapsto M; [x] \mapsto L\} \\ & |\ [\mathcal{E}] \quad |\ \mathcal{E} \oplus M \quad |\ V \oplus \mathcal{E}\end{aligned}$$

Fig. 3. Reduction rules of the explicit language

applications, inside casts, pattern matches, lift operations, and base type operations. The Ctx rule specifies reduction inside an evaluation context. The Err rule specifies that a cast failure inside an evaluation context floats up to the top level, terminating the reduction sequence.

3 PROPERTIES OF THE EXPLICIT LANGUAGE

3.1 Type Safety

We have proven type safety of the explicit language following the syntactic approach of Wright and Felleisen [1994].

Theorem 3.1 (Preservation). *(Coq:* `preservation`*)*
If $\Gamma \vdash N : A$ *and* $N \longrightarrow M$*, then either* $M = \mathsf{blame}\ p$ *for some* p *or* $\Gamma \vdash M : A$*.*

Proof. The proof is by induction on the typing derivation. The App and Case-NonNull cases depend on a substitution lemma, shown below. The Cast-App case depends on symmetry of the compatibility relation $\sim$. □

Lemma 3.2 (Substitution). *(Coq:* `substitution`*)*
If $\Gamma, x : B \vdash N : A$ *and* $\Gamma \vdash M : B$*, then* $\Gamma \vdash N[x \mapsto M] : A$*.*

Proof. The proof is standard, by induction on the typing of N. The Var case depends on a weakening lemma, also proved by straightforward induction. □

Lemma 3.3 (Compatibility Symmetry). *(Coq:* `compat_sym`*) If $A \sim B$ then $B \sim A$.*

Proof. The proof is by straightforward induction on the derivation of $A \sim B$. □

Theorem 3.4 (Progress). *(Coq:* `progress`*)*
If $\vdash N : A$ then either N is a value, $N \longrightarrow M$ for some M, or $N = blame\ p$ for some p.

Proof. The proof is by induction on the typing derivation. The App case depends on a canonical forms lemma for function types, shown below. □

Lemma 3.5 (Canonical Forms Arrow). *(Coq:* `canonical_forms_arrow`*)*
If $\vdash V : A \to B$, then either $V = \lambda x{:}A.N$ for some x and N, or $V = W : A' \to B' \Longrightarrow^{p} A'' \to B''$ for some W, A', B', A'', B'', and p.

Proof. The proof is by straightforward induction on the typing derivation. □

3.2 Blame Safety

In addition to type safety, we have also proved blame safety following directly the approach of Wadler and Findler [2009] (see also Wadler [2015] for a more accessible summary of the approach). The applicability of this standard approach is one of the benefits of the explicit language relative to the calculus of Nieto et al. [2020a].

The approach depends on four subtyping relations, defined for the explicit language in Figure 4. Intuitively, a cast between types related by positive subtyping cannot give rise to positive blame (blame with the same label as the cast) and a cast between types related by negative subtyping cannot give rise to negative blame (blame with a label that is the complement of that on the cast). Ordinary subtyping is an intersection of these two relations, so a cast between types related by ordinary subtyping cannot give rise to any blame. We will discuss naive subtyping and its relationship to the other three subtyping relations in Section 3.3.

We make the intuitive understanding of positive and negative subtyping precise as follows:

Definition 3.6 (Safe Term). (Coq: `sfor`) A term N is *safe* for blame label p, written N safe p, if N has no subterm of the form blame p, every cast in N of the form $M : A \Longrightarrow^{p} B$ satisfies $A <:^{+} B$, and every cast in N of the form $M : A \Longrightarrow^{\overline{p}} B$ satisfies $A <:^{-} B$.

With this definition, we can prove blame safety of the explicit language, that when N safe p, N cannot reduce to blame p in any number of steps.

Theorem 3.7 (Safe Term Preservation). *(Coq:* `sfor_preservation`*)*
If $\Gamma \vdash N : A$, N safe p, and $N \longrightarrow M$, then M safe p.

Proof. The proof is by induction on the derivation of $N \longrightarrow M$. Each case is straightforward except in the App and Case-NonNull cases, we need the following lemma to show that the safety relation is preserved by substitution. □

Lemma 3.8 (Substitution Preserves Safe Terms). *(Coq:* `subst_pres_sfor`*)*
If N safe p and M safe p, then $N[x \mapsto M]$ safe p.

Proof. By straightforward induction on the structure of N. □

Corollary 3.9 (Safe Term Progress). *(Coq:* `sfor_progress`*)*
If $\vdash N : A$, N safe p, and $N \longrightarrow$ blame q, then $p \neq q$.

Proof. This follows directly from Theorem 3.7 and the definition of safe. □

Subtyping

$$\iota <: \iota \qquad \text{(Base)}$$

$$\frac{D <: E}{D <: E?} \qquad \text{(Null-Sup)}$$

$$\frac{D <: E}{D? <: E?} \qquad \text{(Null)}$$

$$\frac{A' <: A \qquad B <: B'}{A \to B <: A' \to B'} \qquad \text{(Arrow)}$$

Naive Subtyping

$$\iota <:_n \iota \qquad \text{(Naive-Base)}$$

$$\frac{D <:_n E}{D <:_n E?} \qquad \text{(Naive-Null-Sup)}$$

$$\frac{D <:_n E}{D? <:_n E?} \qquad \text{(Naive-Null)}$$

$$\frac{A <:_n A' \qquad B <:_n B'}{A \to B <:_n A' \to B'} \qquad \text{(Naive-Arrow)}$$

Positive and Negative Subtyping

$$\iota <:^+ \iota \qquad \text{(Positive-Base)}$$

$$\frac{D <:^+ E}{D <:^+ E?} \qquad \text{(Positive-Null-Sup)}$$

$$\frac{D <:^+ E}{D? <:^+ E?} \qquad \text{(Positive-Null)}$$

$$\frac{A' <:^- A \qquad B <:^+ B'}{A \to B <:^+ A' \to B'} \qquad \text{(Positive-Arrow)}$$

$$\iota <:^- \iota \qquad \text{(Negative-Base)}$$

$$\frac{D <:^- E}{D <:^- E?} \qquad \text{(Negative-Null-Sup)}$$

$$\frac{D <:^- E}{D? <:^- E?} \qquad \text{(Negative-Null)}$$

$$\frac{A' <:^+ A \qquad B <:^- B'}{A \to B <:^- A' \to B'} \qquad \text{(Negative-Arrow)}$$

Fig. 4. Subtyping rules of the explicit language

Theorem 3.10 (Blame Safety). *(Coq:* `safety`*)*
If $\vdash N : A$*,* N *safe* p*, and* $N \longrightarrow^* \text{blame } q$*, then* $p \neq q$*.*

Proof. The proof is by induction on the transitive reduction relation. In the inductive case, it uses Theorems 3.1 and 3.7. □

3.3 Naive Subtyping and the Tangram Lemma

Naive subtyping relates types according to how *definite* they are in the sense of gradual typing. In our specific setting, A is a naive subtype of B if they have the same structure, but some non-nullable components D of A may be replaced by nullable components $D?$ in B, regardless of whether they occur covariantly or contravariantly. In a language with implicit nulls, the less definite type $D?$ appearing in a program might be intended to mean either a non-nullable type D or a nullable type $D?$; the distinction would be expressible using these more definite types in a language with explicit nulls.

The Tangram Lemma of Wadler and Findler [2009] relates these four subtyping relations. We show that it holds for the specific relations defined in Figure 4.

Theorem 3.11 (Tangram Lemma). *(Coq:* `tangram_fwd`*) (Coq:* `tangram_rev`*) (Coq:* `tangram_naive_fwd`*) (Coq:* `tangram_naive_rev`*)*

1. $A <: B$ *if and only if* $A <:^+ B$ *and* $A <:^- B$*.*
2. $A <:_n B$ *if and only if* $A <:^+ B$ *and* $B <:^- A$*.*

Proof. Each of the four cases is proved by a straightforward induction on the derivation of $A <: B$, the derivation of $A <:_n B$, or mutual induction on the derivations of $A <:^+ B$ and $A <:^- B$. □

4 THE IMPLICIT LANGUAGE

Having defined an explicit language with casts (like Scala), we now define an implicit language that ignores nullability in its types (like Java).

4.1 Syntax

The syntax of the implicit language is defined in Figure 5. In general, we use a h$\hat{\text{a}}$t to mark elements of the implicit language. The syntax of terms of the implicit language $\hat{L}, \hat{M}, \hat{N}$ mirrors that of the explicit language, but omits lifting, pattern matching, casts, and blame, since those are useless without the distinction between nullable and non-null types.

Types $\hat{A}, \hat{B}, \hat{C}$ of the implicit language are not distinguished as nullable or non-null. All types in the implicit language admit the null constant.

TODO: Consider changing the notation for types from the implicit language to write them *always* with question marks: $\iota?$ and $(A \rightarrow B)?$.

4.2 Typing

The typing rules of the implicit language are standard, and are shown in Figure 6. The $\widehat{\text{null}}$ constant can have any type $\hat{A}$.

4.3 Semantics

We define the semantics of the implicit language by translation to the explicit language, whose operational semantics we defined in Section 2. The translation is presented in Figure 7.

Implicit Terms

$\hat{L}, \hat{M}, \hat{N} ::=$	$\hat{x}$	Variable
	$\mid \hat{c}$	Base Constant
	$\mid \hat{M} \oplus \hat{N}$	Base Operation
	$\mid \hat{\mathsf{null}}$	Null Constant
	$\mid \lambda x{:}\hat{A}.\hat{N}$	Function Abstraction
	$\mid \hat{L}\ \hat{M}$	Function Application

Implicit Types

$\hat{A}, \hat{B}, \hat{C} ::=$	ι	Base Type
	$\mid \hat{A} \to \hat{B}$	Function Type

Fig. 5. Syntax of the language with implicit nulls

$$\frac{x{:}\hat{A} \in \Gamma}{\Gamma \vdash \hat{x} : \hat{A}} \qquad (\textsc{Imp-Var})$$

$$\Gamma \vdash \hat{c} : \iota \qquad (\textsc{Imp-Base})$$

$$\frac{\Gamma \vdash \hat{N} : \iota \qquad \Gamma \vdash \hat{M} : \iota}{\Gamma \vdash \hat{N} \oplus \hat{M} : \iota} \qquad (\textsc{Imp-Binop})$$

$$\Gamma \vdash \hat{\mathsf{null}} : \hat{A} \qquad (\textsc{Imp-Null})$$

$$\frac{\Gamma, x{:}\hat{A} \vdash \hat{N} : \hat{B}}{\Gamma \vdash \lambda x{:}\hat{A}.\hat{N} : \hat{A} \to \hat{B}} \qquad (\textsc{Imp-Abs})$$

$$\frac{\Gamma \vdash \hat{N} : \hat{A} \to \hat{B} \qquad \Gamma \vdash \hat{M} : \hat{A}}{\Gamma \vdash \hat{N}\ \hat{M} : \hat{B}} \qquad (\textsc{Imp-App})$$

Fig. 6. Typing rules for language with implicit nulls

Types of the implicit language are translated to types in the explicit language that are equivalent in that they admit equivalent sets of values: in particular, the translated types admit the null constant.

In the translation of terms of the implicit language, we will make frequent use of pattern matching. We introduce as shorthand an Elvis operator M ?: N that takes a term M of nullable type D? and reduces to N when M evaluates to null and to V when M evaluates to $[V]$.

Translation of Implict Types

$$|\iota| = \iota?$$
$$|\hat{A} \to \hat{B}| = (|\hat{A}| \to |\hat{B}|)?$$

Elvis Operator Syntactic Sugar

$$M \;?\!:\; N = \mathsf{case}\ M\ \mathsf{of}\ \{\mathsf{null} \mapsto N; [x] \mapsto x\}$$

Translation of Implict Terms

$$\begin{aligned}
|\hat{x}| &= x \\
|\hat{\mathsf{null}}| &= \mathsf{null} \\
|\hat{c}| &= [c] \\
|\hat{N} \oplus \hat{M}| &= [(|\hat{N}| \;?\!:\; \mathsf{blame}\ op) \oplus (|\hat{M}| \;?\!:\; \mathsf{blame}\ op)] \\
|\lambda x{:}\hat{A}.\hat{N}| &= [\lambda x{:}|\hat{A}|.|\hat{N}|] \\
|\hat{N}\ \hat{M}| &= (|\hat{N}| \;?\!:\; \mathsf{blame}\ deref)\ |\hat{M}|
\end{aligned}$$

Fig. 7. Translation from implicit language to explicit language

Variable references and the null constant are just translated to themselves. Base constants $\hat{c}$, base operations $\hat{N} \oplus \hat{M}$, and function abstractions $\lambda x{:}\hat{A}.\hat{M}$ have types in the implicit language that translate to nullable types in the explicit language, so these terms are translated to lifted terms in the explicit language. The translation of a base operation $\oplus$ uses the Elvis operator to check the whether the operands $\hat{N}$ and $\hat{M}$ are null before performing the operation $\oplus$. The blame label *op* is used to signal that a null check in a base operation failed. Similarly, the translation of a function application $\hat{N}\ \hat{M}$ first checks whether $\hat{N}$ (the function) evaluates to null before evaluating the argument $\hat{M}$ and performing the application.

4.4 Type Preservation of the Translation

The translation from the implicit language to the explicit language preserves typing:

THEOREM 4.1 (TRANSLATION PRESERVES TYPING). *(Coq:* `desugaring_typing`*)*
If $\Gamma \vdash \hat{N} : \hat{A}$*, then* $|\Gamma| \vdash |\hat{N}| : |\hat{A}|$*, where a typing context* $|\Gamma|$ *is obtained by replacing each binding of the form* $x : \hat{A}$ *in* Γ *with* $x : |\hat{A}|$*.*

PROOF. The proof is by induction on the derivation of $\Gamma \vdash \hat{N} : \hat{A}$. The IMP-ABS rule is parameterized by an arbitrary fresh variable x, so the case for this rule requires to prove a conclusion that holds for *any* such fresh x. To do so, we need to show that the translation function $|\cdot|$ commutes with α-renaming of variables. In fact, we prove a stronger result, that this function commutes with substitution of an arbitrary term for a free variable, which we show below in Lemma 4.2. All other cases are straightforward. □

LEMMA 4.2 (SUBSTITUTION COMMUTES WITH TRANSLATION). *(Coq:* `open_trm_of_itrm`*)*

$$|\hat{N}[\hat{x} \mapsto \hat{M}]| = |\hat{N}|[x \mapsto |\hat{M}|]$$

Proof. The proof is by straightforward induction on the structure of $\hat{N}$. □

5 INTEROPERABILITY

In this section, we will explore how terms of the explicit language can use terms of the implicit language and vice versa.

5.1 Implicit Terms within Explicit Terms

To use a term $\hat{M}$ of the implicit language within the explicit language, we just translate the term first, and use $|\hat{M}|$ within the explicit language. However, the translated term has an inconvenient type, so it cannot be used directly. For example, the translated implicit language constant term $|\hat{c}|$ has the nullable type $\iota?$, so it cannot be an operand of the explicit language $\oplus$ operator, which requires operands of type ι. Similarly, the translated implicit language function term $|\lambda x{:}\hat{A}.\hat{N}|$ has the nullable type $(|\hat{A}| \to |\hat{B}|)?$ (where $\hat{B}$ is a type of the body $\hat{N}$), so it cannot be used directly in a function application.

One safe solution is to explicitly handle the possibility of an implicit language subterm evaluating to null using the case construct. In case $|\hat{c}|$ of $\{\mathsf{null} \mapsto N; [x] \mapsto M\}$, the constant $\hat{c}$ is available in M through the variable x with the convenient type ι. More complicated types require additional pattern matching. For example, a pattern match can extract a term of function type $|\hat{A}| \to |\hat{B}|$ from one of nullable type $(|\hat{A}| \to |\hat{B}|)?$, but additional lifts and pattern matching are required to deal with the remaining nullable types $|\hat{A}|$ and $|\hat{B}|$. Although inconvenient, this approach is safe: since it uses only pattern matches but no casts, there are no casts that could fail.

If we are confident that the implicit language subterm will not evaluate to null, a more convenient approach is to cast it to a more suitable type. We define a *naive* translation of implicit types as follows:

$$\lfloor \iota \rfloor = \iota$$
$$\lfloor \hat{A} \to \hat{B} \rfloor = \lfloor \hat{A} \rfloor \to \lfloor \hat{B} \rfloor$$

The naive translation maps a base type of the implicit language to a base type of the core language and it maps a function type of the implicit language to a function type of the core language. Thus, given a term $\hat{N}$ of type $\hat{A}$ in the implicit language, we can use it directly in the explicit language if we embed it using the following cast:

$$|\hat{N}| : |\hat{A}| \Longrightarrow^{ie} \lfloor \hat{A} \rfloor$$

Here, the blame label *ie* stands for a cast from the implicit language to the explicit language.

This is convenient but unsafe, since the cast could fail. However, the cast can fail only with positive blame, blaming the implicit language subterm $|\hat{N}|$ rather than the surrounding explicit language context. This is because $\lfloor \hat{A} \rfloor$ is a naive subtype of $|\hat{A}|$:

Theorem 5.1 (Implicit Naive Subtyping). *(Coq:* `imp_naive_subtyp`*)* *For every implicit type $\hat{A}$, $\lfloor \hat{A} \rfloor <:_n |\hat{A}|$.*

Proof. The proof is by straightforward induction on the structure of $\hat{A}$. □

Then by the Tangram Lemma, $|\hat{A}| <:^- \lfloor \hat{A} \rfloor$, so a term containing this form of cast is safe for $\overline{ie}$, so it cannot reduce to blame $\overline{ie}$.

5.2 Explicit Terms within Implicit Terms

It is also possible to embed an explicit language term within an implicit language term by applying the translation to the surrounding implicit term. The general pattern is let $x : A = M$ in $|\hat{N}|$, which can be desugared as $(\lambda x{:}A.|\hat{N}|)\ M$.

However, we must again adapt the types. Typing the implicit language term $\hat{N}$ requires a typing context that binds x to some implicit language type $\hat{B}$. By Theorem 4.1, the translated term $|\hat{N}|$ can be typed in a translated context $|\Gamma|$ that binds x to $|\hat{B}|$. Thus, the overall lambda abstraction $(\lambda x{:}A.|\hat{N}|)\ M$ can be typed as long as there is some $\hat{B}$ such that $A = |\hat{B}|$, in other words, as long as the explicit language type A of M is the image of some implicit type $\hat{B}$ under the translation.

For an explicit term M of base type ι, this is easy to achieve using just lifting, since $[M]$ has type $\iota? = |\iota|$. For an explicit term of function type, however, ensuring that its type is the image of some implicit language type requires adjusting its domain and codomain types. Specifically, if the domain type of an explicit language function is a definite (non-null) type, pattern matching needs to be added before applying the function to handle the case that the actual argument from the implicit language could be null. Although it is possible to use a term of the explicit language in the implicit language without adding any casts, it may require adding multiple lifts and pattern matching, and is thus inconvenient.

If we are confident about lack of null references, we can again use a cast to conveniently allow an explicit language term of any type to be embedded in the implicit language. To do so, we first define the *erasure* of an explicit language type to be the implicit language type determined as follows:

$$\lceil D? \rceil = \lceil D \rceil$$
$$\lceil \iota \rceil = \iota$$
$$\lceil A \to B \rceil = \lceil A \rceil \to \lceil B \rceil$$

Then for any explicit type A, the translation of its erasure $|\lceil A \rceil|$ is the image of an implicit language type. Therefore, any explicit term M of any explicit type A can be embedded in an implicit language term $\hat{N}$ using the following pattern:

$$(\lambda x{:}|\lceil A \rceil|.|\hat{N}|)\ (M : A \Longrightarrow^{ei} |\lceil A \rceil|)$$

Here, the blame label *ei* stands for a cast from the explicit language to the implicit language.

Again, this is convenient but unsafe, since the cast may fail. However, the cast may fail only with *negative* blame, placing the blame on the implicit language context. For example, if the type A is a function type with a non-null domain type, the cast could fail if the surrounding implicit context invokes the function with a null argument. The cast cannot fail with positive blame because A is a naive subtype of $|\lceil A \rceil|$:

Theorem 5.2 (Explicit Naive Subtyping). *(Coq:* `exp_naive_subtyp`*)* *For every explicit type A, $A <:_n |\lceil A \rceil|$.*

Proof. The proof is by straightforward induction on the structure of A. □

Then by the Tangram Lemma, $A <:^+ |\lceil A \rceil|$, so a term containing this form of cast is safe for *ei*, so it cannot reduce to blame *ei*.

6 RELATED WORK

Siek and Taha [2006, 2007] introduced the concept of *gradual typing* to enable interoperability between parts of a program with and without static types. Findler and Felleisen [2002] introduced the concept of *blame* to function contracts, allowing to assign responsibility for a runtime failure either to a function itself or to the arguments passed to the function. Wadler [2015]; Wadler and

Findler [2009] combined the two concepts and proved that in a gradually typed program, any cast failure on the boundary can always be blamed on the untyped (or, more generally, the less-precisely typed) part of the program. They generalized their result in the Tangram Lemma, which can be instantiated for other gradually typed calculi. Garcia et al. [2016] formalized the notion of *precision* of a gradual type in the framework of abstract interpretation [Cousot and Cousot 1977], formally defining which part of a program is less-precisely typed and can therefore be blamed.

Nieto et al. [2020a] instantiated the concepts of gradual typing and blame for their explicit-null extension of the Scala language [Nieto et al. 2020b]. There, the less-precisely typed parts of a program are those written in Java or older versions of Scala, and the more-precisely typed parts are those written in the new version of Scala in which the possibility of a reference being null is made explicit in its type. Similar issues occur in other languages that make nulls explicit in their type system but interoperate with older code in type systems agnostic to null. The Kotlin language [JetBrains 2022] aims for null safety within Kotlin code, but adapts Java types to avoid any compile-time errors related to nullability at the boundary between code written in Kotlin and Java. It uses a concept called *platform types*, which are a subtype of a non-null type but a supertype of a nullable type, to avoid reporting errors in both covariant and contravariant contexts. Recent versions of the C# language [Microsoft 2022] have nullable types that indicate that a reference can be null. Types in code written in older versions of the language are interpreted to mean that references are non-null. To enable interoperability, conversions from a nullable to a non-null type and vice versa are allowed, but generate a compile-time warning in areas of code designated to issue such warnings. The Swift language [Apple 2022] has *optionals* similar to discriminated options like Scala's Option and Haskell's Maybe, and *implicitly unwrapped optionals* which are automatically cast to a non-null type in contexts that require one. When Swift code interoperates with code in Objective-C, which does not make nullability explicit in its types, Objective-C expressions are given an implicitly unwrapped optional type in Swift.

7 CONCLUSION

We have defined a pair of core calculi for modelling interoperability between languages that track null references explicitly in their type systems and ones that do not. Our definitions follow the standard blame calculus of Wadler and Findler [2009]; in particular, their Tangram Lemma approach can be used to assign blame for cast failures to the less precise language whose type system ignores nullability. These core calculi can serve as a basis for modelling nullness interoperability in larger languages. Our development is formalized in Coq, and we will submit the formalization to the OOPSLA Artifact Evaluation process.

REFERENCES

Apple. 2022. *The Switft Programming Language*. https://docs.swift.org/swift-book/ (accessed 17 March 2022).

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, Mitchell Wand and Simon L. Peyton Jones (Eds.). ACM, 48–59.

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 429–442. https://doi.org/10.1145/2837614.2837670

JetBrains. 2022. *Kotlin Programming Language*. https://kotlinlang.org/ (accessed 17 March 2022).

Microsoft. 2022. *C# Language Specification*. https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification (accessed 17 March 2022).

Abel Nieto, Marianna Rapoport, Gregor Richards, and Ondřej Lhoták. 2020a. Blame for Null. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:28. https://doi.org/10.4230/LIPIcs.ECOOP.2020.3

Abel Nieto, Yaoyu Zhao, Ondrej Lhoták, Angela Chang, and Justin Pu. 2020b. Scala with Explicit Nulls. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 25:1–25:26. https://doi.org/10.4230/LIPIcs.ECOOP.2020.25

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

Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4609)*, Erik Ernst (Ed.). Springer, 2–27. https://doi.org/10.1007/978-3-540-73589-2_2

Philip Wadler. 2015. A Complement to Blame. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA (LIPIcs, Vol. 32)*, Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 309–320.

Philip Wadler and Robert Bruce Findler. 2009. Well-typed programs can't be blamed. In *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 (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 1–16.

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94.