JFP **34**, e15, 26 pages, 2025.

doi:10.1017/S0956796824000121

A simple blame calculus for explicit nulls

ONDŘEJ LHOTÁK
University of Waterloo, Waterloo, Canada
(*e-mail:* olhotak@uwaterloo.ca)

PHILIP WADLER
School of Informatics, University of Edinburgh, Edinburgh, UK
(*e-mail:* wadler@inf.ed.ac.uk)

Abstract

Gradual typing provides a model for when a legacy language with less precise types interacts with a newer language with more precise types. Casts mediate between types of different precision, allocating blame when a value fails to conform to a type. The blame theorem asserts that blame always falls on the less-precisely typed side of a cast. One instance of interest is when a legacy language (such as Java) permits null values at every type, while a newer language (such as Scala or Kotlin) explicitly indicates which types permit null values. Nieto et al. in 2020 introduced a gradually typed calculus for just this purpose. The calculus requires three distinct constructors for function types and a non-standard proof of the blame theorem; it can embed terms from the legacy language into the newer language (or vice versa) only when they are closed. Here, we define a simpler calculus that is more orthogonal, with one constructor for function types and one for possibly nullable types, and with an entirely standard proof of the blame theorem; it can embed terms from the legacy language into the newer language (and vice versa) even if they are open. All results in the paper have been mechanized in Coq.

1 Introduction

Null pointers are infamous for causing software errors. Hoare (2009) characterized them as "The Billion Dollar Mistake."

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. (In Scala, the type system with explicit nulls (Nieto et al., 2020*b*) is available in versions 3.0.0 and later and is enabled is enabled by the compiler flag `-Yexplicit-nulls`.) 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 Siek and Taha (2006) and Matthews and Findler (2007). They

introduce casts to model monitoring the barrier between the two languages. A cast is introduced at every place where the boundary is crossed. 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. Tobin-Hochstadt and Felleisen (2006) and Matthews and Findler (2007) 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. (2020*a*) applied gradual typing and blame to the case of type systems that track null references. Their $\lambda_{\texttt{null}}$ calculus supports three function types, which vary in the guarantees they provide about whether a variable or field of that type could hold the null value instead of a function value:

- $\#(S \to T)$ is a non-nullable function type, corresponding to a non-nullable object reference 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 type, corresponding to a nullable object reference 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 type, corresponding to an object reference 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.

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 where nulls are implicitly permitted everywhere (like Java) and one with nulls reflected explicitly in its types (like Scala or Kotlin). 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

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

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, a closure in one language cannot close over bindings from the other language. The semantics of $\lambda^{s}_{\mathtt{null}}$ is defined by translation to $\lambda_{\mathtt{null}}$, 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, where nulls are implicitly permitted everywhere.

The metatheory of Nieto et al. (2020*a*) was mechanized in Coq in an accompanying artifact (Nieto et al., 2020*c*). The Coq mechanization closely follows the development as presented in that paper.

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 $\mathsf{A} \to \mathsf{B}$, and there is a nullable type $\mathsf{D?}$, 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 $\mathsf{D??}$.) The values of type $\mathsf{D?}$ are either null or of the form $\langle \mathsf{V} \rangle$, where V is a value of type D. The angle brackets designate lifting from type D to $\mathsf{D?}$.
- 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 ($\mathsf{L\ M}$) and case analysis for nullable values ($\mathsf{case\ L\ of\ \{null \mapsto M;\ \langle x \rangle \mapsto N\}}$).
- Instead of a high-level language $\lambda^{s}_{\mathtt{null}}$ with explicit and implicit sublanguages that translates into a core language $\lambda_{\mathtt{null}}$, we define an *explicit* language EN that fulfills the role of both $\lambda_{\mathtt{null}}$ and the explicit half of $\lambda^{s}_{\mathtt{null}}$ and we define an *implicit* language *IN* that fulfills the role of the implicit half of $\lambda^{s}_{\mathtt{null}}$.
- The implicit language *IN* is given a semantics by translation into the explicit language EN, making it 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 did not use 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.

Our development extends the simply typed lambda calculus with only those features that are motivated by adding nulls. Thus, it is an initial foundation to which other language features can be added in future work to study any hypothetical feature interactions. While our theoretical results can guide the design of full programming languages, a caveat of all work on core language calculi is that it is possible for larger language to violate properties established on a core calculus. For example, although Featherweight Java was proven sound (Igarashi et al., 2001), the presence of null values in the full Java language was found to undermine that soundness (Amin and Tate, 2016). There is value both in more

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

complete semantics, in which feature interactions can be explored, and in featherweight calculi, which elucidate the essence of a feature. The value of featherweight calculi was discussed further by Griesemer et al. (2020), who as evidence compare citations counts for the paper on Featherweight Java, Igarashi et al. (2001), with the four most-cited papers on more complete models, Flatt et al. (1998), Nipkow and von Oheimb (1998), Drossopoulou and Eisenbach (1997), and Syme (1999): 1226 as compared with 592, 270, 188, and 172, respectively (Google Scholar, May 2024).

This paper is organized as follows. Section 2 defines the explicit language EN. Section 3 proves its key properties: type safety, blame safety, and the Tangram Lemma. Section 4 defines the implicit language *IN* and its translation to EN 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 *IN*. Section 6 further compares our calculi to those of Nieto et al. (2020*a*) and discusses connections to full languages such as Scala and Java. Section 7 surveys related work. Section 8 concludes.

We have mechanized all of our lemmas and propositions in Coq and included the mechanization with the paper as additional material. The mechanization follows the locally nameless approach of Aydemir et al. (2008) and uses the Ott (Sewell et al., 2010) and LNGen (Aydemir and Weirich, 2010) tools to automatically generate the associated Coq infrastructure and lemmas from a file of definitions that directly follow those in the paper. Aside from the use of a locally nameless representation, the mechanization directly follows the development as it is presented in the paper.

2 The explicit language EN

In this section, we introduce a language that tracks the possibility of null references explicitly in types. We call it EN for short. Following the advice of Patrignani (2021), we use a blue color and an upright sans-serif font for elements of EN to distinguish them from those of another language that we will introduce in Section 4. In this section, we define syntax and typing rules, a reduction relation, and four blame subtyping relations for EN. In Section 3, we prove standard type safety and blame safety properties.

The syntax of EN is shown in Figure 1. The basic values are constants c of a base type ι, function abstractions $\lambda \mathsf{x}{:}\mathsf{A}.\mathsf{N}$ of a function type $\mathsf{A} \to \mathsf{B}$, and the null constant null. In addition to the two *definite types* ι and $\mathsf{A} \to \mathsf{B}$, the type system includes *nullable types* $\mathsf{D?}$, where D is any definite type. The constructors of $\mathsf{D?}$ are the null constant null and the lift operation $\langle \mathsf{M} \rangle$, where M is a term of type D. When V is a value, $\langle \mathsf{V} \rangle$ is also considered a value, and when V is a function value, the cast $\mathsf{V} : \mathsf{A} \to \mathsf{B} \Longrightarrow^{\mathsf{p}} \mathsf{A}' \to \mathsf{B}'$ is also a function value.

In addition to values, the calculus includes terms for function application $\mathsf{L}\ \mathsf{M}$, casts $\mathsf{M} : \mathsf{A} \Longrightarrow^{\mathsf{p}} \mathsf{B}$, a case construct $\mathsf{case}\ \mathsf{L}\ \mathsf{of}\ \{\mathsf{null} \mapsto \mathsf{M};\ \langle \mathsf{x} \rangle \mapsto \mathsf{N}\}$ that destructs terms of nullable types $\mathsf{D?}$, and a failure result $\mathsf{blame\ p}$. 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 cast with blame label p may fail at run time, yielding either $\mathsf{blame\ p}$ or $\mathsf{blame}\ \overline{\mathsf{p}}$. Here, $\mathsf{blame\ p}$ is *positive blame* and indicates that fault lies with the term contained

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

Labels and Variables

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

Terms

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

Values

$V, W ::=$	c	Base Constant
	$\lambda x{:}A.N$	Function Abstraction
	null	Null Constant
	$\langle V \rangle$	Lift of a Value
	$V : A \to B \Longrightarrow^{p} A' \to B'$	Function-typed Cast of a Value

Types

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

Fig. 1. EN syntax.

in the cast, while $\mathsf{blame}\ \overline{p}$ is *negative blame* and indicates that fault lies with the context containing the cast. The overbar is an involutive operator on blame labels: $\overline{\overline{p}} = p$.

As an example of positive blame, consider the term:

$$((\lambda x{:}\iota.\mathsf{null}) : (\iota \to \iota?) \Longrightarrow^{p} (\iota? \to \iota))\ \langle c \rangle$$

During reduction, the function will be applied to a constant c of type ι, yielding null, causing a cast failure because null is not a value of the definite type ι given as the return type of the function in the target type $(\iota? \to \iota)$ of the cast. The fault lies with the function within the

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

cast, which returned null although the cast promised that it would return a non-null value. This example term evaluates to positive blame p.

As an example of negative blame, consider the similar term:

$$((\lambda x{:}\iota.\mathsf{null}) : (\iota \to \iota?) \Longrightarrow^{\mathsf{p}} (\iota? \to \iota))\ \mathsf{null}$$

During reduction, the function will be applied to null, causing a cast failure because null is not a value of the definite type ι specified as the parameter type of the function. The fault lies with the context containing the cast, namely the null value to which the cast term is being applied. This example term evaluates to negative blame $\overline{\mathsf{p}}$.

The typing rules of EN 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 $\langle \mathsf{M} \rangle$ as the constructors of a nullable type D?. The CASE rule specifies that case destructs terms of a nullable type D?. The CAST rule allows casts from type A to type B as long as A and B are *compatible*, written $\mathsf{A} \sim \mathsf{B}$. Informally, two types are compatible if they have the same structure but differ only in the nullability of their components. Finally, the BLAME rule specifies that a failure result blame p is possible at any type A.

The operational semantics of EN is shown in Figure 3. The BINOP rule reduces a base operation applied to values V and W to a base constant specified by an external base operation evaluation function $[\![\oplus]\!](\mathsf{V}, \mathsf{W})$ (which requires and returns only base constants, not nulls). The APP rule is standard β-reduction. Two rules reduce a case. When the scrutinee is null, the case reduces to the term in the null branch (CASE-NULL). When the scrutinee is a lifted value $\langle \mathsf{V} \rangle$, the case reduces to the term in the non-null branch, with V substituted for the parameter x (CASE-LIFT). The WRAP rule defines β-reduction for a function wrapped in a cast, ensuring that the argument W and the final result of the function application are cast accordingly. During reduction, 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′. 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). A cast of null to a non-nullable type E reduces to blame p (DOWNCAST-NULL). A cast of a lifted value $\langle \mathsf{V} \rangle$ from type D? to a non-nullable type E evaluates to V wrapped in a cast from D to E (DOWNCAST-LIFT). When such a lifted value is cast to a *nullable* type E?, this result is additionally lifted (CAST-LIFT). A cast from base type to itself is the identity (CAST-BASE). A grammar of evaluation contexts $\mathcal{E}$ ensures call-by-value reduction in function applications, inside casts, cases, lift operations, and base type operations. The CTX rule specifies reduction inside an evaluation context. The ERR rule specifies that a failure inside an evaluation context floats up to the top level, terminating the reduction sequence.

3 Properties of the explicit language EN

3.1 Type safety

We prove type safety of EN by proving preservation and progress (Wright and Felleisen, 1994). For each result, we specify the name of the corresponding result in the Coq mechanization accompanying the paper.

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

Typing

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

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

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

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

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

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

$$\frac{\Gamma \vdash M : D}{\Gamma \vdash \langle M \rangle : D?} \quad \text{(Lift)}$$

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

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

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

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. EN typing rules.

Proposition 1 (Preservation)**.** *(Coq: `preservation`)*
If $\Gamma \vdash M : A$ *and* $M \longrightarrow N$ *then* $\Gamma \vdash N : A$.

Proof The proof is by induction on the reduction relation. The App and Case-Lift cases depend on a substitution lemma, which is shown below. The Wrap case depends on symmetry of the compatibility relation $\sim$. ■

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

$$
\begin{array}{rcll}
V \oplus W & \longrightarrow & [\![\oplus]\!](V, W) & (\textsc{Binop}) \\
(\lambda x{:}A.N)\ V & \longrightarrow & N[x \mapsto V] & (\textsc{App}) \\
\mathsf{case\ null\ of}\ \{\mathsf{null} \mapsto M;\ \langle x \rangle \mapsto N\} & \longrightarrow & M & (\textsc{Case-Null}) \\
\mathsf{case}\ \langle V \rangle\ \mathsf{of}\ \{\mathsf{null} \mapsto M;\ \langle x \rangle \mapsto N\} & \longrightarrow & N[x \mapsto V] & (\textsc{Case-Lift}) \\
(V : A \to B \Longrightarrow^{p} A' \to B')\ W & \longrightarrow & (V\ (W : A' \Longrightarrow^{\bar{p}} A)) : B \Longrightarrow^{p} B' & (\textsc{Wrap}) \\
\mathsf{null} : D? \Longrightarrow^{p} E? & \longrightarrow & \mathsf{null} & (\textsc{Cast-Null}) \\
\langle V \rangle : D? \Longrightarrow^{p} E? & \longrightarrow & \langle V : D \Longrightarrow^{p} E \rangle & (\textsc{Cast-Lift}) \\
\mathsf{null} : D? \Longrightarrow^{p} E & \longrightarrow & \mathsf{blame\ p} & (\textsc{Downcast-Null}) \\
\langle V \rangle : D? \Longrightarrow^{p} E & \longrightarrow & V : D \Longrightarrow^{p} E & (\textsc{Downcast-Lift}) \\
V : D \Longrightarrow^{p} E? & \longrightarrow & \langle V : D \Longrightarrow^{p} E \rangle & (\textsc{Upcast}) \\
V : \iota \Longrightarrow^{p} \iota & \longrightarrow & V & (\textsc{Cast-Base}) \\
\dfrac{M \longrightarrow N}{\mathscr{E}[M] \longrightarrow \mathscr{E}[N]} & & & (\textsc{Ctx}) \\
\mathscr{E}[\mathsf{blame\ p}] & \longrightarrow & \mathsf{blame\ p} & (\textsc{Err})
\end{array}
$$

$$
\begin{array}{l}
\mathscr{E} ::= \ \Box \ \mid \mathscr{E}\ N \ \mid V\ \mathscr{E} \ \mid \mathscr{E} : A \Longrightarrow^{p} B \ \mid \mathsf{case}\ \mathscr{E}\ \mathsf{of}\ \{\mathsf{null} \mapsto M;\ \langle x \rangle \mapsto N\} \\
\qquad \mid \langle \mathscr{E} \rangle \ \mid \mathscr{E} \oplus N \ \mid V \oplus \mathscr{E}
\end{array}
$$

Fig. 3. EN reduction rules.

Lemma 2 (Substitution). *(Coq: `substitution`)*
If $\Gamma, x : A \vdash N : B$ *and* $\Gamma \vdash M : A$, *then* $\Gamma \vdash N[x \mapsto M] : B$.

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

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

Proposition 4 (Progress). *(Coq: `progress`)*
If $\vdash M : A$ *then either* M *is a value,* $M \longrightarrow N$ *for some* N, *or* $M = \mathit{blame}\ p$ *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, which is shown below. ■

Lemma 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 the approach of Wadler and Findler (2009) (see also Wadler (2015) for a more accessible summary of the

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

Subtyping

$$\iota <: \iota \quad \text{(BASE)}$$

$$\frac{D <: E}{D <: E?} \text{ (NULL-SUP)}$$

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

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

Naive Subtyping

$$\iota <:_n \iota \quad \text{(NAIVE-BASE)}$$

$$\frac{D <:_n E}{D <:_n E?} \text{ (NAIVE-NULL-SUP)}$$

$$\frac{D <:_n E}{D? <:_n E?} \text{ (NAIVE-NULL)}$$

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

Positive Subtyping

$$\iota <:^+ \iota \quad \text{(POS-BASE)}$$

$$\frac{D <:^+ E}{D <:^+ E?} \text{ (POS-NULL-SUP)}$$

$$\frac{D <:^+ E}{D? <:^+ E?} \text{ (POS-NULL)}$$

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

Negative Subtyping

$$\iota <:^- \iota \quad \text{(NEG-BASE)}$$

$$\frac{D <:^- E}{D? <:^- E} \text{ (NEG-NULL-SUB)}$$

$$\frac{D <:^- E}{D <:^- E?} \text{ (NEG-NULL-SUP)}$$

$$\frac{D <:^- E}{D? <:^- E?} \text{ (NEG-NULL)}$$

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

Fig. 4. EN subtyping rules.

approach). The applicability of this standard approach is one of the benefits of EN relative to the calculus of Nieto et al. (2020*a*).

The approach depends on four subtyping relations, defined for EN 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. Positive, negative, and ordinary subtyping on functions are contravariant in the domain and covariant in the range, while naive subtyping is covariant in both the domain and the range.

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

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

Safe Terms

$$\frac{}{x \text{ safe } p} \quad \text{(SAFE-VAR)}$$

$$\frac{}{c \text{ safe } p} \quad \text{(SAFE-CONSTANT)}$$

$$\frac{M \text{ safe } p \qquad N \text{ safe } p}{M \oplus N \text{ safe } p} \quad \text{(SAFE-BINOP)}$$

$$\frac{N \text{ safe } p}{\lambda x{:}A.N \text{ safe } p} \quad \text{(SAFE-ABS)}$$

$$\frac{L \text{ safe } p \qquad M \text{ safe } p}{L\ M \text{ safe } p} \quad \text{(SAFE-APP)}$$

$$\frac{}{\mathsf{null} \text{ safe } p} \quad \text{(SAFE-NULL)}$$

$$\frac{M \text{ safe } p}{\langle M \rangle \text{ safe } p} \quad \text{(SAFE-LIFT)}$$

$$\frac{L \text{ safe } p \qquad M \text{ safe } p \qquad N \text{ safe } p}{(\mathsf{case}\ L\ \mathsf{of}\ \{\mathsf{null} \mapsto M;\ \langle x \rangle \mapsto N\}) \text{ safe } p} \quad \text{(SAFE-CASE)}$$

$$\frac{A <:^{+} B \qquad M \text{ safe } p}{(M : A \Longrightarrow^{p} B) \text{ safe } p} \quad \text{(SAFE-CAST-POS)}$$

$$\frac{A <:^{-} B \qquad M \text{ safe } p}{(M : A \Longrightarrow^{\overline{p}} B) \text{ safe } p} \quad \text{(SAFE-CAST-NEG)}$$

$$\frac{p \neq q \qquad p \neq \overline{q} \qquad M \text{ safe } p}{(M : A \Longrightarrow^{q} B) \text{ safe } p} \quad \text{(SAFE-CAST-DIFF)}$$

$$\frac{p \neq q}{(\mathsf{blame}\ p) \text{ safe } q} \quad \text{(SAFE-BLAME-DIFF)}$$

Fig. 5. Definition of safe terms.

Definition 6 (Safe Term). *(Coq: `safe`) A term* M *is safe for blame label* p*, written* M *safe* p*, if* M *has no subterm of the form* $\mathsf{blame}\ p$*, every cast in* M *of the form* $N : A \Longrightarrow^{p} B$ *satisfies* $A <:^{+} B$*, and every cast in* M *of the form* $N : A \Longrightarrow^{\overline{p}} B$ *satisfies* $A <:^{-} B$*. An explicit inductive definition is given in Figure 5.*

With this definition, we can prove blame safety of EN, that when M safe p, M cannot reduce to $\mathsf{blame}\ p$ in any number of steps. Note, however, that M can reduce to $\mathsf{blame}\ q$ with a different label $q \neq p$, and, in particular, M can reduce to $\mathsf{blame}\ \overline{p}$.

Proposition 7 (Safe Term Preservation). *(Coq: `safe_preservation`)*
If $\Gamma \vdash M : A$ *and* M *safe* p *and* $M \longrightarrow N$*, then* N *safe* p*.*

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

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

Lemma 8 (Substitution Preserves Safe Terms)**.** *(Coq: `subst_pres_safe`)*
If M *safe* p *and* N *safe* p*, then* $N[x \mapsto M]$ *safe* p*.*

Proof By straightforward induction on the structure of N. ■

Corollary 9 (Safe Term Progress)**.** *(Coq: `safe_progress`)*
If $\vdash M : A$ *and* M *safe* p *and* $M \longrightarrow$ *blame* q*, then* $p \neq q$*.*

Proof This follows directly from Proposition 7 and the definition of safe. ■

Proposition 10 (Blame Safety)**.** *(Coq: `safety`)*
If $\vdash M : A$ *and* M *safe* p *and* $M \longrightarrow^* N$*, then* $N \neq$ *blame* p*.*

Proof The proof is by induction on the transitive reduction relation. In the inductive case, it uses Propositions 1 and 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. Where we write $A <:_n B$, Siek et al. (2015) write $A \sqsubseteq B$. 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. One function is a naive subtype of another if both the domains and ranges are naive subtypes; note that this is covariant in both the domain and range of the function, as opposed to the other three subtyping relations which are contravariant in the domain and covariant in the range.

The Tangram Lemma of Wadler and Findler (2009) relates these four subtyping relations. Part 1 concerns ordinary subtyping and part 2 concerns naive subtyping; note that we have $A <:^- B$ in part 1 and $B <:^- A$ in part 2. We show that the Tangram Lemma holds for the relations defined in Figure 4.

Proposition 11 (Tangram Lemma)**.** *(Coq: `tangram`)*

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 $A <:^- B$. ■

The practical consequence of the Tangram Lemma is that if $A <:_n B$ and M is safe for p, then the two casts $M : A \Longrightarrow^p B$ and $M : B \Longrightarrow^{\overline{p}} A$ are also safe for p. When we study interoperability between languages with implicit and explicit nulls in Section 5, we will

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

***IN* Terms**

$$
\begin{array}{rll}
L, M, N ::= & x & \text{Variable} \\
\mid & c & \text{Base Constant} \\
\mid & M \oplus N & \text{Base Operation} \\
\mid & \lambda x{:}A.N & \text{Function Abstraction} \\
\mid & L\ M & \text{Function Application} \\
\mid & \mathit{null} & \text{Null Constant}
\end{array}
$$

***IN* Types**

$$
\begin{array}{rll}
A, B, C ::= & \iota & \text{Base Type} \\
\mid & A \to B & \text{Function Type}
\end{array}
$$

Fig. 6. *IN* syntax.

use second half of the Tangram Lemma to conclude that all such casts are safe for blame labels that assign blame to the language with explicit nulls. If something goes wrong, it will be blamed on the language with implicit nulls.

4 The implicit language *IN*

Having defined EN, an explicit language with casts (like Scala), we now define an implicit language that ignores nullability in its types (like Java). We call the implicit language *IN* for short.

4.1 Syntax

The syntax of *IN* is defined in Figure 6. Following the advice of Patrignani (2021), we use a *red-orange color and a slanted serif font* to distinguish elements of *IN* from EN. The syntax of *IN* terms L, M, N mirrors that of EN, but omits lifting, case, casts, and blame, since those are useless without the distinction between nullable and non-null types.

Types A, B, C of *IN* are not distinguished as nullable or non-null. All types in *IN* admit the null constant.

4.2 Typing

The typing rules of *IN* are standard and are shown in Figure 7. The *null* constant can have any type A.

4.3 Semantics

We define the semantics of *IN* by translation to EN, whose operational semantics we defined in Section 2. The translation is presented in Figure 8.

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash x : A} \quad \text{(IMP-VAR)}$$

$$\Gamma \vdash c : \iota \quad \text{(IMP-BASE)}$$

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

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

$$\frac{\Gamma \vdash L : A \to B \qquad \Gamma \vdash M : A}{\Gamma \vdash L\ M : B} \quad \text{(IMP-APP)}$$

$$\Gamma \vdash \mathit{null} : A \quad \text{(IMP-NULL)}$$

Fig. 7. *IN* typing rules.

Translation of *IN* Types

$$|\iota| = \iota?$$
$$|A \to B| = (|A| \to |B|)?$$

Elvis Operator Syntactic Sugar

$$\mathsf{L\ ?{:}\ M} = \mathsf{case\ L\ of\ \{null \mapsto M;\ \langle x \rangle \mapsto x\}}$$

Translation of *IN* Terms

$$|x| = \mathsf{x}$$
$$|c| = \langle \mathsf{c} \rangle$$
$$|M \oplus N| = \langle (|M|\ \mathsf{?{:}\ blame}\ op) \oplus (|N|\ \mathsf{?{:}\ blame}\ op) \rangle$$
$$|\lambda x{:}A.N| = \langle \lambda \mathsf{x}{:}|A|.|N| \rangle$$
$$|L\ M| = (|L|\ \mathsf{?{:}\ blame}\ \mathit{deref})\ |M|$$
$$|\mathit{null}| = \mathsf{null}$$

Fig. 8. Translation from *IN* to EN.

Types of *IN* are translated to types in EN 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 *IN*, we will make frequent use of case. We introduce as shorthand the Elvis operator M ?: N (as in Kotlin and other languages) that takes a term M of nullable type D? and reduces to V when M evaluates to ⟨V⟩ and to N when M evaluates to null.

Variable references and the *null* constant are just translated to themselves. Base constants c, base operations $M \oplus N$, and function abstractions $\lambda x{:}A(N$ have types in *IN* that translate to nullable types in EN, so these terms are translated to lifted terms in EN. The translation of a base operation $\oplus$ uses the Elvis operator to check the whether the operands

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

N and 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. (Some might prefer the blame labels $\overline{op}$ and $\overline{deref}$ to indicate that the fault lies with the containing context, but arguably the blame lies with the contained term for not providing a non-null value. In any event, the desirability of using overbars is not clear in the absence of a cast, so we follow the advice of Tufte (2001) to minimize the ink on the page.) Similarly, the translation of a function application $L\ M$ first checks whether L (the function) evaluates to null before evaluating the argument M and performing the application.

4.4 Type preservation of the translation

The translation from IN to EN preserves typing:

Proposition 12 (Translation Preserves Typing)**.** *(Coq: `desugaring_typing`)*
If $\Gamma \vdash M : A$, then $|\Gamma| \vdash |M| : |A|$, where a typing context $|\Gamma|$ is obtained by replacing each binding of the form $x : A$ in Γ with $\mathsf{x} : |A|$.

Proof The proof is by induction on the derivation of $\Gamma \vdash M : A$. The IMP-ABS rule is parameterized by an arbitrary fresh variable x, so a proof for this case must be valid for *any* such fresh x. To prove this case, 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 13. All other cases are straightforward. ■

Lemma 13 (Substitution Commutes With Translation)**.** *(Coq: `open_trm_of_itrm`)*

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

Proof The proof is by straightforward induction on the structure of N. ■

5 Interoperability

In this section, we will explore how terms of EN can use terms of IN and vice versa.

5.1 Implicit terms within explicit terms

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

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

We can use an *IN* term M of type A in EN directly by embedding it within a cast to the desired type:

$$|M| : |A| \Longrightarrow^{implicit} \lfloor A \rfloor \tag{1}$$

Here, $\lfloor A \rfloor$ is a *naive* translation of the *IN* type A, defined in general as follows:

$$\lfloor \iota \rfloor = \iota$$
$$\lfloor A \to B \rfloor = \lfloor A \rfloor \to \lfloor B \rfloor$$

The naive translation maps a base type of *IN* to a base type of EN, and it maps a function type of *IN* to a function type of EN. Thus, the cast enables the *IN* term M to be used in EN directly with a definite base or function type. The blame label *implicit* indicates that the cast is from *IN* to EN.

The cast could fail but only with positive blame, blaming the *IN* subterm $|M|$ rather than the surrounding EN context. This is because $\lfloor A \rfloor$ is a naive subtype of $|A|$:

Proposition 14 (*IN* Naive Subtyping)**.** *(Coq: `imp_naive_subtyp`)*
For every IN type A, $\lfloor A \rfloor <:_{\mathsf{n}} |A|$.

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

Then by the Tangram Lemma, term (1) may reduce to blame *implicit* but can never reduce to blame $\overline{implicit}$.

5.2 Explicit terms within implicit terms

To use a term M of EN within a term of *IN*, we need to apply the translation to the *IN* context that surrounds M. In symbols, our approach is to translate a mixed term $C[\mathsf{M}]$ into the EN term $|C|[\mathsf{M}]$ using a natural extension of the translation from Figure 8 to contexts.

In comparison, the approach taken in the previous section was to translate a mixed term $\mathsf{C}[M]$ into the EN term $\mathsf{C}[|M|]$. That only required applying the translation to the *IN* subterm M rather than to an *IN* context C.

Suppose the EN subterm M has some type A. Typing the surrounding *IN* context C requires an *IN* type for the term intended to plug the hole. We define the *erasure* of a type from EN to *IN* as follows:

$$\lceil \mathsf{D?} \rceil = \lceil \mathsf{D} \rceil$$
$$\lceil \iota \rceil = \iota$$
$$\lceil \mathsf{A} \to \mathsf{B} \rceil = \lceil \mathsf{A} \rceil \to \lceil \mathsf{B} \rceil$$

We then say that an EN subterm of type A can be used within an *IN* context that can be typed under the assumption that an *IN* term that plugs its hole has type $\lceil \mathsf{A} \rceil$.

When we translate the *IN* context C, the resulting EN context $|C|$ will be typable assuming that the term that plugs its hole has type $|\lceil \mathsf{A} \rceil|$. Note that it is possible to define $|\lceil \mathsf{A} \rceil|$ directly as follows:

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

$$|\lceil D? \rceil| = |\lceil D \rceil|$$
$$|\lceil \iota \rceil| = \iota?$$
$$|\lceil A \to B \rceil| = (|\lceil A \rceil| \to |\lceil B \rceil|)?$$

We can use an EN term M of type A in *IN* directly by embedding it within a cast to the desired type:

$$M : A \Longrightarrow^{explicit} |\lceil A \rceil| \tag{2}$$

The cast could fail but only with *negative* blame, blaming the surrounding *IN* context rather than the EN subterm M. This is because A is a naive subtype of $|\lceil A \rceil|$:

Proposition 15 (EN Naive Subtyping)**.** *(Coq:* `exp_naive_subtyp`*)*
For every EN type A, $A <:_n |\lceil A \rceil|$.

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

Then by the Tangram Lemma, term (2) may reduce to blame $\overline{explicit}$ but can never reduce to blame *explicit*.

5.3 Metatheory of contexts

So far, we have taken for granted that when a term is embedded in a context, $C[M]$, the term M can refer to free variables bound in the context C. This deserves to be established formally, particularly when we mix contexts and subterms from EN and *IN*. We make this precise in this section.

One may be tempted by a simpler construction. In an earlier version of this paper, we used the construct $\mathsf{let}\ x = (M : A \Longrightarrow^{explicit} |\lceil A \rceil|)\ \mathsf{in}\ |N|$, where $\mathsf{let}\ x = M\ \mathsf{in}\ N$ abbreviates $(\lambda x{:}A.N)\ M$ when A is the type of M. However, that approach has the significant disadvantage of requiring M to be closed. Here, in the mixed term $C[M]$, the subterm M can refer to free variables bound in the context C, an important generalization.

Figure 9 defines grammars of EN and *IN* contexts following the syntax of terms, but with a hole $\Box$ that can be plugged by a subterm.

We introduce the notation $\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash C : A$ for typing judgments for EN contexts and similarly for *IN* contexts. Such a judgment is to be interpreted as follows: if the hole $\Box$ in context C is filled with some term M' such that $\Gamma' \vdash M' : A'$, then the term that results from plugging the hole in context C with the term M' has type A in typing context Γ. Inductive definitions of the typing relations for EN and *IN* contexts are presented in Figures 10 and 11. By design, both the EN and *IN* context typing rules satisfy the intended property:

Proposition 16 (EN Plugged Typing)**.** *(Coq:* `typing_plugged`*)*
For any EN context C *and term* M'*, if* $\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash C : A$ *and* $\Gamma' \vdash M' : A'$ *for some typing contexts* Γ*,* Γ' *and types* A*,* A'*, then* $\Gamma \vdash C[M'] : A$.

Proof The proof is by induction on the typing relation for EN contexts (Figure 10). ■

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

EN Contexts

$\mathsf{C} ::= \Box$	Hole
$\mid \mathsf{C} \oplus \mathsf{N}$	Base Operation Left
$\mid \mathsf{M} \oplus \mathsf{C}$	Base Operation Righ
$\mid \lambda \mathsf{x}{:}\mathsf{A}.\mathsf{C}$	Function Abstraction
$\mid \mathsf{C}\ \mathsf{M}$	Function Application Left
$\mid \mathsf{L}\ \mathsf{C}$	Function Application Right
$\mid \langle \mathsf{C} \rangle$	Lift
$\mid \mathsf{case}\ \mathsf{C}\ \mathsf{of}\ \{\mathsf{null} \mapsto \mathsf{M};\ \langle \mathsf{x} \rangle \mapsto \mathsf{N}\}$	Case Scrutinee
$\mid \mathsf{case}\ \mathsf{L}\ \mathsf{of}\ \{\mathsf{null} \mapsto \mathsf{C};\ \langle \mathsf{x} \rangle \mapsto \mathsf{N}\}$	Case Null
$\mid \mathsf{case}\ \mathsf{L}\ \mathsf{of}\ \{\mathsf{null} \mapsto \mathsf{M};\ \langle \mathsf{x} \rangle \mapsto \mathsf{C}\}$	Case Non-Null
$\mid \mathsf{C} : \mathsf{A} \Longrightarrow^{\mathsf{p}} \mathsf{B}$	Cast

***IN* Contexts**

$C ::= \Box$	Hole
$\mid C \oplus N$	Base Operation Left
$\mid M \oplus C$	Base Operation Right
$\mid \lambda x{:}A.C$	Function Abstraction
$\mid C\ M$	Function Application Left
$\mid L\ C$	Function Application Right

Fig. 9. Syntax of EN and *IN* contexts.

Proposition 17 (*IN* Plugged Typing)**.** *(Coq: `ityping_plugged`)*
For any IN context C and term M', if $\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash C : A$ and $\Gamma' \vdash M' : A'$ for some typing contexts Γ, Γ' and types A, A', then $\Gamma \vdash C[M'] : A$.

Proof The proof is by induction on the typing relation for *IN* contexts (Figure 11). ■

Having defined typing for contexts, we can state a formal criterion for allowing an *IN* term to be used within an EN context, as we sketched in Section 5.2:

Definition 18 (Allowable *IN* term for EN context)**.** *An IN term M may plug an EN context C if there exist EN typing contexts Γ, Γ', IN type A, and EN type B such that $\lceil \Gamma' \rceil \vdash M : A$ and $|\lceil \Gamma' \rceil| \vdash \Box : \lfloor A \rfloor \Vdash \Gamma \vdash \mathsf{C} : \mathsf{B}$.*

In the definition, Γ' records the types of the variable bindings in C. The *IN* subterm M has some type A in the context $\lceil \Gamma' \rceil$, which applies erasure to the types in Γ'.

The EN context C needs to be typed under the assumption that the hole has type $\lfloor A \rfloor$, the naive translation of A, in the typing context $|\lceil \Gamma' \rceil|$, which is the translation of the erased context $\lceil \Gamma' \rceil$. To see why $|\lceil \Gamma' \rceil|$ is necessary instead of just Γ', consider that an *IN* context $\lceil \Gamma' \rceil$ cannot distinguish whether Γ' binds some variable x to a definite type such as ι or a

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

$$\Gamma \vdash \Box : A \Vdash \Gamma \vdash \Box : A \qquad \text{(ExpC-Hole)}$$

$$\frac{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash C : \iota \qquad \Gamma \vdash N : \iota}{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash C \oplus N : \iota} \qquad \text{(ExpC-Binop-Left)}$$

$$\frac{\Gamma \vdash M : \iota \qquad \Gamma' \vdash \Box : A' \Vdash \Gamma \vdash C : \iota}{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash M \oplus C : \iota} \qquad \text{(ExpC-Binop-Right)}$$

$$\frac{\Gamma' \vdash \Box : A' \Vdash \Gamma, x{:}A \vdash C : B}{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash \lambda x{:}A.C : A \to B} \qquad \text{(ExpC-Abs)}$$

$$\frac{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash C : A \to B \qquad \Gamma \vdash M : A}{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash C\ M : B} \qquad \text{(ExpC-App-Left)}$$

$$\frac{\Gamma \vdash L : A \to B \qquad \Gamma' \vdash \Box : A' \Vdash \Gamma \vdash C : A}{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash L\ C : B} \qquad \text{(ExpC-App-Right)}$$

$$\frac{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash C : D}{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash \langle C \rangle : D?} \qquad \text{(ExpC-Lift)}$$

$$\frac{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash C : D? \qquad \Gamma \vdash M : A \qquad \Gamma, x{:}D \vdash N : A}{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash \mathsf{case}\ C\ \mathsf{of}\ \{\mathsf{null} \mapsto M;\ \langle x \rangle \mapsto N\} : A} \qquad \text{(ExpC-Case-Scrutinee)}$$

$$\frac{\Gamma \vdash L : D? \qquad \Gamma' \vdash \Box : A' \Vdash \Gamma \vdash C : A \qquad \Gamma, x{:}D \vdash N : A}{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash \mathsf{case}\ L\ \mathsf{of}\ \{\mathsf{null} \mapsto C;\ \langle x \rangle \mapsto N\} : A} \qquad \text{(ExpC-Case-Null)}$$

$$\frac{\Gamma \vdash L : D? \qquad \Gamma \vdash M : A \qquad \Gamma' \vdash \Box : A' \Vdash \Gamma, x{:}D \vdash C : A}{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash \mathsf{case}\ L\ \mathsf{of}\ \{\mathsf{null} \mapsto M;\ \langle x \rangle \mapsto C\} : A} \qquad \text{(ExpC-Case-Non-Null)}$$

$$\frac{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash C : A \qquad A \sim B}{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash (C : A \Longrightarrow^{p} B) : B} \qquad \text{(ExpC-Cast)}$$

Fig. 10. EN context typing rules.

nullable type such as $\iota?$; both are erased to ι. Typing the hole in the translated erased typing context solves the problem, since $|\lceil \iota \rceil| = |\lceil \iota? \rceil| = \iota?$.

This criterion ensures that when an implicit term M is plugged into an explicit context C, its translation $C[\ |M| : |A| \Longrightarrow^{implicit} \lfloor A \rfloor\ ]$ is well typed.

Proposition 19 (Typing of EN term in *IN* context)**.** *(Coq: `nest_itrm_in_ctx`)*
For any EN context C and IN term M that satisfy Definition 18 with EN typing contexts Γ, Γ', IN type A, and EN type B,

$$\Gamma \vdash C[\ |M| : |A| \Longrightarrow^{implicit} \lfloor A \rfloor\ ] :\ B$$

Proof This follows directly by Proposition 12 and Proposition 16. Proposition 14 is needed to justify the compatibility of the types $|A|$ and $\lfloor A \rfloor$ in the cast. ■

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

$$\Gamma \vdash \Box : A \Vdash \Gamma \vdash \Box : A \qquad \text{(ImpC-Hole)}$$

$$\frac{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash C : \iota \qquad \Gamma \vdash N : \iota}{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash C \oplus N : \iota} \qquad \text{(ImpC-Binop-Left)}$$

$$\frac{\Gamma \vdash M : \iota \qquad \Gamma' \vdash \Box : A' \Vdash \Gamma \vdash C : \iota}{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash M \oplus C : \iota} \qquad \text{(ImpC-Binop-Right)}$$

$$\frac{\Gamma' \vdash \Box : A' \Vdash \Gamma, x{:}A \vdash C : B}{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash \lambda x{:}A.C : A \to B} \qquad \text{(ImpC-Abs)}$$

$$\frac{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash C : A \to B \qquad \Gamma \vdash M : A}{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash C\ M : B} \qquad \text{(ImpC-App-Left)}$$

$$\frac{\Gamma \vdash L : A \to B \qquad \Gamma' \vdash \Box : A' \Vdash \Gamma \vdash C : A}{\Gamma' \vdash \Box : A' \Vdash \Gamma \vdash L\ C : B} \qquad \text{(ImpC-App-Right)}$$

Fig. 11. *IN* context typing rules.

Turning our attention to EN terms embedded in *IN* contexts, we define a similar typing criterion for such mixed terms $C[\mathsf{M}]$:

Definition 20 (Allowable EN term for *IN* context)**.** *An EN term* M *may plug an IN context* C *if there exist IN typing contexts* Γ, Γ', *EN type* A, *and IN type* B *such that* $|\Gamma'| \vdash \mathsf{M} : \mathsf{A}$ *and* $\Gamma' \vdash \Box : \lceil \mathsf{A} \rceil \Vdash \Gamma \vdash C : B$.

The EN term M is typed in a typing context $|\Gamma'|$, which contains the variables bound in C, but with their types translated from *IN* to EN using the translation from Figure 8. In that context, M has some EN type A, which erases to the *IN* type $\lceil \mathsf{A} \rceil$, so the *IN* context C must be typable under the assumption that its hole has the erased type $\lceil \mathsf{A} \rceil$.

We can apply the translation from Figure 8 to the *IN* context C to obtain an EN context $|C|$. The translation of contexts is defined to be the same as the translation of terms, with the additional case that a hole translates to a hole. The resulting EN context will satisfy $|\Gamma'| \vdash \Box : |\lceil \mathsf{A} \rceil| \Vdash |\Gamma| \vdash |C| : |B|$ by the following proposition:

Proposition 21 (Translated Context Typing)**.** *(Coq:* `ictrm_desugaring_typing`*) For any IN context* C*, types* A, B*, and typing contexts* Γ, Γ'*, if* $\Gamma' \vdash \Box : A \Vdash \Gamma \vdash C : B$*, then* $|\Gamma'| \vdash \Box : |A| \Vdash |\Gamma| \vdash |C| : |B|$.

Proof The proof is by induction on the derivation of the context typing of C. The case of function abstraction requires the following Lemma 22, an analog of Lemma 13 for contexts rather than terms. ■

Lemma 22 (Substitution Commutes With Translation of Contexts)**.** *(Coq:* `open_ctrm_of_ictrm`*)*

$$|C[x \mapsto C']| = |C|[\mathrm{x} \mapsto |C'|]$$

Proof The proof is by straightforward induction on the structure of C. ■

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

To plug the resulting EN context $|C|$ with the term M, we must use a cast to adapt the type of M from A to the type $|\lceil A \rceil|$ that the context $|C|$ requires. We summarize the type safety of the translation as follows.

Proposition 23 (Typing of EN term in *IN* context). *(Coq: `nest_trm_in_ictx`)*
For any IN context C and EN term M that satisfy Definition 20 with IN typing contexts Γ, Γ', *EN type A, and IN type B,*

$$|\Gamma| \vdash |C|[\, M : A \Longrightarrow^{explicit} |\lceil A \rceil|\,] : |B|$$

Proof This follows directly by Proposition 21 and Proposition 16. Proposition 15 is needed to justify the compatibility of the types A and $|\lceil A \rceil|$ in the cast. ■

5.4 A defensive alternative

Although the casts presented in the previous two subsections always blame the implicit language when they fail, it is also possible to prevent any possibility of failure by using case to explicitly test for null and manually (and tediously) providing explicit error-handling terms to be evaluated when a null value is encountered.

To show how tedious, consider the *IN* term $M = \lambda x{:}(\iota \to \iota)(x\ c$ to be embedded in the EN context $C = \Box\ (\lambda x'{:}\iota.x')$. The translated term $|M|$ has type $((\iota? \to \iota?)? \to \iota?)?$, but the context requires the hole to be plugged by a term of type $(\iota \to \iota) \to \iota$. Instead of using a cast to mediate between $|M|$ and C, we could interpose the context

$$\begin{array}{l} C' = \ \mathsf{case}\ \Box\ \mathsf{of} \\ \left\{ \begin{array}{l} \mathsf{null} \mapsto \mathsf{blame\ p}; \\ \\ \langle \mathsf{x} \rangle \mapsto \lambda \mathsf{x}'{:}\iota \to \iota.\ \ \mathsf{case} \left(\mathsf{x} \left\langle \lambda \mathsf{x}''{:}\iota?. \left\langle \mathsf{x}' \ \begin{array}{l} \mathsf{case\ x}''\ \mathsf{of} \\ \left\{ \begin{array}{l} \mathsf{null} \mapsto \mathsf{blame\ p}; \\ \langle \mathsf{x}''' \rangle \mapsto \mathsf{x}''' \end{array} \right\} \end{array} \right\rangle \right\rangle \right) \mathsf{of} \\ \qquad\qquad \left\{ \begin{array}{l} \mathsf{null} \mapsto \mathsf{blame\ p}; \\ \langle \mathsf{x}'''' \rangle \mapsto \mathsf{x}'''' \end{array} \right\} \end{array} \right. \end{array}$$

The resulting term would be $C[C'[|M|]]$. The occurrences of blame p in C' identify the places where *null* can occur at run time and could be replaced by user-defined defensive compensation code.

In the other direction, consider the EN term $M = \lambda x{:}(\iota \to \iota).x\ c$ to be embedded in the *IN* context $C = \Box\ (\lambda x'{:}\iota(x')$. The term M has type $(\iota \to \iota) \to \iota$, but the translated context $|C|$ requires the hole to be plugged by a term of type $((\iota? \to \iota?)? \to \iota?)?$. Instead of using a cast to mediate between M and $|C|$, we could interpose the context

$$C' = \left\langle \lambda x{:}(\iota? \to \iota?)?. \left\langle \Box \ \begin{array}{l} \mathsf{case\ x}\ \mathsf{of} \\ \left\{ \begin{array}{l} \mathsf{null} \mapsto \mathsf{blame\ p}; \\ \langle \mathsf{x}' \rangle \mapsto \lambda \mathsf{x}''{:}\iota.\ \ \mathsf{case\ x}'\ \langle \mathsf{x}'' \rangle\ \mathsf{of} \\ \qquad\qquad \left\{ \begin{array}{l} \mathsf{null} \mapsto \mathsf{blame\ p}; \\ \langle \mathsf{x}''' \rangle \mapsto \mathsf{x}''' \end{array} \right\} \end{array} \right\} \end{array} \right\rangle \right\rangle$$

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

The resulting term would be $|C|[C'[M]]$. The occurrences of $\mathsf{blame\ p}$ in C' identify the places where *null* can occur at run time and could be replaced by user-defined defensive compensation code.

Given any pair of compatible EN types, we can synthesize the necessary compensation code using the following function, which follows the structure of the inductive definition of compatibility from Figure 2:

$$\begin{aligned}
\|\iota \Longrightarrow \iota\| &= \Box \\
\|A \Longrightarrow D?\| &= \langle \|A \Longrightarrow D\| \rangle \\
\|D? \Longrightarrow A\| &= \mathsf{case}\ \Box\ \mathsf{of}\ \{\mathsf{null} \mapsto \mathsf{blame\ p};\ \langle x \rangle \mapsto \|D \Longrightarrow A\|[x]\} \\
\|(A \to B) \Longrightarrow (A' \to B')\| &= \lambda x{:}A'.\|B \Longrightarrow B'\|[\Box\ (\|A' \Longrightarrow A\|[x])]
\end{aligned}$$

As before, occurrences of $\mathsf{blame\ p}$ can be replaced by user-defined defensive compensation code.

6 Connections with other languages

Now that we have presented our explicit and implicit languages in full, we briefly discuss the connections with the languages of Nieto et al. (2020*a*) and with complete languages like Scala and Java.

6.1 Connections with $\lambda_{\mathtt{null}}$ and $\lambda^{s}_{\mathtt{null}}$

Recall that the core language $\lambda_{\mathtt{null}}$ of Nieto et al. (2020*a*) defines three function types, which correspond to types in EN and *IN* as follows:

- $\#(S \to T)$ in $\lambda_{\mathtt{null}}$ corresponds to $A \to B$ in EN.
- $?(S \to T)$ in $\lambda_{\mathtt{null}}$ corresponds to $(A \to B)?$ in EN.
- $!(S \to T)$ in $\lambda_{\mathtt{null}}$ corresponds to $A \to B$ in *IN*.

Here, we assume that S and T correspond to A and B or A and B.

The semantics of *safe application* $app(s, t, u)$ in $\lambda_{\mathtt{null}}$ is analogous to the EN term $\mathsf{case\ L\ of}\ \{\mathsf{null} \mapsto \mathsf{N};\ \langle x \rangle \mapsto x\ \mathsf{M}\}$, where s, t, u correspond to L, M, N. This term requires L to be of type $(A \to B)?$, while standard function application requires a term of type $A \to B$. Since a $\lambda_{\mathtt{null}}$ term of type $!(S \to T)$ can be used in both forms of application, it does not correspond to any one type in EN, but more closely to the *IN* type $A \to B$: values of that type include both *null* and functions, and terms of that type can be used in *IN* function application.

This illuminates a key difference between EN and $\lambda_{\mathtt{null}}$. In EN, all implicit features have been desugared away into casts, and casts are the only terms that can fail with blame. In contrast, $\lambda_{\mathtt{null}}$ retains unsafe nullable function types, an implicit language feature, so applications there can also fail with blame. This difference motivates the various auxiliary relations needed in $\lambda_{\mathtt{null}}$, such as blame assignment and normalization.

The surface language $\lambda^{s}_{\mathtt{null}}$ provides import_e and import_i terms that enable an explicit term to be used within an implicit term and vice versa, like the embeddings that we discussed in Sections 5.1 and 5.2. A key difference is that the $\lambda^{s}_{\mathtt{null}}$ typing rules require the

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

embedded subterms to be closed, so they cannot refer to variables bound in the surrounding context in the other sublanguage.

6.2 Connections with Scala and Java

In this section, we briefly compare the core explicit and implicit languages with Scala and Java.

The types of the implicit language correspond to the reference types of Java, in that every type admits the null value. The nullable types D? of the explicit language correspond to union types T|Null in Scala, where T is a Scala type corresponding to the definite type D. Thus, interoperability between Scala and Java directly follows the interoperability between the explicit and implicit languages. We can embed a Java term, such as a call to a Java method, in Scala following the translation from Section 5.2. We can embed a Scala term, such as a call to a Scala method, in Java following the translation from Section 5.1. Such embedded terms are not limited to calls and can refer to variables bound in the surrounding context in the opposite language, to model higher-level language features such as terms referring to fields and variables defined in other classes, including in code written in the opposite language. The same applies to Kotlin instead of Scala, where a nullable type is written as T?.

There is a minor difference in the values. The explicit language distinguishes a value V of definite type D and a lifted value ⟨V⟩ of nullable type D?. In Scala, both values have the same runtime representation and the lifting operation is a no-op at run time.

As core calculi based on simply typed lambda calculus, the explicit and implicit languages obviously omit many features present in Scala and Java, notably objects, classes and class types, and generic methods and generic class types. The calculi serve as a foundation to guide the design of null safety for these additional language features. In particular, one practical challenge in a full language arises when a single type describes a large, linked data structure, such as a linked list or tree. If the type does not say anything about null values within the structure, such values could occur anywhere within it. Extensions of the core languages for classes and objects would add casts to identify places where blame must be assigned for inter-language operations on objects. Those casts would identify the places in practical programs where nulls can arise.

7 Related work

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. Siek and Taha (2006, 2007) introduced the concept of *gradual typing* to enable interoperability between parts of a program with and without static types. 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. Also see Wadler (2015).

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

Garcia et al. (2016) used the framework of abstract interpretation (Cousot and Cousot, 1977) to explain what it means for part of a program to be more-precisely or less-precisely typed. This abstract interpretation notion of the precision of a type is identical to naive subtyping. Estep et al. (2021) applied this methodology to a static analysis that determines which expressions in a program may evaluate to null at run time. Their baseline static analysis works for an intermediate language in which every variable is explicitly annotated to be either non-null or nullable; they systematically derive a gradual static analysis that works for an intermediate language that allows a subset of variables to be left unannotated. Malewski et al. (2021) applied the abstract interpretation framework to a calculus with algebraic data types (ADTs) to systematically derive a calculus with gradual ADTs. Their *(open) gradual datatype* designates expressions that evaluate to a value of some ADT, but the specific ADT and its set of constructors is not known statically and can even be open in the sense of allowing extension with new constructors.

Nieto et al. (2020*a*) instantiated the concepts of gradual typing and blame for their explicit-null extension of the Scala language (Nieto et al., 2020*b*). 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. Platform types are inherently unsound since, by transitivity, they make a nullable type a subtype of the corresponding non-null type, but they are necessary in practice to avoid the overwhelming number of compile-time errors that would result if we insisted on static null safety at the boundary between Java and Kotlin.

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. A null value may still flow at run time to a context in which a non-null value is expected, resulting in a run-time exception.

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. Attempting to use an implicitly unwrapped optional that is nil at run time results in a run-time exception.

https://doi.org/10.1017/S0956796824000121 Published online by Cambridge University Press

8 Conclusion

We have defined a pair of core calculi for modeling 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 modeling nullness interoperability in larger languages, in the same way that foundations such as Featherweight Java (Igarashi et al., 2001) and DOT (Amin et al., 2016) have guided the design of Java and Scala. Our development is formalized in Coq and is included as an artifact accompanying the paper.

Conflict of interest.

The authors report no conflict of interest.

Supplementary material

For supplementary material for this article, please visit https://doi.org/10.1017/S0956796824000121.

References

Amin, N., Grütter, S., Odersky, M., Rompf, T. & Stucki, S. (2016) The essence of dependent object types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 9600)*, Lindley, S., McBride, C., Trinder, P. W. & Sannella, D. (eds.), Springer, pp. 249–272. https://doi.org/10.1007/978-3-319-30936-1_14

Amin, N. & Tate, R. (2016) Java and Scala's type systems are unsound: The existential crisis of null pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 – November 4, 2016*, Visser, E. & Smaragdakis, Y. (eds.), ACM, pp. 838–848.

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

Aydemir, B. & Weirich, S. (2010) *LNgen: Tool Support for Locally Nameless Representations*. Technical Report MS-CIS-10-24. Computer and Information Science, University of Pennsylvania.

Aydemir, B. E., Charguéraud, A., Pierce, B. C., Pollack, R. & Weirich, S. (2008) Engineering formal metatheory. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7–12, 2008*, Necula, G. C. & Wadler, P. (eds.), ACM, pp. 3–15. https://doi.org/10.1145/1328438.1328443

Cousot, P. & Cousot, R. (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*, Graham, R. M., Harrison, M. A. & Sethi, R. (eds.), ACM, pp. 238–252. https://doi.org/10.1145/512950.512973

Drossopoulou, S. & Eisenbach, S. (1997) Java is type safe - Probably. In *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings (Lecture Notes in Computer Science, Vol. 1241)*, Aksit, M. & Matsuoka, S. (eds.), Springer, pp. 389–418. https://doi.org/10.1007/BFB0053388

Estep, S., Wise, J., Aldrich, J., Tanter, É., Bader, J. & Sunshine, J. (2021) Gradual program analysis for null pointers. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Møller, A. & Sridharan, M. (eds.), Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 3:1–3:25. https://doi.org/10.4230/LIPICS.ECOOP.2021.3

Findler, R. B. & Felleisen, M. (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*, Wand, M. & Peyton Jones, S. L. (eds.), ACM, pp. 48–59.

Flatt, M., Krishnamurthi, S. & Felleisen, M. (1998) Classes and Mixins. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, MacQueen, D. B. & Cardelli, L. (eds.), ACM, pp. 171–183. https://doi.org/10.1145/268946.268961

Garcia, R., Clark, A. M. & Tanter, É. (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*, Bodík, R. & Majumdar, R. (Eds.). ACM, pp. 429–442. https://doi.org/10.1145/2837614.2837670

Griesemer, R., Hu, R., Kokke, W., Lange, J., Taylor, I. L., Toninho, B., Wadler, P. & Yoshida, N. (2020) Featherweight go. *Proc. ACM Program. Lang.* **4**, 149:1–149:29. https://doi.org/10.1145/3428217

Hoare, T. (2009) *Null References: The Billion Dollar Mistake*. https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/ (accessed 17 March 2022).

Igarashi, A., Pierce, B. C. & Wadler, P. 2001. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* **23**(3), 396–450. https://doi.org/10.1145/503502.503505

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

Malewski, S., Greenberg, M. & Tanter, É. (2021) Gradually structured data. *Proc. ACM Program. Lang.* **5**, 1–29. https://doi.org/10.1145/3485503

Matthews, J. & Findler, R. B. (2007) Operational semantics for multi-language programs. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, Hofmann, M. & Felleisen, M. (eds.), ACM, pp. 3–10. https://doi.org/10.1145/1190216.1190220

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

Nieto, A., Rapoport, M., Richards, G. & Lhoták, O. (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)*, R. Hirschfeld & T. Pape (eds.), Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:28. https://doi.org/10.4230/LIPIcs.ECOOP.2020.3

Nieto, A., Zhao, Y., Lhoták, O., Chang, A. & Pu, J. (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)*, Hirschfeld, R. & Pape, T. (eds.), Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 25:1–25:26. https://doi.org/10.4230/LIPIcs.ECOOP.2020.25

Nieto, A., Zhao, Y., Lhoták, O., Chang, A. & Pu, J. (2020c) Scala with explicit nulls (artifact). *Dagstuhl Artifacts Ser.* **6**(2), 14:1–14:2. https://doi.org/10.4230/DARTS.6.2.14

Nipkow, T. & von Oheimb, D. (1998) Java$_{light}$ is Type-Safe - Definitely. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, MacQueen, D. B. & Cardelli, L. (eds.), ACM, pp. 161–170. https://doi.org/10.1145/268946.268960

Patrignani, M. (2021) Why Should Anyone use Colours? or, Syntax Highlighting Beyond Code Snippets. arXiv:2001.11334 [cs.SE]

Sewell, P., Nardelli, F. Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S. & Strnisa, R. (2010) Ott: Effective tool support for the working semanticist. *J. Funct. Program.* **20**(1), 71–122.

Siek, J. G. & Taha, W. (2006) Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Vol. 6, pp. 81–92.

Siek, J. G. & Taha, W. (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)*, Ernst, E. (ed.), Springer, pp. 2–27. https://doi.org/10.1007/978-3-540-73589-2_2

Siek, J. G., Vitousek, M. M., Cimini, M. & Boyland, J. T. (2015) Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA (LIPIcs, Vol. 32)*, Ball, T., Bodík, R., Krishnamurthi, S., Lerner, B. S. & Morrisett, G. (eds.), Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 274–293.

Syme, D. (1999) Proving Java type soundness. In *Formal Syntax and Semantics of Java (Lecture Notes in Computer Science, Vol. 1523)*, Alves-Foss, J. (ed.), Springer, pp. 83–118. https://doi.org/10.1007/3-540-48737-9_3

Tobin-Hochstadt, S. & Felleisen, M. (2006) Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium, OOPSLA Companion*. ACM, pp. 964–974.

Tufte, E. R. (2001) *The Visual Display of Quantitative Information* (second ed.). Graphics Press, Cheshire, Connecticut.

Wadler, P. (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)*, Ball, T., Bodík, R., Krishnamurthi, S., Lerner, B. S. & Morrisett, G. (eds.), Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 309–320.

Wadler, P. & Findler, R. B. (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)*, Castagna, G. (ed.), Springer, pp. 1–16.

Wright, A. K. & Felleisen, M. (1994) A syntactic approach to type soundness. *Inf. Comput.* **115**(1), 38–94.