Robust Dynamic Embedding for Gradual Typing*

KOEN JACOBS, Inria, France
MATÍAS TORO, University of Chile, Chile
NICOLAS TABAREAU, Inria, France
ÉRIC TANTER, University of Chile, Chile

Gradual typing has long been advocated as a means to bridge the gap between static and dynamic typing disciplines, enabling a range of use cases such as the gradual migration of existing dynamically typed code to more statically typed code, as well as making advanced static typing disciplines more accessible. To assess whether a given gradual language can effectively support these use cases, several formal properties have been proposed, most notably the refined criteria set forth by Siek et al. One criterion asserts that the dynamic extreme of the spectrum should be expressible in the gradual language, formalized by the existence of an adequate embedding from the corresponding dynamic language.

We observe that the existing dynamic embedding criterion does not capture the desirable property of being able to ascribe embedded code to a static type that it semantically satisfies, and ensure reliable interactions with other components within the gradual language. Specifically, we introduce the notion of *robustness* for gradual terms, meaning that when interacting with any gradual context, runtime failures that may occur ought to be caused by the context, not by the robust term itself. We then formulate the *robust dynamic embedding* criterion: if a dynamic component semantically satisfies a given static type, then its embedding subsequently ascribed to that static type should be a robust term. We demonstrate that robust dynamic embedding is not implied by any existing metatheoretical property from the literature, and is not upheld by various existing gradual languages. We show that robust dynamic embedding is achievable with a gradualized simply-typed language. All the results are formalized in the Rocq proof assistant. This novel criterion complements the set of criteria for gradual languages and opens several venues for further exploration, in particular for typing disciplines that enforce rich semantic properties.

CCS Concepts: • **Theory of computation → Semantics and reasoning**; **Type structures**.

Additional Key Words and Phrases: Gradual typing, dynamic typing, semantic typing

ACM Reference Format:
Koen Jacobs, Matías Toro, Nicolas Tabareau, and Éric Tanter. 2025. Robust Dynamic Embedding for Gradual Typing. *Proc. ACM Program. Lang.* 9, ICFP, Article 238 (August 2025), 27 pages. https://doi.org/10.1145/3747507

1 Introduction

Dynamic and static typing are known for their complementary strengths and weaknesses. With static typing, one can statically rely on the interesting guarantees provided by the typing discipline. Given a type τ, and a syntactically well-typed expression $\vdash e : \tau$, one gets for free that e satisfies

*This work has been partially funded by the Inria Équipe Associée GRAPA, ANID DFG 220011 and ANID FONDECYT Iniciación 11250054, and the Millennium Science Initiative Program: code ICN17_002.

Authors' Contact Information: Koen Jacobs, Inria, France, koen.jacobs@inria.fr; Matías Toro, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Santiago, Chile, mtoro@dcc.uchile.cl; Nicolas Tabareau, Inria, France, nicolas.tabareau@inria.fr; Éric Tanter, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Santiago, Chile, etanter@dcc.uchile.cl.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/8-ART238
https://doi.org/10.1145/3747507

whatever semantic property ($\vDash e : \tau$) the static type system enforces at τ. However, a sound static type checker is necessarily conservative and therefore rejects some well-behaved programs.

Example 1. Consider the following function defined in a dynamically-typed language:

$$g \triangleq \left| \begin{array}{l} \texttt{let}\ taut = \lambda n, h.\ \texttt{if}\ n = 0\ \texttt{then}\ h\ \texttt{else}\ (taut\ (n-1)\ (h\ \texttt{false}) \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad taut\ (n-1)\ (h\ \texttt{true}))\ \texttt{in} \\ \quad taut\ 3 \end{array} \right.$$

Function g tests whether a given 3-argument boolean function h is a tautology, *i.e.*, that h returns $\texttt{true}$ for any combination of arguments. As such, g morally satisfies the simple type $(\mathbb{B} \to \mathbb{B} \to \mathbb{B} \to \mathbb{B}) \to \mathbb{B}$, even though a non-dependent type system would not be able to statically validate it, given that the expected arity of h depends on n. One says that g is *semantically* typed at this type, written $\vDash g : (\mathbb{B} \to \mathbb{B} \to \mathbb{B} \to \mathbb{B}) \to \mathbb{B}$, capturing the appropriate "semantic contract" satisfied by g, despite the fact that the *syntactic* judgment $\vdash g : (\mathbb{B} \to \mathbb{B} \to \mathbb{B} \to \mathbb{B}) \to \mathbb{B}$ is not derivable in a non-dependent type system.

Gradual Typing. Gradually typed languages [33] alleviate the conservativity issue of static typing by offering programmers fine-grained control over when to abide by static typechecking, and when to defer to runtime typechecking. A gradual type system supports *imprecise* type annotations, where imprecision is typically introduced through the use of an "unknown" type, often denoted by ?, which can be thought of as a wildcard representing any possible static type [14]. With the introduction of imprecise type information, static typechecking is then dealt with *optimistically*. That is, a gradual expression at a type τ may optimistically be assumed to be of a type τ', as long as τ and τ' are "plausibly equal" or "consistent" written $\tau \sim \tau'$ (e.g. $? \sim \mathbb{B} \to ?$ because $? \sim ? \to ?$ and $\mathbb{B} \sim ?$, but $? \to \mathbb{B} \nsim \mathbb{Z} \to \mathbb{Z}$ because $\mathbb{B} \nsim \mathbb{Z}$).

As such, one can express Example 1 in a *gradually* typed language as follows:[1]

$$f \triangleq \left| \begin{array}{l} \texttt{let}\ taut = \lambda n, h.\ \texttt{if}\ n = 0\ \texttt{then}\ h\ \texttt{else}\ (taut\ (n-1)\ (h\ \texttt{false}) \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad taut\ (n-1)\ (h\ \texttt{true}))\ \texttt{in} \\ \quad taut\ 3 \end{array} \right.$$

Here, f is well-typed thanks to the use of the unknown type ? for h. Indeed, we have $\vdash taut : \mathbb{Z} \to ? \to ?$ because in the else branch, h may optimistically be assumed to be of type $\mathbb{B} \to ?$ Moreover, we have $\vdash taut\ 3 : ? \to ?$, which in turn may be assumed to be of the more precise (and therefore consistent) type $(\mathbb{B} \to \mathbb{B} \to \mathbb{B} \to \mathbb{B}) \to \mathbb{B}$.

The price for this optimism is then paid in the form of runtime checks at the boundaries between expressions of different precision. These runtime checks validate whether the static optimism turns out to be justified and if not, a runtime error is raised.

Formally, the dynamics of a gradual language is often defined in terms of a translation into a cast calculus, where each use of optimism gets translated into an explicit cast. If during typechecking e of type τ is optimistically assumed to be of consistent type τ', the expression gets translated to $e' : \tau \Rightarrow \tau'$, where e' is the recursive translation of e. A possible translation of the gradually-typed function f is as follows:

$$f \triangleq \left| \begin{array}{l} \texttt{let}\ taut = \lambda n, h.\ \texttt{if}\ n = 0\ \texttt{then}\ h\ \texttt{else}\ (taut\ (n-1)\ ((h : ? \Rightarrow^{\ell_1} \mathbb{B} \to ?)\ \texttt{false}) \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad taut\ (n-1)\ ((h : ? \Rightarrow^{\ell_2} \mathbb{B} \to ?)\ \texttt{true}))\ \texttt{in} \\ \quad (taut\ 3) : ? \to ? \Rightarrow^{\ell_3} (\mathbb{B} \to \mathbb{B} \to \mathbb{B} \to \mathbb{B}) \to \mathbb{B} \end{array} \right. \quad (1)$$

[1]For clarity, we use separate colors for terms of the untyped language and terms of the gradual language.

Each cast also carries a label, which propagates at runtime (for higher-order casts), so that when a gradual program fails with a cast error, **error** ℓ, one can pinpoint the original failing cast, namely the cast with label ℓ. Following Wadler and Findler [41], we represent gradual programs with *explicit* casts for the sake of clarity, at the expense of some verbosity. As standard, we abbreviate two consecutive casts as $e : A \Rightarrow^{\ell_1} B \Rightarrow^{\ell_2} C$.

Programming in the Extremes. To assess the adequacy of gradual languages, a number of different criteria have been introduced in the literature, most notably the refined criteria of Siek et al. [34]. For now, we focus on the two criteria that formalize that a gradual language supports both fully statically and fully dynamically typed programming.

(1) *The gradual language conservatively extends the static language.* That is, programming in the purely static end of the spectrum is indeed supported. On static terms, typing in the gradual language and static language coincide. Moreover, static terms behave the same when evaluated using the dynamics of either the static or gradual language [34].
(2) *Similarly, the gradual language should support programming in the dynamic extreme.* Formally, there exists a dynamic embedding, denoted $\lceil\!\lceil _ \rceil\!\rceil$, that maps an arbitrary dynamic program e to a gradually well-typed program at the unknown type, $\vdash \lceil\!\lceil e \rceil\!\rceil : ?$ [33]. Moreover, e and $\lceil\!\lceil e \rceil\!\rceil$ behave equivalently when run in isolation—either both diverge, both error out, or both produce the same values (modulo embedding).

Coming back to Example 1, the conservative extension of the static typing discipline does not apply, as function g is not syntactically well-typed ($\vdash$), although it is semantically well-typed ($\vDash$). The dynamic embedding criterion tells us that we ought to be able to embed g into the gradual language, obtaining the gradually well-typed expression $\vdash \lceil\!\lceil g \rceil\!\rceil : ?$. The fact that the embedding is a systematic procedure releases programmers from the responsibility of manually coming up with an equivalent gradual counterpart, as we did for f above.

Reasoning about embedded terms. The dynamic embedding criterion [34] does not provide any insight into the behavior of the embedded term *when used at the static type* that it semantically satisfies (if any). For instance, consider a scenario in which the embedding of g is used at the static type $(\mathbb{B} \to \mathbb{B} \to \mathbb{B} \to \mathbb{B}) \to \mathbb{B}$ as part of some larger program defined as follows:

$$P \triangleq \left|\begin{array}{l} \textbf{let}\ g' = \lceil\!\lceil g \rceil\!\rceil : ? \Rightarrow^{\ell} (\mathbb{B} \to \mathbb{B} \to \mathbb{B} \to \mathbb{B}) \to \mathbb{B}\ \textbf{in} \\ \quad \textbf{let}\ h_1 = \ldots\ \textbf{in} \\ \quad\quad \textbf{let}\ h_2 = \ldots\ \textbf{in} \\ \quad\quad\quad g'\ h_1 \wedge g'\ h_2 \end{array}\right. \tag{2}$$

If P throws a cast error, one would expect the guilty cast to stem from the definitions of h_1 or h_2, as g satisfies the semantic contract of type $(\mathbb{B} \to \mathbb{B} \to \mathbb{B} \to \mathbb{B}) \to \mathbb{B}$. Otherwise, if one of the casts in g' (*i.e.,* the outer cast or any of the internal casts introduced by the dynamic embedding procedure) had failed, this would question whether g actually is semantically well-typed. Such reasoning is however not formally provided by the dynamic embedding criterion. This work proposes a refined dynamic embedding criterion that does enable such reasoning.

Robustness. To reason about the behavior of the embedding of semantically well-typed terms once ascribed, we first present a notion of *robustness* for gradual terms. Intuitively, a gradual expression is *robust* for some type τ if *its internal casts can never fail* when used within any surrounding gradual code that uses it at type τ.[2] To formally define robustness, we use the standard notion of *contexts.* A

[2]The use of the adjective "robust" is inspired by Abate et al. [1], where a property (*e.g.,* compiler correctness) is said to be robust whenever it holds in the face of arbitrary surrounding contexts (*e.g.,* linking).

context C is an expression with a hole, noted $\square$, which can be filled with an expression e to obtain $C[e]$. For instance, the program P may be defined by taking the following context:

$$C \triangleq \mathbf{let}\, g' = \square \,\mathbf{in}\,\mathbf{let}\, h_1 = \ldots \,\mathbf{in}\,\mathbf{let}\, h_2 = \ldots \,\mathbf{in}\, g'\ h_1 \wedge g'\ h_2 \tag{3}$$

and filling the hole with $\lceil\!\lceil g \rceil\!\rceil : ? \Rightarrow^{\ell} (\mathbb{B} \to \mathbb{B} \to \mathbb{B} \to \mathbb{B}) \to \mathbb{B}$. The typing judgment for contexts is standard: $\vdash C : (\Gamma; \tau) \Rightarrow (\Gamma'; \tau')$ denotes that if $\Gamma \vdash e : \tau$, then $\Gamma' \vdash C[e] : \tau'$.

Definition 1 (Robustness of gradual expressions)**.** A gradual expression e is *robust* at $\Gamma\&\tau$ iff $\Gamma \vdash e : \tau$ and for any well-typed context $\vdash C : (\Gamma; \tau) \Rightarrow (\cdot; \tau')$, if $C[e]$ reduces to $\mathbf{error}\ \ell$, then ℓ denotes one of the casts in C. If e is closed, we say that e is *robust* at τ.

Having defined robustness, we can now state the robust dynamic embedding (RDE) criterion:

Statement 1 (Robust Dynamic Embedding Criterion). *Consider a dynamic expression* e *that is* semantically *well-typed at some static type* τ, $\vDash e : \tau$. *Then the typed embedding of* e *at* τ, $\lceil\!\lceil e \rceil\!\rceil : ? \Rightarrow^{\ell} \tau$, *is* robust at τ.

RDE entails robustness of the typed embedding $\lceil\!\lceil g \rceil\!\rceil : ? \Rightarrow^{\ell} (\mathbb{B} \to \mathbb{B} \to \mathbb{B} \to \mathbb{B}) \to \mathbb{B}$, and as such, if P reduces to $\mathbf{error}\ \ell$, then the reported label ℓ must indeed denote a cast in either h_1 or h_2.

Additionally, if h_1 and h_2 are fully static boolean terms—hence do not contain any cast—one would know that P simply cannot error. This observation suggests the following corollary of the robust dynamic embedding criterion:

Corollary 1. *Consider a dynamic expression* e *that is* semantically *well-typed at some static type* τ, $\vDash e : \tau$. *Given any cast-free static context* C_s, *then* $C_s[\lceil\!\lceil e \rceil\!\rceil : ? \Rightarrow^{\ell} \tau]$ *cannot reduce to an* $\mathbf{error}$.

Critique of Criteria. A criterion is a meta-theoretical property meant as a formal way to characterize and compare language designs. The criteria set forth by Siek et al. [34] are widely used in the literature to better understand the design space of gradually typed languages, but many other criteria have been proposed, such as graduality [29], the fully abstract embedding of the static language into the gradual language [19], complete monitoring [16], open world soundness [39], as well as properties related to blame assignment [16, 41], among others.

As such, a criterion captures an (arguably) desirable property, not an absolute rule. For instance, the full abstraction criterion of Jacobs et al. [19] is not satisfied in many existing designs. Also, the different criteria set forth by Siek et al. [34] do not always combine well with specific typing disciplines and their expected semantic guarantees, as shown for instance by the extensive literature on both gradual parametricity [2, 17, 22, 29, 38] and gradual security typing [5, 6, 9, 13, 37], as well as on gradual dependent types [11, 12, 23–25]. In cases of tension between different criteria, language designers have to make some compromises, either explicitly sacrificing some criterion, or adjusting the language syntax and semantics by imposing restrictions that help resolve the tensions.

This work proposes a new criterion for gradually typed languages, and as such must address a number of key questions, beyond its informal motivation: Is RDE *trivial* and therefore universally satisfied? Is RDE *redundant i.e.,* just a corollary of some criteria already formulated in the literature? Is RDE *realistic* or an unattainable ideal?

Contributions. The specific contributions of this work are as follows:

- We motivate a novel criterion for gradually typed languages, called the *robust dynamic embedding* (RDE) criterion, which specifies hat the runtime enforcement mechanism of

type discipline	untyped function f	static type τ
(§2.1) sum types	$\lambda s.\,\mathbf{case}\ s\ \mathbf{of}\ (\mathbf{inl}\ _ \Rightarrow s \mid \mathbf{inr}\ _ \Rightarrow \mathbf{inl}\ ())$	$(\mathbb{1}+\mathbb{B}) \longrightarrow (\mathbb{1}+\mathbb{1})$
(§2.2) subtypes	$\lambda z.\,\mathbf{if}\ 0 \le z\ \mathbf{then}\ z\ \mathbf{else}\ 0$	$\mathbb{Z} \longrightarrow \mathbb{N}$
(§2.3) information flow	$\lambda x.\ x - x$	$\mathbb{Z}_H \longrightarrow_L \mathbb{Z}_L$
(§2.4) sensitivity	$\lambda x.\ \mathbf{if}\ 0 \le x\ \mathbf{then}\ x\ \mathbf{else}\ -x$	$(x:\mathbb{Z}) \longrightarrow^{1x} \mathbb{Z}$

Table 1. The static typechecker rejects f to be of type τ (shaded areas), despite having $\vDash f : \tau$.

the gradual language must not entail failure of terms embedded from the corresponding dynamically typed language when ascribed a static type that they semantically satisfy.

- To show that RDE is not trivial, we present several examples of languages from the literature, in a variety of typing disciplines, that do not satisfy it (§2).
- To show that RDE is not redundant, we explain why it is not implied by any of the existing criteria in the literature (§3).
- To show that RDE is realistic, we first describe a simple dynamically typed language λ_d and a corresponding gradual language λ_g, whose semantics are mostly standard, along with an embedding from λ_d to λ_g (§4). We then define semantic typing (for static types) for λ_d using a logical relation (§5). With all these ingredients at hand, we describe the formal proof of the criterion for λ_g (§6). The logical relation is novel in that it allows for the compositional and relational reasoning about programs that may fail only at some chosen set of cast labels.

Finally, §7 discusses some design choice and related work, and §8 concludes. The formal development has been mechanized in Rocq, and is provided as supplementary material [20].

2 The Robust Dynamic Embedding Criterion in the Wild

We now explore the robust dynamic embedding criterion in different contexts, in order to demonstrate that the criterion is not trivially satisfied. Each subsection considers a static typing discipline and fixes a semantically (but not syntactically) well-typed function f at some particular static type τ (Table 1). We then consider possible gradualizations of the typing discipline, the dynamic embedding of f and its subsequent use at τ, obtaining the gradual program $\llbracket f \rrbracket : ? \Rightarrow \tau$. Contemplating different dynamic semantics for the gradual language, using references from the literature, we show that the criterion is not necessarily satisfied, and might even be very challenging or impossible to achieve depending on the semantic property denoted by the typing discipline.

2.1 Sum Types

Consider a dynamic lambda calculus with injections and a case analysis construct, the unit value $()$, and f and τ as defined in Table 1 (type $\mathbb{1}$ is the unit type).

A Semantically Typed Function. With standard sum types [31], one can see that $\vDash f : (\mathbb{1}+\mathbb{B}) \to (\mathbb{1}+\mathbb{1})$: When applying it with any value at $\mathbb{1}+\mathbb{B}$ (i.e. $\mathbf{inl}\ ()$ or $\mathbf{inr}\ b$ with b a boolean), the function successfully computes the value $\mathbf{inl}\ ()$, indeed of type $\mathbb{1}+\mathbb{1}$. Yet, the same function is typically not *syntactically* typed at $(\mathbb{1}+\mathbb{B}) \to (\mathbb{1}+\mathbb{1})$ because the typing rules are typically conservative: the first branch after case-analysis is typed at $\mathbb{1}+\mathbb{B}$ and not $\mathbb{1}+\mathbb{1}$.

Gradualizations. Consider a gradualization of this language with sum types, and the program: $P \triangleq \mathbf{let}\ f' = (\llbracket f \rrbracket : ? \Rightarrow (\mathbb{1}+\mathbb{B}) \to (\mathbb{1}+\mathbb{1}))\ \mathbf{in}\ f'\ (\mathbf{inl}\ ())$. Informally, one might consider P to reduce to a term of the form: $\mathbf{inl}\ () : \mathbb{1}+\mathbb{B} \Rightarrow ? \Rightarrow \mathbb{1}+\mathbb{1}$. As such, P immediately directly contradicts RDE if such a sequence of casts *is not allowed to go beyond type-level information.* Specifically, in a standard gradualization obtained via the AGT methodology [14], P indiscriminately

throws a cast error given that $\mathbb{1} + \mathbb{B}$ is not consistent with $\mathbb{1} + \mathbb{1}$. Indeed, while semantically these types do overlap, this is not reflected in the consistency relation. In contrast, in semantics where such a sequence of casts can inspect whether the injection is a left or a right injection [7, 8, 19], the casts may factor through the injected value reducing to $\mathbf{inl}\ (() : \mathbb{1} \Rightarrow ? \Rightarrow \mathbb{1})$ and thus to $\mathbf{inl}\ ()$, thereby respecting RDE.

2.2 Subtyping

We now turn to a simple static type discipline extended with subtyping, where naturals are a subtype of integers ($\mathbb{N} <: \mathbb{Z}$).

A Semantically Typed Function. Function f (Table 1) is *semantically* typed at $\mathbb{Z} \to \mathbb{N}$: when applied to any integer, f returns a natural, specifically because in the then-branch, z is definitely positive. In any flow-insensitive type system, this function is however not syntactically well-typed at $\mathbb{Z} \longrightarrow \mathbb{N}$, as the meaning of the condition is not taken into account, and the returned type is the subtyping join of both branches, $\mathbb{Z}$.

Gradualizations. In a gradualization with an appropriate dynamic embedding, take the following gradual program: $P \triangleq \mathbf{let}\ f' = (\lceil\!\lceil f \rceil\!\rceil : ? \Rightarrow \mathbb{Z} \to \mathbb{N})\ \mathbf{in}\ f'\ 1$ in it. Informally, in a cast calculus P reduces to a term of the following form: $1 : \mathbb{Z} \Rightarrow ? \Rightarrow \mathbb{N}$.

As in §2.1, P contradicts RDE if such a combination of casts is not allowed to go beyond type-level information, as is the case of AGT-derived languages. More precisely, any semantics where such a combination of casts indiscriminately fails regardless of the integer it is casting due to the type-level fact that $\mathbb{Z}$ is not a subtype of $\mathbb{N}$ contradicts the criterion. In contrast, in the semantics described by Gierczak et al. [15], the latter downcast would actually look at the value at hand and fail only if it were a negative integer; as such, the criterion is not contradicted in that design.

2.3 Security Types for Information-Flow Control

Security typing is a discipline to reason about confidentiality [40]. Types are tagged by a confidentiality level, such as H for private and L for public data. The security type system prevents private values to flow to public channels.

A Semantically Typed Function. The function f in Table 1 does satisfy the semantic contract of the type $\mathbb{Z}_H \longrightarrow_L \mathbb{Z}_L$: this type says that the underlying value is a public function (L on the arrow) that takes a private input ($\mathbb{Z}_H$) and returns a public value ($\mathbb{Z}_L$). The semantic property of *noninterference* implies that the function must be constant, which is indeed the case here as f always returns 0. no information about the secret is ever leaked. Syntactically, however, a security type system will reject f because the public return value is derived from a private argument.

Gradualizations. Consider now a possible gradual counterpart where imprecision is introduced at the confidentiality levels of types, *e.g.,* [6, 9, 37]. For instance, $\mathbb{Z}_? \longrightarrow_? \mathbb{Z}_?$ represents a function of unknown confidentiality, whose domain and codomain are also of unknown confidentiality. Using said annotations, one can consider the dynamic embedding $\lceil\!\lceil f \rceil\!\rceil$ and consider it as part of following program $P \triangleq \mathbf{let}\ f' = (\lceil\!\lceil f \rceil\!\rceil : \mathbb{Z}_? \longrightarrow_? \mathbb{Z}_? \Rightarrow \mathbb{Z}_H \longrightarrow_L \mathbb{Z}_L)\ \mathbf{in}\ f'\ 1$. The above-cited gradualizations all evaluate P to a result of the form $0 : \mathbb{Z}_H \Rightarrow \mathbb{Z}_? \Rightarrow \mathbb{Z}_L$, which produces a runtime security cast error, thereby violating RDE.

Admittedly, the possibility of designing the runtime semantics of a gradual security language such that the robust dynamic embedding criterion is satisfied is an open issue, considering that noninterference is a hyperproperty. While one can devise strategies to achieve this goal, supporting RDE for any static security type is a challenging, potentially out of reach, objective.

2.4 Sensitivity typing

As a last example, we look at sensitivity type systems, which control the *sensitivity* of expressions, *i.e.,* how much the output of a function can change if its input changes [26, 32, 36].

A Semantically Typed Function. The semantic contract for the type τ in Table 1 is called *metric preservation* [32], denoting a bound on the output variation of a function given an input variation. Here, the type $(x : \mathbb{Z}) \rightarrow^{1x} \mathbb{Z}$ specifies a 1-sensitive function, so the semantic contract for f is that for any x, y, $|f(x) - f(y)| \leq |x - y|$, which can easily be established for f by case analysis on whether the arguments have the same sign. This function is not syntactically well-typed, because most sensitivity typing conservatively assuming branching on a sensitive value as ∞-sensitive, because both branches could provide arbitrarily distant results.

Gradualizations. In a gradual sensitivity language such as GSoul [4], imprecision is introduced exclusively at the level of sensitivity annotations. So, f can be dynamically embedded to obtain $\vdash \lceil\!\lceil f \rceil\!\rceil : (x : \mathbb{Z}) \rightarrow^{?x} \mathbb{Z}$, an integer function of *unknown* sensitivity. The gradual program $P \triangleq \mathbf{let}\, f' = \lceil\!\lceil f \rceil\!\rceil : (x : \mathbb{Z}) \rightarrow^{?x} \mathbb{Z} \Rightarrow^{\ell} (x : \mathbb{Z}) \rightarrow^{1x} \mathbb{Z}\ \mathbf{in}\, f'\ 1$ reduces to a term of the form $1 : \infty x \Rightarrow ?\,x \Rightarrow 1x$, which fails as the dynamic semantics of conditionals inherit the conservative nature of the static typing of conditionals. Like for noninterference (§2.3), metric preservation is a hyperproperty, so developing any gradualization that satisfies RDE might be extremely challenging or plain impossible without imposing some severe restrictions.

3 Robust Dynamic Embedding vs Other Criteria

The literature on gradual typing already offers a large range of meta-theoretical properties to guide and evaluate the design of gradual languages. We now explain why the proposed robust dynamic embedding criterion is not implied by existing properties, but rather complements them. In this section we focus on the most established criteria, and discuss the remaining ones in §7.2.1.

Type safety and type soundness. Type safety in a gradual language means that well-typed expressions either reduce to a value, diverge, or raise a cast error at runtime [33]. As such, type safety does not address when a cast *cannot* fail. When gradualizing languages where static types enforce more than just safety, one usually establishes a more involved soundness theorem—for instance parametricity, noninterference, or metric preservation—formalized by a type-indexed logical relation [2, 4, 29, 37, 38]. Similar to type safety, these semantics contracts are relaxed to always permit the possibility of a runtime error. As a result, they are too coarse to address any particular circumstance in which a cast *cannot* fail.

Dynamic embedding. As explained earlier, the dynamic embedding criterion [34] only talks about the embedding of closed dynamic programs without considering their interaction with other gradual code. RDE refines this criterion via the lense of robustness (Definition 1) when ascribed a semantically-satisfied static type.

Gradual guarantee and graduality. The gradual guarantee states that typing and reduction are monotone with respect to imprecision [34]. That is, if the type annotations of a gradually-typed expression are made less precise, the resulting expression should still be well-typed, and cannot fail more that its more precise version. Compared to RDE, this theorem relates two *gradual* expressions, as opposed to a semantically-typed *dynamic* expression and its ascribed dynamic embedding in the gradual language. The examples described in §2 all satisfy the gradual guarantee, yet not necessarily RDE. The gradual guarantee captures the expected smoothness of the static-to-dynamic spectrum afforded by gradual typing, whereas RDE focuses on importing well-behaved dynamic components.

The gradual guarantee may *partially* imply the robust dynamic embedding criterion *when restricted in two important dimensions.* First, the gradual guarantee does not specify *where* a novel

runtime error may be introduced when making types more precise. Therefore, only the special case Corollary 1 might be provable. The second restriction is more fundamental and becomes clear when trying to prove Corollary 1 with the gradual guarantee. Given a semantically well-typed expression $\vDash e : \tau$, one might start by making the *gradual* expression $\llbracket e \rrbracket : ? \Rightarrow \tau$ more precisely typed with the objective to end up with a statically typed expression. As the translation of the latter would not contain casts, $C_s[\llbracket e \rrbracket : ? \Rightarrow \tau]$ (where C_s is an arbitrary precisely-typed context) would not either, and hence, cannot throw a runtime error. However, such a migration (starting from $\llbracket e \rrbracket : ? \Rightarrow \tau$) can reach a statically-typed expression *only if* e is deemed syntactically well-typed at type τ by the static type system. RDE is more interestingly applicable to scenarios where the imported term is not syntactically well-typed at the static type it satisfies semantically.

New and Ahmed [28] generalize the dynamic gradual guarantee to a semantic property named graduality, and prove it by relying on the observation that casts between a less and a more precise type should form embedding-projection pairs according to observational error approximation. The relation to RDE is similar to that of the gradual guarantee. §7 discusses a more technical example to demonstrate how RDE can still be violated, even though casts form embedding-projection pairs.

Blame theorem. The blame theorem [41] states that statically typed regions of a program can never be blamed for runtime type errors. More precisely, characterizes whether and, if so, how a cast may fail, considering only the involved types, not the specific expression that is being cast. For instance, a cast from $? \rightarrow \mathbb{B}$ to $\mathbb{Z} \rightarrow ?$ will never fail, as it merely restricts the values that can be applied to the function that is being cast to integers, and if the function successfully computes a final value, the resulting value must be a boolean and can always be considered at the unknown type. Technically, Wadler and Findler [41] develop a blame calculus that allows every cast to fail with two polarities, denoting whether the expression or the context is to blame. For example, a cast $\mathbb{Z} \rightarrow \mathbb{B} \Rightarrow ? \rightarrow \mathbb{B}$ can only fail by blaming its context if the latter does not provide an integer.

While there is some overlap with the criterion we propose, the blame theorem does not imply RDE. Consider, for instance, the cast $\llbracket g \rrbracket : ? \Rightarrow ((\mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}) \rightarrow \mathbb{B})$ (Equation 2). The blame theorem would only indicate that this cannot fail while blaming the context. It does not state that said cast cannot fail blaming $\llbracket g \rrbracket$, a fact that can only be derived from knowing that $\llbracket g \rrbracket$ is semantically well-typed in the first place. In that sense, RDE is a more specific term-level property, compared to the type-level blame theorem.

4 Setting the Scene: Gradual Types and Dynamic Embedding

This section and the following two describe a concrete instance in which the robust dynamic embedding (RDE) criterion is formally established. A roadmap of these sections is provided in Figure 1, mentioning the main definitions and (links to) the main results.

This section focuses on the two languages under study: a dynamic language λ_d (§4.1) and a gradually typed language λ_g (§4.2), together with a dynamic embedding (§4.3). While both languages are standard, two key variations are adopted to support the proof of RDE.

(1) *Lazy arrow downcasts.* In λ_g, downcasts on values of function types are deferred until the function is applied. This is necessary for RDE to hold, as discussed in §7.1.
(2) *Labeled dynamic semantics.* In λ_d, every destructor is annotated with a label, and runtime errors propagate this label during execution.

These decisions make the following definitions possible:

(1) An alternative semantics of the gradual language—aside from the usual one based on casts—by an emulation function into the dynamic language (§4.2.3). casts in λ_g are translated as ordinary functions that preserve the original labels (Proposition 1).

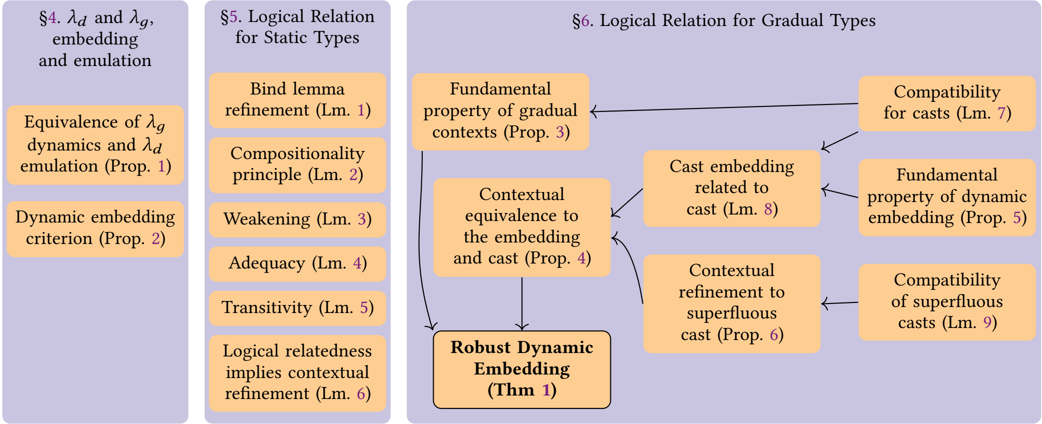

Fig. 1. Roadmap of technical sections. Key dependencies between results of §6 are indicated by arrows.

(2) A slightly more precise version of the usual dynamic embedding from λ_d to λ_g (§4.3). For every destructor, the introduced downcast inherits the original label of the destructor.
(3) A logical relation to prove *label-aware* refinements in λ_d (§5).

These definitions significantly simplify the proof effort described in §6 and mechanized in Rocq.

4.1 The Dynamic Language λ_d

λ_d is a call-by-value left-to-right lambda calculus that throws errors on terms that would otherwise get stuck, such as applying a boolean to an integer. As is natural for a dynamic language, errors also report the source location of the corresponding destructor that was problematic during execution. The syntax and dynamic semantics of λ_d are presented in Figure 2.

Syntax. The syntax of expressions is fairly standard: unit values, a sequence operator, booleans and their corresponding destructors, integers and binary operations, lambdas and applications, pairs and projections, injections and case analysis, and finally, the error term. Note that every destructor in λ_d, as well as the error term, carries a label ℓ.

Dynamics semantics. The dynamic semantics of λ_d is defined in terms of head steps and evaluation contexts K. Evaluation contexts are defined as usual, with context hole noted $\square$, and are extended to carry labels. A head step, denoted by $e_1 \to e_2$, is either a successful step (where the labels are completely irrelevant), or one that that throws an error if the destructor acts upon a value of a wrong shape. Moreover, the error term inherits the label of the corresponding destructor.

An evaluation step, denoted by $e_1 \mapsto e_2$, is either a head-step under an evaluation context or an error under a non-empty evaluation context that gets propagated.

4.2 The Gradual Language λ_g

We now describe the static and dynamic semantics of the gradual language λ_g. In addition to the standard cast-based reduction, we describe an alternative semantics via emulation in λ_d (§4.2.3), which is useful to prove RDE.

4.2.1 Static Semantics. The static semantics of λ_g is quite standard and defined in Figure 3. The expressions mirror the definition of λ_d but with two differences. First, destructors do not carry labels because typing excludes gradual programs where destructors could get stuck. Secondly,

$$
\begin{array}{rrl}
\text{expressions} & e ::= & () \mid e ;^{\ell} e \mid b \mid \textbf{if}^{\ell}\ e\ \textbf{then}\ e\ \textbf{else}\ e \mid z \mid e \circledcirc^{\ell} e \mid \lambda x.\, e \mid (e\ e)^{\ell} \mid (e, e) \mid \pi_1^{\ell}\ e \mid \\
& & \pi_2^{\ell}\ e \mid \textbf{inl}\ e \mid \textbf{inr}\ e \mid \textbf{case}^{\ell} e\ \textbf{of}\ (\textbf{inl}\ x \Rightarrow e \mid \textbf{inr}\ x \Rightarrow e) \mid \textbf{error}\ \ell \\
\text{values} & v ::= & () \mid b \mid z \mid \lambda x.\, e \mid (v, v) \mid \textbf{inl}\ v \mid \textbf{inr}\ v \\
\text{eval. contexts} & K ::= & \square \mid K ;^{\ell} e \mid \textbf{if}^{\ell} K\ \textbf{then}\ e\ \textbf{else}\ e' \mid (K\ e)^{\ell} \mid (v\ K)^{\ell} \mid (K, e) \mid (v, K) \mid \pi_1^{\ell} K \mid \pi_2^{\ell} K \mid \ldots
\end{array}
$$

$\boxed{e \rightarrow e}$

$$
\begin{array}{rcl}
((); ^{\ell} e) & \rightarrow & e \\
(z_1 \circledcirc^{\ell} z_2) & \rightarrow & [\![\circledcirc]\!](z_1, z_2) \\
(\textbf{if}^{\ell}\ \textbf{true}\ \textbf{then}\ e_1\ \textbf{else}\ e_2) & \rightarrow & e_1 \\
(\textbf{if}^{\ell}\ \textbf{false}\ \textbf{then}\ e_1\ \textbf{else}\ e_2) & \rightarrow & e_2 \\
((\lambda x.e)\ v)^{\ell} & \rightarrow & e[x \mapsto v] \\
\pi_1^{\ell}\ (v_1, v_2) & \rightarrow & v_1 \\
\pi_2^{\ell}\ (v_1, v_2) & \rightarrow & v_2 \\
\textbf{case}^{\ell} \textbf{inl}\ v\ \textbf{of}\ (\textbf{inl}\ x \Rightarrow e \mid \ldots) & \rightarrow & e[x \mapsto v] \\
\textbf{case}^{\ell} \textbf{inr}\ v\ \textbf{of}\ (\ldots \mid \textbf{inr}\ x \Rightarrow e) & \rightarrow & e[x \mapsto v]
\end{array}
$$

$$
\begin{array}{rcll}
(v ;^{\ell} e) & \rightarrow & \textbf{error}\ \ell & \text{if } v \notin \mathbb{1} \\
(v_1 \circledcirc^{\ell} v_2) & \rightarrow & \textbf{error}\ \ell & \text{if } v_1 \notin \mathbb{Z} \vee v_2 \notin \mathbb{Z} \\
(\textbf{if}^{\ell}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2) & \rightarrow & \textbf{error}\ \ell & \text{if } v \notin \mathbb{B} \\
(v_1\ v_2)^{\ell} & \rightarrow & \textbf{error}\ \ell & \text{if } v_1 \neq \lambda x.\, e \\
\pi_1^{\ell}\ v & \rightarrow & \textbf{error}\ \ell & \text{if } v \neq (v_1, v_2) \\
\pi_2^{\ell}\ v & \rightarrow & \textbf{error}\ \ell & \text{if } v \neq (v_1, v_2) \\
\textbf{case}^{\ell} v\ \textbf{of}\ \ldots & \rightarrow & \textbf{error}\ \ell & \text{if } v \neq \textbf{in*}\ w
\end{array}
$$

$\boxed{e \mapsto e}$

$$
\frac{e_1 \rightarrow e_2}{K[e_1] \mapsto K[e_2]} \qquad \frac{K \neq []}{K[\textbf{error}\ \ell] \mapsto \textbf{error}\ \ell}
$$

Fig. 2. Syntax and dynamic semantics of λ_d *[dyn_lang/definition.v]*

expressions include the cast primitive, $e : \tau \Rightarrow^{\ell} \tau'$, which casts a given expression e of type τ to type τ', and report the label ℓ if failing, as described in §1.

Types include base types, sums, products, arrows, and finally the unknown type. The typing judgment $\Gamma \vdash e : \tau$ means that expression e has type τ under type environment Γ. The type rules are mostly standard except for the fact that consistency is always explicit through the use of casts. Recall that consistency captures plausible equality [14], and is defined following Siek et al. [34].

4.2.2 Dynamics. The dynamics of λ_g are standard call-by-value left-to-right evaluation. The most interesting cases is the evaluation of casts. We follow a completely standard approach as per Siek et al. [34], with the only deviation being that casts to function types always happen lazily. This design choice is dictated by the robust dynamic embedding criterion, as eager function casts break the criterion unless the dynamic language is adjusted (see §7.1 for a more detailed discussion).

Rules R1 and R2 handle the elimination of redundant casts. Rules R3 and R4 start from a term $v : G_1 \Rightarrow^{\ell} ? \Rightarrow^{\ell'} G_2$, a value v that is first upcast from a type G_1 to ? and then downcast from ? to G_2. Rule R3 handles the case of successfully combining casts, when $G_1 = G_2$, eliminating both casts. Rule R4 captures incompatible combinations ($G_1 \neq G_2$) and reduces the term to an error with blame label ℓ'. For instance $1 : \mathbb{Z} \Rightarrow^{\ell} ? \Rightarrow^{\ell'} \mathbb{B}$ represents an upcast from $\mathbb{Z}$ to ? followed by a downcast from ? to $\mathbb{B}$, which reduces to an error because $\mathbb{Z} \neq \mathbb{B}$.

The side conditions of Rules R3 and R4 ensure that any cast to a function type is treated as a value, and only reduces when applied. Rules R5, R6, and R7 specify this lazy casting semantics. Rule R5 handles the case where the value being applied is upcast and downcast to and from the ground function type $? \rightarrow ?$. In this case, the casts are eliminated. Rule R6 addresses the case where the value is upcast from a type that is not a ground function type. In this case, the application reduces to an error. Finally, Rule R7 covers the case of casting between function types, where both the argument and the result are ascribed appropriately.

$$
\begin{array}{rrl}
\text{expressions} & e ::= & () \mid e;e \mid b \mid \textbf{if } e \textbf{ then } e \textbf{ else } e \mid z \mid e \circledcirc e \mid \lambda x.\, e \mid e\ e \mid (e,e) \mid \pi_1\, e \mid \\
& & \pi_2\, e \mid \textbf{inl } e \mid \textbf{inr } e \mid \textbf{case } e \textbf{ of } (\textbf{inl } x \Rightarrow e \mid \textbf{inr } x \Rightarrow e) \mid e : \tau \Rightarrow^{\ell} \tau \mid \textbf{error } \ell \\
\text{base types} & B ::= & \mathbb{1} \mid \mathbb{B} \mid \mathbb{Z} \\
\text{gradual types} & \tau ::= & B \mid \tau + \tau \mid \tau \times \tau \mid \tau \to \tau \mid ? \\
\text{ground types} & G ::= & B \mid ?+? \mid ?\times? \mid ? \to ? \\
\text{values} & v ::= & () \mid b \mid z \mid \lambda x.\, e \mid (v,v) \mid \textbf{inl } v \mid \textbf{inr } v \mid \\
& & v : G \Rightarrow^{\ell} ? \mid v : ? \Rightarrow^{\ell} ? \to ? \mid v : \tau_1 \to \tau_2 \Rightarrow^{\ell} \tau_3 \to \tau_4 \\
\text{eval. contexts} & K ::= & \square \mid K;e \mid \textbf{if } K \textbf{ then } e \textbf{ else } e' \mid (K\ e) \mid v\ K \mid (K,e) \mid (v,K) \mid \pi_i K \mid \ldots \mid K : \tau \Rightarrow^{\ell} \tau' \\
\text{contexts} & C ::= & \square \mid C;e \mid e;C \mid \textbf{if } C \textbf{ then } e \textbf{ else } e' \mid \textbf{if } e \textbf{ then } C \textbf{ else } e' \mid \\
& & \textbf{if } e \textbf{ then } e' \textbf{ else } C \mid \lambda x.C \mid C\ e \mid e\ C \mid C : \tau \Rightarrow^{\ell} \tau \mid \ldots
\end{array}
$$

$\boxed{\Gamma \vdash e : \tau}$

$$
\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}
\qquad
\frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2}
\qquad
\frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \sim \tau_2}{\Gamma \vdash e : \tau_1 \Rightarrow^{\ell} \tau_2 : \tau_2}
$$

$\boxed{\tau \sim \tau}$

$$
\tau \sim ? \qquad ? \sim \tau \qquad B \sim B \qquad
\frac{\tau_1 \sim \tau_3 \qquad \tau_2 \sim \tau_4}{\tau_1 \to \tau_2 \sim \tau_3 \to \tau_4}
\qquad
\frac{\tau_1 \sim \tau_3 \qquad \tau_2 \sim \tau_4}{\tau_1 + \tau_2 \sim \tau_3 + \tau_4}
\qquad
\frac{\tau_1 \sim \tau_3 \qquad \tau_2 \sim \tau_4}{\tau_1 \times \tau_2 \sim \tau_3 \times \tau_4}
$$

$$v : B \Rightarrow^{\ell} B \to v \tag{R1}$$

$$v : ? \Rightarrow^{\ell} ? \to v \tag{R2}$$

$$v : G \Rightarrow^{\ell} ? \Rightarrow^{\ell'} G \to v \quad \text{if } G \neq ? \to ? \tag{R3}$$

$$v : G_1 \Rightarrow^{\ell} ? \Rightarrow^{\ell'} G_2 \to \textbf{error } \ell' \quad \text{if } G_2 \neq ? \to ?, G_1 \neq G_2 \tag{R4}$$

$$(v : ? \to ? \Rightarrow^{\ell} ? \Rightarrow^{\ell'} ? \to ?)\ w \to v\ w \tag{R5}$$

$$(v : G \Rightarrow^{\ell} ? \Rightarrow^{\ell'} ? \to ?)\ w \to \textbf{error } \ell' \quad \text{if } G \neq ? \to ? \tag{R6}$$

$$(v : \tau_1 \to \tau_2 \Rightarrow^{\ell} \tau_3 \to \tau_4)\ w \to (v\ (w : \tau_3 \Rightarrow^{\ell} \tau_1)) : \tau_2 \Rightarrow^{\ell} \tau_4 \tag{R7}$$

$$(v_1, v_2) : \tau_1 \times \tau_2 \Rightarrow^{\ell} \tau_1' \times \tau_2' \to (v_1 : \tau_1 \Rightarrow^{\ell} \tau_1', v_2 : \tau_2 \Rightarrow^{\ell} \tau_2') \tag{R8}$$

$$(\textbf{inl } v) : \tau_1 + \tau_2 \Rightarrow^{\ell} \tau_1' + \tau_2' \to \textbf{inl } (v : \tau_1 \Rightarrow^{\ell} \tau_1') \tag{R9}$$

$$v : \tau \Rightarrow^{\ell} ? \to v : \tau \Rightarrow^{\ell} G \Rightarrow^{\ell} ? \quad \text{if } \tau \sim G, \tau \neq G, \tau \neq ? \tag{R10}$$

$$v : ? \Rightarrow^{\ell} \tau \to v : ? \Rightarrow^{\ell} G \Rightarrow^{\ell} \tau \quad \text{if } \tau \sim G, \tau \neq G, \tau \neq ? \tag{R11}$$

$$
\frac{e \to e'}{K[e] \mapsto K[e']}
\qquad
\frac{K \neq \square}{K[\textbf{error } \ell] \mapsto \textbf{error } \ell}
$$

Fig. 3. Semantics of λ_g (other standard rules omitted) *[cast_calc/definition.v]*, *[std.v]*

Rule R8 handles the case where a cast is applied to a pair. The cast is decomposed into two separate casts, one for each component of the pair. Rule R9 handles the case where a cast is applied to a left injection. As specified in §2.1, only the underlying value is cast between the left-hand components of the sum types, while the right-hand components are ignored. Finally, Rules R10

$\boxed{C^\ell_{G\Rightarrow ?}, C^\ell_{?\Rightarrow ?}}$

$$C^\ell_{G\Rightarrow ?} \triangleq \lambda x.\, x$$
$$C^\ell_{?\Rightarrow ?} \triangleq \lambda x.\, x$$

$\boxed{C^\ell_{?\Rightarrow G}}$

$$C^\ell_{?\Rightarrow \mathbb{B}} \triangleq \lambda x.\, \mathbf{if}^\ell\ x\ \mathbf{then\ true\ else\ false}$$
$$C^\ell_{?\Rightarrow ?\times ?} \triangleq \lambda x.\, (\pi^\ell_1\ x, \pi^\ell_2\ x)$$
$$C^\ell_{?\Rightarrow ?\to ?} \triangleq \lambda x.\, (\lambda a.\, (x\ a)^\ell)$$

$\boxed{C^\ell_{\tau_1\Rightarrow\tau_2}(\tau_1 \neq ?, \tau_2 \neq ?)}$

$$C^\ell_{\tau_1\to\tau_2\Rightarrow\tau_3\to\tau_4} \triangleq \lambda f.\, \lambda x.\, (C^\ell_{\tau_2\Rightarrow\tau_4}\ (f\ (C^\ell_{\tau_3\Rightarrow\tau_1}\ x)))$$
$$C^\ell_{\tau_1\times\tau_2\Rightarrow\tau_3\times\tau_4} \triangleq \lambda p.\, (C^\ell_{\tau_1\Rightarrow\tau_3}\ (\pi_1\ p),\ C^\ell_{\tau_2\Rightarrow\tau_4}\ (\pi_2\ p))$$
$$C^\ell_{\tau_1+\tau_2\Rightarrow\tau_3+\tau_4} \triangleq \lambda s.\, \mathbf{case}\ s\ \mathbf{of}\ (\mathbf{inl}\ x \Rightarrow \mathbf{inl}\ (C^\ell_{\tau_1\Rightarrow\tau_3}\ x) \mid \mathbf{inr}\ x \Rightarrow \mathbf{inr}\ (C^\ell_{\tau_2\Rightarrow\tau_4}\ x))$$

$\boxed{C^\ell_{\tau\Rightarrow ?}(\tau \neq G)}$

$$C^\ell_{\tau\times\tau'\Rightarrow ?} \triangleq C^\ell_{?\times ?\Rightarrow ?} \circ C^\nu_{\tau\times\tau'\Rightarrow ?\times ?}$$
$$C^\ell_{\tau\to\tau'\Rightarrow ?} \triangleq C^\nu_{\tau\to\tau'\Rightarrow ?\to ?} \circ C^\ell_{?\to ?\Rightarrow ?}$$

$\boxed{C^\ell_{?\Rightarrow\tau}(\tau \neq G)}$

$$C^\ell_{?\Rightarrow\tau\times\tau'} \triangleq C^\ell_{?\times ?\Rightarrow\tau\times\tau'} \circ C^\ell_{?\Rightarrow ?\times ?}$$
$$C^\ell_{?\Rightarrow\tau\to\tau'} \triangleq C^{?\to ?}_{?\Rightarrow\tau\to\tau'} \circ C^\ell_{?\Rightarrow ?\to ?}$$

Fig. 4. Implementation of casts with λ_d functions *[cast']*

and R11 handle casts to or from the unknown type when the other type is not a ground type. In these cases, an intermediate cast to a ground type is introduced.

4.2.3 Alternative dynamics of λ_g by emulation in λ_d. Alternatively, we define an emulation function $\langle\!\langle _ \rangle\!\rangle : \lambda_g \longrightarrow \lambda_d$, with which the dynamics of a gradual program is given by that of its emulation in the dynamic language (formalized in *[trns]*). This emulation is mostly trivial, except for the case of casts, which get replaced by regular functions, for instance:

$$\langle\!\langle e\ e' \rangle\!\rangle \triangleq (\langle\!\langle e \rangle\!\rangle\ \langle\!\langle e' \rangle\!\rangle) \qquad \langle\!\langle \lambda x.\, e \rangle\!\rangle \triangleq \lambda x.\, \langle\!\langle e \rangle\!\rangle \qquad \langle\!\langle (e, e') \rangle\!\rangle \triangleq (\langle\!\langle e \rangle\!\rangle, \langle\!\langle e' \rangle\!\rangle) \qquad \langle\!\langle \pi_i\ e \rangle\!\rangle \triangleq \pi_i \langle\!\langle e \rangle\!\rangle$$

$$\langle\!\langle e : \tau \Rightarrow^\ell \tau' \rangle\!\rangle \triangleq (C^\ell_{\tau_1\Rightarrow\tau_2}\ \langle\!\langle e \rangle\!\rangle)$$

Note that on any destructor—apart from the cast primitive—the image of the emulation requires choosing some label. Because the typing rules of the gradual language impede that these destructors ever fail, the particular label chosen becomes irrelevant, and is therefore omitted in this article.

Specifically, a cast $e : \tau \Rightarrow^\ell \tau'$, is translated into a dynamic function written $C^\ell_{\tau\Rightarrow\tau'}$. The implementation of these functions are shown in Figure 4, and are chosen to closely resemble the dynamics in Figure 3. An upcast from a ground type to unknown is simply emulated by an identity function, as casting to the unknown type is always valid. Conversely, a downcast from the unknown type to a ground type G must check that the given value conforms to the expected form of G. This is done by performing an η-expansion of the value under consideration, which behaves as the identity function if the value is valid, and throws an error with the specified label ℓ if not and $G \neq ? \to ?$.

Like in the original dynamics (Figure 3), a downcast to $? \longrightarrow ?$ happens lazily. For example, applying such a downcast with label ℓ to an integer 6 results in the expression $C^\ell_{?\Rightarrow ?\to ?}\ 6$, which evaluates to a function $\lambda a.\, (6\ a)^\ell$. This function throws an error with label ℓ when applied.

The emulation of casts between two types of the same top-level type constructor are defined inductively by destructing the term, applying casts on subterms, and construct a term back in the case of sum and product types. Note the contravariance in the argument when dealing with function types. Casts between the unknown and non-ground types are obtained by introducing the ground type with the corresponding top-level type constructor as an intermediate cast.

Both versions of the dynamic semantics correspond on closed programs (proven using a classical simulation argument):[3]

Proposition 1 (Equivalence on closed programs *[equi_eval]*). *Given any* $\cdot \vdash e : \tau$. *Evaluating* e *with the* λ_g *dynamics (Figure 3) and* $\langle\langle e \rangle\rangle$ *with the* λ_d *dynamics, we have: either both diverge, both error* with the same label*, or both terminate to* equivalent *values.*[4]

4.3 Dynamic Embedding from λ_d to λ_g

Having defined both the dynamic and gradual language, the dynamic embedding from λ_d into λ_g is defined as follows (some excerpts) (formalized in *[dyn_emb]*).

$$\lceil\!\lceil b \rceil\!\rceil \triangleq b : \mathbb{B} \Rightarrow ? \qquad \lceil\!\lceil \mathbf{if}^{\ell}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rceil\!\rceil \triangleq \mathbf{if}\ (\lceil\!\lceil e_1 \rceil\!\rceil : ? \Rightarrow^{\ell} \mathbb{B})\ \mathbf{then}\ \lceil\!\lceil e_2 \rceil\!\rceil\ \mathbf{else}\ \lceil\!\lceil e_3 \rceil\!\rceil$$

$$\lceil\!\lceil x \rceil\!\rceil \triangleq x \qquad \lceil\!\lceil \lambda x.\, e \rceil\!\rceil \triangleq (\lambda x.\, \lceil\!\lceil e \rceil\!\rceil) : ? \to ? \Rightarrow ? \qquad \lceil\!\lceil (e_1\ e_2)^{\ell} \rceil\!\rceil \triangleq (\lceil\!\lceil e_1 \rceil\!\rceil : ? \Rightarrow^{\ell} ? \to ?)\ \lceil\!\lceil e_2 \rceil\!\rceil$$

The dynamic embedding is defined by structural induction on expressions, establishing the invariant of well-typedness at the unknown type. Constructors are ascribed to the unknown type, while destructors are ascribed to the bare minimum required to ensure typing. Note that the label of a destructor is used for the corresponding cast in the embedding.

We can now establish the dynamic embedding criterion as formulated by Siek et al. [34].

Proposition 2 (*[dyn_emb_typed]*). *Consider an arbitrary well-scoped* λ_d *expression* e *with free variables* $x_1, \ldots, x_n$, *then* $x_1 : ?, \ldots, x_n : ? \vdash \lceil\!\lceil e \rceil\!\rceil : ?$.

Moreover, given a closed dynamic expression e*, either both* e *and* $\lceil\!\lceil e \rceil\!\rceil$ *diverge, both terminate to equivalent values, or both fail with the same label.*

Compared to the formulation of Siek and Taha [33], we additionally require that in the error case, both expressions report the same label—a minor but necessary refinement to formalize RDE.

5 Semantic Typing for the Dynamic Language

The main purpose of this section is to define semantic typing for λ_d at *static* types, characterizing dynamic programs that run safely (do not throw errors) as if they were of a static type. §5.1 introduces the logical relation with which we will define this semantic typing and §5.2 follows with a number of interesting properties.

Instead of defining a completely canonical logical relation however, we enhance it as follows:

(1) The relation models not only error-free terms but also terms that may fail on a specific, permitted set of labels. This parameterization allows it to relate programs with controlled errors, which is crucial for proving RDE: only errors caused by the gradual context are allowed, while errors from the embedded expression are disallowed.
(2) Instead of a unary logical predicate, we define a binary relation allowing relational reasoning (semantic typing is then be defined as self-relatedness).

At the end of §5.2, we shall summarize the insights of these lemmas by noting that the logical relation models a novel notion of *logical label-aware contextual refinement* formalized in Definition 6.

[3] Most important definitions/theorems are *[hyperlinked]* for easy access to the relevant part in the Rocq formalization.
[4] Intuitively, a gradual value v is *equivalent* to a dynamic value v' means that v corresponds to the evaluation of $\langle\langle v' \rangle\rangle$, modulo minor syntactic differences. For instance, $5 : \mathbb{Z} \Rightarrow^{\ell} ?$ is equivalent to 5. The precise invariant used to relate such values is formalized in *[Invariant]* but not conceptually relevant for RDE.

As can be seen in Figure 1, this notion (together with its associated lemmas Lemma 6 and Lemma 5) will serve as the conceptual linchpin to structure our proof in §6.

Up front, we may also motivate the aforementioned enhancements intuitively by noting that the novel logical relation allows us to precisely specify:

- (the emulation of) arbitrary gradually-typed code (which of-course, may throw errors with the label from their casts)
- the relation of arbitrary dynamic code (which of course may error) and (the emulation of) its dynamic embedding

As can also be seen in the same figure, the formal properties (see Proposition 3 and Proposition 5 that correspond to these intuitive descriptions will also be very useful in the proof in §6.

5.1 Formal Definition of Logical Relations & Semantic Typing

The complete definition is shown in Figure 5. As mentioned, it is fairly standard except for two additions: binary form of the relation and the ability to allow only errors with some particular set of labels.

Top-down Intuitive Account. Before going over the formal definition, we present an intuitive account of the statement (on closed terms), putting emphasis on the novel parts. Given a *binary label predicate* Ψ, two dynamic terms e and e', and a static type τ, we have that $\vDash e \preccurlyeq_{\Psi} e' : \tau$ models a refinement as follows (roughly):

- if e terminates to a value, so does e' with the resulting values related at τ (to be defined)
- if e throws an error with label ℓ, so does e' with label ℓ' say and $\Psi(\ell, \ell')$
- (no restriction on e' if e diverges)

Hence, Ψ encodes a predicate of *allowed labels* for this refinement. A key aspect of this intuitive presentation is that relatedness is compositional: it is preserved under the join of label predicates. This is not only conceptually important, but also plays a direct role in motivating the final definition of relatedness in Equation L4.

Compositionality Expectation. Consider the dynamic expressions e and e' related at the higher-order type $(\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Z}$ under the predicate Ψ, i.e. $\vDash e \preccurlyeq_{\Psi} e' : (\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Z}$. As mentioned, e and e' themselves can only error with labels in Ψ. However, upon interaction with related arguments at $(\mathbb{Z} \to \mathbb{Z})$ under a *different* predicate Ψ', e.g. $\vDash t \preccurlyeq_{\Psi'} t' : \mathbb{Z} \to \mathbb{Z}$, the resulting applications may error with labels outside of Ψ as long as they stem from the arguments. That is, we have the following:

$$\frac{\vDash e \preccurlyeq_{\Psi} e' : (\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Z} \qquad \vDash t \preccurlyeq_{\Psi'} t' : (\mathbb{Z} \to \mathbb{Z})}{\vDash (e\ t) \preccurlyeq_{\Psi \sqcup \Psi'} (e'\ t') : \mathbb{Z}} \tag{4}$$

The resulting applications are related at $\Psi \sqcup \Psi'$, defined as follows.

Definition 2 (Pointwise Join *[join_LabelRel]*)**.** Predicates on labels form a lattice with the $\sqsubseteq$-operator as a pre-order. The join operator $\Psi \sqcup \Psi'$, is defined by point-wise disjunction.

$$\Psi \sqcup \Psi' \triangleq \forall \ell, \ell'.\ \Psi(\ell, \ell') \lor \Psi'(\ell, \ell')$$

The formal definition of the logical relation is unpacked in the following paragraphs.

Weakest precondition. Equation L1 presents the definition of weakest-precondition, which takes an expression e, a *unary value predicate* ϕ (encoding appropriateness at a particular type), a *unary label predicate* ψ encoding which labels are permitted, and returns a predicate on expressions. Intuitively, given some expression e, wp e $\{\phi\}\{\psi\}$ encodes that if e returns a value, it must satisfy

Related Values [valrel_typed] ·

$$\mathcal{V}_\Psi[\![\tau]\!] : Val \to Val \to siProp$$

$$\mathcal{V}_\Psi[\![\mathbb{1}]\!](v, v') \triangleq v = v' = ()$$

$$\mathcal{V}_\Psi[\![\mathbb{B}]\!](v, v') \triangleq \exists b.\, v = v' = b$$

$$\mathcal{V}_\Psi[\![\mathbb{Z}]\!](v, v') \triangleq \exists z.\, v = v' = z$$

$$\mathcal{V}_\Psi[\![\tau_1 + \tau_2]\!](v, v') \triangleq \bigvee_{i \in \{\mathbf{l}, \mathbf{r}\}} \exists w, w'.\, v = \mathbf{in}i\ w \wedge v' = \mathbf{in}i\ w' \wedge \mathcal{V}_\Psi[\![\tau_i]\!](w, w')$$

$$\mathcal{V}_\Psi[\![\tau_1 \times \tau_2]\!](v, v') \triangleq \left| \begin{array}{l} \exists v_1, v_2, v_1', v_2'.\, v = (v_1, v_2) \wedge v' = (v_1', v_2') \wedge \\ \quad \mathcal{V}_\Psi[\![\tau_1]\!](v_1, v_1') \wedge \mathcal{V}_\Psi[\![\tau_2]\!](v_2, v_2') \end{array} \right.$$

$$\mathcal{V}_\Psi[\![\tau \to \tau']\!](v, v') \triangleq \left| \begin{array}{l} \exists e, e'.\, v = \lambda x.\, e \wedge v' = \lambda x.\, e' \wedge \\ \quad \forall w, w'.\, \mathcal{V}_\Psi[\![\tau]\!](w, w') \Rightarrow \mathsf{rfn}\ (\mathcal{V}_\Psi[\![\tau']\!])\ \Psi\ (e[w/x], e'[w'/x]) \end{array} \right.$$

Related Expressions [wp], [rfn], [exprel_typed] ·

$$\mathsf{wp} : Expr \to (Val \to siProp) \to (\mathcal{L} \to Prop) \to siProp \tag{L1}$$

$$\mathsf{wp}\ e\ \{\phi\}\{\psi\} \triangleq \left| \begin{array}{l} (e\ a\ value \wedge \phi(e)) \vee \\ (\exists e'.\, (e \mapsto^p e') \wedge \triangleright\, \mathsf{wp}\ e'\ \{\phi\}\{\psi\}) \vee \\ (\exists \ell.\, \mho_\ell(e) \wedge \psi(\ell))\ where\ \mho_\ell(e) \triangleq \exists K.\, (e = K[\mathbf{error}\ \ell] \vee e \mapsto K[\mathbf{error}\ \ell]) \end{array} \right.$$

$$\mathsf{rfn} : (Val \to Val \to siProp) \to (\mathcal{L} \to \mathcal{L} \to Prop) \to Expr \to Expr \to siProp \tag{L2}$$

$$\mathsf{rfn}\ \Phi\ \Psi\ e\ e' \triangleq \mathsf{wp}\ e\ \left\{v.\, \exists v'.\, e' \mapsto^* v' \wedge \Phi(v, v')\right\}\left\{\ell.\, \exists \ell', e''.\, (e' \mapsto^* e'') \wedge \mho_{\ell'}(e'') \wedge \Psi(\ell, \ell')\right\}$$

$$\mathcal{E}_\Psi[\![\tau]\!] : (Expr \to Expr \to siProp) \tag{L3}$$

$$\mathcal{E}_\Psi[\![\tau]\!]\ e\ e' \triangleq \mathsf{rfn}\ \mathcal{V}_\Psi[\![\tau]\!]\ \Psi\ e\ e'$$

Logical Relation [open_exprel_typed] and Semantic Typing [sem_typed] ·

$$\Gamma \vDash e \preccurlyeq_\Psi e' : \tau \triangleq \forall \Psi', \Psi \sqsubseteq \Psi'. \forall \vec{v}, \vec{v}'. \Big(\bigwedge_{\tau_i \in \Gamma} \mathcal{V}_{\Psi'}[\![\tau_i]\!](\vec{v}_i, \vec{v}_i') \Big) \vdash \mathcal{E}_{\Psi'}[\![\tau]\!](e[\vec{x} \mapsto \vec{v}], e'[\vec{x} \mapsto \vec{v}']) \tag{L4}$$

$$\Gamma \vDash e : \tau \triangleq \Gamma \vDash e \preccurlyeq_\perp e : \tau\ where\ \perp(\ell, \ell') \triangleq False \tag{L5}$$

Fig. 5. Binary logical relation for semantic typing on static types

ϕ, and if it throws an error, its label must satisfy ψ (with no restriction if it diverges). It does so by maintaining a simple invariant upon pure evaluation steps, in which either of the following holds:

(1) It is value, and the post-condition ϕ holds
(2) It is able to take a pure step to e', and the same invariant holds on e'
(3) It is about to throw an error with a label satisfying ψ

Due to the apparent circularity in this invariant (its definition refers to itself without any decreasing argument), the logical relation is defined in terms of a step-indexed logic using the *later modality* $\triangleright$. The recursive occurrence in the second case may be read as wp e' $\{\phi\}\{\psi\}$ holds *one step later*. The use of explicit step-indexing or that of a step-indexed logic (as is the case here, using a small fragment of the Iris programming logic [21]) to successfully model cyclic features is not novel [3, 10]. In Figure 5, *siProp* is the step-indexed propositional universe, which embeds the standard (non-step-indexed) universe *Prop*. We refer the reader to Jung et al. [21] for details.

Related expressions. The definition of related expressions in Equation L3 is defined using a refinement operator rfn. This operator, defined in Equation L2, lifts a binary value predicate Φ (*e.g.,* appropriateness on values at some type) and a binary label predicate Ψ, to a binary one on expressions. The behavior of the first expression is enforced by the weakest-precondition predicate, and the behavior of the second expression is enforced to follow the behavior of the first one in a meaningful sense. Intuitively,

- if e terminates to a value say v, so does e' to v' say and $\Phi(v, v')$
- if e throws an error with label ℓ, so does e' with label ℓ' say and $\Psi(\ell, \ell')$
- (no restriction on e' if e diverges)

Related values. For each static type τ, we define the binary predicate $\mathcal{V}_\Psi[\![\tau]\!]$ (by induction). Two values are related at a base type if they are the same base value of that type. Two values are related at a sum type if they are corresponding injections of values that are related at the corresponding type. Two values are related at a product type if they are both pairs whose corresponding projections are both related at the corresponding types. Finally, two values are related at an arrow type if they are both lambdas, and if the substitutions of their binders with arbitrary related values at the domain type yield two expressions that are related at the codomain type.

Definition Predicate Indexed Logical Relation. The final definition of logical relatedness, given in Equation L4, is standard when disregarding the label-related components. Intuitively, if related values are substituted at the context, then the resulting pair of expressions should be related at the return type.

Given a static context Γ, a static type τ, and Ψ a binary label predicate, $\Gamma \vDash e \preccurlyeq_\Psi e' : \tau$ denotes that, for any predicate Ψ' that is "more permissible" than Ψ, formally defined as:

$$\Psi \sqsubseteq \Psi' \triangleq \forall \ell, \ell'. \Psi(\ell, \ell') \Rightarrow \Psi'(\ell, \ell') \tag{5}$$

the expressions obtained by substituting variables with values that are related at Γ under Ψ' are themselves logically related at type τ under Ψ'.

The quantification over binary label predicates. One might initially expect that the relation $\Gamma \vDash e \preccurlyeq_\Psi e' : \tau$ could be defined more directly as follows:

$$\forall \vec{v}, \vec{v}'. \Big(\bigwedge_{\tau_i \in \Gamma} \mathcal{V}_\Psi[\![\tau_i]\!](\vec{v}_i, \vec{v}'_i) \Big) \vdash \mathcal{E}_\Psi[\![\tau]\!](e[\vec{x} \mapsto \vec{v}], e'[\vec{x} \mapsto \vec{v}']) \tag{6}$$

However, this definition would not be suitable in our setting, because it fails to support compositional reasoning. In particular, it would prevent us from deriving results like the one in Equation 4 when the label predicates Ψ and Ψ' are different.

Semantic Typing. Finally, semantic typing is defined as follows (repeated from Equation L5).

Definition 3 (Semantic Typing *[sem_typed]*)**.** Given a static context Γ, a static type τ, and a dynamic term e, we say that e is semantically typed at Γ and τ, written $\Gamma \vDash e : \tau$, if we have:

$$\Gamma \vDash e \preccurlyeq_\bot e : \tau \textit{ where } \bot(\ell, \ell') \triangleq \textit{False} \tag{7}$$

That is, if e is self-related at Γ and τ at the trivially false predicate.

The quantification over arbitrary predicates (making the condition $\bot \sqsubseteq \Psi'$ redundant in Equation L4) includes the trivially false predicate. As a result, we immediately recover the notion that semantically typed programs do not error.

Example 2 (*[taut_sem_typed]*,*[example3_sem_typed]*)**.** Consider the dynamic function *taut* from §1, defined in λ_d using an untyped fixpoint operator to express recursion. We have proven that for each natural n, $\vDash$ *taut* n : $(\mathbb{B}_0 \to \ldots \to \mathbb{B}_n) \to \mathbb{B})$.

Moreover, the function f discussed in §2.1 is directly expressible in λ_d, and we have $\vDash f : (\mathbb{1} + \mathbb{B}) \to (\mathbb{1} + \mathbb{1})$.

Furthermore, the quantification over the label predicate makes the proof obligation for semantic typing appear slightly more complex compared to more canonical formulations found in the literature. This may raise the concern that such a formulation could complicate the task of establishing semantic typing for programs. However, this is not the case in practice, as demonstrated by the bind lemma at the beginning of §5.2, which is applied in Example 2.

5.2 Properties of Logical Relations

We begin by presenting a few basic properties of the logical relation defined in Figure 5. These properties lead to the notion of label-aware contextual refinement, introduced in Definition 6, along with the associated results Lemma 6 and Lemma 5. As shown in Figure 1, these lemmas will be used directly in the proof presented in §6.

Abstracting the interaction with foreign code. Ideally, reasoning about interactions with potentially error-throwing code should not increase the complexity of the logical relation. Fortunately, this interaction can be abstracted cleanly using the following refined bind lemma, which captures the essence of label-sensitive reasoning in the presence of partial behavior.

Lemma 1 (Bind lemma refinement *[rfn_bind']*)**.**

$$\frac{\mathsf{rfn}\ \Phi'\ \Psi\ e\ e' \qquad \forall v, v'.\ \Phi'(v, v') \Rightarrow \mathsf{rfn}\ \Phi\ \Psi\ K[v]\ K'[v']}{\mathsf{rfn}\ \Phi\ \Psi\ K[e]\ K'[e']}$$

Intuitively, when reasoning about foreign or external code that may throw an error, it is always safe to assume that it terminates normally. If it were to throw an error instead, then the proof obligations would trivially hold. This abstraction plays a role analogous to the standard bind lemma in settings that account for divergence: the reasoning assumes success, since failure leads to vacuously satisfied obligations.

Compositionality. Equation 4 can stated for arbitrary function types (*[compat_app]*).

$$\frac{\Gamma \vDash e_1 \preccurlyeq_{\Psi} e_1' : \tau \to \tau' \qquad \Gamma \vDash e_2 \preccurlyeq_{\Psi'} e_2' : \tau}{\Gamma \vDash (e_1\ e_2) \preccurlyeq_{\Psi \sqcup \Psi'} (e_1'\ e_2') : \tau'}$$

Remark 1. Notice, moreover, that the conclusion remains strong as the join operator "remembers" how labels can be linked together. For instance, if Ψ and Ψ' are "disjoint" (they do not share any labels), the join never links a label at the function type with a label at the argument type.

Remark 2. As $\bot \sqcup \bot = \bot$, semantically typed expressions are composed as usual.

We now generalize this result to a compositional lemma. As a preliminary step, we define logical relatedness on contexts as follows.

Definition 4 (*[ctx_rel_typed]*)**.** Given two well-scoped dynamic context C and C', logical relatedness at $\Gamma; \tau \Rightarrow \Gamma'; \tau'$ on a binary label predicate Ψ is defined as follows:

$$\vDash C \preccurlyeq_{\Psi} C' : (\Gamma; \tau) \Rightarrow (\Gamma'; \tau') \triangleq \left(\forall \Psi', \Psi \sqsubseteq \Psi', e, e'.\ \Gamma \vDash e \preccurlyeq_{\Psi'} e' : \tau \Rightarrow \Gamma' \vDash C[e] \preccurlyeq_{\Psi'} C'[e'] : \tau'\right)$$

With this definition in place, the compositionality lemma can be stated as follows.

Lemma 2 (Compositionality Principle *[rel_ctx_fill_expr]*).

$$\frac{\vDash C \preccurlyeq_{\Psi} C' : (\Gamma;\tau) \Rightarrow (\Gamma';\tau') \qquad \Gamma \vDash e \preccurlyeq_{\Psi'} e' : \tau}{\Gamma' \vDash C[e] \preccurlyeq_{\Psi \sqcup \Psi'} C'[e'] : \tau'}$$

Weakening. As intuitively expected, logically related expressions under some predicate Ψ remain to be related under any more permissible predicate Ψ'.

Lemma 3 (Weakening *[open_exprel_typed_weaken]*). Given any two binary predicates on labels Ψ and Ψ' where $\Psi \sqsubseteq \Psi'$, and any two related expressions $\Gamma \vDash e \preccurlyeq_{\Psi} e' : \tau$, then $\Gamma \vDash e \preccurlyeq_{\Psi'} e' : \tau$.

The proof of this lemma is trivial due to the quantification over all more-permissible predicates. Moreover, given Lemma 3, the proof of Lemma 2 follows naturally.

Remark 3. In §5.1, the quantification over more-permissible Ψ' predicates in Equation L4 was motivated by the need for Equation 4 to hold. Here, it is shown that the weakening principle, used in the proof of Lemma 2, crucially depends on this quantification. Without it, the property does not hold.

To illustrate a specific counter-example, consider the following "non-quantified" logical relations. An "error-excluding" logical relation is defined as follows:

$$\Gamma \vDash_{\not\mho} e \preccurlyeq e' : \tau \triangleq \left(\forall \vec{v}, \vec{v}'. \Big(\bigwedge_{\tau_i \in \Gamma} \mathcal{V}_{\bot}[\![\tau_i]\!](\vec{v}_i, \vec{v}'_i) \Big) \vdash \mathcal{E}_{\bot}[\![\tau]\!](e[\vec{x} \mapsto \vec{v}], e'[\vec{x} \mapsto \vec{v}'])\right)$$

where we write $\bot$ to denote the trivially false binary predicate, and an "error-including" logical relation as follows:

$$\Gamma \vDash_{\mho} e \preccurlyeq e' : \tau \triangleq \left(\forall \vec{v}, \vec{v}'. \Big(\bigwedge_{\tau_i \in \Gamma} \mathcal{V}_{\top}[\![\tau_i]\!](\vec{v}_i, \vec{v}'_i) \Big) \vdash \mathcal{E}_{\top}[\![\tau]\!](e[\vec{x} \mapsto \vec{v}], e'[\vec{x} \mapsto \vec{v}'])\right)$$

where we write $\top$ to denote the trivially true binary predicate.

As such, $\vDash_{\not\mho} \lambda f.\, f\ ();\ \Omega \preccurlyeq \lambda f.\, \Omega : (\mathbb{1} \to \mathbb{1}) \to \mathbb{1}$, where Ω is a diverging expression. However, these expressions are not related in the latter "error-including" relation. In particular, consider substituting the function variable with $\lambda x.\ (\pi_1^{\ell}\ 6)$. The first expression then throws a run-time error, while the second diverges.

Adequacy. The adequacy lemma for the logical relation formalizes the informal prose in §5.1 whilst incorporating Definition 4. To succinctly state it and to easily refer to it later, we first define the concept of a Ψ-refinement, where Ψ is any binary label predicate.

Definition 5 (Ψ-refinements *[RefineL]*). Take two closed dynamic expressions e and e' and a binary predicate Ψ on labels. We say that e Ψ-refines e', denoted $e \leq_{\Psi} e'$, if the following are true:

- If e halts to a value, so does e'.
- If e throws an error with label ℓ say; then e' throws an error with label ℓ' say and $\Psi(\ell, \ell')$.
- (No restriction on e' if e diverges.)

Lemma 4 (Adequacy *[logrel_adequacy_alt]*). For any two related expressions, e.g. $\Gamma \vDash e \preccurlyeq_{\Psi} e' : \tau$, then we have that for all Ψ', C, C' such that $\vDash C \preccurlyeq_{\Psi'} C' : (\Gamma;\tau) \Rightarrow (\cdot;\tau')$, that $C[e] \leq_{\Psi \sqcup \Psi'} C'[e']$.

The proof follows from Lemma 2, together with the fact that the weakest precondition accurately models the intended behavior, as claimed in §5.1.

The logical relation developed in this section captures a form of "logical label-aware contextual refinement" (Definition 6). As depicted in §1, this notion, together with its associated Lemma 5 and Lemma 6, plays a central role in the presentation of the proof.

To define this notion, given any unary label predicate ψ, we define $\delta(\psi) \triangleq \{(\ell, \ell').\ell = \ell' \wedge \psi(\ell)\}$ to be its diagonal as a binary predicate.

Definition 6 (Logical label-aware ctx. refinement *[lla_ctx_refinement]*)**.** Let any unary predicate ψ. We define $\Gamma \vDash e \preccurlyeq^{ctx}_{\psi} e' : \tau$ as: for any C and unary predicate ψ' such that $\vDash C \preccurlyeq_{\delta(\psi')} C : (\Gamma; \tau \Rightarrow []; \tau')$, we have $C[e] \leq_{\delta(\psi \sqcup \psi')} C[e']$.

In other words, this is the biggest relation that satisfies the conclusion of Lemma 4 whilst restricting ourselves to unary predicates and self-related contexts.

The advantage of this relation (as a proxy for logical relatedness) is that it is easily proven to be transitive over terms whilst taking the meet of the predicates.

Lemma 5 (Transitivity *[lla_ctx_refinement_trans]*)**.** If $\Gamma \vDash e_1 \preccurlyeq^{ctx}_{\psi} e_2 : \tau$ and $\Gamma \vDash e_2 \preccurlyeq^{ctx}_{\psi'} e_3 : \tau$, then $\Gamma \vDash e_1 \preccurlyeq^{ctx}_{\psi \sqcap \psi'} e_3 : \tau$.

The meet between two unary label predicates is defined as follows:

Definition 7 (Pointwise Meet)**.** The meet operator $\psi \sqcap \psi'$, is defined by point-wise conjunction.

$$\psi \sqcap \psi' \triangleq \forall \ell. \psi(\ell) \wedge \psi'(\ell)$$

The proof follows directly from unfolding the definition and the fact that for any two unary predicates ψ and ψ', $t_1 \leq_{\delta(\psi)} t_2$ and $t_2 \leq_{\delta(\psi')} t_3$ composes to $t_1 \leq_{\delta(\psi \sqcap \psi')} t_3$.

Moreover, logical relatedness at a diagonal actually models logical contextual label-aware contextual refinement, as stated below (the result follows from Lemma 4):

Lemma 6 (Logical relatedness implies ctx. refinement *[lr_lla_ctx_refinement]*)**.** Let ψ a unary label predicate, and two logically related expressions $\Gamma \vDash e \preccurlyeq_{\delta(\psi)} e' : \tau$, then $\Gamma \vDash e \preccurlyeq^{ctx}_{\psi} e' : \tau$.

6 The Robust Dynamic Embedding Criterion

Having defined λ_d, λ_g and the dynamic embedding $\llbracket _ \rrbracket$ in §4, and semantic typing for static types (Definition 3), we can formalize the robust dynamic embedding criterion for λ_g (§6.1), generalized to open terms.

The proof of RDE is then presented in terms of two high-level properties, as indicated in Figure 1:

(1) A fundamental property characterizing (the emulation of) gradual contexts (Proposition 3).
(2) A logical label-aware contextual refinement from semantically-typed expressions to (the emulation of) their embedding and subsequent casting (Proposition 4).

To establish these properties, the logical relation, originally defined for static types, is extended to also handle gradual types. This is achieved by introducing an additional case for the unknown type, as explained in §6.2.

With this extension in place, §6.3 outlines the proofs of the two key lemmas. Their dependencies are also illustrated in Figure 1.

(1) Proposition 3 is proven by induction on the typing derivation. The case for casts is handled via the corresponding compatibility lemma (Lemma 7).
(2) Proposition 4 is proven by combining two auxiliary results (and Lemma 5 from §5.2):
   (a) Proposition 5, which characterizes *arbitrary dynamic expressions* and their (emulations after) dynamic embedding (together with lemma 7)
   (b) Proposition 6, which states that for *semantic typed code*, adding casts to the appropriate types has no effect

6.1 Main Theorem and High-Level Proof

This section presents a formal version of Statement 1, generalizing it to open terms. A high-level proof is then provided, assuming two novel properties.

Theorem Statement. To state the theorem on open terms, we first generalize the casting operation in *Statement* 1 to one on open terms. Since dynamically embedded open terms expect free variables of type ?, the cast operator must upcast them accordingly.

Notation 1 (Casts to τ under Γ at label ℓ *[linker]*). Let $\Gamma = x_1 : \tau_1, ..., x_n : \tau_n$.

$$cast_grad(t, \Gamma, \tau, \ell) \triangleq \left(\lambda x_1, ..., x_n.\, (t : ? \Rightarrow^{\ell} \tau)\right)\, (x_1 : \tau_1 \Rightarrow^{\ell} ?) \ldots (x_n : \tau_n \Rightarrow^{\ell} ?)$$

To define robustness (Definition 1) in λ_g, let $\mathcal{A}$ be the function that, given a gradual context, collects all labels appearing on its casts.

Definition 8 (Robustness for λ_g *[robust]*).

$$t \text{ is robust at } \Gamma\&\tau \iff \Gamma \vdash t : \tau \wedge \left(\forall \vdash C : (\Gamma;\tau) \Rightarrow (\cdot;\tau').C[t] \mapsto^* \mathtt{error}\,\ell \implies \ell \in \mathcal{A}(C)\right)$$

Theorem 1 (Robust Dynamic Embedding for λ_g *[robust_dyn_emb_criterion]*). *Consider Γ a static context, τ a static type, and e a semantically typed expression, $\Gamma \vDash e : \tau$. Then for any fresh ℓ, $cast_grad(\lceil e \rceil, \Gamma, \tau, \ell)$ is robust at $\Gamma\&\tau$.*

High-level proof. The high-level proof proceeds under the assumption of the following two propositions.

Proposition 3 (Fundamental Property of Gradual Contexts *[fundamental_ctx]*). *Any gradually-typed context $\Gamma \vdash C : (\Gamma;\tau) \Rightarrow (\Gamma';\tau')$, is self-related under $\delta(\mathcal{A}(C))$*

$$\vDash \langle\!\langle C \rangle\!\rangle \preccurlyeq_{\delta(\mathcal{A}(C))} \langle\!\langle C \rangle\!\rangle : (\Gamma;\tau) \Rightarrow (\Gamma';\tau')$$

Notation 2. For any dynamic term e we define the following.

$$cast_dyn(e, \Gamma, \tau, \ell) \triangleq \left(\lambda x_1, ..., x_n.\, (C^{\ell}_{? \Rightarrow \tau} e)\right)\, (C^{\ell}_{\tau_1 \Rightarrow ?}\, x_1) \ldots (C^{\ell}_{\tau_n \Rightarrow ?}\, x_n) \tag{8}$$

Note that for any gradual term t, $\langle\!\langle cast_grad(t, \Gamma, \tau, \ell) \rangle\!\rangle = cast_dyn(\langle\!\langle t \rangle\!\rangle, \Gamma, \tau, \ell)$.

Proposition 4 (*[lla_ctx_rfn_cast_embd_sem_e]*). *Given any $\Gamma \vDash e : \tau$, then:*

$$\Gamma \vDash cast_dyn(\langle\!\langle \lceil e \rceil \rangle\!\rangle, \Gamma, \tau, \ell) \preccurlyeq^{\mathbf{ctx}}_{\bot} e : \tau \tag{9}$$

Assuming Proposition 3 and Proposition 4, the proof of Theorem 1 now follows.

Proof. Take an arbitrary dynamic expression e, semantically typed at some static interface Γ, τ, i.e. $\Gamma \vDash e : \tau$, we shall prove that $cast_grad(\lceil e \rceil, \Gamma, \tau, \ell)$ is robust at $\Gamma\&\tau$.[5]

Take an arbitrary gradually typed context $\vdash C : (\Gamma;\tau) \Rightarrow (\cdot;\tau')$ and suppose the following:

$$C[cast_grad(\lceil e \rceil, \Gamma, \tau, \ell)] \mapsto^* \mathtt{error}\,\ell' \quad \textit{for some } \ell' \tag{10}$$

We shall now have to prove that $\ell' \in \mathcal{A}(C)$. Note that because of Proposition 1, one may reduce Equation 10 to the simulated evaluation in the dynamic language:

$$\langle\!\langle C \rangle\!\rangle[cast_dyn(\langle\!\langle \lceil e \rceil \rangle\!\rangle, \Gamma, \tau, \ell)] \mapsto^* \mathtt{error}\,\ell' \tag{11}$$

Combining Proposition 4 and Proposition 3, and unfolding Definition 6, we immediately obtain:

$$\langle\!\langle C \rangle\!\rangle[cast_dyn(\langle\!\langle \lceil e \rceil \rangle\!\rangle, \Gamma, \tau, \ell)] \leq_{\delta(\mathcal{A}(C))} \langle\!\langle C \rangle\!\rangle[e] \tag{12}$$

From Equation 11 and Equation 12, one immediately obtains that $\delta(\mathcal{A}(C))(\ell)$ is satisfied, hence $\ell \in \mathcal{A}(C)$. □

[5] Typing trivially follows from the original dynamic embedding criterion of Siek and Taha [33] (or Proposition 2) so we ignore it here

Even though a gradual context exposes a static interface, it may internally rely on imprecise types. Therefore, to prove Proposition 3, it is necessary to extend the logical relation to gradual types by introducing a case for the unknown type. Moreover, Proposition 4 involves reasoning about the dynamic embedding for *arbitrary* dynamic code, rather than only semantically well-typed expressions. This also requires supporting logical relatedness at the unknown type.

6.2 Extending the Logical Relation to Gradual Types

The logical relation is extended to gradual types by including an additional case for the unknown type (formalized in *[valrel_unknown]*). This extensions allows to simultaneously model the emulation of gradually typed code at the unknown type, as well as arbitrary (untyped) dynamic code and its dynamic embedding (needed for Proposition 3 and Proposition 4, the two remaining ingredients).[6]

$$
\begin{aligned}
&\mathcal{V}_\Psi[\![?]\!](v, v') \triangleq \\
&\quad\left|\begin{array}{l}
(v = v' = ()) \lor (\exists b.\, v = v' = b) \lor (\exists z.\, v = v' = z) \lor \\
\left(\bigvee_{*=\mathtt{l},\mathtt{r}} \exists w, w'.\, v = \mathtt{in}*\ w \land v' = \mathtt{in}*\ w' \land \triangleright \mathcal{V}_\Psi[\![?]\!](w, w')\right) \lor \\
\left(\exists v_1, v_2, v_1', v_2'.\, v = (v_1, v_2) \land v' = (v_1', v_2') \land \triangleright \mathcal{V}_\Psi[\![?]\!](v_1, v_1') \land \triangleright \mathcal{V}_\Psi[\![?]\!](v_2, v_2')\right) \lor \\
\left(\exists e, e'.\, v = \lambda x.\, e \land v' = \lambda x.\, e' \land \triangleright \left(\forall w, w'.\, \mathcal{V}_\Psi[\![?]\!](w, w') \Rightarrow \mathsf{rfn}_\Psi(\mathcal{V}_\Psi[\![?]\!])(e[w/x], e'[w'/x])\right)\right)
\end{array}\right.
\end{aligned}
$$

Two values are related at the unknown type if one of the following conditions is met:

- They are both the same base value.
- They are both the same type of injection and the corresponding values are related at the unknown type one step later.
- They are both pairs and the corresponding projections are related at the unknown type one step later.
- They are both lambdas and, and one step later, when substituting values related at the unknown in their bodies, the resulting expressions are related at the unknown type.

As such, the logical relation is now indexed by gradual types (instead of only by static ones). Notice that every occurrence of a recursive call is guarded by a later modality, justifying the existence of the fixpoint.

6.3 Final Ingredients

The logical relation extended over gradual types can now be used to prove Proposition 3 and Proposition 4.

6.3.1 Proving Proposition 3. This is done by first stating the analogue of this lemma for gradually typed expressions, after which the lemma for contexts follows naturally. The former is proven by straightforward induction on the typing derivation, where the only interesting case is the typing rule for casts. Proving this case essentially boils down to proving the following.

Lemma 7 (Compatibility for casts *[compat_cast]*)**.**

$$
\frac{\mathcal{E}_\Psi[\![\tau]\!](e, e') \qquad \tau \sim \tau' \qquad \Psi(\ell, \ell')}{\mathcal{E}_\Psi[\![\tau']\!](C^{\ell}_{\tau \Rightarrow \tau'}\ e, C^{\ell'}_{\tau \Rightarrow \tau'}\ e')}
$$

Notice that premise $\Psi(\ell, \ell')$ is needed, as generally, casts may actually fail.

[6]The relation $\mathcal{V}_\Delta[\![?]\!](v, v')$ is very similar to a disjunction over static types, e.g., $\mathcal{V}_\Delta[\![\mathbb{1}]\!](v, v') \lor \mathcal{V}_\Delta[\![\mathbb{B}]\!](v, v') \lor \cdots$, but the two are not equivalent due to the differing placement of later modalities. In the Rocq formalization, the common structure among these cases is factored out using helper functions.

6.3.2 Proving Proposition 4. This is more subtle. The fundamental difficulty lies in the fact that semantic typing is not downward compositional. For example, from $\vDash$ **if** e **then** e' **else** $e'' : \tau$ it does not follow that $\vDash e' : \tau$ or $\vDash e'' : \tau$, as e might just evaluate to **true** in which case e'' can be anything. Consequently, no inductive structure on the term is available, and semantically-typed programs may contain subterms of arbitrary behavior.

We overcome this challenge by proving this lemma in two stages (Lemma 8 & Proposition 6), combining them through Lemma 5. First, we do not rely on semantic typing; instead, we first use the following specification for the dynamic embedding on *arbitrary* dynamic code.

Proposition 5 (Fundamental property of the dynamic embedding *[fundamental_l]*). *Consider e an arbitrary dynamic expression with free variables $x_1, ..., x_n$, we have:*

$$x_1 : ?, ..., x_n : ? \vDash \langle\!\langle \lceil\!\lceil e \rceil\!\rceil \rangle\!\rangle \preccurlyeq_{\delta(\mathcal{D}(e))} e : ?$$

where $\mathcal{D}(e)$ collects every label on every destructor of e.

This is proven by structural induction on e. Note again that this specification is only possible due to generalization of logical relatedness over a predicate of permitted labels.

Using Proposition 5 together Lemma 7, one can now prove the first stage.

Lemma 8 (*[lla_ctx_rfn_cast_embd_arb_e]*)**.** Consider a dynamic expression e with free variables $x_1, ..., x_n$. We have $\Gamma \vDash \ cast_dyn(\langle\!\langle \lceil\!\lceil e \rceil\!\rceil \rangle\!\rangle, \Gamma, \tau, \ell) \preccurlyeq_{\delta(\mathcal{D}(e)\cup\{\ell\})} cast_dyn(e, \Gamma, \tau, \ell) \ : \tau$.

At this stage, semantic typing for e is still not assumed. Consequently, the predicate includes all labels associated with the destructors in e, as well as ℓ, the label of the outer casts.

Second, semantic typing of e is finally used.

Proposition 6 (Contextual refinement to superfluous cast *[linker_superfluous_l]*). *Let $\Gamma \vDash e : \tau$. We have $\Gamma \vDash \ cast_dyn(e, \Gamma, \tau, \ell) \preccurlyeq^{ctx}_{\bot} e : \tau$.*

This lemma essentially states that the outer casts are superfluous, as e is already respecting the interface. Proving this boils down to the following lemma.

Lemma 9 (Compatibility of superfluous casts *[cast_upwards_superfluous_l]*)**.**

$$\frac{\mathcal{E}_\Psi[\![\tau]\!](e, e')}{\mathcal{E}_\Psi[\![\tau]\!](C^{\ell}_{?\Rightarrow\tau}\ e, e)} \qquad\qquad \frac{\mathcal{E}_\Psi[\![\tau]\!](e, e')}{\mathcal{E}_\Psi[\![\tau]\!](C^{\ell}_{\tau\Rightarrow ?}\ e, e)}$$

Note that in this case, no assumptions about Ψ are needed. Using Lemma 5, one may verify that indeed, Lemma 8 and Proposition 6 give Proposition 4, as for any predicate ψ, we have that $\psi \sqcap \bot$ is equivalent to $\bot$.

7 Discussion & Related Work

We first discuss the design choice regarding lazy downcasts in λ_g, and then review related work.

7.1 Lazy Downcasts to Function Types

As mentioned in §4.2.2, the runtime semantics of the gradual language λ_g follows the formalization by Siek et al. [34], with the exception of lazy downcasts to function types. This choice is directly dictated by the robust dynamic embedding criterion. In the contrary (assuming λ_g with eager downcasts to function types), the criterion is broken, as illustrated next.

Example 3 (Term in λ_d semantically-typed at $((\mathbb{1} \to \mathbb{1}) \to \mathbb{1}) \to \mathbb{1}$ *[example_sem_typed]*). Consider the λ_d expression $f \triangleq \lambda h.\, h\ (\lambda_.\, \Omega); h\ ()$, where Ω any trivially-diverging expression (for simplicity, we omit labels). One can prove that $\vDash f : ((\mathbb{1} \to \mathbb{1}) \to \mathbb{1}) \to \mathbb{1}$. Indeed, the first application of h considers the following two possibilities. Either h non-trivially uses its argument ($\lambda x.\, \Omega$ here) by applying it to some value: if so, the expression diverges and, therefore, automatically satisfies the semantic contract. If h does *not* use its argument, then the behavior of $h\ (\lambda x.\, \Omega)$ does not depend on its argument. As such, if $h\ (\lambda x.\, \Omega)$ terminates to $()$, then $h\ ()$ must do so as well.

Now consider the following program P in λ_g, which applies the embedding of f to function $g = \lambda x.\, ()$, after appropriate casting: $P \triangleq (\llbracket f \rrbracket : ? \Rightarrow^{\ell} ((\mathbb{1} \to \mathbb{1}) \to \mathbb{1}) \to \mathbb{1})\ g$. The reduction proceed as follows:

$$
\begin{aligned}
\llbracket f \rrbracket &= \overbrace{(\lambda h.\, \llbracket h\ (\lambda_.\, \Omega) \rrbracket; (h : ? \Rightarrow^{\ell_1} ? \to ?)(() : \mathbb{1} \Rightarrow ?))}^{f'} : ? \to ? \Rightarrow ? \\
P &= (f' : ? \to ? \Rightarrow ? \Rightarrow^{\ell} ((\mathbb{1} \to \mathbb{1}) \to \mathbb{1}) \to \mathbb{1})\ g \\
&\mapsto^* (f'\ (g : (\mathbb{1} \to \mathbb{1}) \to \mathbb{1} \Rightarrow^{\ell} ? \to ? \Rightarrow^{\ell} ?)) : ? \Rightarrow^{\ell} \mathbb{1} \\
&\mapsto^* ((g : (\mathbb{1} \to \mathbb{1}) \to \mathbb{1} \Rightarrow^{\ell} ? \to ? \Rightarrow^{\ell} ? \Rightarrow^{\ell_1} ? \to ?)(() : \mathbb{1} \Rightarrow ?)) : ? \Rightarrow^{\ell} \mathbb{1} \\
&\mapsto ((g : (\mathbb{1} \to \mathbb{1}) \to \mathbb{1} \Rightarrow^{\ell} ? \to ?)(() : \mathbb{1} \Rightarrow ?)) : ? \Rightarrow^{\ell} \mathbb{1} \\
&\mapsto (g\ (\boxed{() : \mathbb{1} \Rightarrow ? \Rightarrow^{\ell} \mathbb{1} \to \mathbb{1}})) : ? \Rightarrow^{\ell} \mathbb{1} \to \mathbb{1} \Rightarrow^{\ell} \mathbb{1}
\end{aligned}
$$

If λ_g employed an eager cast semantics on function types, then in the next step the highlighted subexpression would throw a runtime error with blame label ℓ. In short, although f is semantically well-behaved, embedding and appropriately casting it in a gradual language with eager downcasts to arrow types would result in a non-robust gradual expression, thereby violating RDE.

Reconciling eager downcasts with RDE. Assuming a version of λ_g with eager downcasts to function types, we conjecture that RDE can be recovered by extending the dynamic language to include a simple primitive for testing whether a value is a function (dynamically throwing an error otherwise), or a primitive to perform case analysis on the shape of the value, similar to `callable` in Python and `procedure?` in Racket. In such an extended language, example 3 becomes irrelevant as f is not semantically-typed anymore. This suggests that the dynamic extreme of a gradual language with eager downcasts to arrow types might be more naturally identified with a dynamic language that includes such primitives.

7.2 Related Work

We first discuss the relation between the robust dynamic embedding criterion and other gradual typing criteria not covered in §3. We then briefly reflect on the use of Iris in the formalization, and the relation of this work to secure compilation.

7.2.1 More Gradual Typing Criteria.

Complete monitoring. Greenman et al. [16] develop a generalized framework to formally compare different evaluation strategies for the interaction between typed and untyped code. They introduce *complete monitoring*, which formalizes the intuition that untyped code gets completely "accounted for" by the dynamic checks when it crosses the boundary into the typed world. They show moreover that this property can distinguish between different evaluation strategies. While their setup is not strictly about gradual typing, RDE may fit into their framework. Instead of "dynamically embedding

and casting" an untyped expression however, the expression can be used as a boundary-expression annotated with the desired static type. However, there does not seem to be a straightforward relation between RDE and complete monitoring, as the latter does not talk about when a runtime check *cannot* fail due to specific knowledge about the boundary expression in question. To study RDE within this framework, a promising direction would be to first develop appropriate notions for robustness and semantic typing, generic enough to apply to each evaluation strategy.

In their framework for typed/untyped interactions, Greenman et al. [16] also define *blame soundness* and *blame completeness*, two properties related to the quality of error messages. RDE does not aim to address such a concern, which is orthogonal.

Gradual type theory. New et al. [30] lays out an axiomatic theory for program equivalences, demonstrating that, under the assumption of graduality, the semantics for casts can largely be derived from this theory. They apply this approach in a setting with simple types and call-by-push-value. They note that the behavior of casts between ground types and unknown types are not *uniquely* determined by their derivation; as such, their theory permits casts of the form $? \Rightarrow ? \to ?$ to evaluate eagerly. In the particular setting of §4, such casts must behave lazily for RDE to hold, as discussed in §7.1. There is a fundamental difference in the denoted properties of our logical relation and that of New [27]. New's relation captures *error approximation*: if the left-hand side errors, the right-hand side may behave arbitrarily. In contrast, our logical relation enforces a form of *label-sensitive refinement* (cf. Lemma 8): if the left-hand side errors, the right-hand side must also error, and do so with labels allowed by the predicate Ψ under which the two expressions are related. Thus, the two relations denote different notions and are logically independent—neither one implies the other.

Fully abstract embedding of static terms. Jacobs et al. [19] argue that equivalences in the statically typed language should be preserved upon gradualization. This means that gradual contexts should not enjoy more distinguishing power over static expressions than static contexts do, implying that binary reasoning principles on static expressions should remain valid in the gradual language.

At first glance, there may seem to be a possible source of tension here, informally described as follows: if both criteria are to be satisfied, the semantics of the gradual language must be (1) not too "powerful", so as not to have more distinguishing power than the syntactically-typed static language does (over statically-typed expressions), and (2) "powerful enough" to allow for "semantic runtime checks" on dynamically embedded expressions. However, if the expressiveness of semantic typing closely aligns with that of syntactic typing—as seems to be suggested by the work of Jacobs et al. [18] in the context of a simply-typed lambda calculus with iso-recursive types—this tension may be alleviated. Note that even if every semantically well-typed dynamic expression theoretically has a syntactically well-typed analogue (as suggested by the semantic back-translation of Jacobs et al. [18]), the latter might be so complicated that refactoring the former into the latter might simply be too complex and unfeasible in practice.

Open world soundness. Vitousek et al. [39] examine the context of a gradual language and a dynamic language that, aside from throwing controlled errors, may also go wrong by getting stuck. They state and prove an *open world soundness* theorem about a translation from the gradual language to the dynamic language. Given any gradual expression translated to a dynamic expression, the theorem implies that upon arbitrary interaction with *any* dynamic context C, if the program gets stuck, it must have happened due to the context itself. The theorem is formalized by marking terms into two modes, denoting the possibility (or lack thereof) of stuckness. It would be interesting to explore whether the statement and/or proof of RDE could be simplified by using this technique.

Vigilance. A novel property coined *vigilance* [15] ensures that the runtime semantics of gradual languages enforce the complete runtime typing history of values. There does not appear to be a clear link between this property and the robust dynamic embedding criterion. In §2.2, we cite an example of a vigilant gradual language (described in [15]), which suggests both vigilence and RDE can hold at the same time; conversely, in §2.3, we cite a vigilant gradualization for information flow control [6] that cannot satisfy RDE. Investigating the precise relation between the two properties is left for future work.

7.2.2 Iris. This work uses the Iris program logic [21] to simplify the definition of the logical relations that define semantic typing and its use. This approach is not novel and has significant precedent; for a gentle introduction, we refer the interested reader to Timany et al. [35]. An analogy can be drawn between how our generalized logical relations enable a *composition principle* (Lemma 2) and how Iris allows for composable proofs, each relying on a different kind of ghost state encoded by a different algebraic structure. In the latter, each individual proof is *parametrized* by a general algebraic structure, with the assumption that it *at least* "contains" the necessary algebraic structure at hand. Such proofs can then be composed by combining the assumptions trivially, facilitating the modular reasoning.

7.2.3 Secure Compilation. The statement of the robust dynamic embedding criterion echoes the literature on secure compilation. In fact, the technical development of this paper allows us to prove some additional properties that may readily be interpreted as secure compilation results. For instance, we may establish robust relational hyperproperty preservation (RrHP) as laid out by Abate et al. [1]—rather, its property-free characterization (RrHC)—of the dynamic embedding (§4.3), where the existential is simply determined by the emulation function (§4.2.3) and hence can be thought of as a "back-translation" *[rrhc_dyn_emb_untyped]*. This implies for instance that equivalent dynamic terms remain equivalent once embedded in the gradual language. Moreover, RDE can also be derived as a corollary of RrHC on the type-directed compiler on self-related expressions (on some label predicate that is not necessarily the trivially false one) by dynamic embedding and subsequent casting, which we have also stated and proven formally *[rrhc_import_sem_typed]*. The latter would allow us to prove the preservation of other label-sensitive properties. We leave the investigation of the practical relevance of such properties to future work.

8 Conclusion

We have motivated a novel criterion to assess gradually-typed languages, specifically if components developed in the associated dynamic language that are semantically well-behaved at some static type can seamlessly be embedded and used at that static type in the gradual language. This robust dynamic embedding criterion is not entailed by existing criteria, and is not satisfied by some gradual languages in the literature. We show that it is possible to satisfy robust dynamic embedding by formally presenting a simple gradual language, its dynamic counterpart, and an embedding, for which the criterion holds. To achieve this, we use a novel logical relation that characterizes robustness of gradual terms by limiting the set of casts that are allowed to fail. As such, the robust dynamic embedding criterion complements the existing set of criteria that have been proposed in the gradual typing literature.

Future work includes extending the notion of robust embedding to gradual types, not only static types, as well as studying in more depth if and how the criterion can be achieved in advanced typing disciplines that capture strong properties such as parametricity, noninterference, or metric preservation. Studying the criterion in the challenging setting of gradual dependent types is also a promising venue to explore.

References

[1] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *Computer Security Foundations Symposium*. doi:10.1109/CSF.2019.00025

[2] Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for free for free: parametricity, with and without types. *Proc. ACM Program. Lang.* 1, ICFP (2017), 39:1–39:28. doi:10.1145/3110283

[3] Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *ACM Trans. Program. Lang. Syst.* 23, 5 (Sept. 2001), 657–683. doi:10.1145/504709.504712

[4] Damián Arquez, Matías Toro, and Éric Tanter. 2025. Gradual Sensitivity Typing. In *Proceedings of the 38th Computer Security Foundations Symposium (CSF)*.

[5] Abhishek Bichhawat, McKenna McCall, and Limin Jia. 2021. Gradual Security Types and Gradual Guarantees. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. 1–16. doi:10.1109/CSF51468.2021.00015

[6] Tianyu Chen and Jeremy G. Siek. 2024. Quest Complete: The Holy Grail of Gradual Security. *Proc. ACM Program. Lang.* 8, PLDI, Article 212 (June 2024), 24 pages. doi:10.1145/3656442

[7] Matteo Cimini and Jeremy G. Siek. 2016. The gradualizer: a methodology and algorithm for generating gradual type systems. 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, 443–455. doi:10.1145/2837614.2837632

[8] Matteo Cimini and Jeremy G. Siek. 2017. Automatically generating the dynamic semantics of gradually typed languages. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 789–803. doi:10.1145/3009837.3009863

[9] Arthur Azevedo de Amorim, Matt Fredrikson, and Limin Jia. 2020. Reconciling noninterference and gradual typing. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science* (Saarbrücken, Germany) *(LICS '20)*. Association for Computing Machinery, New York, NY, USA, 116–129. doi:10.1145/3373718.3394778

[10] D. Dreyer, A. Ahmed, and L. Birkedal. 2009. Logical Step-Indexed Logical Relations. In *2009 24th Annual IEEE Symposium on Logic In Computer Science*. 71–80. doi:10.1109/LICS.2009.34

[11] Joseph Eremondi, Ronald Garcia, and Éric Tanter. 2022. Propositional equality for gradual dependently typed programming. *Proc. ACM Program. Lang.* 6, ICFP (2022), 165–193. doi:10.1145/3547627

[12] Joseph Eremondi, Éric Tanter, and Ronald Garcia. 2019. Approximate Normalization for Gradual Dependent Types. *Proceedings of the ACM on Programming Languages* 3, ICFP (Aug. 2019), 88:1–88:30.

[13] Luminous Fennell and Peter Thiemann. 2013. Gradual Security Typing with References. In *Proceedings of the 26th Computer Security Foundations Symposium (CSF)*. 224–239.

[14] Ronald Garcia, Alison M Clark, and Éric Tanter. 2016. Abstracting gradual typing. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 429–442. doi:10.1145/2914770.2837670

[15] Olek Gierczak, Lucy Menon, Christos Dimoulas, and Amal Ahmed. 2024. Gradually Typed Languages Should Be Vigilant! *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 125 (apr 2024), 29 pages. doi:10.1145/3649842

[16] Ben Greenman, Christos Dimoulas, and Matthias Felleisen. 2023. Typed–Untyped Interactions: A Comparative Analysis. *ACM Trans. Program. Lang. Syst.* 45, 1, Article 4 (mar 2023), 54 pages. doi:10.1145/3579833

[17] Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On Polymorphic Gradual Typing. *Proceedings of the ACM on Programming Languages* 1, ICFP (Sept. 2017), 40:1–40:29.

[18] Koen Jacobs, Dominique Devriese, and Amin Timany. 2022. Purity of an ST Monad: Full Abstraction by Semantically Typed Back-Translation. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 82 (apr 2022), 27 pages. doi:10.1145/3527326

[19] Koen Jacobs, Amin Timany, and Dominique Devriese. 2021. Fully Abstract from Static to Gradual. *Proc. ACM Program. Lang.* 5, POPL, Article 7 (jan 2021), 30 pages. doi:10.1145/3434288

[20] Koen Jacobs, Matías Toro, Nicolas Tabareau, and Éric Tanter. 2025. *Rocq Artifact: Robust Dynamic Embedding for Gradual Typing*. doi:10.5281/zenodo.16265382

[21] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018).

[22] Elizabeth Labrada, Matías Toro, Éric Tanter, and Dominique Devriese. 2022. Plausible Sealing for Gradual Parametricity. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 70 (apr 2022), 28 pages. doi:10.1145/3527314

[23] Kenji Maillard, Meven Lennon-Bertrand, Nicolas Tabareau, and Éric Tanter. 2022. A reasonably gradual type theory. *Proc. ACM Program. Lang.* 6, ICFP (2022), 931–959. doi:10.1145/3547655

[24] Kenji Maillard, Meven Lennon-Bertrand, Nicolas Tabareau, and Éric Tanter. 2022. A Reasonably Gradual Type Theory. *Proceedings of the ACM on Programming Languages* 6, ICFP (Aug. 2022), 931–959.

[25] Mara Malewski, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. 2024. Gradual Indexed Inductive Types. *Proceedings of the ACM on Programming Languages* 8, ICFP (Aug. 2024), 255:1–255:29.

[26] Joseph P. Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan, and Dawn Song. 2019. Duet: an expressive higher-order language and linear type system for statically enforcing differential privacy. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 172:1–172:30. doi:10.1145/3360598

[27] Max S. New. 2020. *A Semantic Foundation for Sound Gradual Typing*. Ph.D. thesis. Northeastern University. https://maxsnew.com/docs/dissertation.pdf

[28] Max S. New and Amal Ahmed. 2018. Graduality from Embedding-Projection Pairs. *Proc. ACM Program. Lang.* 2, ICFP, Article 73 (July 2018), 30 pages. doi:10.1145/3236768

[29] Max S. New, Dustin Jamner, and Amal Ahmed. 2020. Graduality and parametricity: together again for the first time. *Proc. ACM Program. Lang.* 4, POPL (2020), 46:1–46:32. doi:10.1145/3371114

[30] Max S. New, Daniel R. Licata, and Amal Ahmed. 2021. Gradual type theory. *J. Funct. Program.* 31 (2021), e21. doi:10.1017/S0956796821000125

[31] Benjamin C. Pierce. 2002. *Types and programming languages.* MIT press.

[32] Jason Reed and Benjamin C. Pierce. 2010. Distance makes the types grow stronger: a calculus for differential privacy. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 157–168. doi:10.1145/1863543.1863568

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

[34] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32)*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 274–293. doi:10.4230/LIPIcs.SNAPL.2015.274

[35] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *J. ACM* 71, 6 (2024).

[36] Matías Toro, David Darais, Chike Abuah, Joseph P. Near, Damián Árquez, Federico Olmedo, and Éric Tanter. 2023. Contextual Linear Types for Differential Privacy. *ACM Trans. Program. Lang. Syst.* 45, 2 (2023), 8:1–8:69. doi:10.1145/3589207

[37] Matías Toro, Ronald Garcia, and Éric Tanter. 2018. Type-Driven Gradual Security with References. *ACM Trans. Program. Lang. Syst.* 40, 4 (Dec. 2018), 16:1–16:55. doi:10.1145/3229061

[38] Matías Toro, Elizabeth Labrada, and Éric Tanter. 2019. Gradual Parametricity, Revisited. *Proc. ACM Program. Lang.* 3, POPL (Jan. 2019), 17:1–17:30. doi:10.1145/3290330

[39] Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big types in little runtime: open-world soundness and collaborative blame for gradual type systems. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 762–774. doi:10.1145/3009837.3009849

[40] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.* 4, 2/3 (1996), 167–188. doi:10.3233/JCS-1996-42-304

[41] Philip Wadler and Robert Bruce Findler. 2009. Well-typed programs can't be blamed. In *European Symposium on Programming*. Springer, 1–16.

Received 2025-02-27; accepted 2025-06-27