Plausible Sealing for Gradual Parametricity

ELIZABETH LABRADA[*], University of Chile, Chile
MATÍAS TORO, University of Chile, Chile
ÉRIC TANTER, University of Chile, Chile
DOMINIQUE DEVRIESE, KU Leuven, Belgium

Graduality and parametricity have proven to be extremely challenging notions to bring together. Intuitively, enforcing parametricity gradually requires possibly sealing values in order to detect violations of uniform behavior. Toro et al. (2019) argue that the two notions are incompatible in the context of System F, where sealing is transparently driven by potentially imprecise type information, while New et al. (2020) reconcile both properties at the cost of abandoning the syntax of System F and requiring user-provided sealing annotations that are not subject to graduality guarantees. Furthermore, all current proposals rely on a global form of dynamic sealing in order to enforce parametric behavior at runtime, which weakens parametric reasoning and breaks equivalences in the static language. Based on the observation that the tension between graduality and parametricity comes from the early commitment to seal values based on type information, we propose *plausible sealing* as a new intermediate language mechanism that allows postponing such decisions to runtime. We propose an intermediate language for gradual parametricity, Funky, which supports plausible sealing in a simplified setting where polymorphism is restricted to instantiations with base and variable types. We prove that Funky satisfies both parametricity and graduality, mechanizing key lemmas in Agda. Additionally, we avoid global dynamic sealing and instead propose a novel lexically-scoped form of sealing realized using a representation of evidence inspired by the category of spans. As a consequence, Funky satisfies a standard formulation of parametricity that does not break System F equivalences. In order to show the practicality of plausible sealing, we describe a translation from Funk, a source language without explicit sealing, to Funky, that takes care of inserting plausible sealing forms. We establish graduality of Funk, subject to a restriction on type applications, and explain the source-level parametric reasoning it supports. Finally, we provide an interactive prototype along with illustrative examples both novel and from the literature.

CCS Concepts: • **Theory of computation → Operational semantics**.

Additional Key Words and Phrases: Gradual typing, polymorphism, parametricity

ACM Reference Format:
Elizabeth Labrada, Matías Toro, Éric Tanter, and Dominique Devriese. 2022. Plausible Sealing for Gradual Parametricity. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 70 (April 2022), 28 pages. https://doi.org/10.1145/3527314

70

[*] Research conducted while Elizabeth was affiliated with Vrije Universiteit Brussel and KU Leuven, Belgium.

This work is partially funded by ANID FONDECYT projects 1190058 and 3200583, Chile.

Authors' addresses: Elizabeth Labrada, University of Chile, PLEIAD Lab, Computer Science Department (DCC), Beauchef 851, Santiago, Chile, elabrada@dcc.uchile.cl; Matías Toro, University of Chile, PLEIAD Lab, Computer Science Department (DCC), Beauchef 851, Santiago, Chile, mtoro@dcc.uchile.cl; Éric Tanter, University of Chile, PLEIAD Lab, Computer Science Department (DCC), Beauchef 851, Santiago, Chile, etanter@dcc.uchile.cl; Dominique Devriese, KU Leuven, imec - DistriNet, Leuven, Belgium, dominique.devriese@kuleuven.be.

2475-1421/2022/4-ART70
https://doi.org/10.1145/3527314

1 INTRODUCTION

Parametric polymorphism enables the generic definition of functions and types, providing as benefits code reusability and representation independence. System F [Girard 1972; Reynolds 1974] is the standard language to formalize this notion of parametric polymorphism. Relational parametricity stipulates that the behavior of polymorphic functions must be independent of the specific types they are instantiated with. For instance, the behavior of a function f of type $\forall X. X \rightarrow X$ should not depend on the type X it is instantiated with and, consequently, should treat the argument of type X opaquely. Hence, f $[\texttt{Int}]$ 42 should never return a different value than 42.

Also, in recent years, gradual typing has become a relevant feature for programming languages because it combines the best of static and dynamic type checking. Central to gradual typing is the notion of *precision* between types, which can range from fully-precise static types to the fully-imprecise unknown type (hereafter written ?), with partially specified types in between, such as $\texttt{Int} \rightarrow ?$. Among the expected properties of gradual languages [Siek et al. 2015], a particularly challenging one is the *dynamic gradual guarantee* (DGG), also called *graduality* [New and Ahmed 2018]. Informally, graduality is a monotonicity property of reduction with respect to precision: introducing imprecision in a program ought not change its behavior. For instance, the function $\lambda x : ?. x + 1$ of type $? \rightarrow \texttt{Int}$ should be transparently usable in place of $\lambda x : \texttt{Int}. x + 1$.

It turns that out that integrating parametric polymorphism and gradual typing into a language while preserving parametricity and graduality is extremely challenging [Ahmed et al. 2009, 2017; Igarashi et al. 2017; Matthews and Ahmed 2008; New et al. 2020; Toro et al. 2019]. The difficulty observed in these efforts is a strong tension between the two desirable properties, considering that functions may attempt to use gradual typing to bypass parametricity. For example, the following function of type $\forall X. X \rightarrow X$ should not be allowed to treat the value x as an integer, even when X happens to be instantiated to $\texttt{Int}$:

$$(\Lambda X. \lambda x : X. ((x :: ?) + 1) :: X)\ [\texttt{Int}]\ 42 \qquad \text{(we write } t :: T \text{ for type ascriptions)}$$

To prevent this application from reducing to $((42 :: ?) + 1) :: \texttt{Int}$, gradual polymorphic languages have generally relied on a form of *dynamic sealing* [Matthews and Ahmed 2008]. Essentially, the function $(\Lambda X. \lambda x : X. ((x :: ?) + 1) :: X)$ is not applied to type $\texttt{Int}$ and value 42; instead, the language generates a fresh seal α and applies the function to α and a sealed version of the value 42, and unseals the result. This approach ensures that effectively-parametric code behaves as usual, but that the above example fails (because addition fails on sealed values):

$$(\Lambda X. \lambda x : X. x)\ [\texttt{Int}]\ 42 \longrightarrow^* \mathit{unseal}_\alpha(\mathit{seal}_\alpha(42)) \longrightarrow^* 42$$
$$(\Lambda X. \lambda x : X. ((x :: ?) + 1) :: X)\ [\texttt{Int}]\ 42 \longrightarrow^* \mathit{unseal}_\alpha((\boxed{(\mathit{seal}_\alpha(42) :: ?) + 1}) :: \alpha) \longrightarrow^* \mathit{error}$$

Unfortunately, when applying a polymorphic function with an imprecise type, the decision of whether arguments should be sealed or not is not so clear-cut. Consider, for example, the functions $f_1 = \Lambda X. \lambda x : ?. x :: X$ and $f_2 = \Lambda X. \lambda x : ?. x :: \texttt{Int}$. By graduality, the two functions should behave like their more precisely-typed versions $\Lambda X. \lambda x : X. x :: X$ and $\Lambda X. \lambda x : \texttt{Int}. x :: \texttt{Int}$, respectively. However, this means that applying both functions to type $\texttt{Int}$ and value 42 should treat their arguments differently even though they have the same parameter type. Applying f_1 $[\texttt{Int}]$ 42 should seal the argument 42, while f_2 $[\texttt{Int}]$ 42 should not. Most proposed gradual parametric languages decide whether to seal or not based on the type of the argument and the function being applied, i.e. they apply *type-driven* sealing. However, there is no way to make this choice *a priori* and modularly, without breaking graduality. For example, GSF [Toro et al. 2019] does not seal the argument here, breaking graduality for f_1.

A recent proposal by New et al. [2020] side-steps the conundrum by shifting the burden of choice to programmers using *term-driven* sealing. PolyG$^\nu$ requires programmers to specify whether arguments should be sealed or not, by writing for example $\mathbf{f}_1$ `[X=Int]` $(\mathit{seal}_{\mathrm{X}}(42))$ and $\mathbf{f}_2$ `[X=Int]` 42, respectively.[1] While this strategy has produced the first parametric gradual calculus, it is important to realize that this calculus does not solve the same problem as the one tackled by other proposals like GSF, λB [Ahmed et al. 2017] or System F_G [Igarashi et al. 2017]. Gradual languages are intended to smoothly support the static-to-dynamic checking spectrum, but as noted by New et al., PolyG$^\nu$ supports this only when the untyped code already contains the right sealing annotations. In other words, in PolyG$^\nu$, sealing annotations are not subject to graduality guarantees, *i.e.* f `[Int]` 42 and f `[Int]` $\mathit{seal}_X(42)$ are unrelated by precision, and therefore graduality does not relate their respective behavior.

In this paper, we revisit the original problem: gradual parametricity with type-driven sealing. Consider again the applications f_1 `[Int]` 42 and f_2 `[Int]` 42. Instead of making an arbitrary choice between sealing or not sealing, we propose to keep both options open, so the decision can be made when the value 42 is actually used. This novel technique, called *plausible sealing*, essentially allows our calculus to treat the applications as f_i `[Int]` $(\mathit{maybeSeal}_X(42))$. The maybe-sealed value 42 embeds the fact that it may be both sealed at X and unsealed, which makes the two applications successfully reduce to 42. To study plausible sealing, we propose an intermediate gradual parametric language, Funky ($\mathrm{F}^?_\varepsilon$), which can be used as the elaboration target of different gradual source languages; we describe one such source language, Funk ($\mathrm{F}^?$), with the familiar syntax of System F.[2] Note that for simplicity, we formalize the approach in a setting where polymorphism is limited to instantiations with base and variable types.

The key novelty of the intermediate language $\mathrm{F}^?_\varepsilon$ is that it introduces maybe-sealing forms, which are interpreted thanks to an innovative runtime tracking technique. Additionally, $\mathrm{F}^?_\varepsilon$ avoids the use of dynamically-generated *global* seals. In previous calculi, a seal α can continue to exist when the type variable X for which it was created goes out of scope: $(\Lambda X.\, \lambda x : X.\, x :: ?)$ `[Int]` $42 \longrightarrow^* \mathit{seal}_\alpha(42)$. In fact, seals in these calculi behave as a form of symbolic cryptography, which makes it possible to embed languages with runtime sealing [Pierce and Sumii 2000; Sumii and Pierce 2004]. But at the same time, global seals have been shown to break equivalences that hold in System F [Devriese et al. 2018]. This global nature of seals is also the reason that parametricity theorems for gradual calculi so far have used formulations based on Kripke worlds containing semantic types for dynamically-allocated seals. $\mathrm{F}^?_\varepsilon$ features *lexically-scoped sealing*, and it is the first to support a stronger formulation of parametricity where semantic types are tracked in a lexical environment, similar to traditional formulations of parametricity [Reynolds 1983]. As such, $\mathrm{F}^?_\varepsilon$ could perhaps satisfy the ambitious criterion for gradual languages recently proposed by Jacobs et al. [2021]: fully abstract embedding of the statically-typed language into the gradually-typed language. This has been disproved by Devriese et al. [2018] for λB, but their counterexample, which essentially relies on the global nature of seals in λB and GSF, does not apply to $\mathrm{F}^?_\varepsilon$.[3] Finally, we prove both graduality and parametricity for the intermediate language $\mathrm{F}^?_\varepsilon$.

The elaboration of the source language $\mathrm{F}^?$ to $\mathrm{F}^?_\varepsilon$ is in charge of introducing maybe-sealing forms when imprecise types occur in type applications. For $\mathrm{F}^?$, we establish graduality, currently subject to a restriction on type applications. Specifically, reasoning about graduality requires users to verify that the types of polymorphic functions being applied have the same shape; for instance, graduality holds between functions of types ∀X.X→X and ∀X.?→?, but not between ∀X.X→X and

[1] As the syntax suggests, type variables in PolyG$^\nu$ are introduced at instantiation time, with outward scoping; this requires linear typing environments and a mechanism to limit their propagation to the current lambda abstraction [New et al. 2020].
[2] Funk is for **F-unk**nown ($\mathrm{F}^?$), and Funky is for Funk with **e**vidence ($\mathrm{F}^?_\varepsilon$).
[3] See the technical report for a proof sketch that their counterexample does not apply to $\mathrm{F}^?_\varepsilon$.

∀X.?. Except for this technical restriction, type and term precision is standard and graduality in $F^?$ allows programmers to reason in much the same way as they would with the natural notion of term precision.

In addition to graduality, we explain the source-level parametric reasoning that $F^?$ offers. It is worth noting that parametric reasoning at the source level of a gradual language is subtle because of another point of tension between parametricity and gradual typing that was pointed out by New et al. [2020]. Consider the two applications: $(\Lambda X.\, \lambda x : ?.\, x :: X)$ `[Int]` 42 and $(\Lambda X.\, \lambda x : ?.\, x :: X)$ `[Bool]` 42. Since the behavior of the polymorphic function $\Lambda X.\, \lambda x : ?.x :: X$ should not depend on the type it is applied to, a strict interpretation of parametricity dictates that both applications should behave the same. At the same time, by graduality, the first application should behave equivalently to the following more precisely typed version, which reduces to 42: $(\Lambda X.\, \lambda x : X.\, x :: X)$ `[Int]` $42 \longrightarrow^* 42$. However, the second application is of type `Bool` and there is no reasonable way to come up with a boolean value to return. Even worse, because parametricity implies preservation of relatedness of values, successfully returning a boolean in the second application would imply a contradiction, because that boolean would have to be related to 42 in an arbitrary, caller-chosen relation, even when that relation is empty. In other words, this strict interpretation of source-level parametricity is incompatible with graduality. However, that is not the end of the story.

In $F^?$, the second application fails at runtime: the value 42 does not have the right type to be sealed at type X, so it is not maybe-sealed, and we simply report an error when it is treated as a value of type X. This means that some polymorphic $F^?$ terms may behave differently depending on the type they are applied to, as we have $(\Lambda X.\lambda x :?.x :: X)$ `[Int]` $42 \longmapsto {}^*42$ and $(\Lambda X.\lambda x :?.x :: X)$ `[Bool]` $42 \longmapsto {}^*$**error**. It would however be incorrect to conclude that $F^?$ is not parametrically polymorphic. First, uniformity of behavior is satisfied for polymorphic functions of fully precise types,[4] even if they internally use type applications that do (!). In these cases, the definition of parametricity coincides with the standard definition for System F—except that related terms may also simultaneously fail with a runtime type error. In other words, the differences in behavior can only occur for imprecise types (and can therefore be avoided using ascriptions to precise types). Intuitively, these differences are a consequence of $F^?$ applying plausible sealing in an attempt to infer whether the programmer intended to treat arguments (or results) as values of the quantified type X, in a maximally permissive way. However, the behavior of plausible sealing is entirely predictable based on type information available *statically* at the call site, and does not depend on runtime type information. When one takes this behavior into account, gradual parametricity in $F^?$ still implies useful free theorems. For example, for any f : ∀X.?→X, f `[Bool]` true may diverge, fail or return the value true, but it can never return false.

Contributions. We develop a novel approach to gradual parametricity based on plausible sealing. Technically, we use lexically-scoped rather than global sealing, and a novel runtime tracking mechanism based on proof-relevant precision to account for postponing sealing decisions. This is achieved using a representation of evidence inspired by the category of spans. Focusing on the new ideas, we formally develop our approach in a simplified setting where polymorphism is restricted to instantiations with base and variable types. We prove that the proposed intermediate language $F^?_\varepsilon$ satisfies both parametricity and graduality, and mechanize the two key lemmas in Agda needed to prove these properties. We illustrate the practicality of $F^?_\varepsilon$ by providing a translation from the source gradual language $F^?$. For $F^?$, we establish graduality, subject to a restriction on type applications, and explain the source-level parametric reasoning it offers.

[4] Later on, we introduce a mechanism to annotate occurrences of the unknown type with the subset of type variables in scope that it might denote, and explain the impact of this feature on parametric reasoning for imprecise types.

Overview. In Section 2, we illustrate the behavior of $F^?_\varepsilon$ programs by starting from their $F^?$ source counterparts, and compare to other approaches. We then formalize the core calculus $F^?_\varepsilon$ (Section 3), describe its novel form of runtime tracking mechanism for plausible sealing (Section 4), and prove parametricity (Section 5) and the gradual guarantees (Section 6). We then formalize the source language $F^?$ and its elaboration to $F^?_\varepsilon$. We discuss the parametric reasoning enjoyed by $F^?$, and the gradual guarantees, subject to a technical restriction on type applications (Section 7). We discuss the lifting of the technical restrictions of this work in Section 8. Section 9 discusses related work and Section 10 concludes.

Full definitions and proofs of the main results can be found in the companion technical report, provided as supplementary material. Mechanized proofs of two key technical results in Agda (Lemmas 4.6 and 6.2, marked with ✓) are also included as supplementary material. The implementation (https://doi.org/10.5281/zenodo.6341550) exhibits typing derivations, the translation from $F^?$ to $F^?_\varepsilon$, and reduction traces, including all the examples mentioned in this paper and of the related literature.

2 BACKGROUND AND OVERVIEW OF $F^?_\varepsilon$

This section recalls the basics of gradual typing, emphasizing the Abstracting Gradual Typing methodology [Garcia et al. 2016], which inspired this work. It also outlines the behavior of $F^?_\varepsilon$ with specific source program examples in $F^?$ from the current state of the art of gradual parametricity, informally shedding light on how plausible sealing is realized and compares to other approaches.

2.1 Background on (Abstracting) Gradual Typing

Basics of gradual typing. Gradual typing smoothly supports the range from static to dynamic type checking by introducing the unknown type (here denoted ?) and allowing types to be partially specified [Siek and Taha 2006]. For instance, $? \rightarrow \texttt{Int}$ is the gradual type of functions whose domain is statically unknown ? and whose codomain is $\texttt{Int}$. This type is said to be less *precise* (or more imprecise) than static types such as $\texttt{Bool} \rightarrow \texttt{Int}$, and more precise than both $? \rightarrow ?$ and ? [Siek et al. 2015]; this is noted $\texttt{Bool} \rightarrow \texttt{Int} \sqsubseteq ? \rightarrow \texttt{Int} \sqsubseteq ? \rightarrow ? \sqsubseteq ?$. Optimistically, a variable of type ? can be used at any type statically; at runtime, some mechanism ensures that a runtime type error is raised before any unsafe operation is performed. For instance, the application $(\lambda x : ?. \, x + 1)$ false is well-typed, but results in a runtime error before addition is performed.

The flexibility of gradual typing is achieved by relaxing type predicates (such as type equality, subtyping, etc.) to optimistically account for imprecision. For example, type *consistency* (denoted $\sim$) is the relaxation of type equality [Garcia et al. 2016; Siek and Taha 2006]. The application example above is well typed because $\texttt{Bool} \sim ?$ and $? \sim \texttt{Int}$ (but $\texttt{Bool} \nsim \texttt{Int}$!). The dynamic semantics of a gradual source language is typically given by elaboration to a cast calculus [Siek and Taha 2006]. The elaboration inserts casts to guarantee that violations of static assumptions are detected, triggering type errors at runtime. For instance, the source term $\mathsf{\lambda x : ?. x{+}1}$ would typically be elaborated to the target term $\boldsymbol{\lambda x : ?. \langle Int \Longleftarrow ? \rangle x{+}1}$, where the cast $\boldsymbol{\langle Int \Longleftarrow ? \rangle}$ ensures that the argument given at runtime is indeed an $\mathbf{Int}$ value, otherwise an error is raised.[5]

Abstracting gradual typing. The Abstracting Gradual Typing framework (AGT) [Garcia et al. 2016] derives the static and dynamic semantics of a gradual language starting from a static language and its type safety argument. The static semantics exploit a Galois connection between gradual types and the sets of static types they denote: predicates on gradual types are obtained by existential lifting of static predicates. For instance, consistency is the lifting of equality: two gradual types

[5] We use the blue color and sans serif fonts for source languages and the red color and bold fonts for target languages.

are consistent iff there exist two static types in their denotations (a.k.a. concretizations) that are equal. More advanced predicates and functions for gradual types, such as consistent subtyping and consistent join, can also be derived following this approach.

In AGT, the dynamic semantics is defined by reduction of gradual typing derivations augmented with *evidence* for consistent judgments. Equivalently, one can understand this approach as defining an elaboration to an *evidence-based* target language, by analogy with cast calculi. The key notion here is that of evidence, which tracks the most precise information regarding a consistent judgment at runtime. During reduction, evidences are combined, just like casts, and this combination may fail with a runtime error, whenever the resulting consistent judgment is not justified anymore. This mechanism at least ensures type safety, and can be adjusted to ensure other properties (*e.g.* noninterference [Toro et al. 2018], parametricity [Toro et al. 2019]).

While the concept of evidence is very general and applies to a variety of typing disciplines, Garcia et al. [2016] observe that for a language with only type consistency, evidence coincides with the middle type of threesomes [Siek and Wadler 2010]. A threesome is a three-place cast, $\langle G_2 \overset{G}{\Longleftarrow} G_1 \rangle$, representing a downcast from the source type G_1 to the middle type G, followed by an upcast from the middle type to the target type G_2. This representation allows for space efficiency of cast calculi: when combining two threesomes, it is sufficient to retain the outermost types and keep the *meet* $\sqcap$ (according to the precision partial order) of the middle types. If such a meet is not defined, the combination of threesomes fails with a cast error. For instance, the combination $\langle \mathbf{Int} \overset{\mathbf{Int}}{\Longleftarrow} ? \rangle \langle ? \overset{\mathbf{Bool}}{\Longleftarrow} \mathbf{Bool} \rangle$ fails because $\mathbf{Int} \sqcap \mathbf{Bool}$ is undefined.

Likewise, an evidence ϵ for a consistency judgment, noted $\epsilon : G_1 \sim G_2$, is naturally represented by a common more precise type G such that $G \sqsubseteq G_1$ and $G \sqsubseteq G_2$; for instance $\mathbf{Int} : \mathbf{Int} \sim ?$. At runtime, reduction proceeds by combining evidences through *consistent transitivity* ($\circ$), which is, like for threesomes, the precision meet of the evidences. For example, term $\epsilon_2\ (\epsilon_1\ \mathbf{x} :: \mathbf{Int}) :: ?$ (with $\epsilon_i = \mathbf{Int}$) reduces to $(\epsilon_1 \circ \epsilon_2)\ \mathbf{x} :: ?$, where $\epsilon_1 \circ \epsilon_2 = \mathbf{Int} \sqcap \mathbf{Int} = \mathbf{Int}$.

The elaboration from the source language to the evidence-based target language simply inserts the *initial evidence* of all consistent judgments used in the gradual typing derivation of the term. For example, if $\vdash t : G$ then the source term $t :: G'$ would elaborate to $\epsilon\ \mathbf{t} :: G'$, where $\mathbf{t}$ is the elaboration of the subterm t, and ϵ is the initial evidence between the type G and the ascribed type G', *i.e.* $G \sqcap G'$. In an elimination form such as a function application, elaboration introduces an ascription to ensure that the top-level type constructor matches.

2.2 Evidence for Plausible Sealing

In this work, we adopt AGT for deriving the static semantics of $F^?$ and $F^?_\varepsilon$, and define the dynamic semantics of $F^?$ by elaboration to the evidence-based target language $F^?_\varepsilon$. Prior work using AGT for gradual parametricity (GSF [Toro et al. 2019]) has shown that the semantics obtained blindly with AGT only ensure type safety, but not parametricity. Ensuring parametricity requires a refined representation of evidence and consistent transitivity. In GSF, evidence is represented not as a single type, but as a pair of types (extended with type names tracked globally), in order to capture the directionality of consistent judgments, which can intuitively denote either *sealing* or *unsealing*. Consistent transitivity is refined to forbid unsound unsealing and hence enforce parametricity. In order to address the limitations discussed in the introduction, we design a novel representation of evidence in $F^?_\varepsilon$, to realize plausible sealing. The rest of this section informally describes this novel representation of evidence and the achieved behavior.

Let us focus on the two terms (1) f_1 `[Int] 42` and (2) f_2 `[Int] 42` used in the introduction. As explained, these are key illustrations of the challenge of type-driven sealing: any early decision to either seal or not seal the argument would make one of these examples fail, thereby breaking graduality. Our approach consists of capturing the different possibilities regarding sealing, and postponing the choice to seal or not to seal until a value is used; as a consequence, both programs successfully reduce to 42. This is achieved by a novel representation of evidence, which accommodates the different valid usages of an argument of unknown type, whenever the unknown type is in scope of some type variables. The first step consists of decorating the unknown type with the type variables that are in scope. So the type of both elaborated polymorphic functions in $F^?_\varepsilon$ are $\forall X.?_X \to X$ and $\forall X.?_X \to Int$, respectively, since X is the only type variable in the scope of the unknown type. The argument 42 is of type `Int`, so upon elaboration an ascription to $?_\emptyset$ is introduced—there are no type variables in scope at that point. Hence, the elaboration of both examples (where G stands for either X (1) or Int (2)) is:

$$(\epsilon_1\ ((\Lambda X.\lambda x : ?_X.\ \epsilon\ x :: G\ )\ [Int]) :: ?_\emptyset \to Int)\ (\epsilon_2\ 42 :: ?_\emptyset) \tag{1}$$

When these polymorphic functions are instantiated at type Int, the decorations of unknown types are enriched with the instantiation information, so the lambda-abstractions both take an argument of type $?_{X:Int}$. To proceed with the beta reduction, the argument $\epsilon_2 42 :: ?_\emptyset$ is ascribed to the expected argument type of the lambda, yielding the value $v = \epsilon' 42 :: ?_{X:Int}$. This value is the $maybeSeal_X(42)$ used in the introduction. Observe that there are *two* ways in which the type of 42, Int, is consistent with $?_{X:Int}$: either because $?_{X:Int}$ stands for Int, or because it stands for X (which happens to be instantiated with Int). So it is *plausible* that the value be sealed at type X, though not mandatory. In order to account for this multiplicity of possibilities, we let ϵ' be a *set* of justifications, rather than a single justification as is standard in AGT (and in GSF). Both justifications support the same consistency judgment $Int \sim ?_{X:Int}$, so using just the meet is insufficient. Instead, we represent a justification of a consistent judgment between types G_1 and G_2 as a triple (G, c_1, c_2), where G is the meet, and c_1 (resp. c_2) is a proof term that characterizes *how* G is more precise than G_1 (resp. G_2). Hence, precision in $F^?_\varepsilon$ is a *proof-relevant* notion, and evidences carry these proofs. In the example, the precision judgments are $\mathtt{inj}_X : Int \sqsubseteq ?_{X:Int}$ and $\mathtt{inj}_{Int} : Int \sqsubseteq ?_{X:Int}$, where the proof terms $\mathtt{inj}_X$ and $\mathtt{inj}_{Int}$ denote the two possible injections of imprecision. We write $\mathtt{refl}_{Int}$ for the proof term of $Int \sqsubseteq Int$. So we have:

$$\epsilon' = \{(Int, \mathtt{refl}_{Int}, \mathtt{inj}_X), (Int, \mathtt{refl}_{Int}, \mathtt{inj}_{Int})\}$$

When v is substituted in the body, reduction proceeds by combining ϵ' with ϵ, the evidence inserted by the elaboration of the ascription in the body (Equation 1), using consistent transitivity. Importantly, in Example (1), ϵ justifies that the unknown type is consistent with X via $\mathtt{inj}_X$, and when Int is substituted for X, the proof term $\mathtt{inj}_X$ in ϵ does not change (although it now justifies the judgment $Int \sqsubseteq ?_{X:Int}$ rather than $X \sqsubseteq ?_X$). Then reduction proceeds by checking that there is at least one justification in ϵ' that is compatible with ϵ; otherwise an error is raised. Because such a justification exists in both examples, they both successfully reduce to 42.

In essence, we treat type precision $\sqsubseteq$ in $F^?_\varepsilon$ not simply as a preorder, but as a category, and we construct evidence as a variant of the category of spans. Spans are the triples (G, c_1, c_2), and evidences are sets of spans. Composition of evidence through consistent transitivity can then be defined in terms of a category-theoretic pullback operation, again generalizing the order-theoretic meet that is used in regular threesomes and AGT.

2.3 Comparing Plausible Sealing and Prior Approaches

We now outline the behavior of $F^?_\varepsilon$, informally shedding light on how plausible sealing is realized and compares to other approaches. For the sake of simplicity and understanding, we use source $F^?$ programs for the comparison. Note that, in order to be well typed, source terms in $F^?$ need to be augmented with evidence in $F^?_\varepsilon$, casts in λB, and seal/unseal terms in PolyG$^\nu$ (possibly yielding

Table 1. Comparisons of gradual parametricity approaches.

	Source term in $\mathrm{F}^?$	$\mathrm{F}^?_\varepsilon$	λB	**System F_G**	**GSF**	**PolyG$^\nu$**
1	$(\Lambda X.\lambda x : ?.x :: X)\ [\mathtt{Int}]\ 42$	42	**error**	**error**	**error**	**error** / 42
2	$(\Lambda X.\lambda x : ?.x :: \mathtt{Int})\ [\mathtt{Int}]\ 42$	42	42	42	42	42 / **error**
3	$(\Lambda X.\lambda x : ?.x :: X)\ [\mathtt{Bool}]\ 42$	**error**	**error**	**error**	**error**	**error**
4	$((\Lambda X.\lambda x : X.x :: ?)\ [\mathtt{Int}]\ 42) + 1$	43	**error**	**error**	**error**	**error** / 43
5	$(\Lambda X.\lambda x : X.(x :: ?) + 1)\ [\mathtt{Int}]\ 3$	**error**	**error**	**error**	**error**	**error**
6	$(\Lambda X.\Lambda Y.\lambda x : ?.\langle x, x\rangle :: X \times Y)\ [\mathtt{Int}]\ [\mathtt{Int}]\ 42$	$\langle 42, 42\rangle$	**error**	**error**	**error**	**error** / **error**

two possible well-typed variants), in addition to superficial syntactic differences. Table 1 compares $\mathrm{F}^?_\varepsilon$ with prior approaches using a number of key examples from the literature—except Example (6)—either adapted or verbatim. Additional examples are provided in the technical report.

Examples (1) and (2) are the key examples discussed in Section 2.2. In GSF, λB and System F_G, Example (1) fails with an error, and Example (2) yields 42, because these systems eagerly choose not to seal the argument when it has the unknown type. In PolyG$^\nu$, programmers have to use explicit sealing to decide to seal or not, but this cannot be done modularly; one can obtain different behaviors accordingly. Example (3) raises a runtime error at the ascription to X, as the type variable is instantiated to `Bool` but a value of type `Int` is provided; together with Example (1), it illustrates the shallow non-parametric behavior of $\mathrm{F}^?$ discussed in the introduction. Note that other approaches also raise an error in this example because the argument is not sealed, which implies that Example (1) fails as well. Example (4) illustrates that, contrary to other approaches that use global type names as a runtime sealing mechanism, sealing in $\mathrm{F}^?_\varepsilon$ is lexically scoped: seals cannot outlive the lexical boundary of a type abstraction. In the example, when 42 is returned by the function, it is automatically unsealed and usable as a regular integer. In PolyG$^\nu$, an explicit unseal is needed to avoid failure. Example (5) illustrates the prevention of a violation of parametricity at runtime. Example (6) illustrates yet another flexibility of plausible sealing that makes it more expressive than prior approaches: evidence as sets of spans can support multiple sealing behaviors. In this example, the argument of the function, 42, is treated as plausibly sealed to both X and Y at the same time. This example fails in GSF. In PolyG$^\nu$, programmers have to pick in advance whether to seal with X or Y, and the example fails in both cases. Observe that this program does not have a fully statically-typed counterpart, and therefore showcases an expressiveness gain of the gradual language, which compromises neither graduality nor parametricity.

These examples illustrate the flexibility afforded by plausible sealing, as a novel point in the design space of gradual parametricity.

3 THE EVIDENCE-BASED LANGUAGE $\mathrm{F}^?_\varepsilon$

Now that we have informally explained our representation of evidence and the obtained behavior, we turn to the formalization of $\mathrm{F}^?_\varepsilon$ and its properties: parametricity and graduality. This section centers on presenting the language without entering into the details of evidence: evidence and its operators are treated abstractly. We provide the full details of evidence for $\mathrm{F}^?_\varepsilon$ in Section 4. Sections 5 and 6 establish parametricity and graduality of $\mathrm{F}^?_\varepsilon$, respectively. Section 7 then studies the source language $\mathrm{F}^?$, its elaboration to $\mathrm{F}^?_\varepsilon$, and its properties.

Syntax and static semantics. Figure 1 presents the syntax and semantics of $\mathrm{F}^?_\varepsilon$. A type G can be either a base type, a type variable, a function type, a polymorphic type, or the unknown

$$X \in \textsc{TypeVar}, \quad G \in \textsc{GType}, \quad \epsilon \in \textsc{Evidence}, \quad t \in \textsc{Term}, \quad \Delta \subset \textsc{TypeVar}, \quad \Gamma \in \textsc{Var} \overset{\text{fin}}{\rightharpoonup} \textsc{GType}$$

$$F ::= B \mid X \qquad G ::= B \mid X \mid G \rightarrow G \mid \forall X.G \mid ?_{\delta} \qquad \delta ::= \delta, X : F \mid \emptyset$$

$$u ::= b \mid \lambda x : G.t \mid \Lambda X.t \qquad v ::= \epsilon\, u :: G \qquad t ::= v \mid x \mid t\ t \mid t\ [F] \mid \epsilon\, t :: G \qquad s ::= u \mid t$$

$\boxed{\Delta; \Gamma \vdash s : G}$ **Term typing**

$$\text{Gasc}\ \frac{\Delta; \Gamma \vdash s : G' \qquad \Delta \vdash G \qquad \epsilon : G' \sim G}{\Delta; \Gamma \vdash \epsilon\, s :: G : G} \qquad\qquad \text{GappG}\ \frac{\Delta; \Gamma \vdash t : \forall X.G \qquad \Delta \vdash F}{\Delta; \Gamma \vdash t\ [F] : G[F/X]}$$

$\boxed{t \longrightarrow t \text{ or } \mathbf{error}}$ **Notion of reduction**

$$\begin{array}{lrcl}
(\text{Rasc}) & \epsilon_2\ (\epsilon_1\, u :: G_1) :: G_2 & \longrightarrow & \begin{cases} \epsilon\, u :: G_2 & \text{if } \epsilon = \epsilon_1 \circ \epsilon_2 \\ \mathbf{error} & \text{otherwise} \end{cases} \\
(\text{Rapp}) & \begin{array}{r} (\epsilon_1\ (\lambda x : G_{11}.t) :: G_1 \rightarrow G_2) \\ (\epsilon_2\, u :: G_1) \end{array} & \longrightarrow & \begin{cases} \mathit{cod}(\epsilon_1)\ (t[(\epsilon\, u :: G_{11})/x]) :: G_2 & \text{if } \epsilon = \epsilon_2 \circ \mathit{dom}(\epsilon_1) \\ \mathbf{error} & \text{otherwise} \end{cases} \\
(\text{RappG}) & (\epsilon\ (\Lambda X.t) :: \forall X.G)\ [F] & \longrightarrow & (\mathit{schm}(\epsilon)\, t :: G)[F/X]
\end{array}$$

Fig. 1. $F_{\varepsilon}^{?}$: Syntax, Static and Dynamic Semantics (fragment).

type. [6] Observe that static types from System F are syntactically included in gradual types G. In $F_{\varepsilon}^{?}$, polymorphic types can only be instantiated with base types and type variables, called *instantiation types*, and denoted by metavariable F. As mentioned in the introduction, this restriction on polymorphism simplifies the already-dense technical development while still manifesting all the subtleties of gradual parametricity identified in prior work. Another distinctive feature of $F_{\varepsilon}^{?}$ is that it avoids the use of a global typename store as used in all prior work on gradual parametricity thanks to the fact that the unknown type is indexed by an environment δ. This instantiation environment keeps track of the static and dynamic information related to type variables in scope: $?_{X:\mathsf{Int}}$ expresses that type variable X is in scope and instantiated to Int. Uninstantiated type variables are associated with themselves $X : X$, which for brevity we simply write as X. It is worth noting that in the type $?_{X:X}$, the two occurrences of X play a different role: the first is merely a label, while the second is an actual occurrence of the type variable X.

A term t can be a value v, a variable, a term application, a type application (to an instantiation type), or an ascription. Note the presence of an evidence ϵ in an ascription, to justify the fact that the underlying term is of a type consistent with the ascribed type. Values v are ascribed raw values $\epsilon u :: G$, where ϵ justifies that the type of u is consistent with G. A raw value u can be a base value b, a function, or a type abstraction. To avoid duplication of typing rules, we use metavariable s to denote both raw values u and terms t.

The typing judgment $\Delta; \Gamma \vdash s : G$ establishes that s has type G, under type variable environment Δ, and type environment Γ. Δ is used to track type variables in scope, and Γ to map variables to their types. Most of the type rules are standard, closely following System F. Note that rule (Gasc) is the only rule that uses the consistency relation; all other elimination rules require types to match exactly. Elaboration from the source language $F^{?}$ is in charge of introducing the necessary ascriptions to safely support the flexibility of gradual typing. Rule (GappG) is almost standard, save for the fact that it restricts instantiations to instantiation types F. The type substitution operator $G[F/X]$

[6] We omit formal definitions for pairs, which are unsurprising and found in the technical report.

is standard, except for occurrences of unknown types, for which type substitution is applied to their instantiation environments δ: $(\delta, X : F')[F/X] = \delta[F/X], X : F'[F/X]$. For instance, $?_{Y:X,X:X}[Int/X] = ?_{Y:Int,X:Int}$. Notice that type substitution on an instantiation environment δ only affects type variable occurrences, not labels.

Dynamic semantics. The dynamic semantics of $F^?_\varepsilon$ are usual for an evidence-based reduction semantics [Garcia et al. 2016; Toro et al. 2019], using reduction frames and notions of reduction. Reduction uses the consistent transitivity operator $\circ$ to combine evidence and justify transitive judgments. If $\epsilon_1 \circ \epsilon_2$ is defined then it yields a more precise evidence, otherwise an error is raised. For example, rule (Rasc) reduces nested ascriptions, such as value $\epsilon_1\, u :: G_1$ ascribed to G_2 using evidence ϵ_2. Recall that ϵ_1 justifies that G_u, the type of u, is consistent with G_1, noted $\epsilon_1 : G_u \sim G_1$, and likewise, $\epsilon_2 : G_1 \sim G_2$. Therefore, if $\epsilon_1 \circ \epsilon_2$ is defined, then the resulting evidence justifies the transitive judgment between G_u and G_2, i.e. $\epsilon_1 \circ \epsilon_2 : G_u \sim G_2$. Rule (Rapp) reduces a term application substituting the argument in the body of the function. It first ascribes the argument to G_{11}, the type of x. To justify the transitive judgment of the beta reduction, it combines ϵ_2, with $dom(\epsilon_1)$. Evidence $dom(\epsilon_1)$ and $cod(\epsilon_1)$ can be extracted from ϵ_1 by reasoning about inversion on consistency ($\epsilon_1 : G_{11} \rightarrow G_{12} \sim G_1 \rightarrow G_2$). Evidence ϵ_1 justifies that $G_{11} \rightarrow G_{12}$, the underlying type of the function, is consistent with $G_1 \rightarrow G_2$. Thus, evidence $dom(\epsilon_1)$ justifies that G_1 is consistent with G_{11}, and therefore $\epsilon \circ dom(\epsilon_1)$, if defined, justifies that the type of the raw value u is consistent with G_{11}. The output of the function is ascribed to the expected return type G_2 using the co-domain evidence $cod(\epsilon_1)$. Rule (RappG) reduces type application by substituting type F in the schema evidence $schm(\epsilon)$, in the body of the type abstraction t and in the scheme type G. By inversion on consistency, if $\epsilon : \forall X.G' \sim \forall X.G$, then $schm(\epsilon) : G' \sim G$. Substitution on evidence $\epsilon[F/X]$ is defined using substitution on types, for all type information that appears in the evidence (Section 4). Substitution on terms $t[F/X]$ is recursively defined over subterms, evidences, and types.

It is worth noting that rule (RappG) is remarkably standard unlike other gradual polymorphic calculi where dynamic type generation happens in this rule, being stored in a global store. This is made possible thanks to the use of the annotated unknown type $?_\delta$. Another point that has relevance in the reduction of type applications is that the type of the redex can contain instantiated type variables in scope. For example, term $(\epsilon_2\, (\Lambda X.\lambda x : ?_X.\epsilon_1\, x :: X) :: \forall X.?_X \rightarrow X)\ [Int]$ has type $?_{X:Int} \rightarrow Int$ with X in the instantiation environment of the unknown type. To obtain a term that can be applied to an argument of type $?_\emptyset$, an external evidence to the application is necessary to justify that $?_{X:Int} \rightarrow Int$ is consistent with $?_\emptyset \rightarrow Int$. As we saw in Section 2 and will explain in detail in Section 7, this evidence is inserted by the elaboration from $F^?$ to $F^?_\varepsilon$.

Properties. As expected from any language, $F^?_\varepsilon$ is type safe (*i.e.* well-typed $F^?_\varepsilon$ terms do not get stuck). Thus, a well-typed program either evaluates to a value, a runtime error, or diverges. In order to prove type safety, it is necessary to have some properties about the evidence, such as the resulting evidence from consistent transitivity supports the transitive consistency judgment and type substitution over the evidence supports the substitution over the judgment (Section 4).

Lemma 3.1 (Type Safety). *If* $\vdash t : G$ *then either* $t \overset{*}{\longmapsto} v$ *with* $\vdash v : G$, $t \overset{*}{\longmapsto} \mathbf{error}$, *or* t *diverges.*

Of course, the most interesting properties of $F^?_\varepsilon$ are parametricity and graduality. We dive into the details of these properties in Section 5 and Section 6 respectively, after giving a detailed account of evidence, including its representation, operations, and properties thereof, in particular *associativity* and *monotonicity* of consistent transitivity.

$$c ::= \mathtt{refl}_F \mid c \to c \mid \forall X.c \mid \mathtt{inj}_F \mid \mathtt{inj}_{\to}(c) \mid \mathtt{inj}_{\forall}(c) \mid \mathtt{inj}_?$$

$\boxed{c : G \sqsubseteq G}$ **Proof-relevant precision**

$$\frac{}{\mathtt{refl}_B : B \sqsubseteq B} \qquad \frac{}{\mathtt{refl}_X : X \sqsubseteq X} \qquad \frac{\delta \subseteq \delta'}{\mathtt{inj}_? : ?_\delta \sqsubseteq ?_{\delta'}}$$

$$\frac{c : G_1 \sqsubseteq G_2 \qquad c' : G_1' \sqsubseteq G_2'}{c \to c' : G_1 \to G_1' \sqsubseteq G_2 \to G_2'} \qquad \frac{c : G_1 \sqsubseteq G_2}{\forall X.c : \forall X.G_1 \sqsubseteq \forall X.G_2} \qquad \frac{X : F \in \delta \qquad \delta \vdash F}{\mathtt{inj}_X : F \sqsubseteq ?_\delta}$$

$$\frac{}{\mathtt{inj}_B : B \sqsubseteq ?_\delta} \qquad \frac{c : G \sqsubseteq ?_\delta \to ?_\delta}{\mathtt{inj}_{\to}(c) : G \sqsubseteq ?_\delta} \qquad \frac{c : G \sqsubseteq \forall X.?_{\delta,X}}{\mathtt{inj}_{\forall}(c) : G \sqsubseteq ?_\delta}$$

$\boxed{c;c = c}$ **Composition of precision proof terms**

$$\mathtt{refl}_B;\mathtt{refl}_B = \mathtt{refl}_B \qquad \mathtt{refl}_X;\mathtt{refl}_X = \mathtt{refl}_X \qquad (c_1 \to c_2);(c_1' \to c_2') = (c_1;c_1') \to (c_2;c_2')$$

$$(\forall X.c);(\forall X.c') = \forall X.(c;c') \qquad \mathtt{refl}_B;\mathtt{inj}_B = \mathtt{inj}_B \qquad \mathtt{refl}_F;\mathtt{inj}_X = \mathtt{inj}_X$$

$$c_1;\mathtt{inj\_}(c_2) = \mathtt{inj\_}(c_1;c_2) \qquad c;\mathtt{inj}_? = c$$

$$\begin{tikzcd} G_1 \arrow[r, "c_1"'] \arrow[rr, bend right, "c_1;c_2 = c"] & G \arrow[r, "c_2"'] & G_2 \end{tikzcd} \qquad \begin{tikzcd} \mathrm{Int} \arrow[r, "\mathtt{refl}_{\mathrm{Int}}"'] \arrow[rr, bend right, "\mathtt{inj}_{\mathrm{Int}}"] & \mathrm{Int} \arrow[r, "\mathtt{inj}_{\mathrm{Int}}"'] & ?_{X:\mathrm{Int}} \end{tikzcd} \qquad \begin{tikzcd} \mathrm{Int} \arrow[r, "\mathtt{inj}_X"'] \arrow[rr, bend right, "\mathtt{inj}_X"] & ?_{X:\mathrm{Int}} \arrow[r, "\mathtt{inj}_?"'] & ?_{X:\mathrm{Int},Y:\mathrm{Int}} \end{tikzcd}$$

Fig. 2. Proof-relevant type precision, composition, and examples.

4 EVIDENCE FOR PLAUSIBLE SEALING IN $F_\varepsilon^?$

We now turn to the key technical innovation that makes $F_\varepsilon^?$ (and by extension, $F^?$) able to address the dilemma presented in Section 1: plausible sealing, implemented via a novel representation of evidence based on a *proof-relevant* notion of gradual type precision. As explained in Section 2, for a consistency judgment $G_1 \sim G_2$, instead of having evidence only track a common more precise type G, evidence is a set of spans, where each span includes a common more precise type G and two proof terms that describe *how* $G \sqsubseteq G_1$ and $G \sqsubseteq G_2$ hold, respectively.

Proof-relevant precision. As mentioned in Section 2.2, there can be multiple ways of satisfying a precision relation $G \sqsubseteq G'$. To differentiate them, we extend the precision relation between types with a proof term c that expresses how G is more precise than G'.

Proof-relevant precision is presented in Figure 2. The proof relevant judgment $c : G \sqsubseteq G'$ denotes that *proof term* c justifies that G is more precise than G'. A reflexive proof term $\mathtt{refl}_F$ justifies that F is more precise than F. A function proof term $c \to c'$ witnesses that a function type is more precise than another function type if their domains and codomains are related; likewise for polymorphic proof terms $(\forall X.c)$. Proof term $\mathtt{inj}_X$ represents an injection from X into $?_\delta$, and witnesses that if X is associated to F in δ and F is well-formed with respect to δ, then F is more precise than $?_\delta$. We say that a type F is well-formed with respect to δ $(\delta \vdash F)$ if F is a base type B or is a type variable X and $X : X \in \delta$. For example, if X is not yet instantiated and $X : X \in \delta$, then $\mathtt{inj}_X : X \sqsubseteq ?_{X:X}$. If $X : \mathrm{Int} \in \delta$, then $\mathtt{inj}_X : \mathrm{Int} \sqsubseteq ?_{X:\mathrm{Int}}$. Injection $\mathtt{inj}_B$ witnesses that a base type B is more precise than any unknown type. Proof term sequence $\mathtt{inj}_{\to}(c)$ justifies that function types are more precise than unknown: if c witnesses that a type G is more precise than

$?_\delta \to ?_\delta$, then $\mathtt{inj}_{\to}(c)$ justifies that G is more precise than $?_\delta$. Similarly, $\mathtt{inj}_{\forall}(c)$ witnesses that polymorphic types are more precise than unknown respectively. $\mathtt{inj}_?$ justifies that an unknown type is more precise than another if the environment of the former is contained in the environment of the latter.

To support transitive judgments of precision, we define the composition of proof terms in Figure 2. A reflexive proof term combined with itself yields the same proof term. The combinations of function, and abstraction proof terms are defined inductively. The combination of a reflexive base type proof term with an injection from that type yields the latter. Similarly, the combination of a reflexive $\mathtt{refl}_F$ proof term with an injection from X, yields just the injection from X. Finally, the combination of an $\mathtt{inj}_?$ from the right can always be dropped. Figure 2 illustrates graphically the composition function along some examples. If $c_1 : G_1 \sqsubseteq G$, and $c : G \sqsubseteq G_2$, then $c_1;c_2 : G_1 \sqsubseteq G_2$. If $\mathtt{refl}_{\mathrm{Int}} : \mathrm{Int} \sqsubseteq \mathrm{Int}$, and $\mathtt{inj}_{\mathrm{Int}} : \mathrm{Int} \sqsubseteq ?_{X:\mathrm{Int}}$, then as $\mathtt{refl}_{\mathrm{Int}};\mathtt{inj}_{\mathrm{Int}} = \mathtt{inj}_{\mathrm{Int}}$, then $\mathtt{inj}_{\mathrm{Int}} : \mathrm{Int} \sqsubseteq ?_{X:\mathrm{Int}}$. Finally, if $\mathtt{inj}_{\mathrm{Int}} : \mathrm{Int} \sqsubseteq ?_{X:\mathrm{Int}}$, and $\mathtt{inj}_? : ?_{X:\mathrm{Int}} \sqsubseteq ?_{X:\mathrm{Int},Y:\mathrm{Int}}$, then $\mathtt{inj}_{\mathrm{Int}} : \mathrm{Int} \sqsubseteq ?_{X:\mathrm{Int},Y:\mathrm{Int}}$. With these reflexivity and composition operators, gradual types and type precision proof terms can be seen as the objects and morphisms of a category, which will be useful in Section 4.

Evidence and consistent transitivity. As mentioned before, evidence is defined as a set of justifications, accounting for the multiple possibilities in which two types can be consistent. Using proof-relevant type precision, evidence ϵ is defined as a non-empty set of spans $\{\, S, \ldots \}$, where a span S is a tuple (G, c_1, c_2) such that if $\epsilon : G_1 \sim G_2$, then G is a common more precise type than G_1 and G_2, and c_1 and c_2 justify "how", respectively.

Definition 4.1. $\epsilon : G_1 \sim G_2$ iff $\forall (G, c_1, c_2) \in \epsilon, c_1 : G \sqsubseteq G_1 \wedge c_2 : G \sqsubseteq G_2$.

For example, $\epsilon = \{(\mathrm{Int}, \mathtt{refl}_{\mathrm{Int}}, \mathtt{inj}_{\mathrm{Int}}), (\mathrm{Int}, \mathtt{refl}_{\mathrm{Int}}, \mathtt{inj}_X)\}$ justifies that $\mathrm{Int} \sim ?_{X:\mathrm{Int}}$, because $\mathtt{refl}_{\mathrm{Int}} : \mathrm{Int} \sqsubseteq \mathrm{Int}$, $\mathtt{inj}_{\mathrm{Int}} : \mathrm{Int} \sqsubseteq ?_{X:\mathrm{Int}}$, and $\mathtt{inj}_X : \mathrm{Int} \sqsubseteq ?_{X:\mathrm{Int}}$. Therefore, as explained in Section 2, the term $\epsilon 42 :: ?_{X:\mathrm{Int}}$ corresponds exactly to the maybe-sealed value $\mathit{maybeSeal}_X(42)$ from the introduction: the evidence holds *both* possible justifications.

The type substitution definition on evidence, and more precisely on proof terms, is fundamental for the plausible sealing mechanism to preserve parametricity. Type substitution for evidence is defined as the type substitution for each of its spans. Type substitution for a span is defined as the type substitution of its components. For example, $(X, \mathtt{refl}_X, \mathtt{refl}_X)[F/X] = (F, \mathtt{refl}_F, \mathtt{refl}_F)$ and $(X, \mathtt{inj}_X, \mathtt{refl}_X)[F/X] = (F, \mathtt{inj}_X, \mathtt{refl}_F)$. Note that $\mathtt{inj}_X[F/X] = \mathtt{inj}_X$ is essential to preserve sealed values; otherwise, we would forget the sealing if we apply the substitution.

To define consistent transitivity for this representation of evidence, we first define the *pullback* operator between proof terms.

Lemma 4.2 (Pullback operator and its universal property). *There exists a partial pullback operator such that if* $c_1 : G_1 \sqsubseteq G$ *and* $c_2 : G_2 \sqsubseteq G$, *and* $\mathit{pullback}(G, (G_1, c_1), (G_2, c_2)) = (G_3, c_3, c_4, c_5)$, *then* $c_3 : G_3 \sqsubseteq G_1$, $c_4 : G_3 \sqsubseteq G_2$, $c_3;c_1 = c_5$ *and* $c_4;c_2 = c_5$. *The pullback operator is universal in the following sense. If there exists* G_3', c_3', c_4' *and* c_5' *such that* $c_3';c_1 = c_5'$ *and* $c_4';c_2 = c_5'$, *then* $\mathit{pullback}(G, (G_1, c_1), (G_2, c_2)) = (G_3, c_3, c_4, c_5)$ *and there exists a unique* $c : G_3' \sqsubseteq G_3$ *such that* $c;c_3 = c_3'$, $c;c_4 = c_4'$ *and* $c;c_5 = c_5'$.

$$\begin{tikzcd}
 & G & \\
G_1 \arrow[ur, "c_1"] & & G_2 \arrow[ul, "c_2"'] \\
 & G_3 \arrow[uu, "c_5"] \arrow[ul, "c_3"] \arrow[ur, "c_4"'] & \\
 & G_3' \arrow[u, "c"] \arrow[uul, bend left, "c_3'"] \arrow[uur, bend right, "c_4'"'] &
\end{tikzcd}$$

Our pullback operator and its universal property are a mild adaptation of the standard definition in category theory [nLab contributors 2021a]. Figure 3 illustrates our *pullback* operator, along with two examples. The first example (second diagram) calculates $\mathit{pullback}(?_{X:\mathrm{Int}}, (\mathrm{Int}, \mathtt{inj}_X), (\mathrm{Int}, \mathtt{inj}_X))$, returning $(\mathrm{Int}, \mathtt{refl}_{\mathrm{Int}}, \mathtt{refl}_{\mathrm{Int}}, \mathtt{inj}_X)$. Note that the diamond diagram commutes, obtaining $\mathtt{inj}_X$ both on the left and on the right. The second example (third diagram) tries to calculate

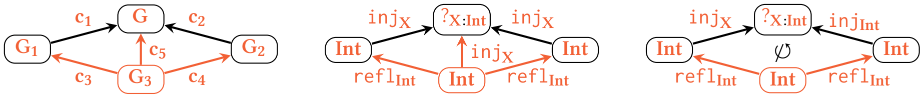

Fig. 3. Pullback and examples.

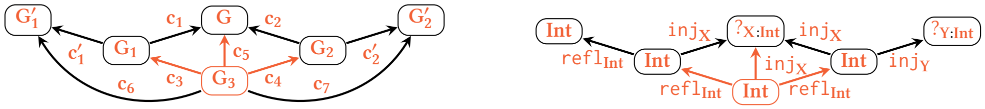

Fig. 4. Consistent transitivity for spans and example.

$\mathit{pullback}(?_{X:\mathsf{Int}}, (\mathsf{Int}, \mathtt{inj}_X), (\mathsf{Int}, \mathtt{inj}_{\mathsf{Int}}))$, but it is undefined since there is no gradual type and proof terms such that the diagram commutes. The definition of the pullback operator is algorithmic, proceeding in most cases congruently.

The universal property in Lemma 4.2 establishes that if G'_3 (with proof terms c'_3 and c'_4) makes the pullback diagram commute, then the pullback is defined, resulting in type G_3 (with proof terms c_3 and c_4), the less precise type that makes the diagram commute. Additionally, there exists a proof term c such that $c : G_3 \sqsubseteq G'_3$ and all sub-diagrams commute. For example, suppose $c_1 = c_2 = \mathtt{inj}_?$ and $G = G_1 = G_2 = ?_{X:\mathsf{Int}}$. Since $G'_3 = \mathsf{Int}$, with $c'_3 = c'_4 = \mathtt{inj}_X$, makes the diagram commute, we know that the pullback exists. In this case, we know that $\mathit{pullback}(?_{X:\mathsf{Int}}, (?_{X:\mathsf{Int}}, \mathtt{inj}_?), (?_{X:\mathsf{Int}}, \mathtt{inj}_?)) = (?_{X:\mathsf{Int}}, \mathtt{inj}_?, \mathtt{inj}_?, \mathtt{inj}_?)$. Observe that there exist two proof terms c such that $c : G_3 \sqsubseteq G'_3$, $c = \mathtt{inj}_{\mathsf{Int}}$ and $c = \mathtt{inj}_X$. However, only $c = \mathtt{inj}_X$ satisfies $c;c_3 = c'_3$, $c;c_4 = c'_4$ and $c;c_5 = c'_5$. Consistent transitivity between spans is then defined as follows:

Definition 4.3 (Consistent transitivity for spans). Let $c_1 : G_1 \sqsubseteq G$, $c'_1 : G_1 \sqsubseteq G'_1$, $c_2 : G_2 \sqsubseteq G$ and $c'_2 : G_2 \sqsubseteq G'_2$. We pose $(G_1, c'_1, c_1) \circ (G_2, c_2, c'_2) = \{(G_3, c_6, c_7) \mid \mathit{pullback}(G, (G_1, c_1), (G_2, c_2)) = (G_3, c_3, c_4, c_5) \wedge c_3;c'_1 = c_6 \wedge c_4;c'_2 = c_7\}$.

The definition is very close to the standard definition of composition of spans [nLab contributors 2021b], except that we are dealing with a *partial* pullback and a *set of spans* rather than a single span. Figure 4 graphically supports the definition of consistent transitivity, along with an example. First the pullback of c_1 and c_2 is computed. If the pullback is defined, then the new evidence type is computed using the common gradual type from the pullback G_3, and new proofs that G_3 is more precise than G'_1 and G'_2 using the proof-relevant composition operator. Note that the result of consistent transitivity for spans is either a singleton set or the empty set. In the example, we have that $(\mathsf{Int}, \mathtt{refl}_{\mathsf{Int}}, \mathtt{inj}_X) : \mathsf{Int} \sim ?_{X:\mathsf{Int}}$ (representing a seal at type X), and $(\mathsf{Int}, \mathtt{inj}_X, \mathtt{inj}_Y) : ?_{X:\mathsf{Int}} \sim ?_{Y:\mathsf{Int}}$ (an unseal at type X, followed by a seal at type Y). Consistent transitivity $(\mathsf{Int}, \mathtt{refl}_{\mathsf{Int}}, \mathtt{inj}_X) \circ (\mathsf{Int}, \mathtt{inj}_X, \mathtt{inj}_Y)$ is computed by first computing $\mathit{pullback}(?_{X:\mathsf{Int}}, (\mathsf{Int}, \mathtt{inj}_X), (\mathsf{Int}, \mathtt{inj}_X)) = (\mathsf{Int}, \mathtt{refl}_{\mathsf{Int}}, \mathtt{refl}_{\mathsf{Int}}, \mathtt{inj}_X)$. As $\mathtt{refl}_{\mathsf{Int}};\mathtt{refl}_{\mathsf{Int}} = \mathtt{refl}_{\mathsf{Int}}$, and $\mathtt{refl}_{\mathsf{Int}};\mathtt{inj}_Y = \mathtt{inj}_Y$, the result is $(\mathsf{Int}, \mathtt{refl}_{\mathsf{Int}}, \mathtt{inj}_Y)$.

Finally, consistent transitivity between evidences is just defined as the natural lifting of consistent transitivity of spans to sets of spans.

Definition 4.4 (Consistent transitivity for evidence). Let $\epsilon_1 : G_1 \sim G$, and $\epsilon_2 : G \sim G_2$.

$$\epsilon_1 \circ \epsilon_2 ::= \begin{cases} \epsilon & \text{if } \epsilon = \{S \mid S \in S_1 \circ S_2, S_1 \in \epsilon_1, S_2 \in \epsilon_2\} \neq \varnothing \\ \textbf{error} & \text{otherwise} \end{cases}$$

Note that if the resulting set is empty, then consistent transitivity is undefined, representing a runtime type error because plausibility of well-typedness has been refuted. Otherwise, if consistent transitivity is defined, then the obtained evidence justifies the transitive judgment.

Lemma 4.5. *Let* $\epsilon_1 : \mathrm{G}_1 \sim \mathrm{G}$, *and* $\epsilon_2 : \mathrm{G} \sim \mathrm{G}_2$. *If* $\epsilon_1 \circ \epsilon_2$ *is defined, then* $\epsilon_1 \circ \epsilon_2 : \mathrm{G}_1 \sim \mathrm{G}_2$.

Associativity of consistent transitivity. Associativity of consistent transitivity is a key property in evidence-based semantics, used to establish type soundness as well as space efficiency optimizations [Bañados Schwerter et al. 2021; Toro and Tanter 2020]. In particular, in this work the associativity lemma is used extensively in the proof of parametricity of $\mathrm{F}_\varepsilon^?$ (Section 5).

Lemma 4.6 (✓ Associativity of Evidence Composition). $(\epsilon_1 \circ \epsilon_2) \circ \epsilon_3 = \epsilon_1 \circ (\epsilon_2 \circ \epsilon_3)$.

The proof of the associativity lemma relies on the universal property of the pullback (Lemma 4.2).

Examples of reduction. Armed with the dynamic semantics of $\mathrm{F}_\varepsilon^?$ and the concrete representation of evidence, we first illustrate the reduction of Example (1) from Section 2.3: $(\Lambda X.\lambda x : ?.x :: X)\ [\mathtt{Int}]\ 42$, which reduces to 42. Its elaboration (omitting some trivial evidence for conciseness) and reduction proceed as follows:

$$\begin{array}{lll}
 & & (\epsilon_2\ (\Lambda \mathbf{X}.\lambda \mathbf{x} : ?_{\mathbf{X}}.\epsilon_1\ \mathbf{x} :: \mathbf{X}\ [\mathbf{Int}]) :: ? \rightarrow \mathbf{Int})\ (\epsilon_3\ 42 :: ?) \quad \text{where } \epsilon_1 = \{(\mathbf{X}, \mathtt{inj}_{\mathbf{X}}, \mathtt{refl}_{\mathbf{X}})\} \text{ and} \\
 & & \epsilon_2 = \{(\mathbf{Int} \rightarrow \mathbf{Int}, \mathtt{inj}_{\mathbf{X}} \rightarrow \mathtt{refl}_{\mathbf{Int}}, \mathtt{inj}_{\mathbf{Int}} \rightarrow \mathtt{refl}_{\mathbf{Int}}), \\
 & & \qquad (? \rightarrow \mathbf{Int}, \mathtt{inj}_? \rightarrow \mathtt{refl}_{\mathbf{Int}}, \mathtt{inj}_? \rightarrow \mathtt{refl}_{\mathbf{Int}})\} \text{ and } \epsilon_3 = \{(\mathbf{Int}, \mathtt{refl}_{\mathbf{Int}}, \mathtt{inj}_{\mathbf{Int}})\} \\
(\text{RappG}) & \longmapsto & (\epsilon_2\ (\lambda \mathbf{x} : ?_{\mathbf{X}:\mathbf{Int}}.\epsilon_1'\ \mathbf{x} :: \mathbf{Int}) :: ? \rightarrow \mathbf{Int})\ (\epsilon_3\ 42 :: ?) \quad \text{where } \epsilon_1' = \{(\mathbf{Int}, \mathtt{inj}_{\mathbf{X}}, \mathtt{refl}_{\mathbf{Int}})\} \\
(\text{Rapp}) & \longmapsto & \mathit{cod}(\epsilon_2)\ (\epsilon_1'\ (\epsilon_3'\ 42 :: ?) :: \mathbf{Int}) :: \mathbf{Int} \quad \text{where } \epsilon_3' = \epsilon_3 \circ \mathit{dom}(\epsilon_2) = \\
 & & \{(\mathbf{Int}, \mathtt{refl}_{\mathbf{Int}}, \mathtt{inj}_{\mathbf{X}}), (\mathbf{Int}, \mathtt{refl}_{\mathbf{Int}}, \mathtt{inj}_{\mathbf{Int}})\} \\
(\text{Rasc}) & \longmapsto & \mathit{cod}(\epsilon_2)\ (\epsilon_4\ 42 :: \mathbf{Int}) :: \mathbf{Int} \quad \text{where } \epsilon_4 = \epsilon_3' \circ \epsilon_1' = \{(\mathbf{Int}, \mathtt{refl}_{\mathbf{Int}}, \mathtt{refl}_{\mathbf{Int}})\} \\
(\text{Rasc}) & \longmapsto & \epsilon_5\ 42 :: \mathbf{Int} \quad \text{where } \epsilon_5 = \epsilon_4 \circ \mathit{cod}(\epsilon_2) = \{(\mathbf{Int}, \mathtt{refl}_{\mathbf{Int}}, \mathtt{refl}_{\mathbf{Int}})\}
\end{array}$$

We now illustrate the reduction of Example (3) from Section 2.3: $(\Lambda X.\lambda x : ?.x :: X)\ [\mathtt{Bool}]\ 42$. The elaboration and reduction are as follows:

$$\begin{array}{lll}
 & & (\epsilon_2\ (\Lambda \mathbf{X}.\lambda \mathbf{x} : ?_{\mathbf{X}}.\epsilon_1\ \mathbf{x} :: \mathbf{X}\ [\mathbf{Bool}]) :: ? \rightarrow \mathbf{Bool})\ (\epsilon_3\ 42 :: ?) \quad \text{where } \epsilon_1 = \{(\mathbf{X}, \mathtt{inj}_{\mathbf{X}}, \mathtt{refl}_{\mathbf{X}})\} \text{ and} \\
 & & \epsilon_2 = \{(\mathbf{Bool} \rightarrow \mathbf{Bool}, \mathtt{inj}_{\mathbf{X}} \rightarrow \mathtt{refl}_{\mathbf{Bool}}, \mathtt{inj}_{\mathbf{Bool}} \rightarrow \mathtt{refl}_{\mathbf{Bool}}), \\
 & & \qquad (? \rightarrow \mathbf{Bool}, \mathtt{inj}_? \rightarrow \mathtt{refl}_{\mathbf{Bool}}, \mathtt{inj}_? \rightarrow \mathtt{refl}_{\mathbf{Bool}})\} \text{ and } \epsilon_3 = \{(\mathbf{Int}, \mathtt{refl}_{\mathbf{Int}}, \mathtt{inj}_{\mathbf{Int}})\} \\
(\text{RappG}) & \longmapsto & (\epsilon_2\ (\lambda \mathbf{x} : ?_{\mathbf{X}:\mathbf{Bool}}.\epsilon_1'\ \mathbf{x} :: \mathbf{Bool}) :: ? \rightarrow \mathbf{Bool})\ (\epsilon_3\ 42 :: ?) \quad \text{where } \epsilon_1' = \{(\mathbf{Bool}, \mathtt{inj}_{\mathbf{X}}, \mathtt{refl}_{\mathbf{Bool}})\} \\
(\text{Rapp}) & \longmapsto & \mathit{cod}(\epsilon_2)\ (\epsilon_1'\ (\epsilon_3'\ 42 :: ?) :: \mathbf{Bool}) :: \mathbf{Bool} \quad \text{where } \epsilon_3' = \epsilon_3 \circ \mathit{dom}(\epsilon_2) = \{(\mathbf{Int}, \mathtt{refl}_{\mathbf{Int}}, \mathtt{inj}_{\mathbf{Int}})\} \\
(\text{Rasc}) & \longmapsto & \mathbf{error} \quad \text{because } \{(\mathbf{Int}, \mathtt{refl}_{\mathbf{Int}}, \mathtt{inj}_{\mathbf{Int}})\} \circ \{(\mathbf{Bool}, \mathtt{inj}_{\mathbf{X}}, \mathtt{refl}_{\mathbf{Bool}})\} \text{ is undefined}
\end{array}$$

Finally, we show the reduction of Example (5) from Section 2.3, which illustrates the prevention of a violation of parametricity at runtime: $(\Lambda X.\lambda x : X.(x :: ?) + 1)\ [\mathtt{Int}]\ 3$. The elaboration and reduction are as follows:

$$\begin{array}{lll}
 & & (\epsilon_3\ ((\Lambda \mathbf{X}.\lambda \mathbf{x} : \mathbf{X}.(\epsilon_2\ (\epsilon_1\ \mathbf{x} :: ?_{\mathbf{X}}) :: \mathbf{Int}) + (\epsilon_{\mathtt{Int}}\ 1 :: \mathbf{Int}))\ [\mathbf{Int}]) :: \mathbf{Int} \rightarrow \mathbf{Int})\ (\epsilon_{\mathtt{Int}}\ 3 :: \mathtt{Int}) \\
 & & \text{where } \epsilon_1 = \{(\mathbf{X}, \mathtt{refl}_{\mathbf{X}}, \mathtt{inj}_{\mathbf{X}})\}, \epsilon_2 = \{(\mathbf{Int}, \mathtt{inj}_{\mathbf{Int}}, \mathtt{refl}_{\mathbf{Int}})\}, \epsilon_{\mathtt{Int}} = \{(\mathbf{Int}, \mathtt{refl}_{\mathbf{Int}}, \mathtt{refl}_{\mathbf{Int}})\} \\
 & & \qquad \text{and } \epsilon_3 = \{(\mathbf{Int} \rightarrow \mathbf{Int}, \mathtt{refl}_{\mathbf{Int}} \rightarrow \mathtt{refl}_{\mathbf{Int}}, \mathtt{refl}_{\mathbf{Int}} \rightarrow \mathtt{refl}_{\mathbf{Int}})\} \\
(\text{RappG}) & \longmapsto & (\epsilon_3\ (\lambda \mathbf{x} : \mathbf{Int}.(\epsilon_2\ (\epsilon_1'\ \mathbf{x} :: ?_{\mathbf{X}:\mathbf{Int}}) :: \mathbf{Int}) + (\epsilon_{\mathtt{Int}}\ 1 :: \mathbf{Int})) :: \mathbf{Int} \rightarrow \mathbf{Int})\ (\epsilon_{\mathtt{Int}}\ 3 :: \mathtt{Int}) \\
 & & \text{where } \epsilon_1' = \{(\mathbf{Int}, \mathtt{refl}_{\mathbf{Int}}, \mathtt{inj}_{\mathbf{X}})\} \\
(\text{Rapp}) & \longmapsto & \mathit{cod}(\epsilon_3)\ ((\epsilon_2\ (\epsilon_1'\ (\epsilon_{\mathtt{Int}}\ 3 :: \mathtt{Int}) :: ?_{\mathbf{X}:\mathbf{Int}}) :: \mathbf{Int}) + (\epsilon_{\mathtt{Int}}\ 1 :: \mathbf{Int})) :: \mathbf{Int} \ \text{ where } \epsilon_{\mathtt{Int}} \circ \mathit{dom}(\epsilon_3) = \epsilon_{\mathtt{Int}} \\
(\text{Rasc}) & \longmapsto & \mathit{cod}(\epsilon_3)\ ((\epsilon_2\ (\epsilon_1'\ 3 :: ?_{\mathbf{X}:\mathbf{Int}}) :: \mathbf{Int}) + (\epsilon_{\mathtt{Int}}\ 1 :: \mathbf{Int})) :: \mathbf{Int} \ \text{ where } \epsilon_{\mathtt{Int}} \circ \epsilon_1' = \epsilon_1' \\
(\text{Rasc}) & \longmapsto & \mathbf{error} \quad \text{because } \epsilon_1' \circ \epsilon_2 = \{(\mathbf{Int}, \mathtt{refl}_{\mathbf{Int}}, \mathtt{inj}_{\mathbf{X}})\} \circ \{(\mathbf{Int}, \mathtt{inj}_{\mathbf{Int}}, \mathtt{refl}_{\mathbf{Int}})\} \text{ is undefined}
\end{array}$$

5 $\mathrm{F}_\varepsilon^?$: GRADUAL PARAMETRICITY

In this section, we present parametricity for $\mathrm{F}_\varepsilon^?$. Since gradual types support non-terminating terms, we use a standard technique for establishing this result: *step-indexed* logical relations [Ahmed 2006;

$$
\begin{array}{ll}
\mathcal{V}_\rho[\![B]\!] & = \{(n, \mathrm{v}, \mathrm{v}) \in \textsc{Atom}_\rho[B]\} \\
\mathcal{V}_\rho[\![\mathrm{G}_1 \rightarrow \mathrm{G}_2]\!] & = \{(n, \mathrm{v}_1, \mathrm{v}_2) \in \textsc{Atom}_\rho[\mathrm{G}_1 \rightarrow \mathrm{G}_2] \ \mid\ \forall n' \leq n, \mathrm{v}_1', \mathrm{v}_2'. \\
& \qquad \triangleright\ (n', \mathrm{v}_1', \mathrm{v}_2') \in \mathcal{V}_\rho[\![\mathrm{G}_1]\!] \Rightarrow (n', \mathrm{v}_1\ \mathrm{v}_1', \mathrm{v}_2\ \mathrm{v}_2') \in \mathcal{T}_\rho[\![\mathrm{G}_2]\!]\} \\
\mathcal{V}_\rho[\![\forall \mathrm{X}.\mathrm{G}]\!] & = \{(n, \mathrm{v}_1, \mathrm{v}_2) \in \textsc{Atom}_\rho[\forall \mathrm{X}.\mathrm{G}] \ \mid\ \forall \vdash \mathrm{B}_1, \vdash \mathrm{B}_2, R \in \textsc{Rel}[\mathrm{B}_1, \mathrm{B}_2]. \\
& \qquad (n, \mathrm{v}_1\ [\mathrm{B}_1], \mathrm{v}_2\ [\mathrm{B}_2]) \in \mathcal{T}_{\rho;\mathrm{X}\mapsto(\mathrm{B}_1,\mathrm{B}_2,R)}[\![\mathrm{G}]\!]\} \\
\mathcal{V}_\rho[\![\mathrm{X}]\!] & = \rho.R(\mathrm{X}) \\
\mathcal{V}_\rho[\![?_\delta]\!] & = \{(n, \mathrm{v}_1, \mathrm{v}_2) \in \textsc{Atom}_\rho[?_\delta] \ \mid\ \forall \mathrm{G_R}, \epsilon, \vdash \epsilon : \delta \rightarrowtriangle \mathrm{G_R}. \\
& \qquad (n, \rho_1(\epsilon)\ \mathrm{v}_1 :: \rho_1(\mathrm{G_R}), \rho_2(\epsilon)\ \mathrm{v}_2 :: \rho_2(\mathrm{G_R})) \in \mathcal{T}_\rho[\![\mathrm{G_R}]\!]\} \\
\hline
\mathcal{T}_\rho[\![\mathrm{G}]\!] & = \{(n, \mathrm{t}_1, \mathrm{t}_2) \in \textsc{Atom}_\rho[\mathrm{G}] \ \mid\ \forall i < n. \\
& \qquad (\forall \mathrm{v}_1.\ \mathrm{t}_1 \longmapsto^{i} \mathrm{v}_1 \Rightarrow \exists \mathrm{v}_2.\ \mathrm{t}_2 \overset{*}{\longmapsto} \mathrm{v}_2 \wedge \triangleright^{i}\ (n, \mathrm{v}_1, \mathrm{v}_2) \in \mathcal{V}_\rho[\![\mathrm{G}]\!]) \wedge \\
& \qquad (\mathrm{t}_1 \longmapsto^{i} \mathbf{error} \Rightarrow \mathrm{t}_2 \overset{*}{\longmapsto} \mathbf{error})\} \\
\hline
\mathcal{D}[\![\emptyset]\!] & = \{(n, \emptyset)\} \\
\mathcal{D}[\![\Delta, \mathrm{X}]\!] & = \{(n, \rho[\mathrm{X} \mapsto (\mathrm{B}_1, \mathrm{B}_2, R)]) \mid (n, \rho) \in \mathcal{D}[\![\Delta]\!] \wedge R \in \textsc{Rel}[\mathrm{B}_1, \mathrm{B}_2]\} \\
\hline
\mathcal{G}_\rho[\![\emptyset]\!] & = \{(n, \emptyset)\} \\
\mathcal{G}_\rho[\![\Gamma, \mathrm{x} : \mathrm{G}]\!] & = \{(n, \gamma[\mathrm{x} \mapsto (\mathrm{v}_1, \mathrm{v}_2)]) \mid (n, \gamma) \in \mathcal{G}_\rho[\![\Gamma]\!] \wedge (n, \mathrm{v}_1, \mathrm{v}_2) \in \mathcal{V}_\rho[\![\mathrm{G}]\!]\} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\Delta; \Gamma \vdash \mathrm{t}_1 \preceq \mathrm{t}_2 : \mathrm{G} \triangleq \Delta; \Gamma \vdash \mathrm{t}_1 : \mathrm{G} \wedge \Delta; \Gamma \vdash \mathrm{t}_2 : \mathrm{G} \wedge \forall n, \rho, \gamma. ((n, \rho) \in \mathcal{D}[\![\Delta]\!] \wedge (n, \gamma) \in \mathcal{G}_\rho[\![\Gamma]\!]) \Rightarrow \\
\qquad\qquad\qquad (n, \rho_1(\gamma_1(\mathrm{t}_1)), \rho_2(\gamma_2(\mathrm{t}_2))) \in \mathcal{T}_\rho[\![\mathrm{G}]\!] \\
\Delta; \Gamma \vdash \mathrm{t}_1 \approx \mathrm{t}_2 : \mathrm{G} \triangleq \Delta; \Gamma \vdash \mathrm{t}_1 \preceq \mathrm{t}_2 : \mathrm{G} \wedge \Delta; \Gamma \vdash \mathrm{t}_2 \preceq \mathrm{t}_1 : \mathrm{G} \\
\hline
\end{array}
$$

$$\textsc{Atom}_\rho[\mathrm{G}] = \{(n, \mathrm{t}_1, \mathrm{t}_2) \in \textsc{Atom}[\rho.1(\mathrm{G}), \rho.2(\mathrm{G})]\}$$

$$\textsc{Atom}[\mathrm{G}_1, \mathrm{G}_2] = \{(n, \mathrm{t}_1, \mathrm{t}_2) \mid\ \vdash \mathrm{t}_1 : \mathrm{G}_1 \wedge\ \vdash \mathrm{t}_2 : \mathrm{G}_2\} \qquad \textsc{Atom}^{\mathrm{val}}[\mathrm{G}_1, \mathrm{G}_2] = \{(n, \mathrm{v}_1, \mathrm{v}_2) \in \textsc{Atom}[\mathrm{G}_1, \mathrm{G}_2]\}$$

$$\textsc{Rel}[\mathrm{G}_1, \mathrm{G}_2] = \{R \subseteq \textsc{Atom}^{\mathrm{val}}[\mathrm{G}_1, \mathrm{G}_2] \mid \forall n' \leq n, \mathrm{v}_1, \mathrm{v}_2.\ (n, \mathrm{v}_1, \mathrm{v}_2) \in R \Rightarrow (n', \mathrm{v}_1, \mathrm{v}_2) \in R\}$$

Fig. 5. Gradual logical relation and auxiliary definitions.

Appel and McAllester 2001]. Step indexing ensures the well-foundedness of the logical relation. We start by defining the logical relation for values and terms, and then we establish the fundamental property or parametricity. Our proposal is the first gradual polymorphic language to support a formulation of parametricity where semantic types are tracked in a lexical environment, similar to traditional formulations of parametricity [Reynolds 1983].

Logical relations. Figure 5 presents the logical relation for parametricity along with some auxiliary definitions. The relational interpretation is presented using *atoms* of the form $(n, \mathrm{t}_1, \mathrm{t}_2) \in \textsc{Atom}[\mathrm{G}_1, \mathrm{G}_2]$, where n denotes the step index, and t_1 and t_2 denote closed well-typed terms at types G_1 and G_2 respectively. The logical relation is defined using two mutually-defined interpretations: one for values $\mathcal{V}_\rho[\![\mathrm{G}]\!]$ and one for computations $\mathcal{T}_\rho[\![\mathrm{G}]\!]$. Both interpretations are indexed by a type G, and an environment ρ, which maps type variables to two types G_1 and G_2 and a relation $R \in \textsc{Rel}[\mathrm{G}_1, \mathrm{G}_2]$. $\textsc{Rel}[\mathrm{G}_1, \mathrm{G}_2]$ defines the set of all admissible relations R such that $R \subseteq \textsc{Atom}^{\mathrm{val}}[\mathrm{G}_1, \mathrm{G}_2]$ (the subset of atoms where terms are values). For convenience, if $\rho = \{\overline{\mathrm{X_i} \mapsto (\mathrm{G_{i1}}, \mathrm{G_{i2}}, R_i)}\}$, then $\rho.1$, $\rho.2$, and $\rho.R$ are abbreviations for $\{\overline{\mathrm{X_i} \mapsto \mathrm{G_{i1}}}\}$ and $\{\overline{\mathrm{X_i} \mapsto \mathrm{G_{i2}}}\}$, and $\{\overline{\mathrm{X_i} \mapsto R_i}\}$ respectively. Thus $\rho.j(\mathrm{G})$ is an abbreviation for multiple substitutions $\mathrm{G}[\overline{\mathrm{X_i} \mapsto \mathrm{G_{ij}}}]$. Finally, $\textsc{Atom}_\rho[\mathrm{G}]$ denotes the set of atoms $\textsc{Atom}[\rho.1(\mathrm{G}), \rho.2(\mathrm{G})]\}$.

Logical relation for values. The definition of related values is standard except for the unknown type. Two base values of type B are related if they are the same. Two functions are related at type $\mathrm{G}_1 \rightarrow \mathrm{G}_2$, if given two related arguments at type G_1 (and a strictly smaller index), the application yields related computations at type G_2. We use notation $\triangleright^i\ (n, v_1, v_2) \in \mathcal{V}_\rho\llbracket G \rrbracket$ as an abbreviation for $(n-i, v_1, v_2) \in \mathcal{V}_\rho\llbracket G \rrbracket$, and $\triangleright\ (n, v_1, v_2) \in \mathcal{V}_\rho\llbracket G \rrbracket$ for $\triangleright^1\ (n, v_1, v_2) \in \mathcal{V}_\rho\llbracket G \rrbracket$. Two type abstractions are related if their instantiations to two arbitrary base types yields related computations for any given relation between the instantiated types. Two values are related at an abstract type X, if they are contained in the relation for X. Two values are related at the unknown type $?_\delta$, if given any evidence ϵ that justifies that any G_R *is ground* with respect to δ, notation $\vdash \epsilon : \delta \rightarrowtriangle G_R$, then both values ascribed to $\rho.1(G_R)$ and $\rho.2(G_R)$, using $\rho.1(\epsilon)$ and $\rho.2(\epsilon)$ respectively, are related computations at type G_R. This definition captures the fact that if two values are related at $?_\delta$, they are also related at some more precise ground type, either X, B, $?_\delta \rightarrow ?_\delta$, or $\forall X.?_{\delta,X}$, after removing the respective injections to the unknown type with the evidences $\rho.1(\epsilon)$ and $\rho.2(\epsilon)$. Both λB and GSF use similar approaches for defining the logical relation for values of type unknown but are formalized differently, according to the syntax of the considered languages. Relation $\vdash \epsilon : \delta \rightarrowtriangle G_R$ is defined such that G_R is a *ground type* restricted to δ, and $\epsilon : ?_\delta \sim G_R$:

$$\frac{}{\vdash \{(B, \mathtt{inj}_B, \mathtt{refl}_B)\} : \delta \rightarrowtriangle B} \qquad \frac{}{\vdash \{(?_\delta \rightarrow ?_\delta, \mathtt{inj}_\rightarrow, \mathtt{inj}_? \rightarrow \mathtt{inj}_?)\} : \delta \rightarrowtriangle ?_\delta \rightarrow ?_\delta}$$

$$\frac{X : F \in \delta \qquad \delta \vdash F}{\vdash \{(F, \mathtt{inj}_X, \mathtt{refl}_F)\} : \delta \rightarrowtriangle F} \qquad \frac{}{\vdash \{(\forall X.?_{\delta,X}, \mathtt{inj}_\forall, \forall X.\mathtt{inj}_?)\} : \delta \rightarrowtriangle \forall X.?_{\delta,X}}$$

Let us illustrate the interpretation of the unknown type with examples. Consider the evidences: $\epsilon_{Int?} = \{(Int, \mathtt{refl}_{Int}, \mathtt{inj}_{Int})\}$, $\epsilon_{Int} = \{(Int, \mathtt{refl}_{Int}, \mathtt{refl}_{Int})\}$ and $\epsilon_{X?} = \{(Int, \mathtt{refl}_{Int}, \mathtt{inj}_X)\}$, where $\delta = X : X$, and $\delta' = X : Int$.

- $(n, \epsilon_{Int?} 42 :: ?_{\delta'}, \epsilon_{Int?} 42 :: ?_{\delta'}) \in \mathcal{V}_\rho\llbracket ?_\delta \rrbracket$ because for $\epsilon = \{(Int, \mathtt{inj}_{Int}, \mathtt{refl}_{Int})\}$ and $\vdash \epsilon : \delta \rightarrowtriangle Int$, as $\epsilon_{Int?} \circ \rho.i(\epsilon) = \epsilon_{Int}$, then $\triangleright\ (n, \epsilon_{Int} 42 :: Int, \epsilon_{Int} 42 :: Int) \in \mathcal{V}_\rho\llbracket Int \rrbracket$ (and for every other evidence ϵ and G_R, such that $\vdash \epsilon : \delta \rightarrowtriangle G_R$, consistent transitivity is not defined).
- $(n, \epsilon_{Int?} 42 :: ?_{\delta'}, \epsilon_{Int?} 43 :: ?_{\delta'}) \notin \mathcal{V}_\rho\llbracket ?_\delta \rrbracket$ because for $\vdash \{(Int, \mathtt{inj}_{Int}, \mathtt{refl}_{Int})\} : \delta \rightarrowtriangle Int$, as $\triangleright\ (n, \epsilon_{Int} 42 :: Int, \epsilon_{Int} 43 :: Int) \notin \mathcal{V}_\rho\llbracket Int \rrbracket$.
- Suppose $\triangleright\ (n, \epsilon_{Int} 42 :: Int, \epsilon_{Int} 43 :: Int) \in \rho.R(X)$. Then $(n, \epsilon_{X?} 42 :: ?_{\delta'}, \epsilon_{X?} 43 :: ?_{\delta'}) \in \mathcal{V}_\rho\llbracket ?_\delta \rrbracket$ because for $\epsilon = \{(X, \mathtt{inj}_X, \mathtt{refl}_X)\}$ and $\vdash \epsilon : \delta \rightarrowtriangle X$, as $\epsilon_{X?} \circ \rho.i(\epsilon) = \epsilon_{Int}$, then $\triangleright$ $(n, \epsilon_{Int} 42 :: Int, \epsilon_{Int} 43 :: Int) \in \mathcal{V}_\rho\llbracket X \rrbracket = \rho.R(X)$ (and for every other evidence ϵ and G_R, such that $\vdash \epsilon : \delta \rightarrowtriangle G_R$, consistent transitivity is not defined).
- But $(n, \epsilon_{X?} 42 :: ?_{\delta'}, \epsilon_{X?} 43 :: ?_{\delta'}) \notin \mathcal{V}_\rho\llbracket ?_{\delta'} \rrbracket$ because for $\epsilon = \{(Int, \mathtt{inj}_X, \mathtt{refl}_{Int})\}$ and $\vdash \epsilon : \delta' \rightarrowtriangle Int$, as $\epsilon_X \circ \rho.i(\epsilon) = \epsilon_{Int}$, but $\triangleright\ (n, \epsilon_{Int} 42 :: Int, \epsilon_{Int} 43 :: Int) \notin \mathcal{V}_\rho\llbracket Int \rrbracket$.
- Suppose $\rho.R(X) = \mathcal{V}_\rho\llbracket Int \rrbracket$, and $\epsilon_m = \epsilon_{Int?} \cup \epsilon_{X?}$. Then $(n, \epsilon_m 42 :: ?_{\delta'}, \epsilon_m 42 :: ?_{\delta'}) \in \mathcal{V}_\rho\llbracket ?_\delta \rrbracket$ because (1) for $\epsilon = \{(X, \mathtt{inj}_X, \mathtt{refl}_X)\}$ and $\vdash \epsilon : \delta \rightarrowtriangle X$, as $\epsilon_m \circ \rho.i(\epsilon) = \epsilon_{X?} \circ \rho.i(\epsilon) = \epsilon_{Int}$, then $\triangleright\ (n, \epsilon_{Int} 42 :: Int, \epsilon_{Int} 42 :: Int) \in \mathcal{V}_\rho\llbracket X \rrbracket = \rho.R(X) = \mathcal{V}_\rho\llbracket Int \rrbracket$; and (2) for $\epsilon = \{(Int, \mathtt{inj}_{Int}, \mathtt{refl}_{Int})\}$, $\vdash \epsilon : \delta \rightarrowtriangle Int$, as $\epsilon_m \circ \rho.i(\epsilon) = \epsilon_{Int?} \circ \rho.i(\epsilon) = \epsilon_{Int}$, then $\triangleright\ (n, \epsilon_{Int} 42 :: Int, \epsilon_{Int} 42 :: Int) \in \mathcal{V}_\rho\llbracket Int \rrbracket$.

Note that for the case of functions, type applications, and the unknown type, although the same step index is used in every recursive reasoning, the relations are well-formed as in each case a single step of reduction is always taken, lowering the index by one.

Logical relation for terms. Two computations are related at n steps if the first term yields a value in $i < n$ reduction steps, then the second must produce a value related at that type at $n - i$ steps; and if the first term fails, then the second also fails.

Logical relation for environments. The interpretation of environment Δ, specifies all type substitutions ρ, such that all type variables in Δ are mapped to a pair of base types and a relation at those types. The interpretation of environment Γ, specifies all value substitution γ, such that every variable of type G is mapped to a pair of related values at that type.

Parametricity. The logical approximation $\Delta; \Gamma \vdash t_1 \preceq t_2 : G$ states that given any step index, any environments ρ and γ that satisfy Δ and Γ respectively, the substituted terms are related computations. Similarly to ρ, for convenience if $\gamma = \{\overline{x \mapsto (v_{i1}, v_{i2})}\}$, then $\gamma_j = \{\overline{x \mapsto v_{ij}}\}$. Finally, the fundamental property states that any well-typed term logically approximates itself.

Theorem 5.1 (Fundamental Property). *If* $\Delta; \Gamma \vdash t : G$ *then* $\Delta; \Gamma \vdash t \preceq t : G$.

As standard [Ahmed 2004], the proofs of the fundamental property depends on numerous compatibility lemmas for each term constructor and the compositionality lemma, which in this work resembles compositionality for System F.

Lemma 5.2 (Compositionality). *Let* $\Delta \vdash F$, $\Delta, X \vdash G$, $(n, \rho) \in \mathcal{D}[\![\Delta]\!]$, *and* $R = \mathcal{V}_\rho[\![F]\!]$, *then* $\mathcal{V}_\rho[\![G[F/X]]\!] = \mathcal{V}_{\rho, X \mapsto (\rho_1(F), \rho_2(F), R)}[\![G]\!]$.

The most important lemma, used by almost all compatibility lemmas and compositionality is the ascription lemma, which says that the ascription of two related values yields related computations.

Lemma 5.3 (Ascription Lemma). *If* $(n, v_1, v_2) \in \mathcal{V}_\rho[\![G]\!]$, $(n, \rho) \in \mathcal{D}[\![\Delta]\!]$, $\Delta \vdash G'$ *and* $\epsilon : G \sim G'$, *then* $(n, \rho_1(\epsilon)v_1 :: \rho_1(G'), \rho_2(\epsilon)v_2 :: \rho_2(G')) \in \mathcal{T}_\rho[\![G']\!]$.

We finalize this section by emphasizing that most of the logical relations and main lemmas are standard and defined just as in System F. In particular, and contrary to other gradual parametricity formulations, the definition of related values at polymorphic types is defined just as in System F, without the need for special notations and cases. The only unusual case is the definition of related values at the unknown type, but that is expected for any gradual language.

6 $F^?_\varepsilon$: GRADUAL GUARANTEES

This section presents graduality for $F^?_\varepsilon$. We start by presenting the definition of evidence and term precision. Similar to type precision, these definitions are also proof-relevant. Then, we show two of the main challenges of proving graduality: monotonicity of consistent transitivity and monotonicity of type substitution over evidence. We end this section by establishing graduality, more specifically, the static and dynamic gradual guarantees [Siek et al. 2015].

Evidence precision. To define precision between evidence we start by stating two intuitive requirements. Suppose $\epsilon_1 : G_1 \sim G_1'$ and $\epsilon_2 : G_2 \sim G_2'$. We say that ϵ_1 is more precise than ϵ_2, if first, the types involved in the judgments are related by precision, *i.e.* $c : G_1 \sqsubseteq G_2$ and $c' : G_1' \sqsubseteq G_2'$ for some c and c'; and second, we require that for all $S_1 \in \epsilon_1$ there exists some $S_2 \in \epsilon_2$, such that S_1 is more precise than S_2. Note that there may be some $S \in \epsilon_2$ not in precision with any element of ϵ_1. This is intuitively expected by graduality, as it may cause ϵ_2 to “fail less” than ϵ_1 when combined with other evidence. Precision between spans $(G_{t1}, c_1, c_1') \sqsubseteq (G_{t2}, c_2, c_2')$ could be naively defined if there exists some proof term c_t that justifies that G_{t1} is more precise than G_{t2}, i.e. $c_t : G_{t1} \sqsubseteq G_{t2}$.

However, the above requirements are not sufficient to define precision among evidences. Suppose that we have $\epsilon_1 = \{(\mathsf{Int}, \mathtt{refl}_{\mathsf{Int}}, \mathtt{inj}_X)\}$ and $\epsilon_2 = \{(\mathsf{Int}, \mathtt{refl}_{\mathsf{Int}}, \mathtt{inj}_{\mathsf{Int}})\}$, where $\epsilon_i : \mathsf{Int} \sim ?_{X:\mathsf{Int}}$ (Figure 6 supports this example). These two evidences meet all the above requirements: there exist $c = \mathtt{refl}_{\mathsf{Int}}$, $c' = \mathtt{inj}_?$, and $c_t = \mathtt{refl}_{\mathsf{Int}}$ such that $c : \mathsf{Int} \sqsubseteq \mathsf{Int}$, $c' : ?_{X:\mathsf{Int}} \sqsubseteq ?_{X:\mathsf{Int}}$ and $c_t : \mathsf{Int} \sqsubseteq \mathsf{Int}$. We may be tempted to say that $\epsilon_1 \sqsubseteq \epsilon_2$ (or vice versa), but then graduality would not hold. In particular, monotonicity of consistent transitivity (MCT), a key lemma used to prove graduality, would be broken. MCT states that given two pairs of evidence related by precision $\epsilon_1 \sqsubseteq \epsilon_2$ and $\epsilon_1' \sqsubseteq \epsilon_2'$, if $\epsilon_1 \circ \epsilon_1'$ is defined, then $\epsilon_1 \circ \epsilon_1' \sqsubseteq \epsilon_2 \circ \epsilon_2'$. In the example, if we take evidence $\epsilon_1' = \epsilon_2' = \{(\mathsf{Int},$

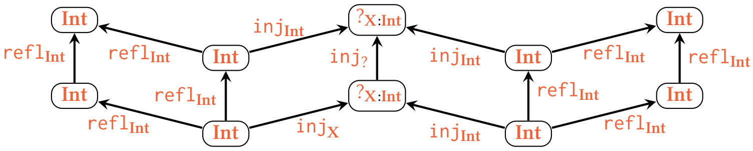
Int
?X:Int
Int
Int
Int
Int
Int
?X:Int
Int
Int
$\mathtt{refl}_{\mathrm{Int}}$
$\mathtt{refl}_{\mathrm{Int}}$
$\mathtt{inj}_{\mathrm{Int}}$
$\mathtt{inj}_{?}$
$\mathtt{inj}_{\mathrm{Int}}$
$\mathtt{refl}_{\mathrm{Int}}$
$\mathtt{refl}_{\mathrm{Int}}$
$\mathtt{refl}_{\mathrm{Int}}$
$\mathtt{refl}_{\mathrm{Int}}$
$\mathtt{refl}_{\mathrm{Int}}$
$\mathtt{inj}_{X}$
$\mathtt{inj}_{\mathrm{Int}}$
$\mathtt{refl}_{\mathrm{Int}}$

Fig. 6. Evidence precision auxiliary example.

$\mathtt{inj}_{\mathrm{Int}}, \mathtt{refl}_{\mathrm{Int}})\}$ that justifies $?_{X:\mathrm{Int}} \sim \mathrm{Int}$ (and $\epsilon_1' \sqsubseteq \epsilon_2'$), then $\epsilon_1 \circ \epsilon_1' = \{(\mathrm{Int}, \mathtt{refl}_{\mathrm{Int}}, \mathtt{refl}_{\mathrm{Int}})\}$ is defined but $\epsilon_2 \circ \epsilon_2'$ is not. We can use an analogous reasoning when assuming $\epsilon_2 \sqsubseteq \epsilon_1$, using $\epsilon_1' = \epsilon_2' = \{(\mathrm{Int}, \mathtt{inj}_{X}, \mathtt{refl}_{\mathrm{Int}})\}$. These two evidences should not be related by precision; we miss a connection between c_t and both c' and c as described next.

Definition 6.1 (Evidence Precision). If $\epsilon_1 : G_1 \sim G_1'$, $\epsilon_2 : G_2 \sim G_2'$, $c : G_1 \sqsubseteq G_2$ and $c' : G_1' \sqsubseteq G_2'$, then we say that $[c]\epsilon_1 \sqsubseteq \epsilon_2[c']$ iff for all $(G_{t1}, c_1, c_1') \in \epsilon_1$ there exists a $(G_{t2}, c_2, c_2') \in \epsilon_2$, c_1'', c_2'' and c_t such that $c_t : G_{t1} \sqsubseteq G_{t2}$, $c_t;c_2 = c_1''$, $c_1;c = c_1''$, $c_t;c_2' = c_2''$ and $c_1';c' = c_2''$.

$$\begin{tikzcd}
G_2 & & & & G_2' \\
& & G_{t2} \arrow[ull, "c_2"'] \arrow[urr, "c_2'"] & & \\
G_1 \arrow[uu, "c"] & & & & G_1' \arrow[uu, "c'"'] \\
& & G_{t1} \arrow[ul, "c_1"] \arrow[uuull, "c_1''"] \arrow[uu, "c_t"'] \arrow[uuurr, "c_2''"'] \arrow[urr, "c_1'"'] & &
\end{tikzcd}$$

In addition to the requirements described above, evidence precision also requires that the combination of c_t and c_2 must commute with the combination of c_1 and c; similarly, the combination of c_t and c_2' must commute with the combination of c_1' and c'. Going back to the example, ϵ_1 and ϵ_2 are not related by precision as the diagram does not commute: $\mathtt{refl}_{\mathrm{Int}};\mathtt{inj}_{\mathrm{Int}} = \mathtt{inj}_{\mathrm{Int}}$ ($c_t;c_2' = c_2''$) and $\mathtt{inj}_{X};\mathtt{inj}_{?} = \mathtt{inj}_{X}$ ($c_1';c' = c_2''$), but $\mathtt{inj}_{\mathrm{Int}} \neq \mathtt{inj}_{X}$. On the other hand, evidence $\{(\mathrm{Int}, \mathtt{refl}_{\mathrm{Int}}, \mathtt{refl}_{\mathrm{Int}})\}$ is more precise than $\{(\mathrm{Int}, \mathtt{refl}_{\mathrm{Int}}, \mathtt{inj}_{X})\}$, because we can choose $c = c_t = \mathtt{refl}_{\mathrm{Int}}$, and $c' = \mathtt{inj}_{X}$, such that the diagram commutes: $\mathtt{refl}_{\mathrm{Int}};\mathtt{inj}_{X} = \mathtt{inj}_{X}$ ($c_t;c_2' = c_2''$), and $\mathtt{refl}_{\mathrm{Int}};\mathtt{inj}_{X} = \mathtt{inj}_{X}$ ($c_1';c' = c_2''$).

Note that the evidence precision judgment $[c]\epsilon_1 \sqsubseteq \epsilon_2[c']$ explicitly tracks proof terms c and c' (we will refer to them as *boundary proofs*). The reason is that monotonicity of consistent transitivity only holds when adjacent boundary proofs match up.

Lemma 6.2 (✓ Monotonicity of Consistent Transitivity). *If* $[c]\epsilon_1 \sqsubseteq \epsilon_2[c']$, $[c']\epsilon_1' \sqsubseteq \epsilon_2'[c'']$ *and* $(\epsilon_1 \circ \epsilon_1')$ *is defined, then* $[c](\epsilon_1 \circ \epsilon_1') \sqsubseteq (\epsilon_2 \circ \epsilon_2')[c'']$.

Let us consider the following example to understand why the "middle" boundary proof terms must match. We have that $[\mathtt{refl}_{\mathrm{Int}}]\{(\mathrm{Int}, \mathtt{refl}_{\mathrm{Int}}, \mathtt{refl}_{\mathrm{Int}})\} \sqsubseteq \{(\mathrm{Int}, \mathtt{refl}_{\mathrm{Int}}, \mathtt{inj}_{X})\}[\mathtt{inj}_{X}]$ and $[\mathtt{inj}_{\mathrm{Int}}]\{(\mathrm{Int}, \mathtt{refl}_{\mathrm{Int}}, \mathtt{refl}_{\mathrm{Int}})\} \sqsubseteq \{(\mathrm{Int}, \mathtt{inj}_{\mathrm{Int}}, \mathtt{refl}_{\mathrm{Int}})\}[\mathtt{refl}_{\mathrm{Int}}]$. The precision proofs do not match ($\mathtt{inj}_{X} \neq \mathtt{inj}_{\mathrm{Int}}$), and even though $\{(\mathrm{Int}, \mathtt{refl}_{\mathrm{Int}}, \mathtt{refl}_{\mathrm{Int}})\} \circ \{(\mathrm{Int}, \mathtt{refl}_{\mathrm{Int}}, \mathtt{refl}_{\mathrm{Int}})\}$ is defined, $\{(\mathrm{Int}, \mathtt{refl}_{\mathrm{Int}}, \mathtt{inj}_{X})\} \circ \{(\mathrm{Int}, \mathtt{inj}_{\mathrm{Int}}, \mathtt{refl}_{\mathrm{Int}})\}$ is not. Similar to consistent transitivity, type substitution over evidence is also monotonous concerning evidence precision (two evidences related by precision remain related after type substitution).

Lemma 6.3 (Monotonicity of Type Substitution). *If* $[\forall X.c']\epsilon_1 \sqsubseteq \epsilon_2[\forall X.c]$, *then* $[c'](schm(\epsilon_1)) \sqsubseteq (schm(\epsilon_2))[c]$ *and* $[c'[F/X]](schm(\epsilon_1)[F/X]) \sqsubseteq (schm(\epsilon_2)[F/X])[c[F/X]]$.

Term precision. Term precision is the natural lifting of type and evidence precision to terms, and is presented in Figure 7. Judgment $\Omega \vdash c : s_1 \sqsubseteq s_2$ denotes that term s_1 is more precise than s_2 justified by proof term c, under precision relation environment Ω. Boundary proof terms c are propagated for types, contexts, evidence, and subterms, justifying that the type of the less precise term is less precise than the type of the more precise term. Ω binds a term variable x to a type precision judgment $c : G_1 \sqsubseteq G_2$. Rule ($\sqsubseteq_x$) establishes that a term variable is related with itself

$\boxed{\Omega \vdash c : s \sqsubseteq s}$ **Term precision**

$$(\sqsubseteq_b)\ \frac{}{\Omega \vdash \mathtt{refl}_B : b \sqsubseteq b} \qquad (\sqsubseteq_x)\ \frac{\Omega(x) = c : G_1 \sqsubseteq G_2}{\Omega \vdash c : x \sqsubseteq x} \qquad (\sqsubseteq_\lambda)\ \frac{\Omega, x \mapsto\ c : G_1' \sqsubseteq G_2' \vdash c' : t_1 \sqsubseteq t_2}{\Omega \vdash c \rightarrow c' : \lambda x : G_1'.t_1 \sqsubseteq \lambda x : G_2'.t_2}$$

$$(\sqsubseteq_\Lambda)\ \frac{\Omega \vdash c : t_1 \sqsubseteq t_2}{\Omega \vdash \forall X.c : \Lambda X.t_1 \sqsubseteq \Lambda X.t_2} \qquad (\sqsubseteq\text{app})\ \frac{\Omega \vdash c' \rightarrow c : t_1 \sqsubseteq t_2 \qquad \Omega \vdash c' : t_1' \sqsubseteq t_2'}{\Omega \vdash c : t_1\ t_1' \sqsubseteq t_2\ t_2'}$$

$$(\sqsubseteq\text{appG})\ \frac{\Omega \vdash \forall X.c : t_1 \sqsubseteq t_2}{\Omega \vdash c[F/X] : t_1\ [F] \sqsubseteq t_2\ [F]} \qquad (\sqsubseteq\text{asc})\ \frac{\Omega \vdash c' : s_1 \sqsubseteq s_2 \qquad c : G_1 \sqsubseteq G_2 \qquad [c']\epsilon_1 \sqsubseteq \epsilon_2[c]}{\Omega \vdash c : \epsilon_1 s_1 :: G_1 \sqsubseteq \epsilon_2 s_2 :: G_2}$$

Fig. 7. $F^?_\varepsilon$: Term Precision (fragment).

along boundary proof term c if $x : (\ c : G_1 \sqsubseteq G_2) \in \Omega$, and Rule ($\sqsubseteq_\lambda$) extends Ω with the judgment that justifies that the argument types are in precision. Analogous to MCT, rule ($\sqsubseteq$app) requires that the domain proof term of the function matches with the proof term of the arguments. Rule ($\sqsubseteq$asc) establishes that two ascriptions are related if the sub-terms, s_1 and s_2, are in precision with proof term c', the ascribed types, G_1 and G_2, are in precision with the proof c, and evidences are in precision with the boundary proof terms c' and c.

Gradual guarantees. Armed with the definition of term precision, we now establish the graduality of $F^?_\varepsilon$ with the gradual guarantees [Siek et al. 2015].

Theorem 6.4. *Suppose* $\vdash t_1 : G_1$ *and* $\vdash c : t_1 \sqsubseteq t_2$. *Then,*

- $\vdash t_2 : G_2$ *and* $c : G_1 \sqsubseteq G_2$.
- $t_1 \overset{*}{\longmapsto} v_1$ *implies* $t_2 \overset{*}{\longmapsto} v_2$ *and* $\vdash c : v_1 \sqsubseteq v_2$.
- t_1 *diverges implies* t_2 *diverges.*

The only peculiarity of this result compared to others in the literature is that the type and term precision judgments are proof-relevant. The static part of graduality (the static gradual guarantee) ensures that if t_1 with type G_1 is more precise than t_2, justified by proof c, then t_2 has a less precise type G_2 justified by c. The dynamic part of graduality (the dynamic gradual guarantee) establishes that if the more precise term reduces to a value, then the less precise term also does, resulting in values in precision with the same type proof term c. The key lemmas to prove graduality, are MCT (Lemma 6.2), and monotonicity of type substitution (evidence precision is monotonous with respect to type substitution) (Lemma 6.3).

7 THE GRADUAL SOURCE LANGUAGE $F^?$

Having formalized the key technical innovation of this work, plausible sealing, and established both graduality and parametricity for the intermediate language $F^?_\varepsilon$, we now turn to the source language $F^?$. This section presents the static semantics of $F^?$ and its translation to $F^?_\varepsilon$. The static semantics of $F^?$ is derived systematically by applying AGT to System F_1, which is a variation of System F where type instantiations are restricted to instantiation types (*i.e.* base types and type variables). The novel translation to $F^?_\varepsilon$ plays a crucial role since it is in charge of statically generating the maybe-sealing evidence for type applications. We study the gradual guarantees for $F^?$ and the resulting source-level parametric reasoning.

$F^?$: Statics. In order to apply AGT to obtain the static semantics (*i.e.* lifting functions and predicates), we use explicit type equalities and partial type functions in the typing rules of System F_1. These partial functions also allow capturing elimination forms in a single rule that accounts for both precise and imprecise type information [Garcia et al. 2016]. For instance, functions *dom* and *cod* extract the

$\mathsf{F} ::= \mathsf{B} \mid \mathsf{X} \qquad \mathsf{G} ::= \mathsf{F} \mid \mathsf{G}{\rightarrow}\mathsf{G} \mid \forall \mathsf{X}.\mathsf{G} \mid ?_{\delta} \qquad \delta ::= \delta, \mathsf{X} : \mathsf{X} \mid \emptyset \; \mathsf{t} ::= \mathsf{b} \mid \lambda \mathsf{x} : \mathsf{G}.\mathsf{t} \mid \Lambda \mathsf{X}.\mathsf{t} \mid \mathsf{x} \mid \mathsf{t}\;\mathsf{t} \mid \mathsf{t}\;[\mathsf{F}] \mid \mathsf{t} :: \mathsf{G}$

$\boxed{\Delta; \Gamma \vdash \mathsf{t} : \mathsf{G}}$ **Term typing**

$$(\mathsf{Gasc})\;\frac{\Delta; \Gamma \vdash \mathsf{t} : \mathsf{G}' \qquad \Delta \vdash \mathsf{G} \qquad \mathsf{G}' \sim \mathsf{G}}{\Delta; \Gamma \vdash \mathsf{t} :: \mathsf{G} : \mathsf{G}} \qquad\qquad (\mathsf{GappG})\;\frac{\Delta; \Gamma \vdash \mathsf{t} : \mathsf{G} \qquad \Delta \vdash \mathsf{F}}{\Delta; \Gamma \vdash \mathsf{t}\;[\mathsf{F}] : \mathit{inst}^{\sharp}(\mathsf{G}, \mathsf{F})}$$

Fig. 8. $\mathsf{F}^{?}$: Syntax and Static Semantics (fragment).

domain and codomain types, and *inst* instantiates a polymorphic type using a substitution function $T[T'/X]$, which replaces T' for X in T. For example, the typing rule for type applications is:

$$(\text{TappT})\;\frac{\Delta; \Gamma \vdash t : T \qquad \Delta \vdash F}{\Delta; \Gamma \vdash t\;[F] : \mathit{inst}(T, F)}$$

Figure 8 presents the syntax of source gradual types G. Source gradual types are syntactically contained in the gradual types of $\mathsf{F}^{?}_{\varepsilon}$, and for simplicity throughout this section, we write G as the $\mathsf{F}^{?}_{\varepsilon}$ counterpart of G. Source gradual types restricts the scope of unknown types to not-instantiated variables only. To represent this, we index unknown types with the δ meta-variable (included in δ); and as every type variable in $?_{\delta}$ is not instantiated (i.e. of the form $\mathsf{X} : \mathsf{X}$), for simplicity we just write δ as a set of type variables. One final note is that $?$ (without a scope δ, as used in previous sections) is syntactic sugar for $?_{\Delta}$, where Δ is the set of all variables in scope at that point. A straightforward and simple translation can insert these annotations before typing.

We give meaning to gradual types G through the concretization function $\gamma(\cdot)$ (omitted for space). The meaning of the unknown type $?_{\delta}$ is the set of all well-formed static types with respect to δ (*i.e.* $\gamma(?_{\delta}) = \{\, T \mid \delta \vdash T \,\}$). The concretization function helps us define precision ($\mathsf{G}_1 \sqsubseteq \mathsf{G}_2$ if and only if $\gamma(\mathsf{G}_1) \subseteq \gamma(\mathsf{G}_2)$) and consistency ($\mathsf{G}_1 \sim \mathsf{G}_2$ if and only if there exists T_1 and T_2 such that $T_1 = T_2$, $T_1 \in \gamma(\mathsf{G}_1)$ and $T_2 \in \gamma(\mathsf{G}_2)$). Precision and consistency resemble their $\mathsf{F}^{?}_{\varepsilon}$ counterpart and can also be inductively defined. For instance, $\mathsf{X} \sim ?_{\mathsf{X}}$, but $\mathsf{X} \nsim ?_{\mathsf{Y}}$ (for $\mathsf{X} \neq \mathsf{Y}$). However, precision in $\mathsf{F}^{?}$ is no longer proof relevant: $\mathsf{F}^{?}$ contains only unknown types $?_{\delta}$ with uninstantiated type variables, so that the precision relation from $\mathsf{F}^{?}_{\varepsilon}$ (which had the structure of a category) reduces to a proof-irrelevant order relation in $\mathsf{F}^{?}$.

Figure 8 presents a fragment of the term typing rules for $\mathsf{F}^{?}$, which are obtained by replacing type predicates and functions with their corresponding liftings. The lifting is straightforward and uses the corresponding abstraction function $\alpha(\cdot)$ of $\gamma(\cdot)$, forming a Galois connection. For example, rule (Gasc) uses type consistency instead of type equality, and rule (GappG) uses the lifting of the function *inst*, defined for polymorphic types and the unknown type (*i.e.* $\mathit{inst}^{\sharp}(\forall \mathsf{X}.\mathsf{G}, \mathsf{G}') = \mathsf{G}[\mathsf{G}'/\mathsf{X}]\backslash\mathsf{X}$, $\mathit{inst}^{\sharp}(?_{\delta}, \mathsf{G}') = ?_{\delta}$ and undefined for other cases). Note that $\mathit{inst}^{\sharp}$ uses the scope removal function $\mathsf{G}\backslash\mathsf{X}$, which is removes X from the scopes of unknown types in G: $?_{\delta_1,\mathsf{X},\delta_2}\backslash\mathsf{X} = ?_{\delta_1,\delta_2}$. For instance, $(\mathsf{X}{\rightarrow}?_{\mathsf{X},\mathsf{Y}})[\mathtt{Int}/\mathsf{X}]\backslash\mathsf{X} = \mathtt{Int}{\rightarrow}?_{\mathsf{Y}}$.

$\mathsf{F}^{?}$: Elaboration to $\mathsf{F}^{?}_{\varepsilon}$. The dynamic semantics of a $\mathsf{F}^{?}$ program is given by a type-directed translation to $\mathsf{F}^{?}_{\varepsilon}$. The rules are mostly standard save for the elaboration rule for type application. Judgment $\Delta; \Gamma \vdash \mathsf{t} : \mathsf{G} \rightsquigarrow \mathsf{t}'$ expresses that term t is elaborated to t', under type variable environment Δ, and type environment Γ. The elaboration rules use the function $\mathrm{initEv}(\mathsf{G}_1, \mathsf{G}_2)$, which stands for the *initial evidence* between G_1 and G_2. It computes the least precise evidence that justifies consistency between the types, and is defined as follows:

$$\begin{array}{rlrl}
\mathrm{instEv}(B, X, F) = \mathrm{reflEv}(B) & & \mathrm{instEv}(Y, X, F) = \mathrm{reflEv}(Y) \text{ if } X \neq Y \\
\mathrm{instEv}(X, X, F) = \mathrm{reflEv}(F) & & \mathrm{instEv}(G_1 \rightarrow G_2, X, F) = \mathrm{instEv}(G_1, X, F) \rightarrow \mathrm{instEv}(G_2, X, F) \\
\mathrm{instEv}(?_\delta, X, F) = \mathrm{reflEv}(?_\delta) \text{ if } X \notin \delta & & \mathrm{instEv}(\forall Y.G, X, F) = \forall Y.\mathrm{instEv}(G, X, F)
\end{array}$$

$$\mathrm{instEv}(?_{\delta_1;X;\delta_2}, X, F) = \{(?_{\delta_1;\delta_2}, \mathtt{inj}_?, \mathtt{inj}_?), (F, \mathtt{inj}_X, \mathtt{inj}_F)\}$$

Fig. 9. Instantiation evidence function.

Definition 7.1 (Initial Evidence). If $G_1 \sim G_2$ then $G = G_1 \sqcap G_2$ and $\mathrm{initEv}(G_1, G_2) = \{(G, \mathrm{initPT}(G, G_1), \mathrm{initPT}(G, G_2))\}$.

This evidence consists of a single span, where the first component is the *meet* (greatest lower bound with respect to precision) $G_1 \sqcap G_2$ between G_1 and G_2, and the second and third components are the *initial proof terms* between the meet and G_1 and G_2, respectively. The meet $G_1 \sqcap G_2$ is a partial function and corresponds formally to $\alpha(\gamma(G_1) \cap \gamma(G_2))$. As for the definition of precision and congruence, we also define an inductive definition for the meet. Note that from AGT, $G \sim G'$ holds if $\gamma(G_1) \cap \gamma(G_2)$ is not empty, then if $G \sim G'$ then $G \sqcap G'$ will always be defined. The initial proof term between two types in precision is computed using the $\mathrm{initPT}(G, G')$ function such that $\mathrm{initPT}(G, G') : G \sqsubseteq G'$. It is important to note that the initial proof term between two types is unique since the type variables within the unknown type scope are not instantiated. Its definition is unsurprising and can be derived from the type precision judgment from Figure 2. For example, $\mathrm{initPT}(\mathtt{Int}, ?_X) = \mathtt{inj}_{\mathbf{Int}}$ and $\mathrm{initPT}(X, ?_X) = \mathtt{inj}_X$.

As $F_\varepsilon^?$ requires all values to be ascribed, the elaboration rules ascribe base values, functions and type abstractions to their own type using the *reflexive evidence* operator $\mathrm{reflEv}(G) \triangleq \mathrm{initEv}(G, G)$. For instance, term 42 is elaborated to $\mathrm{reflEv}(\mathtt{Int})\, 42 :: \mathbf{Int}$, where $\mathrm{reflEv}(\mathtt{Int}) = \{(\mathbf{Int}, \mathtt{refl}_{\mathbf{Int}}, \mathtt{refl}_{\mathbf{Int}})\}$. The elaboration process also inserts ascriptions to equate types in elimination forms. In particular, the translation of a term application (Eapp) ascribes the function term t_1 to a function type that matches [Cimini and Siek 2016] with its own type ($G_1 \rightarrowtriangle G_{11} \rightarrow G_{12}$): its own type if t_1 has a function type; otherwise $?_\delta \rightarrow ?_\delta$ (if the type is $?_\delta$). Also, the argument term t_2 is ascribed to the argument type of the ascribed function.

$$\text{Eapp}\ \frac{\begin{array}{c}\Delta;\Gamma \vdash t_1 : G_1 \rightsquigarrow t_1' \qquad \Delta;\Gamma \vdash t_2 : G_2 \rightsquigarrow t_2' \qquad G_1 \rightarrowtriangle G_{11} \rightarrow G_{12} \\ \epsilon_1 = \mathrm{initEv}(G_1, G_{11} \rightarrow G_{12}) \qquad\qquad \epsilon_2 = \mathrm{initEv}(G_2, G_{11})\end{array}}{\Delta;\Gamma \vdash t_1\ t_2 : G_{12} \rightsquigarrow (\epsilon_1 t_1' :: G_{11} \rightarrow G_{12})\ (\epsilon_2 t_2' :: G_{11})}$$

Rule (EappG) elaborates type applications, and is responsible of inserting “maybe-seal” evidence.

$$\text{EappG}\ \frac{\Delta;\Gamma \vdash t : G \rightsquigarrow t' \qquad \Delta \vdash F \qquad G \rightarrowtriangle \forall X.G' \qquad \epsilon_1 = \mathrm{initEv}(G, \forall X.G') \qquad \epsilon_2 = \mathrm{instEv}(G', X, F)}{\Delta;\Gamma \vdash t\ [F] : G'[F/X]\backslash X \rightsquigarrow \epsilon_2((\epsilon_1 t' :: \forall X.G')\ [F]) :: G'[F/X]\backslash X}$$

The elaborated type application is ascribed to the instantiated scheme type $G'[F/X]\backslash X$ (removing X from the environments of unknown types), using a special evidence that justifies that $G'[F/X]$ is consistent with $G'[F/X]\backslash X$. This evidence is computed using the *instantiation evidence* function instEv defined in Figure 9. Function $\mathrm{instEv}(G, X, F) : G[F/X] \sim G[F/X]\backslash X$ is defined (inductively) almost as the reflexive evidence operator, save for the case when type G is $?_\delta$ and X is in scope δ. Then, instEv generates an evidence which consist of two spans: one span that “seals to X” and another one that does not. More in detail, the first span $(F, \mathtt{inj}_X, \mathtt{inj}_F)$ represents that the unknown type should behave polymorphically in X. On the contrary, the second span $(?_{\delta_1;\delta_2}, \mathtt{inj}_?, \mathtt{inj}_?)$ does not acknowledge the existence of variable X. For example, we have that $(?_X \rightarrow X)[\mathbf{Int}/X] =$

$\boxed{\mathsf{G} \preccurlyeq \mathsf{G}}$ **Shape-restricted type precision**

$$(\preccurlyeq_B)\ \frac{}{\mathsf{B} \preccurlyeq \mathsf{B}} \qquad (\preccurlyeq_X)\ \frac{}{\mathsf{X} \preccurlyeq \mathsf{X}} \qquad (\preccurlyeq_\to)\ \frac{\mathsf{G}_1 \preccurlyeq \mathsf{G}_2 \qquad \mathsf{G}_1' \preccurlyeq \mathsf{G}_2'}{\mathsf{G}_1 \to \mathsf{G}_1' \preccurlyeq \mathsf{G}_2 \to \mathsf{G}_2'} \qquad (\preccurlyeq_\forall)\ \frac{\mathsf{G}_1 \preccurlyeq \mathsf{G}_2}{\forall \mathsf{X}.\mathsf{G}_1 \preccurlyeq \forall \mathsf{X}.\mathsf{G}_2}$$

$$(\preccurlyeq_{B?})\ \frac{}{\mathsf{B} \preccurlyeq ?_\delta} \qquad (\preccurlyeq_{X?})\ \frac{\mathsf{X} \in \delta}{\mathsf{X} \preccurlyeq ?_\delta} \qquad (\preccurlyeq_?)\ \frac{\delta \subseteq \delta'}{?_\delta \preccurlyeq ?_{\delta'}}$$

$\boxed{\Omega \vdash \mathsf{t} : \mathsf{G} \sqsubseteq \mathsf{t} : \mathsf{G}}$ **Term precision**

$$(\sqsubseteq\text{asc})\ \frac{\Omega \vdash \mathsf{t}_1 : \mathsf{G}_1' \sqsubseteq \mathsf{t}_2 : \mathsf{G}_2' \qquad \mathsf{G}_1 \sqsubseteq \mathsf{G}_2}{\Omega \vdash \mathsf{t}_1 :: \mathsf{G}_1 : \mathsf{G}_1 \sqsubseteq \mathsf{t}_2 :: \mathsf{G}_2 : \mathsf{G}_2} \qquad (\sqsubseteq\text{appG})\ \frac{\begin{array}{cc}\Omega \vdash \mathsf{t}_1 : \mathsf{G}_1 \sqsubseteq \mathsf{t}_2 : \mathsf{G}_2 & \mathsf{G}_1 \preccurlyeq \mathsf{G}_2 \\ \mathsf{G}_1 \rightarrowtriangle \forall \mathsf{X}.\mathsf{G}_1' & \mathsf{G}_2 \rightarrowtriangle \forall \mathsf{X}.\mathsf{G}_2'\end{array}}{\Omega \vdash \mathsf{t}_1\ [\mathsf{F}] : \mathsf{G}_1'[\mathsf{F}/\mathsf{X}]\backslash\mathsf{X} \sqsubseteq \mathsf{t}_2\ [\mathsf{F}] : \mathsf{G}_2'[\mathsf{F}/\mathsf{X}]\backslash\mathsf{X}}$$

Fig. 10. $\mathsf{F}^?$: Shape-restricted type precision and term precision (fragment).

$?_{X:\mathbf{Int}} \to \mathbf{Int}$ and $(?_X \to X)[\mathbf{Int}/X]\backslash X = ? \to \mathbf{Int}$. Therefore $\mathrm{instEv}(?_X \to X, X, \mathtt{Int}) : ?_{X:\mathbf{Int}} \to \mathbf{Int} \sim ? \to \mathbf{Int}$, where $\mathrm{instEv}(?_X \to X, X, \mathtt{Int}) = \{(\mathbf{Int} \to \mathbf{Int}, \mathtt{inj}_X \to \mathtt{refl}_{\mathbf{Int}}, \mathtt{inj}_{\mathbf{Int}} \to \mathtt{refl}_{\mathbf{Int}}), (? \to \mathbf{Int}, \mathtt{inj}_? \to \mathtt{refl}_{\mathbf{Int}}, \mathtt{inj}_? \to \mathtt{refl}_{\mathbf{Int}})\}$. Note that this evidence makes it impossible that a sealed value leaks out of a polymorphic function application. The scope of a type variable (label) is limited to the type application in which it appears. For example, consider the type application $\mathsf{t}\ [\mathtt{Int}]$ in $\mathrm{F}^?$, where $\vdash \mathsf{t} : \forall X.?_X \to ?_X$. This term is elaborated to $\mathrm{instEv}(?_X \to ?_X, X, \mathtt{Int})\ (\mathbf{t}\ [\mathtt{Int}]) :: ? \to ?$, where $\mathbf{t}\ [\mathtt{Int}]$ has type $?_{X:\mathbf{Int}} \to ?_{X:\mathbf{Int}}$; the generated instEv evidence is used to coerce this type to $? \to ?$. The label X may appear in unknown types that are used inside $\mathbf{t}\ [\mathbf{Int}]$, but the instEv evidence casts $\mathbf{t}\ [\mathtt{Int}]$ to a type not mentioning the label X, effectively restricting the scope of X to inside the term $\mathbf{t}\ [\mathbf{Int}]$. Even when the resulting term computes further and $\mathbf{t}\ [\mathbf{Int}]$ is reduced and combined with values from the context (for example, in a function application), the instEv evidence protects the scope of X, preventing it from interfering with possible other occurrences of the same name X introduced by other type applications. Because of this, there is no need for alpha-renaming.

It is important to clarify one important limitation: the instantiation evidence $\mathrm{instEv}(\mathsf{G}, \mathsf{X}, \mathsf{F})$ is not general enough when $\mathsf{G} = ?_{\delta_1, X, \delta_2}$. With the current definition, $?_{\delta_1, X, \delta_2}$ intuitively only represents something of type X or other well-formed static type with respect to δ_1, δ_2. This means that at runtime, if $?_{\delta_1, X, \delta_2}$ is used in a consistent judgment with a function such as $\mathsf{X} \to \mathsf{X}$, the program could fail. For instance, the program $(\Lambda X.\lambda x : X.x) :: \forall X.?_X\ [\mathtt{Int}]\ 1$ generates the instantiation evidence $\mathrm{instEv}(?_X, X, \mathtt{Int}) = \{(?, \mathtt{inj}_?, \mathtt{inj}_?), (\mathbf{Int}, \mathtt{inj}_X, \mathtt{inj}_{\mathbf{Int}})\}$, which will not seal argument 1, making this program fail at runtime. To fix the program, and generate appropriate sealing, the type of the ascription had to be changed as follows: $(\Lambda X.\lambda x : X.x) :: \forall X.?_X \to ?_X\ [\mathtt{Int}]\ 1$. The spans generated now include $(\mathbf{Int} \to \mathbf{Int}, \mathtt{inj}_X \to \mathtt{inj}_X, \mathtt{inj}_{\mathbf{Int}} \to \mathtt{inj}_{\mathbf{Int}})$ which makes the program run without errors. However, statically there is no way to know a priori the exact shape of the evidence needed when imprecise information is involved. Consequently, a more general mechanism for the generation of the instantiation evidence is needed (see Section 8).

Finally, we prove that the elaboration preserves typing:

Theorem 7.2 (Elaboration Preserves Typing). *If $\Delta; \Gamma \vdash \mathsf{t} : \mathsf{G}$, then $\Delta; \Gamma \vdash \mathsf{t} : \mathsf{G} \rightsquigarrow \mathbf{t}'$ and $\Delta; \Gamma \vdash \mathbf{t}' : \mathbf{G}$.*

Source-level graduality. Under the natural notion of type precision (Figure 8), some $\mathrm{F}^?$ terms related by precision elaborate to $\mathrm{F}_\varepsilon^?$ terms that are *not* related by precision. Consider program $(\Lambda X.\lambda x : X.x) :: \forall X.?_X \to ?_X\ \ [\mathtt{Int}]\ 1$ more precise than $(\Lambda X.\lambda x : X.x) :: \forall X.?_X\ \ [\mathtt{Int}]\ 1$ (note that

$\forall X.?_X \to ?_X \sqsubseteq \forall X.?_X$). The first program elaborates to a program that reduces correctly, but the second to a program that fails. As explained in the previous section, this is because $\mathrm{instEv}(?_X, X, \mathtt{Int})$ does not generate evidence that contains function spans that seal the argument. This does not mean that there is no source-level graduality in $F^?$ at all; as first explored by Igarashi et al. [2017], the fact that the gradual guarantees are stated relative to a notion of precision means that we may be able to characterize source-level graduality via a restricted notion of precision.

To characterize the $F^?$ programs for which we can guarantee graduality, it is enough to restrict term precision *only* for type applications, enforcing that for such expressions, type precision be restricted to types of the same shape. Notice how rule ($\sqsubseteq$appG) in Figure 10 uses *shape-restricted* type precision $\preccurlyeq$ in its premise, while other rules, such as rule ($\sqsubseteq$asc), use the natural type precision $\sqsubseteq$. Shape-restricted type precision $\preccurlyeq$ is defined similarly to $\sqsubseteq$, but in the case of polymorphic and function types, the type constructors have to match. For example, $\forall X.X \to X \preccurlyeq \forall X.?_X \to ?_X$ but $\forall X.(X \to X) \to X \not\preccurlyeq \forall X.?_X \to ?_X$ and $\forall X.X \to X \not\preccurlyeq \forall X.?_X$. This means that if $G_1 \preccurlyeq G_2$, then all sealing spans included in applying instEv to G_1 will be included in the application of instEv to G_2. The other cases of term precision are derived just as the natural lifting of type precision to terms.

With this notion of term precision, two source terms related by precision elaborate to $F^?_\varepsilon$ terms that are also related by precision:

Lemma 7.3. *If* $\vdash t_1 : G_1 \sqsubseteq t_2 : G_2$, $\vdash t_1 : G_1 \rightsquigarrow t_1$ *and* $\vdash t_2 : G_2 \rightsquigarrow t_2$, *then* $\vdash \mathrm{initPT}(G_1, G_2) : t_1 \sqsubseteq t_2$.

Note that precision in $F^?_\varepsilon$ is proof-relevant, therefore we have to provide a proof term that justifies "how" two terms are related. We do that by using the initial proof term function between the types of the related terms.

The dynamic semantics of a $F^?$ term are given by first elaborating the term to $F^?_\varepsilon$ and then reducing the $F^?_\varepsilon$ term. Hence, for establishing the gradual guarantees in $F^?$, we write $t \Downarrow v$ if $\vdash t : G \rightsquigarrow t$ and $t \overset{*}{\longmapsto} v$. Similarly, we write $t \Uparrow$ if the elaboration of t diverges. Then, using Lemmas 7.2, 7.3 and 6.4 we can prove the gradual guarantees for $F^?$:

Theorem 7.4 (Gradual guarantees). *Suppose* $\vdash t_1 : G_1 \sqsubseteq t_2 : G_2$ *and* $\vdash t_1 : G_1$.

(1) $\vdash t_2 : G_2$ *and* $G_1 \sqsubseteq G_2$.
(2) If $t_1 \Downarrow v_1$, *then* $t_2 \Downarrow v_2$ *and* $\vdash \mathrm{initPT}(G_1, G_2) : v_1 \sqsubseteq v_2$.
If $t_1 \Uparrow$ *then* $t_2 \Uparrow$.

Source-level parametric reasoning. As a first form of parametric reasoning for $F^?$, the elaborations of well-typed $F^?$ terms produces $F^?_\varepsilon$ terms that are also well typed (by Theorem 7.2), and hence related to themselves (by Theorem 5.1):

Corollary 7.5. *If* $\Delta; \Gamma \vdash t : G \rightsquigarrow t$ *then* $\Delta; \Gamma \vdash t \preceq t : G$.

This lemma is powerful, but it is not immediately clear what it means for concrete example terms in $F^?$. We make this clearer as follows: a type abstraction f of type $\forall X.G$ applied to related types, produces related terms whenever X does not occur in the scopes of unknown types in G (a condition written $G \backslash X = G$ below):

Lemma 7.6. $\forall n, \rho$

$$\frac{\begin{array}{ccc} \Delta; \Gamma \vdash f : \forall X.G & \forall B_1, B_2, R \in \mathit{Rel}[B_1, B_2] & ((n, \rho) \in \mathcal{D}[\![\Delta]\!] \wedge (n, \gamma) \in \mathcal{G}_\rho[\![\Gamma]\!]) \\ G \backslash X = G & & \Delta; \Gamma \vdash f\ [B_i] : G[B_i/X] \backslash X \rightsquigarrow t_i \end{array}}{(n, \rho_1(\gamma_1(t_1)), \rho_2(\gamma_2(t_2))) \in \mathcal{T}_{\rho, X \mapsto (B_1, B_2, R)}[\![G]\!]}$$

As a direct consequence of Lemma 7.6, every $F^?$ program ascribed to a static type behaves parametrically, even if it internally uses the unknown type. Since the logical relation for $F^?_\varepsilon$ coincides almost exactly with traditional formulations for System F, this means we get the same reasoning about such applications than in System F itself, save for the possibility that two related terms both raise a runtime error. From Lemma 7.6, we can also derive free theorems involving imprecise types. For instance, given a function of type $\forall X.?_{\emptyset}{\rightarrow}X$, then the application of the function either fails or diverges.

Lemma 7.7. *If* $\vdash f : \forall X.?_{\emptyset}{\rightarrow}X$, $\vdash v : B$ *and* $t = f\ [B]\ v$, *then* $t \Downarrow \mathbf{error}$, *or* $t \Uparrow$.

This behavior is expected because $?_{\emptyset}$ can only stand for a type that does *not* involve X. Intuitively, this means that this gradual polymorphic function type denotes types such as $\forall X.\mathtt{Int}{\rightarrow}X$ and $\forall X.\mathtt{Bool}{\rightarrow}X$, but not $\forall X.X{\rightarrow}X$. Therefore, the function cannot create a value of type X out of thin air, and the argument v cannot possibly be sealed as a value of type X, so the function necessarily fails if it tries to return any value.

Lemma 7.6 does not apply to polymorphic functions whose type mentions unknown types with the quantified variable in scope. To understand why such types require more nuance, remember the example function $f = \Lambda X.\,\lambda x : ?.x :: X$ discussed in the introduction. When applied to type $\mathtt{Int}$ and value 42, this function produces the value 42, but it throws a runtime type error when applied to $\mathtt{Bool}$ and value 42. In such examples, $F^{?}$ inserts plausible sealing evidence, in an effort to guess whether the programmer intended 42 to be treated as a value of type X or not, in a maximally permissive way. However, this does not mean we do not get any form of parametricity for such examples, but rather, we need to keep in mind the intuitive effect of plausible sealing. In other words, $F^{?}$ supports more parametric reasoning than just what Lemma 7.6 expresses. Particularly, when a type variable is in the scope of an unknown type within a function type, we can also derive some free theorems using Corollary 7.5. For instance, if a function f has type $\forall X.?_{X}{\rightarrow}X$, then by parametricity we can deduce that f behaves either as the identity function or fails or diverges:

Lemma 7.8. *If* $\vdash f : \forall X.?_{X}{\rightarrow}X$, $\vdash v : B$ *and* $t = f\ [B]\ v$, *then* $t \Downarrow v$ *with* $\vdash v : B \rightsquigarrow v$, *or* $t \Downarrow \mathbf{error}$, *or* $t \Uparrow$.

Contrary to GSF [Toro et al. 2019], in $F^{?}$ this result holds just by looking at the type of f, without the need to unfold its definition. Intuitively, this lemma takes into account that $F^{?}$ applies plausible sealing to the argument v, so f might return it as the result of type X. The function f can also diverge or fail, but parametricity for $F^{?}$ still implies that f cannot return any value other than v.

8 LIMITATIONS AND PERSPECTIVES

The technical development of plausible sealing in this article suffers from two technical limitations. The first is that we only formalize a simplified form of polymorphism with instantiation types. The second is that graduality for $F^{?}$ is restricted for type applications, since two type applications are only related when the polymorphic types have the same shape (Section 7).

Both limitations manifest in the definition of instantiation evidence for the unknown type (Figure 9): $\mathrm{instEv}(?_{\delta_1;X;\delta_2}, X, F) = \{(?_{\delta_1;\delta_2}, \mathtt{inj}_{?}, \mathtt{inj}_{?}), (F, \mathtt{inj}_{X}, \mathtt{inj}_{F})\}$. Recall that the role of this instantiation evidence is to cast, for example, a function application of type $?_{X:\mathrm{Int}} \rightarrow \mathtt{Int}$ to type $?_{\emptyset} \rightarrow \mathtt{Int}$. The restricted form of polymorphism is apparent because instEv's third argument F is restricted to an instantiation type (a base type or a type variable), and we simply use $\mathtt{inj}_{F} : F \sqsubseteq ?_{\delta_1;\delta_2}$ to inject F into type $?_{\delta_1;\delta_2}$. Generalizing to full polymorphism would require replacing F and $\mathtt{inj}_{F}$ in the above definition with an arbitrary type G and a proof term $\mathbf{c} : G \sqsubseteq ?_{\delta_1;\delta_2}$. This requires to extend the syntax of δ to allow for any type. An initial exploration suggests this is largely unproblematic and in fact, our Agda proofs of consistent transitivity associativity and monotonicity already support such a richer syntax of δ. A problem turns up when one of the other types in δ_1 or δ_2 mentions $?_{\delta'}$ with $X : X \in \delta'$. In that case, it appears we additionally need a proof term that expresses precision between $?_{\delta}$ and $?_{\delta'}$ when δ is not just a subset of δ' but some of the types in δ are themselves strictly more precise than corresponding types in δ'.

Allowing a type abstraction to be instantiated at any type G would require saving the information of the instantiated type in the proof term $\mathtt{inj}_{X}$. This information could then be refined through composition. Therefore, we should transform $\mathtt{inj}_{X}$ to the proof term sequence $\mathtt{inj}_{X}(c)$, where $\mathtt{inj}_{X}(c) : G' \sqsubseteq ?_{\delta}$, $X : G \in \delta$, $c : G' \sqsubseteq G$ and $\delta \vdash G'$. In addition, the composition of proof terms for this case would slightly change: $c_1;\mathtt{inj}_{X}(c_2) = \mathtt{inj}_{X}(c_1;c_2)$. Note that applying the type abstraction to any type G' with rule RappG would not change; the instantiated type G' would continue to be substituted in the body of the type abstraction and the evidence.

The other main limitation of $F^{?}$ is caused by the right-hand-side of the above definition. Problematically, it only mentions two cases: a value of type $?_{\delta_1;\delta_2}$ will be converted into type $?_{\delta_1;X;\delta_2}$ either (1) by not sealing at all and simply extending the scope of the unknown type, or (2) by sealing, converting a value of type F into a value sealed at type X in $?_{\delta_1;X;\delta_2}$. What is missing is a recursive case that would treat, for example, a value of type $F \to F$ as a value of type $X \to X$ and recursively seal it accordingly. In fact, this could be accommodated easily by extending the right-hand-side with an additional case: $(F \to F, \mathtt{inj}_{\to}(\mathtt{inj}_{X} \to \mathtt{inj}_{X}), \mathtt{inj}_{\to}(\mathtt{inj}_{F} \to \mathtt{inj}_{F}))$. we conjecture that a solution is to introduce a syntax of recursive evidence that would allow us to define instEv$(?_{\delta_1;X;\delta_2}, X, F)$ as:
$\mu\epsilon.\, \mathtt{inj}_{\to}(\epsilon \to \epsilon) \uplus \mathtt{inj}_{\times}(\epsilon \times \epsilon) \uplus \mathtt{inj}_{\forall}(\forall Y.\, \epsilon) \uplus \{(B, \mathtt{inj}_{B}, \mathtt{inj}_{B})\} \uplus \{(F, \mathtt{inj}_{X}, \mathtt{inj}_{F})\}$. In this notation, we construct an evidence $\mathtt{inj}_{\to}(\epsilon \to \epsilon) : ?_{\delta} \sim ?_{\delta'}$ from $\epsilon : ?_{\delta} \sim ?_{\delta'}$ by combining two spans (G_1, c_1, c_2) and (G_2, c_3, c_4) from ϵ into the span $(G_1 \to G_2, \mathtt{inj}_{\to}(c_1 \to c_3), \mathtt{inj}_{\to}(c_2 \to c_4))$ and similarly for $\mathtt{inj}_{\times}$ and $\mathtt{inj}_{\forall}$. We leave the definition of the operational behavior of such recursive evidence and the proofs of its properties to future work.

9 DISCUSSION AND RELATED WORK

Gradual parametricity has been intensively studied [Ahmed et al. 2009, 2017; Igarashi et al. 2017; Matthews and Ahmed 2008; New et al. 2020; Toro et al. 2019; Xie et al. 2018]. We have already discussed in detail related work, emphasizing the most recent proposals. Recall that Section 2.3 compares existing languages via examples (additional examples are included in the technical report).

Sealing for Parametricity. Dynamic sealing, originally proposed by Morris [1973] to dynamically enforce type abstraction, has been widely used to guarantee parametricity in gradual languages. The notion of dynamic sealing combined with global runtime type name generation has driven the dynamic semantics of polymorphic gradual languages such as λB [Ahmed et al. 2017], CSA [Xie et al. 2018], GSF [Toro et al. 2019] and PolyG$^{\nu}$ [New et al. 2020]. Type names are dynamically generated in each type application and are kept in a global store, making the dynamic semantics and the definitions and proof of parametricity less standard and more complex. $F^{?}_{\varepsilon}$ avoids using type names generation and, therefore, a global store thanks to the fact that the unknown type is decorated by an environment and the sealing/unsealing mechanism is generated statically. It is worth noting that, unlike other developed gradual polymorphic languages, PolyG$^{\nu}$ also includes explicit seal and unseal terms in its syntax. In this sense, we can say that $F^{?}_{\varepsilon}$ also includes in its syntax explicit forms of sealing and unsealing, since for a program with an imprecise type to behave in a parametric way, it is necessary to introduce the evidence of sealing and unsealing statically. The key novelty of $F^{?}_{\varepsilon}$ is to support evidence with *multiple* sealing justifications, which makes it possible to avoid to eagerly choose a sealing strategy when interacting with the unknown type.

Strict Precision. Igarashi et al. [2017] first proposed using a non-standard notion of precision in System F_G to address some problems with the dynamic gradual guarantee when the unknown type is allowed to stand for a type variable. Consequently, in System F_G the unknown type is not consistent with any type variable. Here, the source language $F^?$ also restricts precision, but in a much less drastic manner: type precision is the standard precision, and only term precision is restricted, in order to only relate type applications to types of the same shape. We explain in the previous section how this restriction could be lifted. Finally, $F^?_\varepsilon$ has no such restriction, and satisfies the gradual guarantees with respect to the standard notion of precision, for both types and terms.

Proof Terms for Type Precision. PolyG$^\nu$, inspired by previous work [New and Ahmed 2018], adds a proof term to the type precision relation as a technical intermediate representation for the translation from PolyG$^\nu$ to PolyC$^\nu$, a cast calculus that gives meaning to PolyG$^\nu$ programs. It is important to note that proof terms in the type precision relation are canonical, *i.e.* there is at most one proof term that proves any given type precision judgment. Likewise, $F^?_\varepsilon$ language indexes the type precision relation with proof terms, but contrary to PolyG$^\nu$, proof terms are relevant, *i.e.* there can be multiple proof terms for the same precision judgment. Also, as a difference with PolyG$^\nu$, term precision in $F^?_\varepsilon$ is indexed by a relevant proof term.

Family of Unknown Types. Devriese et al. [2018] proposed to decorate the unknown type with the set of type variables in scope, as we do here, thus limiting the expressiveness of the unknown type by forming different families. This proposal aims to potentially reestablish the fully abstract embedding property of System F into λB. Devriese et al. [2018] and more recently Jacobs et al. [2021] proposed a new criterion for gradual typing named the fully abstract embedding (FAE) property: the embedding from the static to the gradual language should be fully abstract in order to preserve the semantics properties of the static languages. We conjecture that this criterion holds for the language $F^?$, being a gradual version of System F that preserves its main semantic property (*i.e.* parametricity), although we leave a proof of FAE as future work.

Implicit Polymorphism. Several polymorphic gradual languages have explored implicit polymorphism present in languages such as Haskell. Xie et al. [2018] developed a gradual source language with implicit polymorphism, where the runtime semantics are given by compilation to λB. λB and System F_G, in turn, are languages with explicit polymorphism that accommodate some form of implicit polymorphism. $F^?$ and $F^?_\varepsilon$, as well as PolyG$^\nu$ and GSF, only support explicit polymorphism; exploring implicit polymorphism for $F^?$ is an interesting venue for future work, in order to enhance interoperability between typed and untyped code.

Evidence Representation. Evidence has been used in different scenarios, varying its representation according to the semantic properties to be preserved in the gradual language. For instance, Lehmann and Tanter [2017] develop a gradual language with refinement types, allowing smooth evolution and interoperability between simple types and logically refined types. In this case, the evidence for consistent subtyping is represented by a triple, where the first component accounts for the logical environment, and the second and third are types. Toro et al. [2018] develop a gradual language with security types and references, indexing types with gradual security labels. Driven by noninterference, types in evidence are indexed with *intervals* of security labels, representing (bounded) ranges of possible static types. Likewise, $F^?_\varepsilon$ represents evidence in a novel way. First, it enriches evidence with proof terms relevant that we call span and then generalizes the evidence to a set of spans, building evidence with the expressiveness to ensure both graduality and parametricity. This theory may be applicable in other complex settings as well. We conjecture, and leave as future work, that by representing the evidence of instantiation recursively when imprecise types occur, we can lift the restriction on the term precision relation, thus correctly running all programs that by graduality must end in a value. In addition, it would be interesting to explore if there is a way to systematically derive proof-relevant consistency with AGT.

Performance. Gradual parametricity is a very challenging topic at the theoretical level, with all current efforts trying to figure out how to achieve a good design backed by a strong metatheory. This work likewise focuses on the theory of gradual parametricity, contributing a novel approach and technique. We leave the study of the performance and efficiency of a practical implementation as an open question to be addressed. Nevertheless, it is worth mentioning that the proposed language design satisfies a relevant criterion for space efficiency [Herman et al. 2010], namely associativity of evidence composition, which is known to allow for space efficiency in evidence-based semantics [Bañados Schwerter et al. 2021; Toro and Tanter 2020]. Whether the algorithmic definition of consistent transitivity can be efficiently implemented depends on whether evidence can be represented in memory in a form that uses space efficiently and allows an efficient implementation of evidence composition. All these are open research questions.

10 CONCLUSION

Previous work on gradual parametricity has had to compromise on important design goals like graduality [Toro et al. 2019] or type-driven sealing [New et al. 2020]. Rather than accepting these compromises, this paper attempts to revisit accepted wisdom like the use of globally scoped sealing and contribute new ideas like plausible sealing and the set-of-spans representation of evidence for proof-relevant precision. Although the results presented here still have some restrictions, they open a new path towards the goal of reconciling parametricity, graduality and type-driven sealing. Additionally, some of our novel techniques are potentially reusable in other settings. Finally, our use of lexically scoped sealing invalidates the counterexample of fully-abstract embedding put forth by Devriese et al. [2018], and thereby offers new hope of constructing a gradual language that satisfies the ambitious goal of embedding System F fully abstractly [Jacobs et al. 2021].

ACKNOWLEDGMENTS

This work was partially supported by the United States Air Force Office of Scientific Research under award number FA9550-21-1-0054, by the Flemish Research Programme Cybersecurity, and by ANID FONDECYT projects 1190058 and 3200583, Chile.

REFERENCES

Amal Ahmed. 2004. *Semantics of Types for Mutable State.* Ph.D. Dissertation. Princeton University.

Amal Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *Proceedings of the 15th European Symposium on Programming Languages and Systems (ESOP 2006) (Lecture Notes in Computer Science, Vol. 3924)*, Peter Sestoft (Ed.). Springer-Verlag, Vienna, Austria, 69–83.

Amal Ahmed, Robert Bruce Findler, Jacob Matthews, and Philip Wadler. 2009. Blame for All. In *Workshop on Script to Program Evolution (STOP)*. Genova, Italy.

Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for Free for Free: Parametricity, with and Without Types. See[ICFP 2017 2017], 39:1–39:28.

Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-carrying Code. *ACM Transactions on Programming Languages and Systems* 23, 5 (Sept. 2001), 657–683.

Felipe Bañados Schwerter, Alison M. Clark, and Jafery. 2021. Abstracting Gradual Typing Moving Forward: Precise and Space-Efficient. See[POPL 2021 2021], 61:1–61:28.

Rastislav Bodík and Rupak Majumdar (Eds.). 2016. *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. ACM Press, St Petersburg, FL, USA.

Matteo Cimini and Jeremy Siek. 2016. The gradualizer: a methodology and algorithm for generating gradual type systems, See [Bodík and Majumdar 2016], 443–455.

Dominique Devriese, Marco Patrignani, and Frank Piessens. 2018. Parametricity versus the universal type. *Proceedings of the ACM on Programming Languages* 2, POPL (Jan. 2018), 38:1–38:23.

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing, See [Bodík and Majumdar 2016], 429–442. See erratum: https://www.cs.ubc.ca/ rxg/agt-erratum.pdf.

Jean-Yves Girard. 1972. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur.* Ph.D. Dissertation. Université de Paris VII, Paris, France.

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Sympolic Computation* 23, 2 (June 2010), 167–189.

ICFP 2017 2017.

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On Polymorphic Gradual Typing. See[ICFP 2017 2017], 40:1–40:29.

Koen Jacobs, Amin Timany, and Dominique Devriese. 2021. Fully abstract from static to gradual. See[POPL 2021 2021], 7:1–7:30.

Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*. ACM Press, Paris, France, 775–788.

Jacob Matthews and Amal Ahmed. 2008. Parametric Polymorphism Through Run-Time Sealing, or, Theorems for Low, Low Prices!. In *Proceedings of the 17th European Symposium on Programming Languages and Systems (ESOP 2008) (Lecture Notes in Computer Science, Vol. 4960)*, Sophia Drossopoulou (Ed.). Springer-Verlag, Budapest, Hungary, 16–31.

James H. Morris. 1973. Protection in Programming Languages. *Commun. ACM* 16, 1 (Jan. 1973), 15–21.

Max S. New and Amal Ahmed. 2018. Graduality from Embedding-Projection Pairs. *Proceedings of the ACM on Programming Languages* 2, ICFP (Sept. 2018), 73:1–73:30.

Max S. New, Dustin Jamner, and Amal Ahmed. 2020. Graduality and Parametricity: Together Again for the First Time. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 46:1–46:32.

nLab contributors. 2021a. pullback. https://ncatlab.org/nlab/show/pullback

nLab contributors. 2021b. span. https://ncatlab.org/nlab/show/span

Benjamin Pierce and Eijiro Sumii. 2000. Relating Cryptography and Polymorphism. Manuscript.

POPL 2021 2021.

John C. Reynolds. 1974. Towards a Theory of Type Structure. In *Porceedings of the Programming Symposium (Lecture Notes in Computer Science, Vol. 19)*. Springer-Verlag, 408–423.

John C. Reynolds. 1983. Types, abstraction, and parametric polymorphism. In *Information Processing 83*, R. E. A. Mason (Ed.). Elsevier, 513–523.

Jeremy Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Proceedings of the Scheme and Functional Programming Workshop*. 81–92.

Jeremy Siek and Philip Wadler. 2010. Threesomes, with and without blame. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*. ACM Press, Madrid, Spain, 365–376.

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)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Asilomar, California, USA, 274–293.

Eijiro Sumii and Benjamin C. Pierce. 2004. A Bisimulation for Dynamic Sealing. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*. ACM Press, Venice, Italy, 161–172.

Matías Toro, Ronald Garcia, and Éric Tanter. 2018. Type-Driven Gradual Security with References. *ACM Transactions on Programming Languages and Systems* 40, 4 (Nov. 2018), 16:1–16:55.

Matías Toro, Elizabeth Labrada, and Éric Tanter. 2019. Gradual Parametricity, Revisited. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 17:1–17:30.

Matías Toro and Éric Tanter. 2020. Abstracting Gradual References. *Science of Computer Programming* 197 (Oct. 2020), 1–65.

Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira. 2018. Consistent Subtyping for All. In *Proceedings of the 27th European Symposium on Programming Languages and Systems (ESOP 2018) (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer-Verlag, Thessaloniki, Greece, 3–30.