Gradual System F

ELIZABETH LABRADA, MATÍAS TORO, and ÉRIC TANTER, PLEIAD Lab, Computer Science Department, University of Chile, Chile

Bringing the benefits of gradual typing to a language with parametric polymorphism like System F, while preserving relational parametricity, has proven extremely challenging: first attempts were formulated a decade ago, and several designs have been recently proposed, with varying syntax, behavior, and properties. Starting from a detailed review of the challenges and tensions that affect the design of gradual parametric languages, this work presents an extensive account of the semantics and metatheory of GSF, a gradual counterpart of System F. In doing so, we also report on the extent to which the Abstracting Gradual Typing methodology can help us derive such a language. Among gradual parametric languages that follow the syntax of System F, GSF achieves a unique combination of properties. We clearly establish the benefits and limitations of the language, and discuss several extensions of GSF towards a practical programming language.

CCS Concepts: • **Theory of computation → Type structures**; **Program semantics;**

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

ACM Reference format:
Elizabeth Labrada, Matías Toro, and Éric Tanter. 2022. Gradual System F. *J. ACM* 69, 5, Article 38 (October 2022), 78 pages.
https://doi.org/10.1145/3555986

1 INTRODUCTION

There are many approaches to integrate static and dynamic type checking [Abadi et al. 1991; Bierman et al. 2010; Cartwright and Fagan 1991; Matthews and Findler 2007; Tobin-Hochstadt and Felleisen 2006]. In particular, gradual typing supports the smooth integration of static and dynamic type checking by introducing the notion of *imprecision* at the level of types, which induces a notion of *consistency* between plausibly equal types [Siek and Taha 2006]. A gradual type checker treats imprecision optimistically, and the runtime of the gradual language detects when optimistic static assumptions are invalid. Such detection is usually achieved by compilation to an internal language with explicit casts, called a cast calculus. In addition to being type safe, a gradually-typed language is expected to satisfy a number of properties that characterize the static-to-dynamic checking spectrum supported by the language [New and Ahmed 2018; Siek et al. 2015a].

Since its early formulation in a simple functional language [Siek and Taha 2006], gradual typing has been explored in a number of increasingly challenging settings such as subtyping [Garcia

Partially funded by ANID/FONDECYT Projects 1190058 & 3200583, ANID/Doctorado Nacional/2015-21150510 & 21151566.
Authors' address: E. Labrada, M. Toro, and É Tanter, PLEIAD Lab, Computer Science Department, University of Chile, Beauchef 851, Santiago, Chile; emails: elilabdeniz@gmail.com, {mtoro, etanter}@dcc.uchile.cl.

0004-5411/2022/10-ART38 $15.00
https://doi.org/10.1145/3555986

et al. 2016; Siek and Taha 2007], references [Herman et al. 2010; Siek et al. 2015b], effects [Bañados Schwerter et al. 2014, 2016], ownership [Sergey and Clarke 2012], typestates [Garcia et al. 2014; Wolff et al. 2011], information-flow typing [Disney and Flanagan 2011; Fennell and Thiemann 2013; Toro et al. 2018], session types [Igarashi et al. 2017b], refinements [Lehmann and Tanter 2017], dependent types [Eremondi et al. 2019], set-theoretic types [Castagna and Lanvin 2017], Hoare logic [Bader et al. 2018], separation logic [Wise et al. 2020], and, most relevant to this article, parametric polymorphism [Ahmed et al. 2011, 2017; Igarashi et al. 2017a; Ina and Igarashi 2011; New et al. 2020; Xie et al. 2018].

In the case of parametric polymorphism, a long-standing challenge has been to prove that the gradual language preserves a rich semantic property known as *relational parametricity* [Reynolds 1983], which dictates that a polymorphic value must behave uniformly for all possible instantiations. Building upon work on dynamic enforcement of parametricity [Guha et al. 2007; Matthews and Ahmed 2008; Pierce and Sumii 2000; Sumii and Pierce 2004], a first version of a parametric cast calculus[1] was proposed, albeit without any proof of parametricity [Ahmed et al. 2011]. Years later, a variant of this cast calculus, λB [Ahmed et al. 2017], did come with a proof of parametricity. λB extends the archetypal polymorphic lambda calculus also known as System F with casts and a dynamic (or unknown) type. λB is also used as a target language by Xie et al. [2018], who explore the treatment of *implicit* polymorphism. Another recent effort is System F_G, a gradual source language that is compiled to a cast calculus akin to λB, called System F_C [Igarashi et al. 2017a]. These efforts highlight several key challenges and tensions in the design of a gradually parametric language, and leave some questions unanswered, most notably the possibility to satisfy both parametricity and the dynamic gradual guarantee [Siek et al. 2015a] (or graduality [New and Ahmed 2018]). Inspired by these challenges, New et al. [2020] propose a different design that forgoes the familiar syntax of System F and instead relies on explicit terms for sealing and unsealing values. This change of perspective allows the language PolyG$^{\nu}$ to satisfy both parametricity and graduality, but at the cost of a different programming model with its own limitations (Section 3).

Contributions. This work starts from the identification of several design issues in existing gradual languages, and studies a gradual parametric language in the style of System F by applying a general methodology to gradualize programming languages. The resulting language, called **GSF** (for **Gradual System F**), embodies a number of important design choices. The first is in its name: it is an extension of System F, and therefore sticks to the traditional syntax of the polymorphic lambda calculus, where terms need not bother with sealing explicitly as in PolyG$^{\nu}$. Two major characteristics differentiate GSF from other gradual parametric languages based on System F. First, GSF ensures that type instantiations on imprecise types are faithfully supported, thereby soundly supporting higher-order polymorphic programming patterns. Second, System F polymorphic values in GSF can flow into imprecise code while preserving their original behavior, because imprecise ascriptions are harmless.

We introduce gradual parametricity informally (Section 2) and discuss its challenges by reviewing closely related work (Section 3), and present a quick tour of GSF, including its design principles and main properties (Section 4). We then explain how we derive GSF from a variant of System F called SF (Section 5), by following the **Abstracting Gradual Typing** methodology **(AGT)** [Garcia

[1]The difference between a gradual (source) language and a cast calculus is that a cast calculus demands explicit use of casts in order to exploit the flexibility of runtime checking—whether a cast calculus is meant to be used directly by programmers is in the eyes of the beholder. A gradual source language is usually defined by a type-directed translation to a cast calculus, with its own typing and reduction rules. In the AGT methodology reduction is instead defined directly for the gradual source language, in terms of the reduction of enriched typing derivations [Garcia et al. 2016]. Toro and Tanter [2017, 2020] prove that both approaches are equivalent in a standard, simply-typed setting.

et al. 2016]. While mostly standard, SF is peculiar in that its dynamic semantics rely on *runtime type generation*. This choice comes in anticipation of the gradualization of SF, informed by prior work that has used runtime type generation to enforce parametricity dynamically [Ahmed et al. 2011, 2017; Matthews and Ahmed 2008]. The statics of GSF then follow naturally from those of SF using the AGT methodology (Section 6), but the dynamics are more challenging (Section 7/Section 8). In particular, satisfying parametricity forces us to sacrifice the dynamic gradual guarantee in certain scenarios. We study this tension in detail and expose a weaker form of the dynamic gradual guarantee that GSF does satisfy (Section 9). While weaker than the dynamic gradual guarantee as originally intended, this guarantee is stronger than what other gradual parametric languages based on System F achieve; in particular, it implies that imprecise ascriptions are harmless. We then review the notions of gradual parametricity from the literature and present the gradual parametricity that GSF satisfies, along with gradual free theorems [Wadler 1989] (Section 10). We show that although the notion of parametricity enjoyed by GSF is based on and similar to that of λB, some polymorphic programs behave differently, due to differences in the dynamic semantics of each language. We then study the dynamic end of the gradual typing spectrum supported by GSF, and show that beyond a standard dynamically-typed language, GSF can faithfully embed a language with dynamic sealing primitives [Sumii and Pierce 2004] (Section 11). The embedding is built using a general seal/unseal generator, which is expressed as a GSF term. The proposed term is a polymorphic pair of functions of type $\forall X.(X \to ?) \times (? \to X)$. The first component of the pair serves a sealing function, producing an opaque value as result, and the second component serves as the corresponding unsealing function, returning the original unsealed value. Finally, we study an extension of GSF with existential types, which are the core of data abstraction mechanisms [Mitchell and Plotkin 1988] (Section 12).

Prior publication. This article substantially revises and extends a prior publication [Toro et al. 2019]. First of all, the presentation of related approaches (Section 2) has been largely revised, with novel illustrations and analysis, and also includes the recent work on PolyG$^{\nu}$ [New et al. 2020]. This revised article presents several novel technical contributions compared to the prior article: the detailed analysis of the dynamic gradual guarantee violation is new, and so is the development of the weaker guarantee that GSF satisfies (Section 9). The presentation of gradual parametricity (Section 10) includes a new comparison between the approaches of Ahmed et al. [2017] and New et al. [2020], shedding light on the current design space. The results that follow from both parametricity and the weaker guarantee subsume those presented in the prior publication. Additionally, the last two sections on embedding dynamic sealing and gradual existential types in GSF are completely novel developments.

Supplementary material. Auxiliary definitions and proofs of the main results can be found in the companion technical report. Additionally, an interactive prototype of GSF is available, which exhibits both typing derivations and reduction traces, and comes with all the examples mentioned in this paper, among others. The supplementary material can be found at https://pleiad.cl/gsf.

2 GRADUAL PARAMETRICITY: BACKGROUND AND BASICS

We start with a quick introduction to parametric polymorphism and parametricity, as well as gradual typing. We then briefly motivate gradual parametricity through a basic example.

2.1 Parametric Polymorphism

Parametric polymorphism enables the definition of terms that can operate over any type, with the introduction of type variables and universally-quantified types. For instance, a function of type $\forall X.X \to X$ can be used at any type, and returns a value of the same type as its actual argument.

For the sake of this work, it is important to recall two crucial distinctions that apply to languages with parametric polymorphism, one syntactic—whether polymorphism is explicit or implicit—and one semantic—whether polymorphic types impose strong behavioral guarantees or not.

Explicit vs Implicit. In a language with *explicit* polymorphism, such as the Girard-Reynolds polymorphic lambda calculus (*a.k.a.* System F) [Girard 1972; Reynolds 1974], the term language includes explicit type abstraction $\Lambda X.e$ and explicit type application $e\ [T]$, as illustrated next:

```
let f : ∀X.X → X = ΛX.λx:X.x in f [Int] 10
```

The function `f` has the polymorphic (or universal) type $\forall X.X \to X$. By applying `f` to type Int (we also say that `f` is *instantiated* to Int), the resulting function has type Int → Int; it is then passed the number `10`. Hence, the program evaluates to `10`.

In contrast to this explicit, Church-style formulation, the Curry-style presentation of *polymorphic type assignment* [Curry et al. 1972] does not require type abstraction and type application to be reflected in terms. This approach, known as *implicit* polymorphism, has inspired many languages such as ML and Haskell, which use the traditional Damas-Milner type system [Damas and Milner 1982]. This type system can generalize the type of a term and can give a term of polymorphic type a (partially) instantiated type. Alternatively, implicit polymorphism can be seen as inducing a notion of subtyping relating polymorphic types to their instantiations [Mitchell 1988; Odersky and Läufer 1996]; e.g., $\forall X.X \to X <:$ Int → Int. Implicitly-polymorphic languages are often compiled into an explicitly-polymorphic language. For instance, the use of the subtyping judgment $\forall X.X \to X <:$ Int → Int is materialized by introducing an explicit instantiation [Int], and vice-versa, the use of the judgment Int → Int $<: \forall X.$Int → Int is materialized by inserting a type abstraction constructor ΛX.

Genericity vs. Parametricity. Some languages with universal type quantification also support intensional type analysis or reflection, which allows a function to behave differently depending on the type to which it is instantiated. For instance, in Java, a generic method of type $\forall X.X \to X$ can use `instanceof` to discriminate the actual type of the argument, and behave differently for `String`, say, than for `Integer`. Therefore these languages only support *genericity*, i.e., the fact that a value of a universal type can be safely instantiated at any type.[2]

Parametricity is a much stronger interpretation of universal types, which dictates that a polymorphic value *must behave uniformly* for all possible instantiations [Reynolds 1983]. This implies that one can derive interesting theorems about the behavior of a program by just looking at its type, hence the name "free theorems" coined by Wadler [1989]. For instance, one can prove using parametricity that any polymorphic function of type $\forall X.$List $X \to$ List X commutes with the polymorphic map function. Technically, parametricity is expressed in terms of a (type-indexed) *logical relation* that denotes when two terms behave similarly when viewed at a given type. All well-typed terms of System F are related to themselves in this logical relation, meaning in particular that all polymorphic terms behave uniformly at all instantiations [Reynolds 1983].

Simply put, if a value `f` has type $\forall X.X \to X$, then—modulo divergence if admitted in the considered language—genericity only tells us that `f [Int] 10` reduces to *some* integer, while parametricity tells the much stronger result that `f [Int] 10` necessarily evaluates to `10`, i.e., `f` has to be the identity function. In the context of gradual typing, Ina and Igarashi [2011] have explored genericity with a gradual variant of Java. All other work has focused on the challenge of enforcing parametricity [Ahmed et al. 2011, 2017; Igarashi et al. 2017a; New et al. 2020; Xie et al. 2018].

[2]We call this property *genericity*, by analogy to the name *generics* in use in object-oriented languages like Java and C#.

2.2 Gradual Typing

Static and dynamic typechecking have dual advantages and limitations. For instance, adopting a static discipline provides early detection of errors at the expense of conservatively rejecting some programs that would go right. On the other hand, adopting a dynamic discipline provides flexibility at the cost of extra checks (and errors!) at runtime. Gradual typing is a specific approach to combine static and dynamic checking within the same language, letting programmers control which checking discipline is used where, and supporting the convenient evolution between both [Siek and Taha 2006]. Specifically, programmers can use the unknown type ? to denote the absence of statically-known type information. Hence, a program without any ? is a statically-typed program, and a program where all binders and constants have type ? is a dynamically-typed program. In between, there is a whole spectrum of flexibility according to the programmer's needs.

Precision and consistency. To support the transition between static and dynamic type-checking, gradual languages rely on an important relation between types called *(im)precision*, which intuitively denotes how much is known about a given type. Of course ? is the least precise type. In the **gradually-typed lambda calculus GTLC**, type precision is defined as [Siek and Taha 2006; Siek et al. 2015a]:

$$\frac{}{B \sqsubseteq B} \qquad \frac{G_1 \sqsubseteq G_2 \qquad G_1' \sqsubseteq G_2'}{G_1 \to G_1' \sqsubseteq G_2 \to G_2'} \qquad \frac{}{G \sqsubseteq ?}$$

Where B stands for base types such as Int and Bool, and G stands for gradual types. For instance, $\mathsf{Int} \to \mathsf{Bool} \sqsubseteq ? \to \mathsf{Bool} \sqsubseteq ? \to ? \sqsubseteq ?$. Observe that, unlike subtyping, precision is covariant for both the domain and codomain of function types. Precision on terms, noted $t_1 \sqsubseteq t_2$, is the natural lifting of type precision to terms:

$$\frac{}{b \sqsubseteq b} \qquad \frac{}{x \sqsubseteq x} \qquad \frac{G_1 \sqsubseteq G_2 \qquad t_1 \sqsubseteq t_2}{\lambda x : G_1.t_1 \sqsubseteq \lambda x : G_2.t_2} \qquad \frac{t_1 \sqsubseteq t_2 \qquad t_1' \sqsubseteq t_2'}{t_1\ t_1' \sqsubseteq t_2\ t_2'}$$

For instance, $\lambda x : \mathsf{Int}.x \sqsubseteq \lambda x : ?.x$.

A gradual type system optimistically deals with imprecision, thereby relaxing standard relations on types. For instance, type equality is relaxed as type *consistency* $\sim$. Two types are consistent if they agree on their known parts. For instance $\mathsf{Int} \to ? \sim ? \to \mathsf{Bool}$, but $\mathsf{Int} \not\sim \mathsf{Bool}$. Type consistency can be formally defined as [Siek and Taha 2006]:

$$\frac{}{B \sim B} \qquad \frac{G_1 \sim G_2 \qquad G_1' \sim G_2'}{G_1 \to G_1' \sim G_2 \to G_2'} \qquad \frac{}{? \sim G} \qquad \frac{}{G \sim ?}$$

Regarding the dynamics semantics, the standard approach consists in elaborating gradual source terms via a typed-driven translation to a *cast calculus*, a core language with explicit runtime typechecks. The translation inserts casts at the boundaries between static and dynamic typing, ensuring at runtime that violations of static assumptions are detected and manifest as errors.

Illustration. Let us consider the following three programs A, B and C.

```
A)  let x:? = true in          B)  let x:? = true in          C)  let x:Bool = true in
      (λy:?.y + 1) x                 (λy:Int.y + 1) x               (λy:Int.y + 1) x
```

All three programs first bind true to x and then pass x as argument to a function that adds one to its argument. They only differ in their precision: i.e., the type annotations range from all statically unknown (A) to all known (C). Program A typechecks and fails at runtime when trying to add 1 to true. Program B is a more precise variant in which the function argument type is now declared to be Int. This program also fails at runtime, but it does so *earlier* than Program A: the error is

detected when trying to apply the function to true. Finally, Program c is fully static, and is ill-typed. So, by augmenting the precision of a program, we may go from failing at runtime to failing statically.

Properties of gradual languages. To characterize the static-to-dynamic checking spectrum afforded by gradual typing, Siek et al. [2015a] summarized and extended the expected properties of gradual languages, recalled hereafter:

- *Type safety* establishes that well-typed programs cannot get stuck, although they can produce runtime errors due to actual violations of (optimistic) assumptions made during type checking.
- *Conservative extension of a static discipline* means that fully-precise terms typecheck and evaluate exactly as they would in the static language. Of course, this criterion is relative to which language is considered as the "static end" of the spectrum.
- *Embedding of a dynamic discipline* characterizes the capability of the gradual language to accommodate (possibly through a syntactic translation) terms of a dynamically-checked language. Like conservative extension, this criterion is relative, this time with respect to the "dynamic end" of the spectrum.
- *Gradual guarantees.* The gradual guarantees capture the smoothness of the static-to-dynamic checking spectrum, requiring both typing (SGG) and evaluation (DGG) to be monotonic with respect to imprecision. Specifically, if a program is well typed, then a less precise version should also be well typed; likewise, if a program runs to completion without errors, so should a less precise version.

In addition to these key formal properties, there are other interesting aspects not explicitly addressed by Siek et al. [2015a] that are worth considering.

- *Harmless imprecise ascriptions.* New and Ahmed [2018] give a semantic account of the dynamic gradual guarantee, called graduality, based on the notion that imprecision induces embedding-projection pairs. A particular consequence of their formulation is that imprecise ascriptions are harmless: given a term $t : A$ and $A \sqsubseteq B$, then $t :: B :: A$ is equivalent to t.[3] Observe that this property is weaker than the DGG, as the latter is not restricted to outer ascriptions.
- *Expressiveness of imprecision.* A gradual language soundly augments the expressiveness of the original static type system. Let us illustrate what we mean in a simply-typed setting (**STLC** refers to the **simply-typed lambda calculus** with base types), and how imprecision allows bridging the gap towards System F:
  (1) Consider the STLC term $t = \lambda x : T.x$, i.e., the identity function for values of some type T. The term t is operationally valid at different types, but it cannot be given a general type in STLC. Its type has to be fixed at either $\mathsf{Int} \to \mathsf{Int}$, $\mathsf{Bool} \to \mathsf{Bool}$, etc.
  (2) Intuitively, a proper characterization of t requires going from simple types to parametric polymorphism, such as System F. In System F, we could use the type $\forall X.X \to X$ to precisely specify that t can be applied with any argument type and return the same type.
  (3) With a gradual variant of STLC, we can give term t the imprecise type $? \to ?$ to statically capture the fact that t is definitely a function, without committing to specific domain and codomain types.
  (4) This lack of precision is soundly backed by runtime enforcement, such that the term $(t\ 3)\ 1$ evaluates to a runtime type error.

[3] In this article, we write $t :: G$ for a type ascription.

2.3 Gradual Parametricity in a Nutshell

Gradual parametricity ought to support *imprecise* type information while ensuring that assumptions about *parametricity* are enforced at runtime whenever they are not definitely provable statically. To illustrate, consider the following program:

```
let g: ? = ⌈λa.λb.if b then a else a + 1⌉ in
let f: ∀X.X→X = ΛX.λx:X.g x ⌈v⌉ in
f [Int] 10
```

Function `f` is given the polymorphic type $\forall X.X \to X$, and is therefore expected to behave parametrically. It is then instantiated at type Int, and applied to the value 10. Note that `f` is implemented using a function `g` of unknown type, which is the result of embedding ($\lceil\cdot\rceil$) untyped code into the gradual language. We write $\lceil v \rceil$ to stand for the value `true` or `false`.

While this program is gradually well-typed, the compliance of `f` with respect to its declared parametric behavior is unknown statically. By parametricity, `f` should behave as the identity function (Section 2.1). But `g` itself behaves as an identity function only if its second argument $\lceil v \rceil$ is `true`. Conversely, if $\lceil v \rceil$ is `false`, a runtime error is raised to report the parametricity violation.

This example highlights two key characteristics of gradual parametricity. First, to enforce parametricity gradually requires more than tracking type safety. If we let the program reduce to 11 when $\lceil v \rceil$ is `false`, then type *safety* is not endangered; only type *soundness* (i.e., parametricity) is. Second, the statement of free theorems must account for the possible effects of the gradual language: gradual programs can produce runtime errors—and usually can also diverge even if the corresponding static language is strongly normalizing [Siek and Taha 2006].

Also, gradualizing a language with parametric polymorphism requires extending the notion of precision to account for both type variables and polymorphic types. The natural definition of precision simply proceeds congruently:

$$\frac{}{X \sqsubseteq X} \qquad \frac{G_1 \sqsubseteq G_2}{\forall X.G_1 \sqsubseteq \forall X.G_2}$$

and likewise for terms:

$$\frac{t \sqsubseteq t'}{\Lambda X.t \sqsubseteq \Lambda X.t'} \qquad \frac{t \sqsubseteq t' \quad G \sqsubseteq G'}{t\ [G] \sqsubseteq t'\ [G']}$$

As we will see, this natural extension of precision to System F is not the only one that has been (and will be) considered, as it exposes a deep tension between parametricity and the gradual guarantees.

3 GRADUAL PARAMETRICITY: CHALLENGES

While the basics of gradual parametricity illustrated above are uncontroversial, the devil is in the details. There are fundamental tensions in the design of gradual parametricity that arise from the desirable metatheoretical properties—both of parametricity and of gradual typing—which do not seem to be simultaneously satisfiable. We now explain how existing languages in the gradual parametricity design space differ, covering each desirable property described in the previous section, plus a few properties previously not covered. Section 4.4 describes how GSF is situated in this space.

Parametricity. Establishing that a gradual parametric language enforces parametricity has been a long-standing open issue: early work on the polymorphic blame calculus did not prove parametricity [Ahmed et al. 2009b, 2011; Matthews and Ahmed 2008], and the first parametricity result was established several years later for a variant of that calculus, λB [Ahmed et al. 2017]. In fact, λB is a cast calculus, not a gradual source language, meaning that explicit casts should be sprinkled in

the program above to achieve the same result. Igarashi et al. [2017a] develop a gradual source language, System F_G, whose semantics are given by translation to a cast calculus, System F_C, which is a close cousin of λB. Igarashi et al. do not prove parametricity, but conjecture that due to the similarity between System F_C and λB, parametricity should hold. Xie et al. [2018] develop a language (here referred to as CSA) with implicit polymorphism, which compiles to λB and therefore inherits its parametricity result. More recently, New et al. [2020] explore a radically different point in the design space with PolyG$^\nu$, a language that requires explicit *sealing* and *unsealing* terms. This choice sidesteps some tensions and yields a *notion* of gradual parametricity that is stronger and more faithful to the original presentation of Reynolds than that of prior work. We come back to the discussion of different notions of gradual parametricity, which gets fairly technical, in Section 10.

Conservative Extension of System F. Most work on gradual parametricity—λB,[4] System F_G, CSA, as well as the present work—consider System F as the starting point, meaning that System F programs should be valid programs in these languages, and behave as they would in System F. System F_G is a conservative extension of System F, and CSA of an implicitly-polymorphic variant of System F [Damas and Milner 1982]. λB is a cast calculus whose syntax of fully-precise cast-free terms also coincides with System F. For such terms, λB is also a conservative extension of System F. However, its compatibility relation for types is not a conservative extension of type equality in System F. For instance, the polymorphic type $\forall X.X \to X$ is not only compatible with $\mathsf{Int} \to \mathsf{Int}$—a defining feature of *implicit* polymorphism—it is also compatible with $\mathsf{Int} \to \mathsf{Bool}$.

PolyG$^\nu$ departs from the syntax of System F and so is not a conservative extension of it. For instance, the following System F program defines a function `f`, which is the identity function, and instantiates it at type Int, applies it to 1, and then adds 1 to the result, yielding 2.

```
let f : ∀X.X → X = ΛX.λx:X.x in (f [Int] 1) + 1
```

This program is rejected statically in PolyG$^\nu$, because the sealing and unsealing that implicitly underlies polymorphic behavior in System F must happen *explicitly* in the syntax of terms, resulting in the more verbose program:

```
let f : ∀X.X → X = ΛX.λx:X.x in unseal_X(f [X=Int] (seal_X 1)) + 1
```

PolyG$^\nu$ forces an outward scoping of type variables, i.e., `[X=Int]` above puts `X` in scope for subsequent use by $\mathtt{seal}_X$ and $\mathtt{unseal}_X$. While not addressed by the authors, it seems reasonable to conjecture that PolyG$^\nu$ is a conservative extension of such an unusual static source language with explicit sealing.

Embedding of a Dynamic Language. Because a polymorphic language includes a simply-typed core, one naturally expects an untyped lambda calculus to be embeddable in a gradual polymorphic language [Ahmed et al. 2011]. As we will see in Section 11, and previously explored by Siek and Wadler [2016], the dynamic end of the spectrum for a gradual parametric language can be even more interesting, accommodating dynamic sealing primitives [Sumii and Pierce 2004].

Gradual Guarantees. A major tension faced by gradual parametric languages in the past decade has been to attempt to reconcile the gradual guarantees with parametricity. While this article will dive into this question repeatedly and at a quite technical level, let us present here what happens on the surface, for a programmer. Consider this program, which is the same as above, except that the return type of the function `f` is now unknown:

```
let f : ∀X.X → ? = ΛX.λx:X.x in (f [Int] 1) + 1
```

[4]Although λB is a polymorphic cast calculus, we include it in the discussion of gradual languages in this work since all other gradual languages like System F_G and CSA translate to λB to establish their dynamic semantics and properties.

Following the motto that imprecision is harmless, a programmer might expect this program to both typecheck and run without errors, yielding 2. However, in λB and System F_C, the (elaboration with casts of the) above program fails with a runtime error, because the result of `f [Int] 1` is *sealed*, and therefore unusable directly.

This failure is a violation of the **dynamic gradual guarantee (DGG)**. But in fact, technically, this behavior only is a violation if we consider the program above to actually be a more imprecise variant of the System F program where the return type of `f` is `X` instead of ?. Faced with this tension between the gradual guarantees and parametricity, Igarashi et al. [2017a] introduce a *stricter* notion of precision in System F_G, which does not allow losses of precision in *parametric* positions of a polymorphic type. For instance, in System F_G, $\forall X.X \to \mathsf{Int}$ is considered more precise than $\forall X.X \to ?$, but $\forall X.X \to X$ is not. Igarashi et al. [2017a] prove the static gradual guarantee for System F_G based on this more restrictive notion of precision, but leave the corresponding dynamic gradual guarantee as a conjecture.

PolyG$^\nu$ addresses this tension between parametricity and the dynamic gradual guarantee by uncoupling sealing and precision, using a syntax with explicit sealing and unsealing. If we start with the following fully-static program:

```
let f : ∀X.X → X = ΛX.λx:X.x in unseal_X(f [X=Int] (seal_X 1)) + 1
```

then making the return type of `f` unknown yields a program that still typechecks and runs successfully. This is clear because the sealing behavior that was causing problems is now explicit in the terms, and therefore not affected by a loss of precision in types. However, this choice of syntax is not innocuous. Consider the following imprecise program, where the body of `f` has been elided:

```
let f : ∀X.X → ? = body in unseal_X(f [X=Int] (seal_X 1)) + 1
```

If `body` is `ΛX.λx:X.x`, then the program evaluates to 2 as expected. However, if `f` is a constant function, e.g., `body` is `ΛX.λx:X.1`, then this PolyG$^\nu$ program fails because the call-site unsealing of the value returned by `f` is now invalid. If one removes `unseal_X` around the application of `f`, the program evaluates to 2, i.e., `f` behaves as a constant function. But now, the case where `body` is the polymorphic identity function fails, when trying to add 1 to a sealed value. This means that in PolyG$^\nu$, the decision to use unsealing or not at a call site cannot be done *modularly*: one needs to know the implementation of `f` to decide.

Faithful Type Instantiations. What should type instantiations on terms of unknown type mean? Below, the polymorphic identity function ends up instantiated to Int and passed a Bool value:

```
let g : ? = ΛX.λx:X.x in g [Int] true
```

This program in System F_G, and a possible adaptation to λB (following the translation proposed by Igarashi et al. [2017a]), both return `true` despite the explicit instantiation to Int. Internally, this happens because `g` is first consistently considered to be of type $\forall X.?$ in order to accommodate the type instantiation, but then the instantiation yields a substitution of Int for X in ?, which in both languages is just ?. There is no tracking of the decision to instantiate the underlying value to Int.

In contrast, if we try to write a similar program in PolyG$^\nu$:

```
let g : ? = ΛX.λx:X.x in g [X=Int] (seal_X true)
```

then this program does not even typecheck, because sealing `true` with `X` requires the types to coincide. If we ascribe `true` to the unknown type before sealing it, then the program typechecks but fails at runtime, thereby respecting the type instantiation to `Int`.

Expressiveness of Imprecision. We can unfold the exact same line of reasoning presented in Section 2.2, but this time starting from System F and bridging the gap towards System F_ω:

(1) Consider the System F term $t = \lambda x : T.(x\ [\mathsf{Int}])$, which behaves as an instantiation function to Int. The term t is operationally valid at different types, but cannot be given a general type in System F. Its type has to be fixed at either $(\forall X.X \to X) \to (\mathsf{Int} \to \mathsf{Int})$, $(\forall XY.X \to Y \to X) \to (\forall Y.\mathsf{Int} \to Y \to \mathsf{Int})$, etc.
(2) Intuitively, a proper characterization of t requires going from System F to higher-order polymorphism, such as System F_ω. In System F_ω, we could use the type $\forall P.(\forall X.P\ X) \to (P\ \mathsf{Int})$ to precisely specify that t instantiates any polymorphic argument to Int. Term t can be expressed in System F_ω as $t_\omega = \Lambda P.\lambda x : (\forall X.P\ X).x\ [\mathsf{Int}]$.
(3) With a gradual variant of System F, we ought to be able to give term t the imprecise type $(\forall X.?) \to ?$ to statically capture the fact that t is definitely a function that operates on a polymorphic argument, without committing to a specific domain scheme and codomain type.
(4) This lack of precision ought to be soundly backed by runtime enforcement, such that, given $id : \forall X.X \to X$, the term $(t\ id)$ `true` should evaluate to a runtime type error. (Observe that in System F_ω, $(t_\omega\ [\forall X.X \to X]\ id)$ `true` is ill-typed.)

The fact that λB and System F_C do not respect type instantiations on imprecise types mean that in these systems, the term $(t\ id)$ `true` does not raise any error.[5] Therefore, while these higher-order polymorphic patterns can be expressed, they are unsound.

Now consider the same example adapted to PolyG$^\nu$:

```
let t : (∀X.?) → ? = λx:(∀X.?). x [X=Int] in (t id) (seal_X true)
```

This program does not typecheck in PolyG$^\nu$ because the type variable `X` used explicitly in the body of the function is *no longer in scope* at the use site to seal the value `true`. Recall that `[X=Int]` puts `X` in scope for the rest of the *lexical* scope of the instantiation, but it does not cross function boundaries. So, in addition to the modularity issues presented in the previous section, the explicit (un)sealing mechanism of PolyG$^\nu$ cannot accommodate higher-order patterns like the above, which requires abstracting over type applications.

Polymorphic Interoperability. λB, System F_G, and PolyG$^\nu$ are languages with *explicit* polymorphism, i.e., with explicit type abstraction and type application terms. Despite this, λB and System F_G accommodate some form of implicit polymorphism, with different flavors. The underlying motivation is to support *interoperability* between typed and untyped code, considering that type abstraction and application are meaningless terms in an untyped language. The archetypal example is the System F polymorphic identity function, which one would like to be able to use in untyped code as standard function, or vice versa, using the untyped identity function as a polymorphic one.

λB features two type compatibility rules to support this kind of implicit polymorphism:

$$\text{(Comp-AllR)}\ \frac{\Sigma; \Delta, X \vdash T_1 <: T_2 \qquad X \notin T_1}{\Sigma; \Delta \vdash T_1 <: \forall X.T_2} \qquad \text{(Comp-AllL)}\ \frac{\Sigma; \Delta \vdash T_1[?/X] <: T_2}{\Sigma; \Delta \vdash \forall X.T_1 <: T_2}$$

These rules permit $\forall X.X \to X$ to be compatible with $? \to ?$, but as first identified by Xie et al. [2018] and recalled above, these rules also imply that the type $\forall X.X \to X$ is compatible with both $\forall X.\mathsf{Int} \to \mathsf{Bool}$ and $\mathsf{Int} \to \mathsf{Bool}$. System F_G does not relate $\forall X.X \to X$ with any of its static instantiations. However, it *does* relate that type with $? \to ?$, considered to be *quasi-polymorphic*, on the basis that using the unknown type should bring some of the flexibility of implicit polymorphism.

Xie et al. [2018] argue that it is preferable to clearly separate the subtyping relation induced by implicit polymorphism from the consistency relation induced by gradual types. As a result, CSA

[5] In System F_C, $(t\ id)$ `true` fails because $\forall X.?$ is not deemed consistent with $\forall X.X \to X$. Consequently, t must be declared to take an argument of type $?$ instead of $\forall X.?$. The result is the same as in λB however: no runtime error is raised.

Table 1. Comparison of Approaches to Gradual Parametricity

	Polym	SF	TS	Param	CE	ED	SGG	DGG	HIA	FTI	EI	PI
λB	mixed	✓	✓	✓	✓	✓σ	-	✗	✗	✗	✗	✓
System F_G	mixed	✓	✓	c	✓	✓	w	c (w)	-	✗	✗	✓
CSA	implicit	✗	✓	c	✓s	-	✓	✗	-	-	-	✓
PolyG$^\nu$	explicit	✗	✓	✓	na	-	-	✓	✓	✓	✗	✗
GSF	explicit	✓	✓	✓	✓	✓σ	✓	w	✓	✓	✓	✗

Polym: form of polymorphism. SF: System F syntax. TS: type safety. Param: parametricity. CE: conservative extension. ED: embedding of dynamic language. SGG: static gradual guarantee. DGG: dynamic gradual guarantee. HIA: harmless imprecise ascriptions. FTI: faithful type instantiations. EI: expressiveness of imprecision. PI: polymorphic interoperability. ✓: the property has been proven. ✗: the property is not satisfied. -: the property is not studied. na: the property does not apply. ✓s: proved only for the static semantics. ✓σ: can embed untyped lambda calculus with dynamic sealing. c: the property is explicitly conjectured but not proven. w: guarantees stated wrt a restricted precision.

features intuitive and straightforward definitions of precision and consistency, while accommodating the flexibility of implicit polymorphism in full.

Summary. Table 1 summarizes the different approaches we reviewed. The last line corresponds to our proposal, GSF, which is informally described in the next section and then formally developed throughout this article. A check denotes a property that is proven. A cross denotes a property that is not satisfied. A question mark is used for results that are explicitly conjectured, while a dash is used for results that are not studied. For the **conservative extension result (CE)** for CSA, the "s" signals that the result is only established with respect to typing, not reduction. For the **embedding of a dynamic language (ED)**, we annotate with a "σ" when the language has been shown to embed an untyped lambda calculus with sealing primitives (Siek and Wadler [2016] for λB, and Section 11.3 for GSF). We use "w" (weak) to denote gradual guarantees stated with respect to a stricter notion of precision than the natural one.

4 GSF, INFORMALLY

This paper presents the design, semantics and metatheory of GSF, a gradual counterpart of System F. Here, we briefly introduce the methodology and principles we follow to design GSF, and briefly review its properties (summarized in Table 1) and examples of use.

4.1 Design Methodology

In order to assist language designers in crafting new gradual languages, Garcia et al. [2016] proposed the **Abstracting Gradual Typing** methodology (**AGT**, for short). The promise of AGT is that, starting from a specification of the *meaning* of gradual types in terms of the set of possible static types they represent, one can systematically derive all relevant notions, including precision, consistent predicates (e.g., consistency and consistent subtyping), consistent functions (e.g., consistent meet and join), as well as a direct runtime semantics for gradual programs, obtained by reduction of gradual typing derivations augmented with evidence for consistent judgments.

The AGT methodology has so far proven effective to assist in the gradualization of a number of disciplines, including effects [Bañados Schwerter et al. 2014, 2016], record subtyping [Garcia et al. 2016], set-theoretic types [Castagna and Lanvin 2017], union types [Toro and Tanter 2017], refinement types [Lehmann and Tanter 2017], and security types [Toro et al. 2018]. The applicability of AGT to gradual parametricity is an open question repeatedly raised in the literature—see for instance the discussions of AGT by Igarashi et al. [2017a] and Xie et al. [2018]. Considering the

variety of successful applications of AGT, and the complexity of designing a gradual parametric language, in this work we decide to adopt this methodology, and report on its effectiveness.

4.2 Design Principles

Considering the many concerns involved in developing a gradual language with parametric polymorphism, we should be very clear about the principles, goals and non-goals of a specific design. In designing GSF, we respect the following design principles:

System F syntax: GSF is meant to be a gradual version of System F, and as such, adopts its syntax of both terms and types. Types are only augmented with the unknown type ? to introduce the imprecision that is at the core of gradual typing. In particular, this precludes the use of unconventional syntactic constructs like the explicit (un)sealing terms of PolyG$^\nu$.
Explicit polymorphism: GSF is a gradual counterpart to System F, and as such, is a fully *explicitly* polymorphic language: type abstraction and type application are part of the term language, reflected in types. GSF gradualizes type information, not term structure.
Simple statics: GSF embodies the complexity of dynamically enforcing parametricity solely in its dynamic semantics; its static semantics is as straightforward as possible.
Natural precision: Precision is intended to capture the level of static typing information of a gradual type, with ? as the most imprecise, and static types as the most precise [Siek et al. 2015a]. GSF preserves this simple intuition.

4.3 Properties

Regarding the challenges and properties discussed previously, here is where GSF stands (Table 1):

Type safety: GSF is type safe, meaning all programs either evaluate to a value, halt with a runtime error, or diverge. Well-typed GSF terms do not get stuck.
Parametricity: GSF enforces a notion of gradual parametricity (Section 10), directly inspired by λB [Ahmed et al. 2017].
Conservative extension: GSF is a conservative extension of System F: both languages coincide in their static and dynamic semantics for fully static programs.
Embedding of a dynamic language: We show that a standard dynamically-typed language can be embedded in GSF. While novel, this result is not particularly surprising. More interestingly, we prove that GSF can embed a dynamically-typed language with runtime sealing primitives [Sumii and Pierce 2004] (Section 11).
Static gradual guarantee: By virtue of the simple statics principle stated above, GSF satisfies the static gradual guarantee, i.e., typeability is monotonic with respect to the natural notion of precision.
Dynamic gradual guarantee: GSF does not satisfy the **dynamic gradual guarantee (DGG)** for the natural notion of precision, but it does satisfy a weaker DGG (Section 9).
Harmless imprecise ascriptions: The weak DGG satisfied by GSF, in particular, implies that imprecise ascriptions are harmless.
Faithful type instantiations: GSF enforces type instantiations of imprecise types.
Expressive imprecision: GSF soundly supports imprecise higher-order polymorphic patterns, bridging the gap towards System F$_\omega$.
Polymorphic interoperability: GSF, like System F and PolyG$^\nu$, only supports explicit polymorphism. This means that certain desirable interoperability scenarios are not supported.

As will be clear by the end of this article, the conflict between the DGG and parametricity in a setting that respects the type-driven approach to sealing seems extremely challenging to address,

if at all possible. Doing so would require significant changes to the semantics of GSF. In contrast, we believe that the limitation regarding polymorphic interoperability is minor—see illustration and discussion below.

4.4 GSF in Action

We now briefly illustrate GSF in action with a number of examples that correspond to the main properties of the language. Other illustrative examples are available with the online interactive prototype. The different sections of the rest of this paper also come back to such representative examples as needed.

First, System F programs are GSF programs, and behave as expected:

```
let f : ∀X.X → X = ΛX.λx:X.x in (f [Int] 1) + 1  ----> 2
```

GSF enforces gradual parametricity. Recall the example from Section 2.3:

```
let g: ? = λa:?.λb:?.if b then a else a + 1 in
let f: ∀X.X→X = ΛX.λx:X.g x ⌈v⌉ in
f [Int] 10
```

As expected, if ⌈v⌉ is `true`, the program reduces to `10`, and if ⌈v⌉ is `false`, the program fails with a runtime error when the body of the function `g` attempts to perform an addition, since this type-specific operation is a violation of parametricity.

In GSF, the natural notion of precision is used for typing, meaning that the following program is a less precise version than the System F program given at the beginning of this section. Also, imprecise ascriptions on values are harmless:

```
let f : ∀X.X → ? = ΛX.λx:X.x in (f [Int] 1) + 1      ----> 2
```

However, as pointed out in the introduction, GSF does not satisfy the dynamic gradual guarantee relative to the natural notion of precision. Consider the following programs:

```
(ΛX.λx:X.x :: X) [Int] 1  ----> 1
(ΛX.λx:?.x :: X) [Int] 1  ----> error
```

Using the natural notion of term precision, the former is more precise than the latter. Therefore, the dynamic gradual guarantee mandates that less precise term should also reduce to a value, instead of failing. Section 9.1 explains the reason for this behavior, after having presented the dynamic semantics of GSF in detail.

GSF enforces type instantiations even when applied to an imprecisely-typed value:

```
let g : ? = ΛX.λx:X.x in g [Int] true                ----> error
```

GSF soundly augments the expressiveness of System F to higher-order polymorphic code:

```
let t : (∀X.?) → ? = λx:(∀X.?).x [Int] in (t id) 1      ----> 1
let t : (∀X.?) → ? = λx:(∀X.?).x [Int] in (t id) true   ----> error
```

In GSF, we can exploit the underlying runtime sealing mechanism used to enforce gradual parametricity in order to emulate the runtime sealing primitives of the cryptographic lambda calculus [Sumii and Pierce 2004] (Section 11). Indeed, we can define a pair of functions of type $\forall X.(X \to ?) \times (? \to X)$. The first component of the pair is a function of type $X \to ?$, which intuitively justifies sealing the argument (of type X) at runtime, but not unsealing the returned value (of type ?). Dually, the type of the second component is $? \to X$, which only justifies unsealing the returned value. The underlying mechanism ensures that the unsealing function only succeeds if its argument was sealed by the first function:

```
let p : ∀X.(X → ?)×(? → X) = ΛX.⟨λx:X.x::?, λx:?.x::X⟩ in
let su = p [?] in let seal = fst su in let unseal = snd su in
(unseal (seal 1)) + 1    ----> 2
```

On the second line, `p [?]` creates a fresh pair of functions with an underlying type name that acts as the runtime sealing key: therefore, `seal` seals the value `1`, and `unseal` unseals it. The whole program reduces to 2. Unsealing the sealed value with any other generated unsealing function, or attempting to add directly to the sealed value, yields a runtime error.

Another extension of GSF studied in this article is that of existential types (Section 12), which are the foundation of data abstraction and information hiding [Mitchell and Plotkin 1988]. As an example, consider a semaphore **abstract datatype (ADT)** with operations: `bit` to create a semaphore, `flip` to produce a semaphore in the inverted state, and `read` to consult the state of the semaphore. This interface can be expressed with the existential type $\texttt{Sem} \triangleq \exists X.\{X, X \to X, X \to \text{Bool}\}$. Consider the embedding of an untyped implementation of a semaphore, `u`, which is then declared to have the static existential type `Sem`, using the unknown type ? as the representation type:

```
let u : ? = ⌈{bit = true,flip = λx.not x,read = λx.x}⌉ in
let t : Sem = pack⟨?, t⟩ as Sem in
unpack⟨X,x⟩ = t in x.read (x.flip arg)
```

If `arg` is `x.bit`, then the program runs properly without error, yields `false`. But, if the programmer tries to violate type abstraction by passing a boolean value such as `true` for `arg`, a runtime error is raised. Gradual existential types accommodate a variety of scenarios, including both imprecise ADT signatures and implementations. As we will see, gradual parametricity also allows proving representation independence results between gradual ADTs.

Regarding the limitation of GSF regarding polymorphic interoperability, the following program fails at runtime:

```
let g : ? = λx:(∀.X.X → X).x [Int] 1
let h : ? = λx:?.x
g h
```

The runtime error is raised when `g` is applied to `h`, because $? \to ?$ (the "underlying type" of `h`) is not consistent with the polymorphic function type $\forall X.X \to X$. In certain simple scenarios, it is possible to address this limitation by manually introducing type abstractions or applications, however a more systematic and generally applicable mechanism is definitely desirable. We are studying an extension of GSF with a dynamic adaptation mechanism that addresses polymorphic interoperability. In essence, in the scenario above, the runtime system automatically wraps a type abstraction around `h` instead of failing at the application. A dual adaptation occurs for missing type applications. We conjecture that this mechanism would enable GSF to smoothly support interaction with untyped code, but the full development of this technique is left for future work.

5 PRELIMINARY: THE STATIC LANGUAGE SF

We systematically derive GSF by applying AGT to a largely standard polymorphic language similar to System F, called SF (Figure 1). In addition to the standard System F types and terms, SF includes base types B inhabited by constants b, typed using the auxiliary function ty, and primitive n-ary operations op that operate on base types and are given meaning by the function δ. SF also includes pairs $\langle t_1, t_2 \rangle$, and the associated projection operations $\pi_i(t)$,[6] as well as type ascriptions $t :: T$.

The statics are standard. The typing judgment is defined over three contexts: a type name store Σ (explained below), a type variable set Δ that keeps track of type variables in scope, and a standard

[6]We omit the constraint $i \in \{1, 2\}$ when operating on pairs throughout this paper.

$x \in \textsc{Var}, X \in \textsc{TypeVar}, \alpha \in \textsc{TypeName} \quad \Sigma \in \textsc{TypeName} \xrightarrow{\text{fin}} \textsc{Type}, \Delta \subset \textsc{TypeVar}, \Gamma \in \textsc{Var} \xrightarrow{\text{fin}} \textsc{Type}$

$$\begin{array}{lll} T & ::= & B \mid T \to T \mid \forall X.T \mid T \times T \mid X \mid \alpha \quad \text{(types)} \\ t & ::= & b \mid \lambda x : T.t \mid \Lambda X.t \mid \langle t, t \rangle \mid x \mid t :: T \mid op(\overline{t}) \mid t\ t \mid t\ [T] \mid \pi_i(t) \quad \text{(terms)} \\ v & ::= & b \mid \lambda x : T.t \mid \Lambda X.t \mid \langle v, v \rangle \quad \text{(values)} \end{array}$$

Well-typed terms $\boxed{\Sigma; \Delta; \Gamma \vdash t : T}$

$$(\text{T}b)\frac{ty(b) = B \qquad \Sigma; \Delta \vdash \Gamma}{\Sigma; \Delta; \Gamma \vdash b : B} \qquad (\text{T}\lambda)\frac{\Sigma; \Delta; \Gamma, x : T \vdash t : T'}{\Sigma; \Delta; \Gamma \vdash \lambda x : T.t : T \to T'}$$

$$(\text{T}\Lambda)\frac{\Sigma; \Delta, X; \Gamma \vdash t : T \qquad \Sigma; \Delta \vdash \Gamma}{\Sigma; \Delta; \Gamma \vdash \Lambda X.t : \forall X.T} \qquad (\text{Tpair})\frac{\Sigma; \Delta; \Gamma \vdash t_1 : T_1 \qquad \Sigma; \Delta; \Gamma \vdash t_2 : T_2}{\Sigma; \Delta; \Gamma \vdash \langle t_1, t_2 \rangle : T_1 \times T_2}$$

$$(\text{Tx})\frac{x : T \in \Gamma \qquad \Sigma; \Delta \vdash \Gamma}{\Sigma; \Delta; \Gamma \vdash x : T} \qquad (\text{Tasc})\frac{\Sigma; \Delta; \Gamma \vdash t : T \qquad \Sigma; \Delta \vdash T = T'}{\Sigma; \Delta; \Gamma \vdash t :: T' : T'}$$

$$(\text{Top})\frac{\Sigma; \Delta; \Gamma \vdash \overline{t} : \overline{T_1} \qquad ty(op) = \overline{T_2} \to T \qquad \Sigma; \Delta \vdash \overline{T_1} = \overline{T_2}}{\Sigma; \Delta; \Gamma \vdash op(\overline{t}) : T} \qquad (\text{Tapp})\frac{\Sigma; \Delta; \Gamma \vdash t_1 : T_1 \qquad \Sigma; \Delta; \Gamma \vdash t_2 : T_2 \qquad \Sigma; \Delta \vdash dom(T_1) = T_2}{\Sigma; \Delta; \Gamma \vdash t_1\ t_2 : cod(T_1)}$$

$$(\text{TappT})\frac{\Sigma; \Delta; \Gamma \vdash t : T \qquad \Sigma; \Delta \vdash T'}{\Sigma; \Delta; \Gamma \vdash t\ [T'] : inst(T, T')} \qquad (\text{Tpair}i)\frac{\Sigma; \Delta; \Gamma \vdash t : T}{\Sigma; \Delta; \Gamma \vdash \pi_i(t) : proj_i(T)}$$

$dom : \textsc{Type} \rightharpoonup \textsc{Type}$	$cod : \textsc{Type} \rightharpoonup \textsc{Type}$	$inst : \textsc{Type}^2 \rightharpoonup \textsc{Type}$	$proj_i : \textsc{Type} \rightharpoonup \textsc{Type}$
$dom(T_1 \to T_2) = T_1$	$cod(T_1 \to T_2) = T_2$	$inst(\forall X.T, T') = T[T'/X]$	$proj_i(T_1 \times T_2) = T_i$
$dom(T)$ undefined o/w	$cod(T)$ undefined o/w	$inst(T, T')$ undefined o/w	$proj_i(T)$ undefined o/w

Type equality $\boxed{\Sigma; \Delta \vdash T = T}$

$$\frac{\vdash \Sigma}{\Sigma; \Delta \vdash B = B} \qquad \frac{\vdash \Sigma \qquad X \in \Delta}{\Sigma; \Delta \vdash X = X} \qquad \frac{\Sigma; \Delta \vdash T_1 = T_1' \qquad \Sigma; \Delta \vdash T_2 = T_2'}{\Sigma; \Delta \vdash T_1 \to T_2 = T_1' \to T_2'}$$

$$\frac{\Sigma; \Delta, X \vdash T_1 = T_2}{\Sigma; \Delta \vdash \forall X.T_1 = \forall X.T_2} \qquad \frac{\Sigma; \Delta \vdash T_1 = T_1' \qquad \Sigma; \Delta \vdash T_2 = T_2'}{\Sigma; \Delta \vdash T_1 \times T_2 = T_1' \times T_2'}$$

$$\frac{\vdash \Sigma \qquad \alpha \in dom(\Sigma)}{\Sigma; \Delta \vdash \alpha = \alpha} \qquad \frac{\Sigma; \Delta \vdash \Sigma(\alpha) = T}{\Sigma; \Delta \vdash \alpha = T} \qquad \frac{\Sigma; \Delta \vdash T = \Sigma(\alpha)}{\Sigma; \Delta \vdash T = \alpha}$$

Notion of reduction $\boxed{\Sigma \triangleright t \longrightarrow \Sigma \triangleright t}$

$$\Sigma \triangleright v :: T \longrightarrow \Sigma \triangleright v \qquad \Sigma \triangleright op(\overline{v}) \longrightarrow \Sigma \triangleright \delta(op, \overline{v}) \qquad \Sigma \triangleright (\lambda x : T.t)\ v \longrightarrow \Sigma \triangleright t[v/x]$$

$$\Sigma \triangleright (\Lambda X.t)\ [T] \longrightarrow \Sigma, \alpha := T \triangleright t[\alpha/X] \quad \text{where } \alpha \notin dom(\Sigma) \qquad \Sigma \triangleright \pi_i(\langle v_1, v_2 \rangle) \longrightarrow \Sigma \triangleright v_i$$

Evaluation frames and reduction $\boxed{\Sigma \triangleright t \longmapsto \Sigma \triangleright t}$

$$f ::= \Box :: T \mid op(\overline{v}, \Box, \overline{t}) \mid \Box\ t \mid v\ \Box \mid \Box\ [T] \mid \langle \Box, t \rangle \mid \langle v, \Box \rangle \mid \pi_i(\Box) \quad \text{(term frames)}$$

$$\frac{\Sigma \triangleright t \longrightarrow \Sigma' \triangleright t'}{\Sigma \triangleright t \longmapsto \Sigma' \triangleright t'} \qquad \frac{\Sigma \triangleright t \longmapsto \Sigma' \triangleright t'}{\Sigma \triangleright f[t] \longmapsto \Sigma' \triangleright f[t']}$$

Fig. 1. SF: Static polymorphic language with runtime type generation.

type environment Γ that associates term variables to types. We adopt the convention of using partial type functions to denote computed types in the rules: *dom* and *cod* for domain and codomain types, *inst* for the resulting type of an instantiation, and $proj_i$ for projected types. These partial functions are undefined if the argument is not of the appropriate shape. We also make the use of type equality explicit as a premise whenever necessary. These conventions are helpful for lifting the static semantics to the gradual setting [Garcia et al. 2016]. For closed terms, we write $\cdot;\cdot;\cdot \vdash t : T$, or simply $\vdash t : T$.

The dynamics are standard call-by-value semantics, specified using reduction frames. The only peculiarity is that they rely on *runtime type generation*. The decision of using type names instead of the traditional substitution semantics is in anticipation of gradualization, and based on prior work that has shown that runtime type generation is key in order to be able to dynamically distinguish between different type variables instantiated with the same type [Ahmed et al. 2011, 2017; Matthews and Ahmed 2008]. We follow the approach already in SF because we want the dynamics and type soundness argument of the static language to help us with GSF, as afforded by AGT [Garcia et al. 2016]. Specifically, upon type application, a fresh type name α is generated and bound to the instantiation type T in a global type name store Σ. A type name store Σ maps type names to types; source terms before reduction are typechecked with an empty name store. The notion of reduction and reduction rules all carry along the type name store. While type names only occur at runtime, and not in source programs, reasoning about SF terms as they reduce requires accounting for programs with type names in them. This is why the typing rules are defined relative to a type name store as well. Similarly, type equality is relative to a type name store: a type name α is considered equal to its associated type in the store. The recursive definition of equality modulo type names is necessary to derive equalities [Igarashi et al. 2017a]. For instance, in the reduction of the well-typed program $(id\ [\mathsf{Int} \to \mathsf{Int}])\ (id\ [\mathsf{Int}])$, where *id* is the polymorphic identity function, the equality $\alpha := \mathsf{Int} \to \mathsf{Int}, \beta := \mathsf{Int}; \Delta \vdash \alpha = \beta \to \beta$ should be derivable.

Rules in Figure 1 appeal to auxiliary well-formedness judgments, omitted for brevity. A type T is well-formed ($\Sigma; \Delta \vdash T$) if it only contains type variables in the type variable environment Δ, and type names bound in a well-formed type name store. A type name store is well-formed ($\vdash \Sigma$) if all type names are distinct, and associated to types that are well-formed with respect to Σ and the empty type variable environment. A type environment Γ binds term variables to types, and is well-formed ($\Sigma; \Delta \vdash \Gamma$) if all types are well-formed.

Unsurprisingly, SF is type safe, and all well-typed terms are parametric. These results also follow from the properties of GSF, and the strong relation between both languages.

6 GSF: STATICS

The first step of the **Abstracting Gradual Typing** methodology **(AGT)** is to define the syntax of gradual types and give them meaning through a concretization function to the set of static types they denote. Then, by finding the corresponding abstraction function to establish a Galois connection, the static semantics of the static language can be lifted to the gradual setting.

6.1 Syntax and Syntactic Meaning of Gradual Types

We introduce the syntactic category of gradual types $G \in \textsc{GType}$, by admitting the unknown type in any position, namely:

$$G ::= B \mid G \to G \mid \forall X.G \mid G \times G \mid X \mid \alpha \mid ?$$

Observe that static types T are syntactically included in gradual types G.

The syntactic meaning of gradual types is straightforward: the unknown type represents any type, and a precise type (constructor) represents the equivalent static type (constructor).

$$
\begin{aligned}
C &: \text{GType} \to \mathcal{P}^*(\text{Type}) \\
C(B) &= \{\, B \,\} \\
C(G_1 \to G_2) &= \{T_1 \to T_2 \mid T_1 \in C(G_1), T_2 \in C(G_2)\} \\
C(G_1 \times G_2) &= \{T_1 \times T_2 \mid T_1 \in C(G_1), T_2 \in C(G_2)\} \\
C(X) &= \{\, X \,\} \\
C(\alpha) &= \{\, \alpha \,\} \\
C(\forall X.G) &= \{\forall X.T \mid T \in C(G)\} \\
C(?) &= \text{Type}
\end{aligned}
\qquad
\begin{aligned}
A &: \mathcal{P}^*(\text{Type}) \to \text{GType} \\
A(\{\, B \,\}) &= B \\
A(\{\, \overline{T_{i1} \to T_{i2}} \,\}) &= A(\{\, \overline{T_{i1}} \,\}) \to A(\{\, \overline{T_{i2}} \,\}) \\
A(\{\, \overline{T_{i1} \times T_{i2}} \,\}) &= A(\{\, \overline{T_{i1}} \,\}) \times A(\{\, \overline{T_{i2}} \,\}) \\
A(\{\, X \,\}) &= X \\
A(\{\, \alpha \,\}) &= \alpha \\
A(\{\, \overline{\forall X.T_i} \,\}) &= \forall X.A(\{\, \overline{T_i} \,\}) \\
A(\{\, \overline{T_i} \,\}) &= ? \ \ \textit{otherwise}
\end{aligned}
$$

Fig. 2. Type concretization (C) and abstraction (A).

For example, Int $\to$? denotes the set of all function types from Int to any static type. Perhaps surprisingly, we can simply extend this syntactic approach to deal with universal types, type variables, and type names; the concretization function C is defined in Figure 2. Note that the definition is purely syntactic and does not even consider well-formedness (? stands for *any* static type); notions built above concretization, such as consistency, will naturally embed the necessary restrictions (Section 6.2). Crucially, choosing to let ? stand for any static type means that ? can in particular stand for a type variable X (because $X \in C(?)$). Therefore, the gradual type $\forall X.? \to X$ includes in its denotation the static types $\forall X.X \to X$ (the identity function), $\forall X.\text{Int} \to X$ (a function that always fails when applied), $\forall X.(X * X) \to X$ (a function that given a pair returns the first or second projection), and so on.

Following the abstract interpretation framework, the notion of precision is not subject to tailoring: precision coincides with set inclusion of the denoted static types [Garcia et al. 2016].

Definition 6.1 (Type Precision). $G_1 \sqsubseteq G_2$ if and only if $C(G_1) \subseteq C(G_2)$.

Proposition 6.2 (Precision, Inductively). *The inductive definition of type precision given in Figure 3 is equivalent to Definition 6.1.*

Observe that both $\forall X.X \to ?$ and $\forall X.? \to X$ are more precise than $\forall X.? \to ?$, and less precise than $\forall X.X \to X$, thereby reflecting the original intuition about precision [Siek and Taha 2006; Siek et al. 2015a]. Also $\forall X.? \to ?$ and $? \to ?$ are unrelated by precision, since they correspond to different constructors (and GSF is a language with *explicit* polymorphism); they are both more precise than ?, of course.

Dual to concretization is abstraction, which produces a gradual type from a non-empty set of static types.[7] The abstraction function A is direct (Figure 2): it preserves type constructors and falls back on the unknown type whenever a heterogeneous set is abstracted. A is both sound and optimal: it produces the *most precise* gradual type that over-approximates a given set of static types.

Proposition 6.3 (Galois Connection). *$\langle C, A\rangle$ is a Galois connection, i.e.:*
(a) (Soundness) for any non-empty set of static types $S = \{\, \overline{T} \,\}$, we have $S \subseteq C(A(S))$
(b) (Optimality) for any gradual type G, we have $A(C(G)) \sqsubseteq G$.

The notion of precision induces a notion of precision *meet* between gradual types, which coincides with the abstraction of the intersection of both concretizations [Garcia et al. 2016].

Definition 6.4 (Precision Meet). $G_1 \sqcap G_2 \triangleq A(C(G_1) \cap C(G_2))$.

[7] There is no gradual type that denotes an empty set of static types; rather, the empty set corresponds to an error [Garcia et al. 2016].

Proposition 6.5 (Meet, Inductively). *The inductive definition of meet below is equivalent to Definition 6.4.*

$$\frac{}{B \sqcap B = B} \qquad \frac{}{X \sqcap X = X} \qquad \frac{G_1 \sqcap G_1' = G_1'' \qquad G_2 \sqcap G_2' = G_2''}{(G_1 \to G_2) \sqcap (G_1' \to G_2') = G_1'' \to G_2''}$$

$$\frac{G_1 \sqcap G_1' = G_1''}{(\forall X.G_1) \sqcap (\forall X.G_1') = \forall X.G_1''} \qquad \frac{G_1 \sqcap G_1' = G_1'' \qquad G_2 \sqcap G_2' = G_2''}{(G_1 \times G_2) \sqcap (G_1' \times G_2') = G_1'' \times G_2''} \qquad \frac{}{\alpha \sqcap \alpha = \alpha}$$

$$\frac{}{G \sqcap ? = G} \qquad \frac{}{? \sqcap G = G}$$

6.2 Lifting the Static Semantics

The key point of AGT is that once the meaning of gradual types is agreed upon, there is no space for ad hoc design in the static semantics of the language. The abstract interpretation framework provides us with the *definitions* of type predicates and functions over gradual types, for which we can then find equivalent inductive or algorithmic *characterizations*.

In particular, a predicate on static types induces a counterpart on gradual types through *existential* lifting. Our only predicate in SF is type equality, whose existential lifting is type consistency:

Definition 6.6 (Consistency). $\Xi; \Delta \vdash G_1 \sim G_2$ if and only if $\Sigma; \Delta \vdash T_1 = T_2$ for some $\Sigma \in C(\Xi)$, $T_i \in C(G_i)$.

For closed types we write $G_1 \sim G_2$. This definition uses a *gradual* type name store Ξ, which binds type names to gradual types. Its concretization is the pointwise concretization:

$$C(\cdot) = \emptyset \qquad C(\Xi, \alpha := G) = \{\, \Sigma, \alpha := T \mid \Sigma \in C(\Xi), T \in C(G) \,\}$$

Note that because consistency is the consistent lifting of static type equality, which does impose well-formedness, consistency is only defined on well-formed types (i.e., $\cdot; \cdot \vdash X \sim X$ does *not* hold).

Proposition 6.7 (Consistency, Inductively). *The inductive definition of type consistency given in Figure 3 is equivalent to Definition 6.6.*

Again, observe that the resulting definition of consistency relates any two types that only differ in unknown type components, without any restriction. Also, because of explicit polymorphism, top-level constructors must match, so $? \to ?$ is not consistent with $\forall X.? \to ?$. However, in line with gradual typing, both are consistent with ?, as expected. Therefore GSF does not treat $? \to ?$ as a special "quasi-polymorphic" type, unlike System F_G [Igarashi et al. 2017a]. Rather, consistency in GSF coincides with that of CSA [Xie et al. 2018].

Lifting type functions such as *dom* requires abstraction: a lifted function is the abstraction of the results of applying the static function to all the denoted static types [Garcia et al. 2016]:

Definition 6.8 (Consistent Lifting of Functions). Let F_n be a function of type $\textsc{Type}^n \to \textsc{Type}$. Its consistent lifting $F_n^\sharp$, of type $\textsc{GType}^n \to \textsc{GType}$, is defined as: $F_n^\sharp(\overline{G}) = A(\{\, F_n(\overline{T}) \mid \overline{T} \in \overline{C(G)} \,\})$

The abstract interpretation framework allows us to prove the following definitions:

Proposition 6.9 (Consistent Type Functions). *The definitions of $dom^\sharp$, $cod^\sharp$, $inst^\sharp$, and $proj_i^\sharp$ given in Figure 3 are consistent liftings, as per Definition 6.8, of the corresponding functions from Figure 1.*

$x \in \textsc{Var}, X \in \textsc{TypeVar}, \alpha \in \textsc{TypeName} \quad \Xi \in \textsc{TypeName} \xrightarrow{\text{fin}} \textsc{GType}, \Delta \subset \textsc{TypeVar}, \Gamma \in \textsc{Var} \xrightarrow{\text{fin}} \textsc{GType}$

$$\begin{array}{rcll} G & ::= & B \mid G \to G \mid \forall X.G \mid G \times G \mid X \mid \alpha \mid ? & \text{(gradual types)} \\ t & ::= & b \mid \lambda x : G.t \mid \Lambda X.t \mid \langle t, t \rangle \mid x \mid t :: G \mid op(\overline{t}) \mid t\ t \mid t\ [G] \mid \pi_i(t) & \text{(gradual terms)} \end{array}$$

$\boxed{\Xi;\Delta;\Gamma \vdash t : G}$ Well-typed terms

$$(\text{G}b)\ \frac{ty(b) = B \qquad \Xi;\Delta \vdash \Gamma}{\Xi;\Delta;\Gamma \vdash b : B} \qquad (\text{G}\lambda)\ \frac{\Xi;\Delta;\Gamma, x : G \vdash t : G'}{\Xi;\Delta;\Gamma \vdash \lambda x : G.t : G \to G'}$$

$$(\text{G}\Lambda)\ \frac{\Xi;\Delta, X;\Gamma \vdash t : G \qquad \Xi;\Delta \vdash \Gamma}{\Xi;\Delta;\Gamma \vdash \Lambda X.t : \forall X.G} \qquad (\text{Gpair})\ \frac{\Xi;\Delta;\Gamma \vdash t_1 : G_1 \qquad \Xi;\Delta;\Gamma \vdash t_2 : G_2}{\Xi;\Delta;\Gamma \vdash \langle t_1, t_2 \rangle : G_1 \times G_2}$$

$$(\text{Gx})\ \frac{x : G \in \Gamma \qquad \Xi;\Delta \vdash \Gamma}{\Xi;\Delta;\Gamma \vdash x : G} \qquad (\text{Gasc})\ \frac{\Xi;\Delta;\Gamma \vdash t : G \qquad \Xi;\Delta \vdash G \sim G'}{\Xi;\Delta;\Gamma \vdash t :: G' : G'}$$

$$(\text{Gop})\ \frac{\Xi;\Delta;\Gamma \vdash \overline{t} : \overline{G_1} \qquad ty(op) = \overline{G_2} \to G \qquad \Xi;\Delta \vdash \overline{G_1} \sim \overline{G_2}}{\Xi;\Delta;\Gamma \vdash op(\overline{t}) : G} \qquad (\text{Gapp})\ \frac{\Xi;\Delta;\Gamma \vdash t_1 : G_1 \qquad \Xi;\Delta;\Gamma \vdash t_2 : G_2 \qquad \Xi;\Delta \vdash dom^\sharp(G_1) \sim G_2}{\Xi;\Delta;\Gamma \vdash t_1\ t_2 : cod^\sharp(G_1)}$$

$$(\text{GappG})\ \frac{\Xi;\Delta;\Gamma \vdash t : G \qquad \Xi;\Delta \vdash G'}{\Xi;\Delta;\Gamma \vdash t\ [G'] : inst^\sharp(G, G')} \qquad (\text{Gpair}i)\ \frac{\Xi;\Delta;\Gamma \vdash t : G}{\Xi;\Delta;\Gamma \vdash \pi_i(t) : proj_i^\sharp(G)}$$

$dom^\sharp : \textsc{GType} \rightharpoonup \textsc{GType}$
$dom^\sharp(G_1 \to G_2) = G_1$
$dom^\sharp(?) = ?$
$dom^\sharp(G)$ undefined o/w

$cod^\sharp : \textsc{GType} \rightharpoonup \textsc{GType}$
$cod^\sharp(G_1 \to G_2) = G_2$
$cod^\sharp(?) = ?$
$cod^\sharp(G)$ undefined o/w

$inst^\sharp : \textsc{GType}^2 \rightharpoonup \textsc{GType}$
$inst^\sharp(\forall X.G, G') = G[G'/X]$
$inst^\sharp(?, G') = ?$
$inst^\sharp(G, G')$ undefined o/w

$proj_i^\sharp : \textsc{GType} \rightharpoonup \textsc{GType}$
$proj_i^\sharp(G_1 \times G_2) = G_i$
$proj_i^\sharp(?) = ?$
$proj_i^\sharp(G)$ undefined o/w

$\boxed{\Xi;\Delta \vdash G \sim G}$ Type consistency

$$\frac{\vdash \Xi}{\Xi;\Delta \vdash B \sim B} \qquad \frac{\vdash \Xi \qquad X \in \Delta}{\Xi;\Delta \vdash X \sim X} \qquad \frac{\Xi;\Delta \vdash G_1 \sim G_1' \qquad \Xi;\Delta \vdash G_2 \sim G_2'}{\Xi;\Delta \vdash G_1 \to G_2 \sim G_1' \to G_2'}$$

$$\frac{\Xi;\Delta, X \vdash G_1 \sim G_2}{\Xi;\Delta \vdash \forall X.G_1 \sim \forall X.G_2} \qquad \frac{\Xi;\Delta \vdash G_1 \sim G_1' \qquad \Xi;\Delta \vdash G_2 \sim G_2'}{\Xi;\Delta \vdash G_1 \times G_2 \sim G_1' \times G_2'} \qquad \frac{\vdash \Xi \qquad \alpha \in dom(\Xi)}{\Xi;\Delta \vdash \alpha \sim \alpha}$$

$$\frac{\Xi;\Delta \vdash \Xi(\alpha) \sim G}{\Xi;\Delta \vdash \alpha \sim G} \qquad \frac{\Xi;\Delta \vdash G \sim \Xi(\alpha)}{\Xi;\Delta \vdash G \sim \alpha} \qquad \frac{\Xi;\Delta \vdash G}{\Xi;\Delta \vdash G \sim ?} \qquad \frac{\Xi;\Delta \vdash G}{\Xi;\Delta \vdash ? \sim G}$$

$\boxed{G \sqsubseteq G}$ Type precision

$$\frac{}{B \sqsubseteq B} \qquad \frac{}{X \sqsubseteq X} \qquad \frac{G_1 \sqsubseteq G_1' \qquad G_2 \sqsubseteq G_2'}{G_1 \to G_2 \sqsubseteq G_1' \to G_2'} \qquad \frac{G_1 \sqsubseteq G_2}{\forall X.G_1 \sqsubseteq \forall X.G_2}$$

$$\frac{G_1 \sqsubseteq G_1' \qquad G_2 \sqsubseteq G_2'}{G_1 \times G_2 \sqsubseteq G_1' \times G_2'} \qquad \frac{}{\alpha \sqsubseteq \alpha} \qquad \frac{}{G \sqsubseteq ?}$$

Fig. 3. GSF: Syntax and static semantics.

The gradual typing rules of GSF (Figure 3) are obtained by replacing type predicates and functions with their corresponding liftings. Note that in (Gapp), the premise $\Xi;\Delta \vdash dom^\sharp(G_1) \sim G_2$ is a compositional lifting of the corresponding premise in (Tapp), as justified by Garcia et al. [2016]. Of particular interest here is the fact that a term of unknown type can be optimistically treated

as a polymorphic term and hence instantiated, yielding ? as the result type of the type application ($inst^\sharp(?, G') = ?$). In contrast, a term of function type, even imprecise, cannot be instantiated because the known top-level constructor does not match (e.g., $inst^\sharp(? \to ?, G')$ is undefined).

6.3 Static Properties of GSF

As established by Siek and Taha [2006] in the context of simple types, we can prove that the GSF type system is equivalent to the SF type system on fully-static terms. We say that a gradual type is static if the unknown type does not occur in it, and a term is static if it is fully annotated with static types. Let $\vdash_S$ denote the typing judgment of SF.[8]

Proposition 6.10 (Static Equivalence for Static Terms). *Let t be a static term and G a static type ($G = T$). We have $\vdash_S t : T$ if and only if $\vdash t : T$.*

The second important property of the static semantics of a gradual language is the static gradual guarantee, which states that typeability is monotonic with respect to precision [Siek et al. 2015a].

Type precision (Definition 6.1) extends to *term* precision. A term t is more precise than a term t' if they both have the same structure and t is more precisely annotated than t'. This means that term precision is essentially syntactic, instead of semantic as considered by New et al. [2020]. For example, $t \not\sqsubseteq t :: ?$ syntactically, because the two terms do not have the same syntactic structure, but $t :: G \sqsubseteq t :: ?$, where $\vdash t : G$.

The static gradual guarantee ensures that removing type annotations does not introduce type errors (or dually, that gradual type errors cannot be fixed by making types more precise).

Proposition 6.11 (Static Gradual Guarantee). *Let t and t' be closed GSF terms such that $t \sqsubseteq t'$ and $\vdash t : G$. Then $\vdash t' : G'$ and $G \sqsubseteq G'$.*

7 GSF: EVIDENCE-BASED DYNAMICS

We now turn to the dynamic semantics of GSF. As anticipated, this is where the complexity of gradual parametricity manifests. Still, in addition to streamlining the design of the static semantics, AGT provides effective (though incomplete) guidance for the dynamics. In this section, we first briefly recall the main ingredients of the AGT approach to dynamic semantics, namely *evidence* for consistent judgments and *consistent transitivity*. We then describe the reduction rules of GSF by treating evidence as an abstract datatype. This allows us to clarify a number of key operational aspects before turning in Section 8 to the details of the representation and operations of evidence that enable GSF to satisfy parametricity while adequately tracking type instantiations.

7.1 Background: Evidence-Based Semantics for Gradual Languages

For obtaining the dynamic semantics of a gradual language, AGT augments a consistent judgment (such as consistency or consistent subtyping) with the *evidence* of *why* such a judgment holds. Then, reduction mimics proof reduction of the type preservation argument of the static language, combining evidences through steps of *consistent transitivity*, which either yield a more precise evidence, or fail if the evidences to combine are incompatible.[9] A failure of consistent transitivity corresponds to a cast error in a traditional cast calculus [Garcia et al. 2016].

Consider the gradual typing derivation of $(\lambda x : ?.x + 1)$ false. In the inner typing derivation of the function, the consistent judgment $? \sim \mathsf{Int}$ supports the addition expression, and at the top-level, the judgment $\mathsf{Bool} \sim ?$ supports the application of the function to false. When two types

[8] As usual, the main propositions are stated over closed terms, but are proven as corollaries of statements over open terms. All statements over open terms can be found in the companion technical report.

[9] In this paper, we refer to the evidence of a consistent judgment as a countable entity. Therefore, we use the plural *evidences*, following the accepted use in academic English [Oxford 2021], instead of writing *pieces of evidence* or *evidence objects*.

are involved in a consistent judgment, we *learn* something about each of these types, namely the justification of *why* the judgment holds. This justification can be captured by a pair of gradual types, $\varepsilon = \langle G_1, G_2 \rangle$, which are at least as precise as the types involved in the judgment [Garcia et al. 2016]. (Throughout this article, we use the blue color for evidence ε to enhance readability of the structure of terms.)

$$\varepsilon \Vdash G_1 \sim G_2 \iff \varepsilon \sqsubseteq A^2(\{\langle T_1, T_2 \rangle \mid T_1 \in C(G_1), T_2 \in C(G_2), T_1 = T_2\})$$
$$\text{where } A^2(\{\overline{\langle T_{i1}, T_{i2} \rangle}\}) = \langle A(\overline{\{T_{i1}\}}), A(\overline{\{T_{i2}\}}) \rangle$$

i.e., if evidence $\langle G_1', G_2' \rangle$ justifies the consistency judgment $G_1 \sim G_2$, then $G_1' \sqsubseteq G_1$ and $G_2' \sqsubseteq G_2$. For instance, by knowing that $? \sim \mathsf{Int}$ holds, we learn that the first type can only possibly be Int, while we do not learn anything new about the right-hand side, which is already fully static. Therefore the evidence of that judgment is $\varepsilon_1 = \langle \mathsf{Int}, \mathsf{Int} \rangle$. Similarly, the evidence for the second judgment is $\varepsilon_2 = \langle \mathsf{Bool}, \mathsf{Bool} \rangle$. Types in evidence can be gradual, e.g., $\langle ? \to ?, ? \to ? \rangle$ justifies that $(? \to ?) \sim ?$. Note that with the lifting of simple static type equality, both components of the evidence always coincide, so evidence can be represented as a single gradual type. But for an asymmetric relation such as subtyping, both components are not the same [Garcia et al. 2016]; e.g., suppose $A <: B$, then $\langle A, B \rangle$ justifies that A is a consistent subtype of B. Similarly, type equality in SF is more subtle because it is relative to a type name store (Section 5), so the general presentation of evidence as pairs is also required. As an informal example, $\langle \mathsf{Int}, \alpha \rangle$ justifies that $\mathsf{Int} \sim \alpha$ relative to a store in which α is instantiated to Int; this will be explained in detail in Section 8.1.

At runtime, reduction rules need to *combine* evidences in order to either justify or refute a use of transitivity in the type preservation argument. In our example, we need to combine ε_1 and ε_2 in order to (try to) obtain a justification for the transitive judgment, namely that $\mathsf{Bool} \sim \mathsf{Int}$. The combination operation, called *consistent transitivity* $⨟$, determines whether two evidences support the transitivity: here, $\varepsilon_2 ⨟ \varepsilon_1 = \langle \mathsf{Bool}, \mathsf{Bool} \rangle ⨟ \langle \mathsf{Int}, \mathsf{Int} \rangle$ is undefined, so a runtime error is raised.

The evidence approach is very general and scales to disciplines where consistent judgments are not symmetric, involve more complex reasoning, and even other evidence combination operations [Garcia et al. 2016; Lehmann and Tanter 2017]. All the definitions involved are justified by the abstract interpretation framework. Also, both type safety and the dynamic gradual guarantee become straightforward to prove. In particular, the dynamic gradual guarantee follows directly from the monotonicity (in precision) of consistent transitivity. In fact, the generality of the approach even admits evidence to range over other abstract domains; for instance, for gradual security typing with references, evidence is defined with *label intervals*, not gradual labels [Toro et al. 2018].

7.2 Reduction for GSF

In order to denote reduction of (evidence-augmented) gradual typing derivations, Garcia et al. [2016] use *intrinsic* terms as a notational device; while appropriate, the resulting description is fairly hard to comprehend and unusual, and it does implicitly involve a (presentational) transformation from source terms to their intrinsic representation. In this work, we simplify the exposition by avoiding the use of intrinsic terms; instead, we rely on a type-directed, straightforward translation to GSFε, a simple variant of GSF in which all values are ascribed, and ascriptions carry evidence. The translation, described formally below (Section 7.3), inserts explicit ascriptions everywhere consistency is used—very much in the spirit of the coercion-based semantics of subtyping [Pierce 2002].

For instance, the small program of Section 7.1 above, $(\lambda x : ?.x + 1)\ \mathsf{false}$, is translated to:

$$(\varepsilon_{?\to\mathsf{Int}}(\lambda x : ?.(\varepsilon_1 x :: \mathsf{Int}) + (\varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int})) :: ? \to \mathsf{Int})\ (\varepsilon_2(\varepsilon_{\mathsf{Bool}}\mathsf{false} :: \mathsf{Bool}) :: ?)$$

where ε_G is the evidence of the reflexive judgment $G \sim G$ (e.g., $\varepsilon_{\mathsf{Int}}$ supports $\mathsf{Int} \sim \mathsf{Int}$). Evidences $\varepsilon_1 = \langle \mathsf{Int}, \mathsf{Int} \rangle$ and $\varepsilon_2 = \langle \mathsf{Bool}, \mathsf{Bool} \rangle$ are the ones from Section 7.1. Recall that ε_1 is evidence of the consistency judgment $? \sim \mathsf{Int}$, where $?$ is the type of x, and Int comes from the ascription; likewise ε_2 is evidence of the consistency judgment $\mathsf{Bool} \sim ?$. Such initial evidences are computed by means of an *interior* function $\mathcal{I}$, given by the abstract interpretation framework [Garcia et al. 2016]: in this setting, the interior coincides with the precision meet (Section 6.1), i.e., $\mathcal{I}(G_1, G_2) = \langle G_1 \sqcap G_2, G_1 \sqcap G_2 \rangle$.

This translation preserves the essence of the AGT dynamics approach in which evidence and consistent transitivity drive the runtime monitoring aspect of gradual typing. Furthermore, by making the translation explicitly ascribe all base values to their base type, GSFε can feature a uniform syntax and greatly simplified reduction rules, compared to the original AGT exposition. This presentation also streamlines the proofs by reducing the number of cases to consider.

Figure 4 presents the syntax and semantics of GSFε, a simple variant of GSF in which all values are ascribed, and ascriptions carry evidence. Key changes with respect to Figure 3 are highlighted in gray. Here, we treat evidence as a pair of elements of an *abstract* datatype; we define its actual representation (and operations) in the next section.

As we will see in Section 7.3, the translation from GSF to GSFε introduces explicit ascriptions everywhere consistency is used, leaving rule (*E*asc) as the only remaining use of consistency in the typing rules. The evidence of the term itself supports the consistency judgment in the premise. All other rules require types to match exactly; the translation inserts ascriptions to ensure that top-level constructors match in every elimination form.

The notion of reduction for GSFε terms deals with evidence propagation and composition with consistent transitivity. Rule (*R*asc) specifies how an ascription around an ascribed value reduces to a single value if consistent transitivity holds: ε_1 justifies that $G_u \sim G_1$, where G_u is the type of the underlying simple value u, and ε_2 is evidence that $G_1 \sim G_2$. The composition via consistent transitivity $\fatsemi$ justifies that $G_u \sim G_2$; if the composition is undefined, reduction steps to **error**. Rule (*R*op) simply strips the underlying simple values, applies the primitive operation, and then wraps the result in an ascription, using a canonical base evidence ε_B (which trivially justifies that $B \sim B$). Rule (*R*app) combines the evidence from the argument value ε_2 with the domain evidence of the function value $dom(\varepsilon_1)$ in an attempt to transitively justify that $G_u \sim G_{11}$. Failure to justify that judgment, as in our example in Section 7.1, produces **error**. The return value is ascribed to the expected return type, using the codomain evidence of the function $cod(\varepsilon_1)$. Rule (*R*pair) produces a pair value when the subterms of a pair have been reduced to values themselves, using the product operator on evidences $\varepsilon_1 \times \varepsilon_2$. This rule is necessary to enforce a uniform presentation of all values as ascribed values, which simplifies technicalities. Dually, Rule (*R*proj$i$) extracts a component of a pair and ascribes it to the projected type, using the corresponding evidence obtained with $p_i(\varepsilon)$ (not to be confused with $\pi_i(\varepsilon)$, which refers to the first or second projection of evidence, itself a metalanguage pair).

Apart from the presentational details, the above rules are standard for an evidence-based reduction semantics. Rule (*R*app*G*) is *the* rule that specifically deals with parametric polymorphism, reducing a type application.

$$\Xi \triangleright (\varepsilon \Lambda X.t :: \forall X.G)\ [G'] \longrightarrow \Xi' \triangleright \varepsilon_{out}(\varepsilon[\hat{\alpha}]t[\hat{\alpha}/X] :: G[\alpha/X]) :: G[G'/X] \quad \text{where } \Xi' \triangleq \Xi, \alpha := G' \text{ for some } \alpha \notin dom(\Xi) \text{ and } \hat{\alpha} = lift_{\Xi'}(\alpha)$$

This is where most of the complexity of gradual parametricity concentrates. Observe that there are two ascriptions in the produced term:

$$
\begin{array}{lcll}
t & ::= & v \mid \langle t, t\rangle \mid x \mid \varepsilon t :: G \mid op(\overline{t}) \mid t\ t \mid t\ [G] \mid \pi_i(t) & \text{(terms)}\\
v & ::= & \varepsilon u :: G & \text{(values)}\\
u & ::= & b \mid \lambda x : G.t \mid \Lambda X.t \mid \langle u, u\rangle & \text{(raw values)}
\end{array}
$$

$\boxed{\Xi;\Delta;\Gamma \vdash s : G}$ **Well-typed terms** (for conciseness, s ranges over both t and u)

$$(Eb)\ \frac{ty(b) = B \qquad \Xi;\Delta \vdash \Gamma}{\Xi;\Delta;\Gamma \vdash b : B} \qquad (E\lambda)\ \frac{\Xi;\Delta;\Gamma, x : G \vdash t : G'}{\Xi;\Delta;\Gamma \vdash \lambda x : G.t : G \to G'}$$

$$(E\Lambda)\ \frac{\Xi;\Delta, X;\Gamma \vdash t : G \qquad \Xi;\Delta \vdash \Gamma}{\Xi;\Delta;\Gamma \vdash \Lambda X.t : \forall X.G} \qquad (E\text{pair})\ \frac{\Xi;\Delta;\Gamma \vdash s_1 : G_1 \qquad \Xi;\Delta;\Gamma \vdash s_2 : G_2}{\Xi;\Delta;\Gamma \vdash \langle s_1, s_2\rangle : G_1 \times G_2}$$

$$(Ex)\ \frac{x : G \in \Gamma \qquad \Xi;\Delta \vdash \Gamma}{\Xi;\Delta;\Gamma \vdash x : G} \qquad (E\text{asc})\ \frac{\Xi;\Delta;\Gamma \vdash s : G \qquad \varepsilon \Vdash \Xi;\Delta \vdash G \sim G'}{\Xi;\Delta;\Gamma \vdash \varepsilon s :: G' \ : G'}$$

$$(E\text{op})\ \frac{\Xi;\Delta;\Gamma \vdash \overline{t} : \overline{G} \qquad ty(op) = \overline{G} \to G'}{\Xi;\Delta;\Gamma \vdash op(\overline{t}) : G'} \qquad (E\text{app})\ \frac{\Xi;\Delta;\Gamma \vdash t_1 : G \to G' \qquad \Xi;\Delta;\Gamma \vdash t_2 : G}{\Xi;\Delta;\Gamma \vdash t_1\ t_2 : G'}$$

$$(E\text{app}G)\ \frac{\Xi;\Delta;\Gamma \vdash t : \forall X.G \qquad \Xi;\Delta \vdash G'}{\Xi;\Delta;\Gamma \vdash t\ [G'] : G[G'/X]} \qquad (E\text{pair}i)\ \frac{\Xi;\Delta;\Gamma \vdash t : G_1 \times G_2}{\Xi;\Delta;\Gamma \vdash \pi_i(t) : G_i}$$

$\boxed{\Xi \triangleright t \longrightarrow \Xi \triangleright t \text{ or } \textbf{error}}$ **Notion of reduction**

$$
\begin{array}{lrcl}
(R\text{asc}) & \Xi \triangleright \varepsilon_2(\varepsilon_1 u :: G_1) :: G_2 & \longrightarrow & \begin{cases} \Xi \triangleright (\varepsilon_1 \fatsemi \varepsilon_2)u :: G_2 \\ \textbf{error} \qquad \text{if not defined} \end{cases}\\
(R\text{op}) & \Xi \triangleright op(\overline{\varepsilon u :: G}) & \longrightarrow & \Xi \triangleright \varepsilon_B\ \delta(op, \overline{u}) :: B \quad \text{where } B \triangleq cod(ty(op))\\
(R\text{app}) & \Xi \triangleright (\varepsilon_1(\lambda x : G_{11}.t) :: G_1 \to G_2)\ (\varepsilon_2 u :: G_1) & \longrightarrow & \begin{cases} \Xi \triangleright cod(\varepsilon_1)(t[(\varepsilon_2 \fatsemi dom(\varepsilon_1))u :: G_{11})/x]) :: G_2 \\ \textbf{error} \qquad \text{if not defined} \end{cases}\\
(R\text{pair}) & \Xi \triangleright \langle \varepsilon_1 u_1 :: G_1, \varepsilon_2 u_2 :: G_2\rangle & \longrightarrow & \Xi \triangleright (\varepsilon_1 \times \varepsilon_2)\langle u_1, u_2\rangle :: G_1 \times G_2\\
(R\text{proj}i) & \Xi \triangleright \pi_i(\varepsilon\langle u_1, u_2\rangle :: G_1 \times G_2) & \longrightarrow & \Xi \triangleright p_i(\varepsilon)u_i :: G_i\\
(R\text{app}G) & \Xi \triangleright (\varepsilon\Lambda X.t :: \forall X.G)\ [G'] & \longrightarrow & \Xi' \triangleright \varepsilon_{out}(\varepsilon[\hat{\alpha}]t[\hat{\alpha}/X] :: G[\alpha/X]) :: G[G'/X]\\
 & & & \text{where } \Xi' \triangleq \Xi, \alpha := G' \text{ for some } \alpha \notin dom(\Xi)\\
 & & & \text{and } \hat{\alpha} = lift_{\Xi'}(\alpha)
\end{array}
$$

$\boxed{\Xi \triangleright t \longmapsto \Xi \triangleright t \text{ or } \textbf{error}}$ **Evaluation frames and reduction**

$$f \quad ::= \quad \varepsilon\Box :: G \mid op(\overline{v}, \Box, \overline{t}) \mid \Box\ t \mid v\ \Box \mid \Box\ [G] \mid \langle \Box, t\rangle \mid \langle v, \Box\rangle \mid \pi_i(\Box)$$

$$(R \longrightarrow)\ \frac{\Xi \triangleright t \longrightarrow \Xi' \triangleright t'}{\Xi \triangleright t \longmapsto \Xi' \triangleright t'} \qquad (Rf)\ \frac{\Xi \triangleright t \longmapsto \Xi' \triangleright t'}{\Xi \triangleright f[t] \longmapsto \Xi' \triangleright f[t']}$$

$$(R\text{err})\ \frac{\Xi \triangleright t \longrightarrow \textbf{error}}{\Xi \triangleright t \longmapsto \textbf{error}} \qquad (Rf\text{err})\ \frac{\Xi \triangleright t \longmapsto \textbf{error}}{\Xi \triangleright f[t] \longmapsto \textbf{error}}$$

Fig. 4. GSFε: Syntax, static and dynamic semantics.

- The *inner* ascription (to $G[\alpha/X]$) is for the body of the polymorphic term, asserting that substituting a fresh type name α for the type variable X preserves typing. The associated evidence $\varepsilon[\hat{\alpha}]$ is the result of instantiating ε (which justifies that the actual type of $\Lambda X.t$ is consistent with $\forall X.G$) with the fresh type name, hence justifying that the body after substitution is consistent with $G[\alpha/X]$. The operator $lift_{\Xi'}(\alpha)$, and substitution operations $t[\hat{\alpha}/X]$ and $\varepsilon[\hat{\alpha}]$ are left abstract for now (as evidence is abstract) and defined later in Section 8.2.

- The *outer* ascription asserts that $G[\alpha/X]$ is consistent with $G[G'/X]$, witnessed by evidence ε_{out}. We define ε_{out} in Section 8.2 below, once the representation of evidence is introduced.

Instead of using these two evidences, we could have used directly their composition, $\varepsilon[\hat{\alpha}] \fatsemi \varepsilon_{out}$. But the approach used here makes the definition of the logical relation clearer and the proofs easier.

The use of $\hat{\alpha}$ is a technicality: because so far we treat evidence as an abstract datatype from an as-yet-unspecified domain, say pairs of ETYPE, we cannot directly use gradual types (GTYPE) inside evidences. The connection between GTYPE and ETYPE is specified by lifting operators, $lift_{\Xi} : \text{GTYPE} \to \text{ETYPE}$ and $unlift : \text{ETYPE} \to \text{GTYPE}$.[10] We define these operators later (Figure 6), after the structure of evidences has been explained in detail. Because type names have meaning related to a store, the lifting is parameterized by the store Ξ. Type substitution in terms is mostly standard: it uses *unlift* to recover α, and is extended to substitute recursively in evidences. Substitution in evidence, also triggered by evidence instantiation, is simply component-wise substitution on evidence types. Both substitution operators are formally defined later (Figure 7).

Finally, the evaluation frames and associated reduction rules in Figure 4 are straightforward; in particular (*R*err) and (*Rf* err) propagate **error** to the top-level.

7.3 Elaborating GSF to GSFε

Figure 5 defines the type-preserving translation from GSF to GSFε by using two mutually-defined translations: $\rightsquigarrow$ translates GSF terms to GSFε terms, and $\rightsquigarrow_v$ translates GSF values to GSFε raw values. Raw values are treated separately as they are not values (and thus not terms) and must be ascribed upon translation. The translation follows naturally from the typing rules of GSF. We use metavariable v in GSF to range over constants, functions, type abstractions and pairs of v, and use $\rightsquigarrow_v$ to translate them to raw values u. Rule (Gu) and (Gascu) translate a value v and an ascribed value in GSF, respectively, to a GSFε value, using $\rightsquigarrow_v$, producing a new ascribed raw value, i.e., a value. Note that we could have translated GSF values v to GSFε values directly but it would have generated redundant ascriptions such as $b :: \text{Bool} \rightsquigarrow \varepsilon_{\text{Bool}}(\varepsilon_{\text{Bool}} b :: \text{Bool}) :: \text{Bool}$. Note that these rules use the interior $\mathcal{I}$ to calculate the initial evidence. The rule (Gasct) is similar to (Gascu), but it uses $\rightsquigarrow$ to translate GSF terms that are not values. Rules (Gapp), (GappG) and (Gpair*i*) use *type matching* $\rightarrowtriangle$ [Cimini and Siek 2016] to ascribe subterms of type ? in elimination positions to the corresponding top-level type constructor, e.g., $\forall X.?$ for a type application, and $? \to ?$ for a function application. For subterms of a more precise type, type matching is the identity. Note that in (Gapp), the argument is also ascribed to the type of the domain G_1' obtained during type matching.

We can show straightforwardly by induction on typing judgments that the translation preserves typing (Lemma 7.1).

LEMMA 7.1 (TRANSLATION PRESERVES TYPING). *Let t be a GSF term. If $\Delta;\Gamma \vdash t : G$ then $\Delta;\Gamma \vdash t \rightsquigarrow t_\varepsilon : G$ and $\Delta;\Gamma \vdash t_\varepsilon : G$.*

8 EVIDENCE FOR GRADUAL PARAMETRICITY

As highlighted in Section 7, AGT provides effective (though incomplete) guidance for the dynamics. The dynamic semantics obtained by applying AGT ensure type safety, but unfortunately not parametricity. Ensuring parametricity requires a refined representation of evidence and definition of consistent transitivity. This can be considered as a shortcoming of the AGT methodology, already observed in the context of security typing [Toro et al. 2018]: some properties of the static language may not be preserved by gradualization. We first explain in Section 8.1 why the standard representation of evidence as pair of gradual types is insufficient for gradual parametricity. We

[10] In standard AGT [Garcia et al. 2016] the lifting is simply the identity, i.e., ETYPE = GTYPE.

$\boxed{\Delta;\Gamma \vdash v \rightsquigarrow_v u : G}$ **Value translation**

$$(\mathrm{G}b)\ \frac{ty(b) = B \qquad \Delta \vdash \Gamma}{\Delta;\Gamma \vdash b \rightsquigarrow_v b : B} \qquad (\mathrm{Gpair}u)\ \frac{\Delta;\Gamma \vdash v_1 \rightsquigarrow u_1 : G_1 \qquad \Delta;\Gamma \vdash v_2 \rightsquigarrow u_2 : G_2}{\Delta;\Gamma \vdash \langle v_1, v_2\rangle \rightsquigarrow_v \langle u_1, u_2\rangle : G_1 \times G_2}$$

$$(\mathrm{G}\lambda)\ \frac{\Delta;\Gamma, x : G \vdash t \rightsquigarrow t' : G'}{\Delta;\Gamma \vdash (\lambda x : G.t) \rightsquigarrow_v (\lambda x : G.t') : G \to G'} \qquad (\mathrm{G}\Lambda)\ \frac{\Delta, X;\Gamma \vdash t \rightsquigarrow t' : G \qquad \Delta \vdash \Gamma}{\Delta;\Gamma \vdash (\Lambda X.t) \rightsquigarrow_v (\Lambda X.t') : \forall X.G}$$

$\boxed{\Delta;\Gamma \vdash t \rightsquigarrow t : G}$ **Term translation**

$$(\mathrm{Gu})\ \frac{\Delta;\Gamma \vdash v \rightsquigarrow_v u : G \qquad \varepsilon = \mathcal{I}(G, G)}{\Delta;\Gamma \vdash v \rightsquigarrow \varepsilon u :: G : G} \qquad (\mathrm{Gascu})\ \frac{\Delta;\Gamma \vdash v \rightsquigarrow_v u : G \qquad \varepsilon = \mathcal{I}(G, G')}{\Delta;\Gamma \vdash v :: G' \rightsquigarrow \varepsilon u :: G' : G'}$$

$$(\mathrm{Gx})\ \frac{x : G \in \Gamma \qquad \Delta \vdash \Gamma}{\Delta;\Gamma \vdash x \rightsquigarrow x : G} \qquad (\mathrm{Gasct})\ \frac{t \neq v \qquad \Delta;\Gamma \vdash t \rightsquigarrow t' : G \qquad \varepsilon = \mathcal{I}(G, G')}{\Delta;\Gamma \vdash t :: G' \rightsquigarrow \varepsilon t' :: G' : G'}$$

$$(\mathrm{Gpair}t)\ \frac{(t_1 \neq v_1 \vee t_2 \neq v_2) \qquad \Delta;\Gamma \vdash t_1 \rightsquigarrow t_1' : G_1 \qquad \Delta;\Gamma \vdash t_2 \rightsquigarrow t_2' : G_2}{\Delta;\Gamma \vdash \langle t_1, t_2\rangle \rightsquigarrow \langle t_1', t_2'\rangle : G_1 \times G_2}$$

$$(\mathrm{Gop})\ \frac{\Delta;\Gamma \vdash \overline{t} \rightsquigarrow \overline{t'} : \overline{G_1} \qquad ty(op) = \overline{G_2} \to G \qquad \overline{\varepsilon} = \overline{\mathcal{I}(G_1, G_2)}}{\Delta;\Gamma \vdash op(\overline{t}) \rightsquigarrow op(\overline{\varepsilon t' :: G_2}) : G}$$

$$(\mathrm{Gapp})\ \frac{\begin{array}{l}\Delta;\Gamma \vdash t_1 \rightsquigarrow t_1' : G_1 \qquad G_1 \twoheadrightarrow G_1' \to G_2' \qquad \varepsilon_1 = \mathcal{I}(G_1, G_1' \to G_1') \\ \Delta;\Gamma \vdash t_2 \rightsquigarrow t_2' : G_2 \qquad\qquad\qquad\qquad\quad \varepsilon_2 = \mathcal{I}(G_2, G_1')\end{array}}{\Delta;\Gamma \vdash t_1\ t_2 \rightsquigarrow (\varepsilon_1 t_1 :: G_1' \to G_2')\ (\varepsilon_2 t_2' :: G_1') : G_2'}$$

$$(\mathrm{GappG})\ \frac{\Delta;\Gamma \vdash t \rightsquigarrow t' : G \qquad G \twoheadrightarrow \forall X.G'' \qquad \Delta \vdash G' \qquad \varepsilon = \mathcal{I}(G, \forall X.G'')}{\Delta;\Gamma \vdash t\ [G'] \rightsquigarrow (\varepsilon t' :: \forall X.G'')\ [G'] : G''[G'/X]}$$

$$(\mathrm{Gpair}i)\ \frac{\Delta;\Gamma \vdash t \rightsquigarrow t' : G \qquad G \twoheadrightarrow G_1 \times G_2 \qquad \varepsilon = \mathcal{I}(G, G_1 \times G_2)}{\Delta;\Gamma \vdash \pi_i(t) \rightsquigarrow \pi_i(\varepsilon t' :: G_i) : G_i}$$

$\boxed{G \twoheadrightarrow G}$ **Type matching**

$G_1 \to G_2 \twoheadrightarrow G_1 \to G_2 \qquad \forall X.G \twoheadrightarrow \forall X.G \qquad G_1 \times G_2 \twoheadrightarrow G_1 \times G_2 \qquad ? \twoheadrightarrow ? \to ? \qquad ? \twoheadrightarrow \forall X.? \qquad ? \twoheadrightarrow ? \times ?$

Fig. 5. Translation from GSF to GSFε.

then introduce the refined representation of evidence to enforce parametricity (Section 8.2), and basic properties of the language. Richer properties of GSF are discussed in Sections 9 and 10.

8.1 Simple Evidence, and Why It Fails

In standard AGT [Garcia et al. 2016], evidence is simply represented as a pair of gradual types: an evidence ε is of the form $\langle G_1, G_2\rangle$. The two constituents of an evidence are not necessarily the same, e.g., when considering non-symmetric judgments such as subtyping. Consistent transitivity is defined through the abstract interpretation framework. Write $\varepsilon \Vdash J$ to denote that ε justifies the consistent judgment J. The definition of consistent transitivity for simple types is as follows:

Definition 8.1 (Consistent Transitivity for Simple Type Equality [Garcia et al. 2016]). Suppose $\varepsilon_{ab} \Vdash G_a \sim G_b$ and $\varepsilon_{bc} \Vdash G_b \sim G_c$. Evidence for consistent transitivity is deduced as $(\varepsilon_{ab} \fatsemi \varepsilon_{bc}) \Vdash G_a \sim G_c$, where:

$$\langle G_1, G_{21}\rangle \fatsemi \langle G_{22}, G_3\rangle = A^2(\{\langle T_1, T_3\rangle \in C(G_1) \times C(G_3) \mid \exists T_2 \in C(G_{21}) \cap C(G_{22}), T_1 = T_2 \wedge T_2 = T_3\})$$

In words, if defined, the evidence that supports the transitive judgment is obtained by abstracting over the pairs of static types denoted by the outer evidence types (G_1 and G_3) *such that* they are

connected through a static type common to both middle evidence types (G_{21} and G_{22}). Note that for consistent transitivity to be defined, $C(G_{21}) \cap C(G_{22})$ must not be empty. This definition can be proven to be equivalent to an inductive definition that proceeds in a syntax-directed manner on the structure of types [Garcia et al. 2016].

Consistent transitivity satisfies some important properties. First, it is associative. Second, the resulting evidence is more precise than the outer evidence types, reflecting that during evaluation, typing justification only gets more precise (or fails). Violating this property breaks type safety. Third, it is monotonic; this property is key for establishing the dynamic gradual guarantee [Garcia et al. 2016]. Here, an evidence ε is more precise that ε', written $\varepsilon \sqsubseteq \varepsilon'$, if $\pi_1(\varepsilon) \sqsubseteq \pi_1(\varepsilon')$ and $\pi_2(\varepsilon) \sqsubseteq \pi_2(\varepsilon')$.

Lemma 8.2 (Properties of Consistent Transitivity).
(a) Associativity. $(\varepsilon_1 \fatsemi \varepsilon_2) \fatsemi \varepsilon_3 = \varepsilon_1 \fatsemi (\varepsilon_2 \fatsemi \varepsilon_3)$*, or both are undefined.*
(b) Optimality. If $\varepsilon = \varepsilon_1 \fatsemi \varepsilon_2$ *is defined, then* $\pi_1(\varepsilon) \sqsubseteq \pi_1(\varepsilon_1)$ *and* $\pi_2(\varepsilon) \sqsubseteq \pi_2(\varepsilon_2)$.
(c) Monotonicity. If $\varepsilon_1 \sqsubseteq \varepsilon_1'$ *and* $\varepsilon_2 \sqsubseteq \varepsilon_2'$ *and* $\varepsilon_1 \fatsemi \varepsilon_2$ *is defined, then* $\varepsilon_1' \fatsemi \varepsilon_2'$ *is defined and* $\varepsilon_1 \fatsemi \varepsilon_2 \sqsubseteq \varepsilon_1' \fatsemi \varepsilon_2'$.

Unfortunately, systematically following the AGT methodology and simply extending the consistent transitivity definition to deal with GSF types and consistency judgments yields a gradual language that breaks parametricity.[11] Let us first adapt the simple definition of consistent transitivity (Definition 8.1) to the GSF consistency judgment, which is stated relative to type names and type variables environments:

Definition 8.3 (Consistent Transitivity for GSF—Simple Attempt). Suppose $\varepsilon_{ab} \Vdash \Xi; \Delta \vdash G_a \sim G_b$ and $\varepsilon_{bc} \Vdash \Xi; \Delta \vdash G_b \sim G_c$. Evidence for consistent transitivity is deduced as $(\varepsilon_{ab} \fatsemi \varepsilon_{bc}) \Vdash \Xi; \Delta \vdash G_a \sim G_c$, where:

$$\langle G_1, G_{21} \rangle \fatsemi \langle G_{22}, G_3 \rangle = \begin{array}{l} A^2(\{\langle T_1, T_3 \rangle \in C(G_1) \times C(G_3) \mid \\ \quad \exists T_2 \in C(G_{21}) \cap C(G_{22}) \wedge \Sigma \in C(\Xi) \wedge \Sigma; \Delta \vdash T_1 = T_2 \wedge \Sigma; \Delta \vdash T_2 = T_3\}) \end{array}$$

where Σ is the static counterpart of Ξ, i.e., a mapping from type names to static types (Section 6.2). As previously mentioned, type equality in SF (Figure 1) is more subtle than the simple static type equality. The general representation of evidence as pairs is required (as opposition to use evidence as a single type) because each type in the evidence corresponds to each type in the judgment, which can be different. For example, suppose that α is equal to Int in the store ($\Xi = \alpha := \mathsf{Int}$). Then $\langle \mathsf{Int}, \alpha \rangle$ is evidence that Int is consistent with α, given Ξ, i.e.,

$$\langle \mathsf{Int}, \alpha \rangle \Vdash \Xi; \cdot \vdash \mathsf{Int} \sim \alpha$$

Evidence as a pair of types is crucial in the representation of the outer evidence ε_{out} during reduction. Remember that if a type abstraction with type $\forall X.G$ is applied to G', the resulting ε_{out} justifies that $G[\alpha/X]$ is consistent with $G[G'/X]$, where α is the generated fresh type name. Informally, if $\varepsilon_{out} \Vdash \Xi, \alpha := G' \vdash G[\alpha/X] \sim G[G'/X]$, then ε_{out} is computed as $\langle G[\alpha/X], G[G'/X] \rangle$. Thus, if a type abstraction with type $\forall X.X \to X$ is applied to Int, ε_{out} is computed as $\langle \alpha \to \alpha, \mathsf{Int} \to \mathsf{Int} \rangle$, where ε_{out} justifies that $(X \to X)[\alpha/X] = \alpha \to \alpha$ is consistent with $(X \to X)[\mathsf{Int}/X] = \mathsf{Int} \to \mathsf{Int}$. An evidence such as $\langle \mathsf{Int}, \alpha \rangle$ can be obtained from the domain information of ε_{out}, namely $dom(\varepsilon_{out})$ (Section 8.2, Figure 8). Taking this definition of ε_{out} into account, we can now illustrate the problem of the consistent transitivity definition derived by the AGT methodology. Consider the following simple program:

[11] The obtained language is type safe, and satisfies the dynamic gradual guarantee. This novel design could make sense to gradualize impure polymorphic languages, which do not enforce parametricity.

```
1  (ΛX.(λx:X.(x :: ? :: ?) + 1)) [Int] 1
```

The function above is not parametric because it ends up adding 1 to its argument—although it does so after two intermediate ascriptions to the type ?. Without further precaution, the parametricity violation of this program would not be detected at runtime. Note that two ascriptions are needed in order to elaborate the evidence $\langle ?, ? \rangle$, used below to illustrate the problem. The following reduction trace illustrates all the important aspects of reduction (assuming that the type application in the program below generates the fresh name α, bound to Int in the store):

$$
\begin{array}{rlcl}
 & \cdot & \triangleright & (\varepsilon_{\forall X.X \to \mathsf{Int}}(\Lambda X.\lambda x : X.(\varepsilon_?(\varepsilon_X x :: ?) :: ?) + \varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int}) :: \forall X.X \to \mathsf{Int})\ [\mathsf{Int}]\ (\varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int}) \\
\longmapsto^* & \alpha := \mathsf{Int} & \triangleright & \varepsilon_{out}(\varepsilon_{\alpha \to \alpha}(\lambda x : \alpha.(\varepsilon_?(\varepsilon_\alpha x :: ?) :: ?) + \varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int}) :: \alpha \to \alpha) :: \mathsf{Int} \to \mathsf{Int}\ (\varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int}) \\
\longmapsto^* & \alpha := \mathsf{Int} & \triangleright & \varepsilon_{out}(\lambda x : \alpha.(\varepsilon_?(\varepsilon_\alpha x :: ?) :: ?) + \varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int}) :: \mathsf{Int} \to \mathsf{Int}\ (\varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int}) \\
\longmapsto^* & \alpha := \mathsf{Int} & \triangleright & cod(\varepsilon_{out})((\varepsilon_?(\varepsilon_\alpha((\varepsilon_{\mathsf{Int}} \fatsemi dom(\varepsilon_{out}))1 :: \alpha) :: ?) :: ?) + \varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int}) :: \mathsf{Int} \\
\multicolumn{4}{l}{\text{But } dom(\varepsilon_{out}) = \langle \mathsf{Int}, \alpha \rangle \text{ and } cod(\varepsilon_{out}) = \langle \mathsf{Int}, \mathsf{Int} \rangle} \\
= & \alpha := \mathsf{Int} & \triangleright & \langle \mathsf{Int}, \mathsf{Int} \rangle((\varepsilon_?(\varepsilon_\alpha(\langle \mathsf{Int}, \alpha \rangle 1 :: \alpha) :: ?) :: ?) + \varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int}) :: \mathsf{Int} \\
\longmapsto^* & \alpha := \mathsf{Int} & \triangleright & \langle \mathsf{Int}, \mathsf{Int} \rangle((\varepsilon_?(\langle \mathsf{Int}, \alpha \rangle 1 :: ?) :: ?) + \varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int}) :: \mathsf{Int} \\
\longmapsto^* & \alpha := \mathsf{Int} & \triangleright & \langle \mathsf{Int}, \mathsf{Int} \rangle(\langle \mathsf{Int}, ? \rangle 1 :: ? + \varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int}) :: \mathsf{Int} \\
\longmapsto^* & \alpha := \mathsf{Int} & \triangleright & \langle \mathsf{Int}, \mathsf{Int} \rangle(\langle \mathsf{Int}, \mathsf{Int} \rangle 2 :: \mathsf{Int}) :: \mathsf{Int} \\
\longmapsto^* & \alpha := \mathsf{Int} & \triangleright & \langle \mathsf{Int}, \mathsf{Int} \rangle 2 :: \mathsf{Int}
\end{array}
$$

The summary below illustrates the main evidences that arise in the above reduction, showing the judgment they justify in each case:

$$
\begin{array}{rcccl}
\varepsilon_{\forall X.X \to \mathsf{Int}} = \langle \forall X.X \to \mathsf{Int}, \forall X.X \to \mathsf{Int} \rangle & \Vdash & \cdot ; \cdot & \vdash & \forall X.X \to \mathsf{Int} \sim \forall X.X \to \mathsf{Int} \\
\varepsilon_{\mathsf{Int}} = \langle \mathsf{Int}, \mathsf{Int} \rangle & \Vdash & \cdot ; \cdot & \vdash & \mathsf{Int} \sim \mathsf{Int} \\
\varepsilon_? = \langle ?, ? \rangle & \Vdash & \cdot ; X & \vdash & ? \sim ? \\
\varepsilon_X = \langle X, X \rangle & \Vdash & \cdot ; X & \vdash & X \sim ? \\
\varepsilon_{out} = \langle \alpha \to \mathsf{Int}, \mathsf{Int} \to \mathsf{Int} \rangle & \Vdash & \alpha := \mathsf{Int}; \cdot & \vdash & \alpha \to \mathsf{Int} \sim \mathsf{Int} \to \mathsf{Int} \\
\varepsilon_{\alpha \to \alpha} = \langle \alpha \to \alpha, \alpha \to \alpha \rangle & \Vdash & \alpha := \mathsf{Int}; \cdot & \vdash & \alpha \to \alpha \sim \alpha \to \alpha \\
\varepsilon_\alpha = \langle \alpha, \alpha \rangle & \Vdash & \alpha := \mathsf{Int}; \cdot & \vdash & \alpha \sim ? \\
\langle \mathsf{Int}, \alpha \rangle & \Vdash & \alpha := \mathsf{Int}; \cdot & \vdash & \mathsf{Int} \sim \alpha \\
\langle \mathsf{Int}, ? \rangle & \Vdash & \alpha := \mathsf{Int}; \cdot & \vdash & \mathsf{Int} \sim ?
\end{array}
$$

Initially, the first ascription to variable x, namely $\varepsilon_X x :: ?$, is deemed well-typed thanks to the following consistent judgment:

$$\varepsilon_X = \langle X, X \rangle \Vdash \cdot ; X \vdash X \sim ?$$

Then, after type application, $\varepsilon_{out} = \langle \alpha \to \mathsf{Int}, \mathsf{Int} \to \mathsf{Int} \rangle$ justifies that $(X \to \mathsf{Int})[\alpha/X] = \alpha \to \mathsf{Int}$ is consistent with $(X \to \mathsf{Int})[\mathsf{Int}/X] = \mathsf{Int} \to \mathsf{Int}$, i.e.,

$$\langle \alpha \to \mathsf{Int}, \mathsf{Int} \to \mathsf{Int} \rangle \Vdash \alpha := \mathsf{Int}; \cdot \vdash \alpha \to \mathsf{Int} \sim \mathsf{Int} \to \mathsf{Int}$$

Upon application, the argument $\varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int}$ is ascribed to the expected type of the function α by combining $\varepsilon_{\mathsf{Int}}$ ($\varepsilon_{\mathsf{Int}} \Vdash .; . \vdash \mathsf{Int} \sim ?$) with the domain information of ε_{out} ($dom(\varepsilon_{out}) \Vdash .; . \vdash \mathsf{Int} \sim \alpha$). Using the definition of consistent transitivity (Definition 8.3), $\langle \mathsf{Int}, \mathsf{Int} \rangle \fatsemi \langle \mathsf{Int}, \alpha \rangle = \langle \mathsf{Int}, \alpha \rangle \Vdash \alpha := \mathsf{Int}; \cdot \vdash \mathsf{Int} \sim \alpha$.[12] This operation corresponds to a sealing of the value. The sealed value is then substituted for x inside the body of the function.

[12]Following Definition 8.3: Let $\Xi = \alpha := \mathsf{Int}$, $\langle \mathsf{Int}, \mathsf{Int} \rangle \fatsemi \langle \mathsf{Int}, \alpha \rangle = A^2(\{\langle T_1, T_2 \rangle \in C(\mathsf{Int}) \times C(\alpha) \mid \exists T_2 \in C(\mathsf{Int}) \cap C(\mathsf{Int}) \wedge \Sigma \in C(\Xi) \wedge \Sigma; \Delta \vdash T_1 = T_2 \wedge \Sigma; \Delta \vdash T_2 = T_3\})$, but $C(\mathsf{Int}) = \{\mathsf{Int}\}$ $C(\alpha) = \{\alpha\}$, and $C(\Xi) = \alpha := \mathsf{Int}$. Then $\langle \mathsf{Int}, \mathsf{Int} \rangle \fatsemi \langle \mathsf{Int}, \alpha \rangle = A^2(\{\langle \mathsf{Int}, \alpha \rangle \mid \Sigma \in C(\Xi) \wedge \Sigma; \Delta \vdash \mathsf{Int} = \mathsf{Int} \wedge \Sigma; \Delta \vdash \mathsf{Int} = \alpha\}) = A^2(\{\langle \mathsf{Int}, \alpha \rangle\}) = \langle \mathsf{Int}, \alpha \rangle$.

For justifying that the value bound to x can be ascribed to ?, we need evidence for Int ∼ ? by composing the two judgments below using consistent transitivity:

$$\langle \mathsf{Int}, \alpha \rangle \Vdash \alpha := \mathsf{Int}; \cdot \vdash \mathsf{Int} \sim \alpha \qquad \langle \alpha, \alpha \rangle \Vdash \alpha := \mathsf{Int}; \cdot \vdash \alpha \sim ?$$

Note that the second judgment is obtained by substituting α for X in ε_X.

Using the definition of consistent transitivity (Definition 8.3), $\langle \mathsf{Int}, \alpha \rangle \fatsemi \langle \alpha, \alpha \rangle = \langle \mathsf{Int}, \alpha \rangle$. Similarly, for justifying the second ascription to ?, $\langle \mathsf{Int}, \alpha \rangle$ must be combined with the evidence of the judgment for the second ascription:

$$\langle ?, ? \rangle \Vdash \alpha := \mathsf{Int}; \cdot \vdash ? \sim ?$$

By Definition 8.3, $\langle \mathsf{Int}, \alpha \rangle \fatsemi \langle ?, ? \rangle = A^2(\{ \langle \mathsf{Int}, \mathsf{Int} \rangle, \langle \mathsf{Int}, \alpha \rangle \}) = \langle \mathsf{Int}, ? \rangle$. This evidence can subsequently be used to produce evidence to justify that the addition is well-typed, since $\langle \mathsf{Int}, ? \rangle \fatsemi \langle \mathsf{Int}, \mathsf{Int} \rangle = \langle \mathsf{Int}, \mathsf{Int} \rangle$. Therefore, the program produces 2, without errors: parametricity is violated.

8.2 Refining Evidence

For gradual parametricity, evidence must do more than just ensure type safety. It needs to safeguard the sealing that type variables are meant to represent, also taking care of unsealing as necessary. First of all, we need to define evidence to adequately represent consistency judgments of GSF.

Evidence Types. Instead of using gradual types in the representation of evidence, we introduce *evidence types* $E \in \text{ETYPE}$ with the following syntax:

$$E \quad ::= \quad B \mid E \to E \mid \forall X.E \mid E \times E \mid \alpha^E \mid X \mid ?$$

Then, we define an evidence ε as a pair of evidence types $\langle E_1, E_2 \rangle$. The only difference between gradual types and evidence types is in type names (highlighted in gray above). SF equality judgments, and hence GSF consistency judgments, are relative to a store. It is therefore not enough to use type names in evidence: we need to keep track of their associated types in the store. An evidence type name α^E captures the type associated to the type name α through the store. For instance, evidence that a variable has a polymorphic type X is initially $\langle X, X \rangle$. When X is instantiated, say to Int, and a fresh type name α is introduced, the evidence becomes $\langle \alpha^{\mathsf{Int}}, \alpha^{\mathsf{Int}} \rangle$. An evidence type name does not only record the end type to which it is bound, but the whole path. For instance, $\alpha^{\beta^{\mathsf{Int}}}$ is a valid evidence type name that embeds the fact that α is bound to β, which is itself bound to Int.

Note that as a program reduces, evidence can not only become more precise than statically-used types, but also more than the global store. For instance, it can be the case that $\alpha := ?$ in the global store Ξ, but that locally, the evidence for α has gotten more precise, such as α^{Int}. We use the definition $\mathit{lift}_\Xi(G)$ to enrich a type G with the type information in Ξ (Figure 6), returning an evidence type E. For instance, a type name is enriched recursively with the type that is instantiated in the store, $\mathit{lift}_\Xi(\alpha) = \alpha^{\mathit{lift}_\Xi(\Xi(\alpha))}$. Dually, unlifting ($\mathit{unlift}(E)$) forgets the additional information related to type instantiations, receiving an evidence type E and returning a gradual type G. For example, $\mathit{unlift}(\alpha^E) = \alpha$. In all other cases, both operations recur structurally (Figure 6).

It is crucial to understand the intuition behind the *position* of type names in a given evidence. The position of α^E in an evidence can correspond to a *sealing*, an *unsealing*, or neither. If α^E is *only* on the right side, e.g., $\langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle$, then the evidence is a sealing (here, of Int with α). Dually, if α^E is *only* on the left side, e.g., $\langle \alpha^{\mathsf{Int}}, \mathsf{Int} \rangle$, the evidence is an unsealing (here, of Int from α). Sealing and unsealing evidences arise through reduction, as will be illustrated later in this section.

Armed with the precise definition of evidence, Figure 7 defines the term, evidence, and evidence type substitution operations, used in the runtime semantics (Figure 4). Type substitution over a term is defined inductively over its subterms and their evidences. Observe that type substitution

$$lift_{\Xi}(G) = \begin{cases} B & G = B \\ X & G = X \\ ? & G = ? \\ lift_{\Xi}(G_1) \to lift_{\Xi}(G_2) & G = G_1 \to G_2 \\ \forall X.lift_{\Xi}(G_1) & G = \forall X.G_1 \\ lift_{\Xi}(G_1) \times lift_{\Xi}(G_2) & G = G_1 \times G_2 \\ \alpha^{lift_{\Xi}(\Xi(\alpha))} & G = \alpha \end{cases} \quad unlift(E) = \begin{cases} B & E = B \\ X & E = X \\ ? & E = ? \\ unlift(E_1) \to unlift(E_2) & E = E_1 \to E_2 \\ \forall X.unlift(E_1) & E = \forall X.E_1 \\ unlift(E_1) \times unlift(E_2) & E = E_1 \times E_2 \\ \alpha & E = \alpha^{E_1} \end{cases}$$

Fig. 6. Lifting operations.

$$s[\alpha^E/X] = \begin{cases} b & s = b \\ \lambda x : G_1[\alpha/X].t[\alpha^E/X] & s = \lambda x : G_1.t \\ \Lambda Y.t[\alpha^E/X] & s = \Lambda Y.t \\ \langle s_1[\alpha^E/X], s_2[\alpha^E/X]\rangle & s = \langle s_1, s_2\rangle \\ x & s = x \\ \varepsilon[\alpha^E/X]t[\alpha^E/X] :: G[\alpha/X] & s = \varepsilon t :: G \\ op(\overline{t[\alpha^E/X]}) & s = op(\bar{t}) \\ t_1[\alpha^E/X]\ t_2[\alpha^E/X] & s = t_1\ t_2 \\ \pi_i(t[\alpha^E/X]) & s = \pi_i(t) \\ t[\alpha^E/X]\ [G[\alpha/X]] & s = t\ [G] \end{cases}$$

$$\varepsilon[\alpha^E/X] = \langle \pi_1(\varepsilon)[\alpha^E/X], \pi_2(\varepsilon)[\alpha^E/X]\rangle$$

$$\pi_1(\langle E_1, E_2\rangle) = E_1 \qquad \pi_2(\langle E_1, E_2\rangle) = E_2$$

$$E[\alpha^{E'}/X] = \begin{cases} B & E = B \\ E_1[\alpha^{E'}/X] \to E_2[\alpha^{E'}/X] & E = E_1 \to E_2 \\ \forall Y.E_1[\alpha^{E'}/X] & E = \forall Y.E_1 \\ E_1[\alpha^{E'}/X] \times E_2[\alpha^{E'}/X] & E = E_1 \times E_2 \\ \alpha^{E_1} & E = \alpha^{E_1} \\ \alpha^{E'} & E = X \\ Y & E = Y \wedge X \neq Y \\ ? & E = ? \end{cases}$$

Fig. 7. Term and evidence type substitution.

$$dom(\langle E_{11} \to E_{12}, E_{21} \to E_{22}\rangle) = \langle E_{21}, E_{11}\rangle \qquad cod(\langle E_{11} \to E_{12}, E_{21} \to E_{22}\rangle) = \langle E_{12}, E_{22}\rangle$$
$$dom(\varepsilon) \text{ undefined o/w} \qquad cod(\varepsilon) \text{ undefined o/w}$$

$$p_i(\langle E_{11} \times E_{12}, E_{21} \times E_{22}\rangle) = \langle E_{1i}, E_{2i}\rangle \qquad \langle E_{11}, E_{21}\rangle \times \langle E_{12}, E_{22}\rangle = \langle E_{11} \times E_{12}, E_{21} \times E_{22}\rangle$$
$$p_i(\varepsilon) \text{ undefined o/w}$$

Fig. 8. Auxiliary functions for evidence.

on the annotated types of a term must transform evidence types into gradual types by using the unlift operator ($unlift(\alpha^E) = \alpha$). This occurs in the type substitution on function, ascription, and type application terms. Type substitution in evidence is defined as the type substitution in each of its components, and evidence type substitution is defined inductively in the expected way.

Figure 8 defines evidence inversion functions. For instance, if ε justifies that $G_{11} \to G_{12} \sim G_{21} \to G_{22}$, then $dom(\varepsilon)$ computes new evidence that justifies that $G_{21} \sim G_{11}$, and $cod(\varepsilon)$ computes new evidence that justifies that $G_{12} \sim G_{22}$. Similarly, if ε justifies that $G_{11} \times G_{12} \sim G_{21} \times G_{22}$, then $p_i(\varepsilon)$ justifies that $G_{1i} \sim G_{2i}$. Finally, if ε_i justifies that $G_{1i} \sim G_{2i}$, then $\varepsilon_1 \times \varepsilon_2$ justifies that $G_{11} \times G_{12} \sim G_{21} \times G_{22}$.

Consistent Transitivity. With the syntactic enrichment of evidence types, consistent transitivity can be strengthened to account for sealing and unsealing, ensuring parametricity. Consistent transitivity is defined inductively in Figure 9. Save for rules (idL) and (idR), these inductive rules are equivalent to the formal definition of consistent transitivity given by AGT (Def. 8.3). We describe the interesting rules next.

Rule (unsl) specifies that when a sealing and an unsealing of the same type name meet in the middle positions of a consistent transitivity step, the type name can be eliminated in order to calculate the resulting evidence. For instance, $\langle \mathsf{Int}, \alpha^{\mathsf{Int}}\rangle \fatsemi \langle \alpha^{?}, ?\rangle = \langle \mathsf{Int}, \mathsf{Int}\rangle \fatsemi \langle ?, ?\rangle = \langle \mathsf{Int}, \mathsf{Int}\rangle$.

$\boxed{\varepsilon \fatsemi \varepsilon = \varepsilon}$ **Consistent transitivity**

$$\text{(base)}\ \frac{}{\langle B, B\rangle \fatsemi \langle B, B\rangle = \langle B, B\rangle} \qquad \text{(typeVar)}\ \frac{}{\langle X, X\rangle \fatsemi \langle X, X\rangle = \langle X, X\rangle}$$

$$\text{(idL)}\ \frac{}{\langle E_1, E_2\rangle \fatsemi \langle ?, ?\rangle = \langle E_1, E_2\rangle} \qquad \text{(idR)}\ \frac{}{\langle ?, ?\rangle \fatsemi \langle E_1, E_2\rangle = \langle E_1, E_2\rangle}$$

$$\text{(sealL)}\ \frac{\langle E_1, E_2\rangle \fatsemi \langle E_3, E_4\rangle = \langle E_1', E_2'\rangle}{\langle E_1, E_2\rangle \fatsemi \langle E_3, \alpha^{E_4}\rangle = \langle E_1', \alpha^{E_2'}\rangle} \qquad \text{(sealR)}\ \frac{\langle E_1, E_2\rangle \fatsemi \langle E_3, E_4\rangle = \langle E_1', E_2'\rangle}{\langle \alpha^{E_1}, E_2\rangle \fatsemi \langle E_3, E_4\rangle = \langle \alpha^{E_1'}, E_2'\rangle}$$

$$\text{(unsl)}\ \frac{\langle E_1, E_2\rangle \fatsemi \langle E_3, E_4\rangle = \langle E_1', E_2'\rangle}{\langle E_1, \alpha^{E_2}\rangle \fatsemi \langle \alpha^{E_3}, E_4\rangle = \langle E_1', E_2'\rangle}$$

$$\text{(func)}\ \frac{\langle E_{41}, E_{31}\rangle \fatsemi \langle E_{21}, E_{11}\rangle = \langle E_3, E_1\rangle \qquad \langle E_{12}, E_{22}\rangle \fatsemi \langle E_{32}, E_{42}\rangle = \langle E_2, E_4\rangle}{\langle E_{11} \to E_{12}, E_{21} \to E_{22}\rangle \fatsemi \langle E_{31} \to E_{32}, E_{41} \to E_{42}\rangle = \langle E_1 \to E_2, E_3 \to E_4\rangle}$$

$$\text{(abst)}\ \frac{\langle E_1, E_2\rangle \fatsemi \langle E_3, E_4\rangle = \langle E_1', E_2'\rangle}{\langle \forall X.E_1, \forall X.E_2\rangle \fatsemi \langle \forall X.E_3, \forall X.E_4\rangle = \langle \forall X.E_1', \forall X.E_2'\rangle}$$

$$\text{(pair)}\ \frac{\langle E_{11}, E_{21}\rangle \fatsemi \langle E_{31}, E_{41}\rangle = \langle E_1, E_3\rangle \qquad \langle E_{12}, E_{22}\rangle \fatsemi \langle E_{32}, E_{42}\rangle = \langle E_2, E_4\rangle}{\langle E_{11} \times E_{12}, E_{21} \times E_{22}\rangle \fatsemi \langle E_{31} \times E_{32}, E_{41} \times E_{42}\rangle = \langle E_1 \times E_2, E_3 \times E_4\rangle}$$

$\boxed{\varepsilon \sqsubseteq \varepsilon}$ **Evidence precision**

$$\frac{E_1 \sqsubseteq E_3 \qquad E_2 \sqsubseteq E_4}{\langle E_1, E_2\rangle \sqsubseteq \langle E_3, E_4\rangle}$$

$\boxed{E \sqsubseteq E}$ **Evidence type precision**

$$\frac{}{B \sqsubseteq B} \qquad \frac{}{X \sqsubseteq X} \qquad \frac{E_1 \sqsubseteq E_2}{\alpha^{E_1} \sqsubseteq \alpha^{E_2}} \qquad \frac{}{B \sqsubseteq ?} \qquad \frac{}{? \sqsubseteq ?} \qquad \frac{E_1 \to E_2 \sqsubseteq ? \to ?}{E_1 \to E_2 \sqsubseteq ?}$$

$$\frac{E_1 \sqsubseteq E_3 \qquad E_2 \sqsubseteq E_4}{E_1 \to E_2 \sqsubseteq E_3 \to E_4} \qquad \frac{E_1 \times E_2 \sqsubseteq ? \times ?}{E_1 \times E_2 \sqsubseteq ?} \qquad \frac{E_1 \sqsubseteq E_3 \qquad E_2 \sqsubseteq E_4}{E_1 \times E_2 \sqsubseteq E_3 \times E_4} \qquad \frac{E_1 \sqsubseteq E_2}{\forall X.E_1 \sqsubseteq \forall X.E_2}$$

$$\frac{\forall X.E \sqsubseteq \forall X.?}{\forall X.E \sqsubseteq ?} \qquad \frac{}{X \sqsubseteq ?} \qquad \frac{}{\alpha^E \sqsubseteq ?}$$

Fig. 9. Consistent transitivity and evidence precision.

As shown in Section 8.1, it is important for consistent transitivity to not lose precision when combining an evidence with an unknown evidence. To this end, and contrary to the formal definition given by AGT shown at the end of Section 8.1, rule (idL) in Figure 9 preserves the left evidence. Going back to the example of Section 8.1, we now have $\langle \mathsf{Int}, \alpha^{\mathsf{Int}}\rangle \fatsemi \langle ?, ?\rangle = \langle \mathsf{Int}, \alpha^{\mathsf{Int}}\rangle$, instead of $\langle \mathsf{Int}, ?\rangle$. Because $\langle \mathsf{Int}, \alpha^{\mathsf{Int}}\rangle \fatsemi \langle \mathsf{Int}, \mathsf{Int}\rangle$ is undefined, reduction steps to **error** as desired.

Rule (sealL) shows that when an evidence is combined with a sealing, the resulting evidence is also a sealing. This sealing can be more precise, e.g., $\langle \mathsf{Int}, \mathsf{Int}\rangle \fatsemi \langle ?, \alpha^{?}\rangle = \langle \mathsf{Int}, \alpha^{\mathsf{Int}}\rangle$.

There is one rule per type constructor. For example, rule (func) corresponds to the function case, where consistent transitivity is computed recursively with the domain and codomain evidences. Also, there are symmetric variants for some rules—such as (idR) and (sealR)—in which the left and right components of each evidence are swapped.

Evidence precision. Precision for evidence and evidence types is defined in Figure 9. The definition of evidence type precision is defined analogous to the definition of type precision, accounting

as well for evidence type names α^E. We say that a type name is more precise than another if their bounded types are related by precision.

Properties. Importantly, this refined definition of consistent transitivity preserves associativity and optimality. It does however break monotonicity, and consequently, the dynamic gradual guarantee (we come back to this in Section 9).

Instantiation and Outer Evidence. The reduction rule of a type application (*R*app*G*) produces two evidences. The first one is the instantiation evidence $\varepsilon[\hat{\alpha}]$ that justifies that the type of $t[\hat{\alpha}/X]$ ($G'[\alpha/X]$) is consistent with $G[\alpha/X]$. The second one is the outer evidence ε_{out} that justifies that $G[\alpha/X]$ is consistent with $G[G'/X]$:

$$\Xi \triangleright (\varepsilon \Lambda X.t :: \forall X.G)\ [G'] \longrightarrow \Xi' \triangleright \varepsilon_{out}\,(\varepsilon[\hat{\alpha}]t[\hat{\alpha}/X] :: G[\alpha/X]) :: G[G'/X]$$
$$\text{where } \Xi' \triangleq \Xi, \alpha := G' \text{ for some } \alpha \notin dom(\Xi) \text{ and } \hat{\alpha} = lift_{\Xi'}(\alpha)$$

Evidence $\varepsilon[\hat{\alpha}]$ is defined as: $\langle E_1, E_2\rangle[\alpha^E] = \langle E_1[\alpha^E], E_2[\alpha^E]\rangle$, where the evidence types $E_i[\alpha^E]$ are obtained by the type application of E_i to α^E. The precise definition of ε_{out} is more delicate, addressing a subtle tension between ensuring the precision required for justifying unsealing when possible and not introducing new runtime errors when combined.

$$\varepsilon_{out} \triangleq \langle E_*[\alpha^E], E_*[E']\rangle \qquad \text{where } E_* = lift_{\Xi}(unlift(\pi_2(\varepsilon))), \alpha^E = lift_{\Xi'}(\alpha), E' = lift_{\Xi}(G')$$

In this definition, ε, α, G', Ξ, and Ξ' come from rule (*R*app*G*). The evidence types $E_*[\alpha^E]$ and $E_*[E']$ are obtained by the type application of E_* to α^E and E', respectively. Observe that E_* is obtained using the second component of evidence ε, and not using the information of the first component. This is because ε justifies that $\forall X.G''$ is consistent with $\forall X.G$, where G'' is the type of the body of the type abstraction, and $\forall X.G$ is the ascribed type. Therefore, evidence $\varepsilon[\hat{\alpha}]$ justifies that $G''[\alpha/X]$ is consistent with $G[\alpha/X]$, where the right (resp. left) component of $\varepsilon[\hat{\alpha}]$ corresponds to the most precise information about $G[\alpha/X]$ (resp. $G''[\alpha/X]$). As ε_{out} must justify that $G[\alpha/X]$ is consistent with $G[G'/X]$, we only use the information of the second component of ε (which corresponds to the most precise information about G). To illustrate this, consider $\varepsilon = \langle \forall X.X \to \mathsf{Bool}, \forall X.X \to \beta^{\mathsf{Bool}}\rangle$ and $\varepsilon[\hat{\alpha}] = \langle \alpha^{\mathsf{Int}} \to \mathsf{Bool}, \alpha^{\mathsf{Int}} \to \beta^{\mathsf{Bool}}\rangle$, then $\varepsilon_{out} = \langle \alpha^{\mathsf{Int}} \to \beta^{\mathsf{Bool}}, \mathsf{Int} \to \beta^{\mathsf{Bool}}\rangle$, and $\varepsilon[\hat{\alpha}] \fatsemi \varepsilon_{out} = \varepsilon_{out}$. If ε_{out} would have been constructed using the first component, then $\varepsilon_{out} = \langle \alpha^{\mathsf{Int}} \to \mathsf{Bool}, \mathsf{Int} \to \beta^{\mathsf{Bool}}\rangle$, but $\varepsilon[\hat{\alpha}] \fatsemi \varepsilon_{out}$ is not defined. Determining E_* is the key challenge. The roundtrip unlift/lift "resets" the information of evidence type names to that contained in the store. As previously mentioned, the local information of evidence can be more precise than in the global store. Therefore, it is not necessarily true that $lift_{\Xi}(unlift(\pi_2(\varepsilon))) = \pi_2(\varepsilon)$. For example, if we take $\varepsilon = \langle \mathsf{Int}, \alpha^{\mathsf{Int}}\rangle$ and $\Xi(\alpha) = ?$, then $\pi_2(\varepsilon) = \alpha^{\mathsf{Int}}$, $unlift(\pi_2(\varepsilon)) = \alpha$ and $lift_{\Xi}(unlift(\pi_2(\varepsilon))) = \alpha^?$. This technicality is crucial for proving parametricity (specifically, the compositionality lemma—see Section 10). It is important to note that although the first component of evidence ε_{out} can lose precision with respect the second component of ε, this local loss does not affect the precision of the entire program since evidence $\varepsilon[\hat{\alpha}]$ maintains the precision achieved so far. Furthermore, $\varepsilon[\hat{\alpha}] \fatsemi \varepsilon_{out}$ never fails: the role of the outer evidence ε_{out} is just to seal arguments supplied to the function, and unseal values returned by the function, and not to introduce new runtime errors; the possible runtime errors must come from the inner evidence $\varepsilon[\hat{\alpha}]$. For instance, evidence $\varepsilon_{out} = \langle \alpha^{\mathsf{Int}} \to \alpha^{\mathsf{Int}}, \mathsf{Int} \to \mathsf{Int}\rangle$ seals arguments when they are combined with $dom(\langle \alpha^{\mathsf{Int}} \to \alpha^{\mathsf{Int}}, \mathsf{Int} \to \mathsf{Int}\rangle) = \langle \mathsf{Int}, \alpha^{\mathsf{Int}}\rangle$, and unseals returned values when combined with $cod(\langle \alpha^{\mathsf{Int}} \to \alpha^{\mathsf{Int}}, \mathsf{Int} \to \mathsf{Int}\rangle) = \langle \alpha^{\mathsf{Int}}, \mathsf{Int}\rangle$.

Note that ε_{out} will never cause a runtime error when combined with the resulting evidence of the parametric term result because both are necessarily related by precision. Indeed, by (RappG):

$$\Xi \triangleright (\varepsilon \Lambda X.t :: \forall X.G)\ [G'] \longrightarrow \Xi' \triangleright \varepsilon_{out}(\varepsilon[\hat{\alpha}]t[\hat{\alpha}/X] :: G[\alpha/X]) :: G[G'/X]$$

and by (Rf):

$$\frac{\Xi' \triangleright \varepsilon[\hat{\alpha}]t[\hat{\alpha}/X] :: G[\alpha/X] \longmapsto^* \Xi'' \triangleright \varepsilon' u :: G[\alpha/X]}{\Xi' \triangleright \varepsilon_{out}(\varepsilon[\hat{\alpha}]t[\hat{\alpha}/X] :: G[\alpha/X]) :: G[G'/X] \longmapsto^* \Xi' \triangleright \varepsilon_{out}(\varepsilon' u :: G[\alpha/X]) :: G[G'/X]}$$

Since $\pi_2(\varepsilon') \sqsubseteq \pi_2(\varepsilon[\hat{\alpha}]) \sqsubseteq \pi_1(\varepsilon_{out})$, the combination $(\varepsilon' \fatsemi \varepsilon_{out})$ through transitivity never fails.

Illustration. The following reduction trace illustrates all the important aspects of reduction. Recall that we use ε_G to denote the evidence that justifies the reflexive judgment $G \sim G$. For example, $\varepsilon_{\alpha\to\alpha}$ justifies the reflexive judgment $\alpha \to \alpha \sim \alpha \to \alpha$ under environment $\Xi = \alpha := \mathsf{Int}$.

$$\varepsilon_{\alpha\to\alpha} \Vdash \Xi; \cdot \vdash \alpha \to \alpha \sim \alpha \to \alpha$$

$$\text{where } \varepsilon_{\alpha\to\alpha} = \langle \mathit{lift}_{\Xi}(\alpha \to \alpha), \mathit{lift}_{\Xi}(\alpha \to \alpha) \rangle = \langle \alpha^{\mathsf{Int}} \to \alpha^{\mathsf{Int}}, \alpha^{\mathsf{Int}} \to \alpha^{\mathsf{Int}} \rangle$$

		$(\varepsilon_{\forall X.X\to X}(\Lambda X.\lambda x : X.x) :: \forall X.X \to ?)\ [\mathsf{Int}]\ (\varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int})$	initial evidence
(R*app*G)	$\longmapsto$	$(\langle \alpha^{\mathsf{Int}} \to \alpha^{\mathsf{Int}}, \mathsf{Int} \to \mathsf{Int} \rangle (\varepsilon_{\alpha\to\alpha}(\lambda x : \alpha.x) :: \alpha \to ?) :: \mathsf{Int} \to ?)\ (\varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int})$	ε_{out} and $\varepsilon[\hat{\alpha}]$ are computed
(R*asc*)	$\longmapsto$	$(\langle \alpha^{\mathsf{Int}} \to \alpha^{\mathsf{Int}}, \mathsf{Int} \to \mathsf{Int} \rangle (\lambda x : \alpha.x) :: \mathsf{Int} \to ?)\ (\varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int})$	consistent transitivity
(R*app*)	$\longmapsto$	$\langle \alpha^{\mathsf{Int}}, \mathsf{Int} \rangle (\langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle 1 :: \alpha) :: ?$	argument is sealed
(R*asc*)	$\longmapsto$	$\langle \mathsf{Int}, \mathsf{Int} \rangle 1 :: ?$	unsealing eliminates α

Crucially, the initial evidence of the identity function is fully precise, even though it is ascribed an imprecise type. Consequently, in the first reduction step above, ε_{out} is calculated as:

$$\varepsilon_{out} \triangleq \langle E_*[\alpha^E], E_*[E'] \rangle = \langle (\forall X.X \to X)[\alpha^{\mathsf{Int}}], (\forall X.X \to X)[\mathsf{Int}] \rangle = \langle \alpha^{\mathsf{Int}} \to \alpha^{\mathsf{Int}}, \mathsf{Int} \to \mathsf{Int} \rangle$$

The application step (*R*app) then gives rise to sealing and unsealing evidences after deconstructing ε_{out}: the inner evidence $\langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle$ seals the number 1 at type α, while the outer evidence $\langle \alpha^{\mathsf{Int}}, \mathsf{Int} \rangle$ allows the subsequent unsealing in the ascription step (*R*asc). As a result, the ascribed identity function yields usable values, because the outer evidence subsequently takes care of unsealing. This addresses the violation of the dynamic gradual guarantee reported with λB and System F_C in Section 3. Note that if the function explicitly introduced imprecision, e.g., $\Lambda X.\lambda x : X.(x :: ?)$, then initial evidence would likewise be imprecise, and deconstructing ε_{out} would *not* justify unsealing the result anymore. The following reduction trace illustrates all the important aspects of reduction.

		$(\varepsilon_{\forall X.X\to ?}(\Lambda X.\lambda x : X.(\varepsilon_X x :: ?)) :: \forall X.X \to ?)\ [\mathsf{Int}]\ (\varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int})$	initial evidence
(R*app*G)	$\longmapsto$	$(\langle \alpha^{\mathsf{Int}} \to ?, \mathsf{Int} \to ? \rangle (\varepsilon_{\alpha\to ?}(\lambda x : \alpha.(\varepsilon_\alpha x :: ?)) :: \alpha \to ?) :: \mathsf{Int} \to ?)\ (\varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int})$	type application
(R*asc*)	$\longmapsto$	$(\langle \alpha^{\mathsf{Int}} \to ?, \mathsf{Int} \to ? \rangle (\lambda x : \alpha.(\varepsilon_\alpha x :: ?)) :: \mathsf{Int} \to ?)\ (\varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int})$	consistent transitivity
(R*app*)	$\longmapsto$	$\varepsilon_?(\varepsilon_\alpha(\langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle 1 :: \alpha) :: ?) :: ?$	argument is sealed
(R*asc*)	$\longmapsto$	$\varepsilon_?(\langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle 1 :: ?) :: ?$	consistent transitivity
(R*asc*)	$\longmapsto$	$\langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle 1 :: ?$	unsealing does not occur

We will return to a similar example in Section 9, which studies the dynamic gradual guarantee and why GSF does not fully satisfy it.

8.3 Basic Properties of GSF Evaluation

The runtime semantics of a GSF term are given by first translating the term to GSFε (noted $\vdash t \rightsquigarrow t_\varepsilon : G$) and then reducing the GSFε term. We write $t \Downarrow \Xi \triangleright v$ (resp. $t \Downarrow$ **error**) if $\vdash t \rightsquigarrow t_\varepsilon : G$ and $\cdot \triangleright t_\varepsilon \longmapsto^* \Xi \triangleright v$ (resp. $\cdot \triangleright t_\varepsilon \longmapsto^*$ **error**) for some resulting store Ξ. We write $\Xi \triangleright v : G$ for $\Xi; \cdot; \cdot \vdash v : G$. We write $t \Uparrow$ if the translation of t diverges, and $t \Downarrow v$ when the store is irrelevant.

The properties of GSF follow from the same properties of GSFε, expressed using the small-step reduction relation, due to the fact that the translation $\rightsquigarrow$ preserves typing. In particular, GSF terms do not get stuck, although they might produce **error** or diverge:

Proposition 8.4 (Type Safety). *If* $\vdash t : G$ *then either* $t \Downarrow \Xi \triangleright v$ *with* $\Xi \triangleright v : G$, $t \Downarrow$ **error**, *or* $t \Uparrow$.

Proposition 6.10 established that GSF typing coincides with SF typing on static terms. A similar result holds considering the dynamic semantics. In particular, static GSF terms never produce **error**:

Proposition 8.5 (Static Terms do not Fail). *Let* t *be a static term. If* $\vdash t : T$ *then* $\neg(t \Downarrow$ **error**$)$.

This result follows from the fact that all evidences in a static program are static, hence never gain precision; the initial type checking ensures that combination through transitivity never fails. This result can be found in the companion technical report.

9 GSF AND THE DYNAMIC GRADUAL GUARANTEE

The previous section clarified several aspects of the semantics of GSF programs, by establishing type safety, and by showing that static terms do not fail. This section studies the **dynamic gradual guarantee (DGG)** [Siek et al. 2015a], also known as graduality [New and Ahmed 2018; New et al. 2020]. In a big-step setting, this guarantee essentially says that if $\vdash t : G$ and $t \Downarrow v$, then for any t' such that $t \sqsubseteq t'$, we have $t' \Downarrow v'$ for some v' such that $v \sqsubseteq v'$. Intuitively: losing precision is harmless, or, reducibility is monotonic with respect to precision.

Unfortunately, in order to enforce parametricity (Section 10), and as already alluded to earlier (Section 8.2), GSF does not satisfy the DGG. First, we exhibit a counterexample, and identify the non-monotonicity of consistent transitivity as the root cause for this behavior. Then, in order to better understand the behavior of GSF programs when losing precision, we study a weaker variant of the DGG—weaker in the sense that it is valid for a stricter notion of precision—first in GSFε (Section 9.3) and then in GSF. The idea of devising a stricter notion of precision for which a variant of the DGG can be satisfied was first explored by Igarashi et al. [2017a], even though they leave the proof of such a result for System F_G as a conjecture; here, we formally prove that GSF satisfies the DGG with respect to the strict notion of precision.

9.1 Violation of the Dynamic Gradual Guarantee in GSF

To show that GSF does not satisfy the dynamic gradual guarantee (DGG), it is sufficient to exhibit two terms in GSF, related by precision, whose behavior contradicts the DGG. Consider the polymorphic identity function $id_X \triangleq \Lambda X.\lambda x : X.x :: X$, and an imprecise variant $id_? \triangleq \Lambda X.\lambda x : ?.x :: X$. Then id_X [Int] $1 \Downarrow 1$, but $id_?$ [Int] $1 \Downarrow$ **error**, despite the fact that id_X [Int] $1 \sqsubseteq id_?$ [Int] 1.

Conceptually, it is interesting to shed light on what causes such a violation. Recall that Garcia et al. [2016] prove the DGG for their language using (mostly) the monotonicity of consistent transitivity (Lemma 8.2 (c)) with respect to imprecision. In fact, while not sufficient, we can prove that monotonicity of ***consistent transitivity*** **(CT)** is a *necessary* condition for the DGG to hold. Intuitively, since two successive ascriptions are collapsed via consistent transitivity, a violation of monotonicity for consistent transitivity immediately implies a violation of monotonicity for reduction, and hence a violation of the DGG. For instance, if $t = \varepsilon_2(\varepsilon_1 u :: G_1) :: G_2$, $t' = \varepsilon_2'(\varepsilon_1' u' :: G_1') :: G_2'$, $t \sqsubseteq t'$ and $t \longmapsto (\varepsilon_1 \fatsemi \varepsilon_2)u :: G_2$, then by the DGG, $t' \longmapsto (\varepsilon_1' \fatsemi \varepsilon_2')u :: G_2'$, and thus $\varepsilon_1' \fatsemi \varepsilon_2'$ should be defined.

Proposition 9.1 ($\neg$ Monotonicity of CT $\Rightarrow$ $\neg$ DGG). *Let* $\varepsilon_1 \sqsubseteq \varepsilon_1'$, $\varepsilon_2 \sqsubseteq \varepsilon_2'$, $\varepsilon_1 \Vdash G_1 \sim G_2$, $\varepsilon_2 \Vdash G_2 \sim G_3$, $\varepsilon_1' \Vdash G_1' \sim G_2'$, $\varepsilon_2' \Vdash G_2' \sim G_3'$, *where* $G_i \sqsubseteq G_i'$.

If $\varepsilon_1 \fatsemi \varepsilon_2 \not\sqsubseteq \varepsilon_1' \fatsemi \varepsilon_2'$, *then* $\exists t \sqsubseteq t'$, *such that* $t \longmapsto v$, $t' \longmapsto v'$ *such that* $v \not\sqsubseteq v'$.

Proof. Let $t \triangleq \varepsilon_2(\varepsilon_1 u :: G_2) :: G_3$, and $t' \triangleq \varepsilon'_2(\varepsilon'_1 u' :: G'_2) :: G'_3$, for some $u \sqsubseteq u'$. We know $t \sqsubseteq t'$. Let $\varepsilon_1 \fatsemi \varepsilon_2 = \varepsilon_{12}$ and $\varepsilon'_1 \fatsemi \varepsilon'_2 = \varepsilon'_{12}$, then $t \longmapsto \varepsilon_{12} u :: G_3$ and $t' \longmapsto \varepsilon'_{12} u :: G'_3$, but as $\varepsilon_{12} \not\sqsubseteq \varepsilon'_{12}$ then $\varepsilon_{12} u :: G_3 \not\sqsubseteq \varepsilon'_{12} u' :: G'_3$ and the result holds. □

Garcia et al. [2016] study a language without universal types. But in GSF, because of universal types, there is an additional monotonicity condition that is necessary for the DGG to hold: monotonicity of ***evidence instantiation*** **(EI)**. Monotonicity of EI states that given two type abstractions related by precision, the new evidences created after type application remain related. Intuitively, since type application uses evidence instantiation, a violation of monotonicity for the latter implies a violation of monotonicity for the former, and hence a violation of the DGG. Formally:

Proposition 9.2 (¬ Monotonicity of EI ⇒ ¬ DGG). *Let $\varepsilon_1 \sqsubseteq \varepsilon_2$, $G_1 \sqsubseteq G_2$, $\Xi_1 \sqsubseteq \Xi_2$, $\alpha := G_1 \in \Xi_1$, $\alpha := G_2 \in \Xi_2$, $\hat{\alpha}_1 = lift_{\Xi_1}(\alpha)$, $\hat{\alpha}_2 = lift_{\Xi_2}(\alpha)$, and $\varepsilon_1[\hat{\alpha}_1]$ is defined.*
If $\varepsilon_1[\hat{\alpha}_1] \not\sqsubseteq \varepsilon_2[\hat{\alpha}_2]$, or $\varepsilon_{1out} \not\sqsubseteq \varepsilon_{2out}$, then $\exists t \sqsubseteq t'$, such that $t \longmapsto v$, $t' \longmapsto v'$ such that $v \not\sqsubseteq v'$.

Proof. Let $t \triangleq (\varepsilon_1(\Lambda X.t_1) :: \forall X.G'_1)\ [G_1]$, and $t' \triangleq (\varepsilon_2(\Lambda X.t_2) :: \forall X.G'_2)\ [G_2]$, for some $t_1 \sqsubseteq t_2$ and $\forall X.G'_1 \sqsubseteq \forall X.G'_2$. We know $t \sqsubseteq t'$. Also, we know that $\Xi_1 \triangleright t \longmapsto \Xi_1, \alpha := G_1 \triangleright \varepsilon_{1out}(\varepsilon_1[\hat{\alpha}_1]t'_1 :: G'_1[\alpha/X]) : G'_1[G_1/X]$ and $\Xi_2 \triangleright t' \longmapsto \Xi_2, \alpha := G_2 \triangleright \varepsilon_{2out}(\varepsilon_2[\hat{\alpha}_2]t'_2 :: G'_2[\alpha/X]) : G'_2[G_2/X]$, but as either $\varepsilon_{1out} \not\sqsubseteq \varepsilon_{2out}$ or $\varepsilon_1[\hat{\alpha}_1] \not\sqsubseteq \varepsilon_2[\hat{\alpha}_2]$, then $\varepsilon_{1out}(\varepsilon_1[\hat{\alpha}_1]t'_1 :: G'_1[\alpha/X]) : G'_1[G_1/X] \not\sqsubseteq \varepsilon_{2out}(\varepsilon_2[\hat{\alpha}_2]t'_2 :: G'_2[\alpha/X]) : G'_2[G_2/X]$, and the result holds. □

As mentioned in Section 8.2, monotonicity of consistent transitivity is broken by the strengthening we impose to enforce parametricity. For instance, consider $\langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle \sqsubseteq \langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle$ and $\langle \alpha^{\mathsf{Int}}, \mathsf{Int} \rangle \sqsubseteq \langle ?, ? \rangle$. By consistent transitivity, $\langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle \fatsemi \langle \alpha^{\mathsf{Int}}, \mathsf{Int} \rangle = \langle \mathsf{Int}, \mathsf{Int} \rangle$ (rule unsl), and $\langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle \fatsemi \langle ?, ? \rangle = \langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle$ (rule idL), but $\langle \mathsf{Int}, \mathsf{Int} \rangle \not\sqsubseteq \langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle$. Therefore, the DGG cannot be satisfied as such. We later on discuss a tension between our notion of parametricity and the DGG (Section 10), but first, we look at how to characterize the set of terms for which loss of precision is indeed harmless in GSF.

9.2 Towards a Weak Dynamic Gradual Guarantee for GSF

One way to accommodate the dynamic gradual guarantee in languages like λB, GSF, and System F_G, would be to change the definition of type (and term) precision. This is the approach taken by Igarashi et al. [2017a], although they do not prove that the DGG holds with this adjusted precision, and leave it as a conjecture. Dually, if one sticks to the natural notion of precision, as adopted by both GSF and CSA, and justified by the AGT interpretation, reconciliation might come from considering other forms of parametricity, or perhaps less flexible gradual language designs [Devriese et al. 2018]. Here, inspired by the approach of Igarashi et al. [2017a], we devise an alternative notion of precision for which the DGG does hold. We call this relation *strict precision* as it relates fewer terms than the natural notion of precision. Conversely to Igarashi et al. [2017a], however, we do not intend strict precision to be the one used to typecheck programs, but only to serve as a technical device to characterize harmless losses of precision in GSF.

External vs internal losses of precision. First of all, it is important to observe that the violation of the DGG from the previous section is due to the interaction between polymorphic types and imprecision, which affects runtime sealing with type names. A first consequence of this observation is that the simply-typed subset of GSF should enjoy the DGG with respect to the standard notion of precision. Said differently, strict precision ought to coincide with natural precision on simply-typed terms. A second consequence is that excluding *any* loss of precision related to type variables

would be a sound approximation to characterize when the DGG holds; this corresponds exactly to the precision relation of System F_G [Igarashi et al. 2017a]. However, while this approach would work for GSF as well, it appears too strict, in that it excludes losses of precision on polymorphic types which are harmless in GSF.

Intuitively, we observe that in GSF, losing precision *internally* (i.e., by modifying the types of binders) has a different impact on reducibility, compared to losing precision *externally* (i.e., through imprecise type ascriptions). Specifically, external loss of precision is harmless in GSF when the ascribed term is closed with respect to type variables. Therefore any fully-static polymorphic function that is imprecisely ascribed and used adequately (type-wise) in a gradual context will behave as expected.[13] In practice, this means that in GSF, the fully precise polymorphic identity function $id_X \triangleq \Lambda X.\lambda x : X.x :: X$ and an imprecisely-ascribed variant such as $id_{X?} \triangleq id_X :: \forall X.? \to X$ have the same behavior—in particular, one can apply $id_{X?}$ to any given type and argument of that type and successfully obtain back that argument as result. In contrast, as we have seen, the internally-imprecise function $id_? \triangleq \Lambda X.\lambda x : ?.x :: X$ fails when applied, because the argument value is not sealed on entry, and hence the unsealing on exit is invalid.

Admittedly, this difference in behavior between internal and external losses of precision might come as a surprise to programmers, but it is the result of type-driven sealing. When applying one of the functions above, the operational semantics must decide whether or not the value bound to x ought to be sealed. If the type of x is *known* to be X, as in id_X and $id_{X?}$, it is clear that the value should be sealed (with the runtime type name corresponding to X). However, for $id_?$, the type of x is ?, so there are two options: not sealing because ? might stand for other types that X, or sealing because ? might stand for X. Always sealing presents two issues. First, if the function was $\Lambda X.\lambda x : ?.x + 1$, then the addition would fail at runtime, and we would have another counterexample of the DGG (because it is less precise than $\Lambda X.\lambda x : \mathsf{Int}.x + 1$). More importantly, we could not know with respect to *which variable* one ought to seal. Indeed, consider a slightly more complex function: $\Lambda X.\Lambda Y.\lambda z : ?.t$. Here, always sealing would require deciding whether to seal with (the runtime names of) X or Y.

This conundrum arises because runtime sealing is type driven, and types can be imprecise. When faced with an imprecise term binder under a type binder, either options of sealing or not sealing would expose a failure of the DGG. Optimistically, not sealing has the advantage of avoiding the ambiguity of which type names to seal with, while still supporting harmless losses of precision externally for polymorphic values. Note that a language design such as PolyG$^\nu$ in which sealing and unsealing are independent of the precision of type information can sidestep the problem, by leaving the task of sealing with explicit terms to programmers (Section 2).

Characterizing strict term precision for GSF. Strict term precision should coincide with natural precision on simply-typed terms, but how should it behave on the polymorphic fragment of GSF? We now provide some intuitive characterization of strict term precision, denoted $\leqslant$, based on the analysis above with the three terms id_X, $id_{X?}$, and $id_?$.

(A) $id_X \not\leqslant id_?$

$id_?$ presents an internal loss of precision compared to id_X, because the term binder changes from type X to type ?, and fails at runtime when applied.

(B) $id_X :: \forall X.X \to X \leqslant id_{X?}$

$id_{X?}$ presents an external loss of precision compared to $id_X :: \forall X.X \to X$, and does not fail when applied.

[13] Our prior work on GSF [Toro et al. 2019] formalizes exactly this result. Here, we go further and establish more general results, from which this sort of preservation of behavior for ascribed static terms easily follows (Section 10.4).

$\boxed{G \leqslant G}$ **Strict type precision**

$$\frac{}{B \leqslant B} \qquad \frac{}{X \leqslant X} \qquad \frac{}{\alpha \leqslant \alpha} \qquad \frac{}{B \leqslant ?} \qquad \frac{G_1 \to G_2 \leqslant ? \to ?}{G_1 \to G_2 \leqslant ?} \qquad \frac{}{? \leqslant ?}$$

$$\frac{G_1 \leqslant G_3 \quad G_2 \leqslant G_4}{G_1 \to G_2 \leqslant G_3 \to G_4} \qquad \frac{G_1 \leqslant G_3 \quad G_2 \leqslant G_4}{G_1 \times G_2 \leqslant G_3 \times G_4} \qquad \frac{G_1 \leqslant G_2}{\forall X.G_1 \leqslant \forall X.G_2}$$

$\boxed{\varepsilon \leqslant \varepsilon}$ **Strict evidence precision**

$$\frac{E_1 \leqslant E_3 \quad E_2 \leqslant E_4}{\langle E_1, E_2 \rangle \leqslant \langle E_3, E_4 \rangle}$$

$\boxed{E \leqslant E}$ **Strict evidence type precision**

$$\frac{}{B \leqslant B} \qquad \frac{}{X \leqslant X} \qquad \frac{E_1 \leqslant E_2}{\alpha^{E_1} \leqslant \alpha^{E_2}} \qquad \frac{}{B \leqslant ?} \qquad \frac{E_1 \to E_2 \leqslant ? \to ?}{E_1 \to E_2 \leqslant ?} \qquad \frac{}{? \leqslant ?}$$

$$\frac{E_1 \leqslant E_3 \quad E_2 \leqslant E_4}{E_1 \to E_2 \leqslant E_3 \to E_4} \qquad \frac{E_1 \leqslant E_3 \quad E_2 \leqslant E_4}{E_1 \times E_2 \leqslant E_3 \times E_4} \qquad \frac{E_1 \leqslant E_2}{\forall X.E_1 \leqslant \forall X.E_2}$$

Fig. 10. GSF: Strict precision.

(C) $id_X :: \forall X.X \to X \leqslant id_? :: \forall X.X \to X$

Although $id_?$ presents an internal loss of precision compared to id_X, the ascription to $\forall X.X \to X$ on $id_?$ imposes this parametricity contract, and hence the resulting term does behave like a proper (static) identity function.

The rest of this section builds upon this informal analysis in order to fully define strict precision and establish the corresponding dynamic gradual guarantee. Of course, because strict precision $\leqslant$ is more restrictive than standard precision $\sqsubseteq$, the dynamic gradual guarantee that one may establish with respect to it is *weaker*; hereafter, we denote it DGG$^{\leqslant}$. The dynamic gradual guarantee appeals to term reduction, so in Section 9.3 we start by defining strict precision for GSFε, prove DGG$^{\leqslant}$ for GSFε, and conclude by establishing DGG$^{\leqslant}$ for GSF. Finally, in Section 9.4 we provide a characterization of $\leqslant$ directly on GSF syntax, i.e., without appealing to elaboration, for which DGG$^{\leqslant}$ holds.

9.3 Weak Dynamic Gradual Guarantee for GSF

Armed with the intuition presented above, we define a strict notion of precision for GSFε, which closely characterizes GSFε terms for which monotonicity of consistent transitivity holds. While not sufficient, monotonicity of consistent transitivity is necessary for the DGG to hold, as established in Proposition 9.1.

Strict precision for gradual types. Strict precision for types (Figure 10) avoids any interference between runtime sealing and loss of precision. As expected, $\leqslant$ coincides with $\sqsubseteq$ except for universal types, type variables and type names: these are not more precise than the unknown type anymore. For instance, $\forall X.X \to X \not\leqslant \forall X.X \to ? \not\leqslant ?$. We say G_1 is "more strictly precise" than G_2 when $G_1 \leqslant G_2$.

Strict precision for evidence. Strict type precision can be naturally lifted to define strict precision for evidence and evidence types (Figure 10). A type name is more strictly precise than another if it is bound to a more strictly precise evidence type. Crucially, monotonicity of consistent transitivity holds with respect to $\leqslant$.

Proposition 9.3 ($\leqslant$-Monotonicity of Consistent Transitivity). *If* $\varepsilon_1 \leqslant \varepsilon_2$, $\varepsilon_3 \leqslant \varepsilon_4$, *and* $\varepsilon_1 \fatsemi \varepsilon_3$ *is defined, then* $\varepsilon_1 \fatsemi \varepsilon_3 \leqslant \varepsilon_2 \fatsemi \varepsilon_4$.

$\boxed{\Omega \vdash \Xi \triangleright s : G \leqslant \Xi \triangleright s : G}$ **Strict term precision** (for conciseness, s ranges over both t and u)

$$(\leqslant b_\varepsilon)\frac{ty(b) = B \qquad \Xi_1 \leqslant \Xi_2}{\Omega \vdash \Xi_1 \triangleright b : B \leqslant \Xi_2 \triangleright b : B}$$

$$(\leqslant \lambda_\varepsilon)\frac{\Omega, x : G_1 \sqsubseteq G_2 \vdash \Xi_1 \triangleright t_1 : G_1' \leqslant \Xi_2 \triangleright t_2 : G_2' \qquad G_1 \sqsubseteq G_2}{\Omega \vdash \Xi_1 \triangleright \lambda x : G_1.t_1 : G_1 \to G_1' \leqslant \Xi_2 \triangleright \lambda x : G_2.t_2 : G_2 \to G_2'}$$

$$(\leqslant \times_\varepsilon)\frac{\Omega \vdash \Xi_1 \triangleright s_1 : G_1 \leqslant \Xi_2 \triangleright s_2 : G_2 \qquad \Omega \vdash \Xi_1 \triangleright s_1' : G_1' \leqslant \Xi_2 \triangleright s_2' : G_2'}{\Omega \vdash \Xi_1 \triangleright \langle s_1, s_1' \rangle : G_1 \times G_1' \leqslant \Xi_2 \triangleright \langle s_2, s_2' \rangle : G_2 \times G_2'}$$

$$(\leqslant \Lambda_\varepsilon)\frac{\Omega \vdash \Xi_1 \triangleright t_1 : G_1 \leqslant \Xi_2 \triangleright t_2 : G_2}{\Omega \vdash \Xi_1 \triangleright \Lambda X.t_1 : \forall X.G_1 \leqslant \Xi_2 \triangleright \Lambda X.t_2 : \forall X.G_2} \qquad (\leqslant \mathrm{x}_\varepsilon)\frac{x : G_1 \sqsubseteq G_2 \in \Omega \qquad \Xi_1 \leqslant \Xi_2}{\Omega \vdash \Xi_1 \triangleright x : G_1 \leqslant \Xi_2 \triangleright x : G_2}$$

$$(\leqslant \mathrm{op}_\varepsilon)\frac{\Omega \vdash \Xi_1 \triangleright t_1 : \overline{G} \leqslant \Xi_2 \triangleright t_2 : \overline{G} \qquad ty(op) = \overline{G} \to G'}{\Omega \vdash \Xi_1 \triangleright op(\overline{t_1}) : G' \leqslant \Xi_2 \triangleright op(\overline{t_2}) : G'}$$

$$(\leqslant \mathrm{app}_\varepsilon)\frac{\Omega \vdash \Xi_1 \triangleright t_1 : G_1' \to G_1 \leqslant \Xi_2 \triangleright t_2 : G_2' \to G_2 \qquad \Omega \vdash \Xi_1 \triangleright t_1' : G_1' \leqslant \Xi_2 \triangleright t_2' : G_2'}{\Omega \vdash \Xi_1 \triangleright t_1\ t_1' : G_1 \leqslant \Xi_2 \triangleright t_2\ t_2' : G_2}$$

$$(\leqslant \mathrm{appG}_\varepsilon)\frac{\Omega \vdash \Xi_1 \triangleright t_1 : \forall X.G_1 \leqslant \Xi_2 \triangleright t_2 : \forall X.G_2 \qquad G_1' \leqslant G_2'}{\Omega \vdash \Xi_1 \triangleright t_1\ [G_1'] : G_1[G_1'/X] \leqslant \Xi_2 \triangleright t_2\ [G_2'] : G_2[G_2'/X]}$$

$$(\leqslant \mathrm{pair}i_\varepsilon)\frac{\Omega \vdash \Xi_1 \triangleright t_1 : G_1 \times G_2 \leqslant \Xi_2 \triangleright t_2 : G_1' \times G_2'}{\Omega \vdash \Xi_1 \triangleright \pi_i(t_1) : G_i \leqslant \Xi_2 \triangleright \pi_i(t_2) : G_i'}$$

$$(\leqslant \mathrm{asc}_\varepsilon)\frac{\varepsilon_1 \leqslant \varepsilon_2 \qquad \Omega \vdash \Xi_1 \triangleright s_1 : G_1' \leqslant \Xi_2 \triangleright s_2 : G_2' \qquad G_1 \sqsubseteq G_2}{\Omega \vdash \Xi_1 \triangleright \varepsilon_1 s_1 :: G_1 : G_1 \leqslant \Xi_2 \triangleright \varepsilon_2 s_2 :: G_2 : G_2}$$

$$(\leqslant \mathrm{Masc}_\varepsilon)\frac{\begin{array}{c}\varepsilon_1 \sqsubseteq \varepsilon_2 \qquad \Omega \vdash \Xi_1 \triangleright t_1 : G_1' \leqslant \Xi_2 \triangleright t_2 : G_2' \qquad G_1 \sqsubseteq G_2 \\ \varepsilon_1 = \mathcal{I}_{\Xi_1}(G_1, G_1) \qquad \varepsilon_2 = \mathcal{I}_{\Xi_2}(G_2, G_2) \qquad G_1' \nrightarrow G_1 \qquad G_2' \nrightarrow G_2\end{array}}{\Omega \vdash \Xi_1 \triangleright \varepsilon_1 t_1 :: G_1 : G_1 \leqslant \Xi_2 \triangleright \varepsilon_2 t_2 :: G_2 : G_2}$$

$\boxed{\Xi \vdash t \leqslant \Xi \vdash t}$ **Configuration precision**

$$\frac{\Xi_1 \leqslant \Xi_2 \qquad \cdot \vdash \Xi_1 \triangleright t_1 : G_1 \leqslant \Xi_2 \triangleright t_2 : G_2 \qquad \Xi_1 \vdash t_1 : G_1 \qquad \Xi_2 \vdash t_2 : G_2}{\Xi_1 \triangleright t_1 \leqslant \Xi_2 \triangleright t_2}$$

$$\frac{\forall \alpha \in dom(\Xi_1).\ \Xi_1(\alpha) \leqslant \Xi_2(\alpha)}{\Xi_1 \leqslant \Xi_2}$$

Fig. 11. GSFε: Strict precision.

For illustration purposes, let us recall the counterexample to monotonicity presented in Section 9.1. Consider $\langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle \sqsubseteq \langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle$ and $\langle \alpha^{\mathsf{Int}}, \mathsf{Int} \rangle \sqsubseteq \langle ?, ? \rangle$. By consistent transitivity, $\langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle \fatsemi \langle \alpha^{\mathsf{Int}}, \mathsf{Int} \rangle = \langle \mathsf{Int}, \mathsf{Int} \rangle$ (rule unsl), and $\langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle \fatsemi \langle ?, ? \rangle = \langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle$ (rule idL), but $\langle \mathsf{Int}, \mathsf{Int} \rangle \not\sqsubseteq \langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle$. This argument is no longer valid with strict precision, as $\alpha^{\mathsf{Int}} \not\leqslant ?$ and therefore $\langle \alpha^{\mathsf{Int}}, \mathsf{Int} \rangle \not\leqslant \langle ?, ? \rangle$.

Strict precision for GSFε terms. Strict precision for GSFε terms relates two possibly-open terms and their respective types (Figure 11). It is worth noting that types can be related by $\leqslant$ or

$\sqsubseteq$ depending on the rule. We use $\sqsubseteq$ to relate every type annotation (save for type instantiations), and $\leqslant$ for (almost) every other relation. Note that we could have obtained a simpler definition by using $\leqslant$ everywhere, but the relation would be overly conservative, for instance rejecting example (C) among many others. Our goal is to design a term precision relation as permissive as possible (close to the natural term precision relation) such that the DGG is satisfied. The precision judgment $\Omega \vdash \Xi_1 \triangleright s_1 : G_1 \leqslant \Xi_2 \triangleright s_2 : G_2$ denotes that term s_1 of type G_1 is more strictly precise than s_2 of type G_2, under store Ξ_1 strictly more precise than Ξ_2, and precision relation environment Ω. It is important to clarify that the judgment $\Omega \vdash \Xi_1 \triangleright s_1 : G_1 \leqslant \Xi_2 \triangleright s_2 : G_2$ does not necessarily imply that $G_1 \leqslant G_2$. In fact, the types G_1 and G_2 are related by the more general relation $\sqsubseteq$. Nevertheless, evidences are the ones that play a crucial role in the term precision relation, which are related in almost every case by $\leqslant$. Ω binds a term variable to a pair of types related by precision $\sqsubseteq$. The intuition about why we use $\sqsubseteq$ and not $\leqslant$ in Ω is that as long as evidence are related by $\leqslant$, we can relax this relation on type annotations. Furthermore, using $\leqslant$ on Ω makes the precision relation on terms overly conservative rejecting example (C) above. Rule ($\leqslant \mathrm{x}_\varepsilon$) establishes $\Omega \vdash \Xi_1 \triangleright x : G_1 \leqslant \Xi_2 \triangleright x : G_2$ if $x : G_1 \sqsubseteq G_2 \in \Omega$, and Rule ($\leqslant \lambda_\varepsilon$) extends Ω with the annotated types of the functions to relate.

Strict term precision is the natural lifting of strict type precision $\leqslant$ to terms, except for types that do not influence evidence in the runtime semantics, namely function argument types and ascription types: for these, we can use the more liberal type precision relation $\sqsubseteq$. Note that these types are used to elaborate evidence, but at runtime once evidence is elaborated, they are no longer relevant, unlike instantiation types which propagate to evidence upon application. For example, Rule ($\leqslant \mathrm{asc}_\varepsilon$) has the premise $G_1 \sqsubseteq G_2$. If we imposed a strict precision relation between ascribed types, then example (B) would not be satisfied as $\forall X.X \to X \not\leqslant \forall X.? \to X$. By ($\leqslant \mathrm{asc}_\varepsilon$) we know that for the elaborated terms $\varepsilon_{\forall X.X \to X}\, id_X :: \forall X.X \to X \leqslant \varepsilon_{\forall X.X \to X}\, id_X :: \forall X.? \to X$ because evidences are the same (and thus related by $\leqslant$), whereas the type annotations are related by $\sqsubseteq$. Furthermore, example (A) is satisfied as the elaborated terms are not related by strict precision because $\varepsilon_{\forall X.X \to X}$ is not related to $\varepsilon_{\forall X.? \to X}$.

Rule ($\leqslant \mathrm{appG}_\varepsilon$) states that types involved in a type application must be related by strict precision because they do influence evidence during reduction: after elimination of type abstractions, new evidences are created using these types, and such evidences need to be related as well. Note that this restriction is sufficient to satisfy monotonicity of evidence instantiation, which is needed for the dynamic gradual guarantee (Proposition 9.2).

Finally, we need to strengthen the relation with an additional rule ($\leqslant \mathrm{Masc}_\varepsilon$) to account for GSF$\varepsilon$ terms that are the result of the elaboration from GSF. This will be important to scale the DGG$^{\leqslant}$ from GSFε to GSF below. Recall that the translation from GSF to GSFε introduces evidences to ensure that GSFε terms are well-typed (Figure 5). In particular, the translation uses type matching $\rightarrowtriangle$ to ascribe subterms of type ? in elimination positions to the corresponding top-level type constructor. When an actual matching expansion occurs, the corresponding evidence is generated such as $\varepsilon_{\forall X.?} = \mathcal{I}_\Xi(\forall X.?, \forall X.?)$, or $\varepsilon_{?\to?} = \mathcal{I}_\Xi(? \to ?, ? \to ?)$. Such evidences are related by $\sqsubseteq$, but not necessarily by $\leqslant$. Rule ($\leqslant \mathrm{Masc}_\varepsilon$) accounts for the case where they are not. Note that evidences ε_1 and ε_2 do not contribute to any increase in precision: when combined with some arbitrary evidence ε during reduction, the combination $\varepsilon \fatsemi \varepsilon_i$ either fails or results in ε. Rule ($\leqslant \mathrm{Masc}_\varepsilon$) is key to satisfy example (B). To see why, consider terms $(id_X :: \forall X.X \to X)$ [Int] and $(id_X :: \forall X.? \to X)$ [Int] and their elaborations:

$$\text{(GappG)}\ \frac{\text{(Gascu)}\ \dfrac{id_X \rightsquigarrow id_X' \qquad \varepsilon_1 = \mathcal{I}(\forall X.X \to X, \forall X.X \to X)}{id_X :: \forall X.X \to X \rightsquigarrow \varepsilon_1\, id_X :: \forall X.X \to X} \qquad \varepsilon_2 = \mathcal{I}(\forall X.X \to X, \forall X.X \to X)}{(id_X :: \forall X.X \to X)\ [\mathsf{Int}] \rightsquigarrow \varepsilon_2(\varepsilon_1\, id_X' :: \forall X.X \to X) :: \forall X.X \to X\ [\mathsf{Int}]}$$

$$\text{(GappG)}\ \frac{\text{(Gascu)}\ \dfrac{id_X \rightsquigarrow id_X' \qquad \varepsilon_1' = \mathcal{I}(\forall X.X \to X, \forall X.? \to X)}{id_X :: \forall X.? \to X \rightsquigarrow \varepsilon_1' id_X :: \forall X.? \to X} \qquad \varepsilon_2' = \mathcal{I}(\forall X.? \to X, \forall X.? \to X)}{(id_X :: \forall X.? \to X)\ [\mathsf{Int}] \rightsquigarrow \varepsilon_2'(\varepsilon_1' id_X' :: \forall X.? \to X) :: \forall X.? \to X\ [\mathsf{Int}]}$$

Note that $\varepsilon_1 = \varepsilon_2 = \varepsilon_1' = \varepsilon_{\forall X.X \to X} = \langle \forall X.X \to X, \forall X.X \to X \rangle$, $\varepsilon_2' = \langle \forall X.? \to X, \forall X.? \to X \rangle$, and $\varepsilon_2 \sqsubseteq \varepsilon_2'$ but $\varepsilon_2 \not\leqslant \varepsilon_2'$.After one step of execution, both pairs of evidences are combined ($\varepsilon_1 \fatsemi \varepsilon_1$ and $\varepsilon_1' \fatsemi \varepsilon_2'$), resulting in both cases in $\varepsilon_{\forall X.X \to X}$. Therefore, these two programs behave identically and are thus related.

Strict precision for configurations. Figure 11 also defines strict type precision for GSFε stores and configurations. A store is more strictly precise than another if it binds each type name to a more strictly precise type. Finally, a configuration is more strictly precise than another if the store and term components are more strictly precise, and the terms are well-typed with their respective stores.

DGG$^{\leqslant}$ for GSFε. Armed with strict precision for GSFε, and the fact that consistent transitivity is monotone with respect to it (Proposition 9.3), we can prove the weak dynamic gradual guarantee DGG$^{\leqslant}$ for GSFε. Given two configurations related by strict precision, small-step reduction ($\longmapsto$) of the most precise one implies that of the less precise one. Alternatively, if the first configuration is already a value, then so is the second.

Proposition 9.4 (Small-step DGG$^{\leqslant}$ for GSFε). *Suppose* $\Xi_1 \triangleright t_1 \leqslant \Xi_2 \triangleright t_2$.

(a) If $\Xi_1 \triangleright t_1 \longmapsto \Xi_1' \triangleright t_1'$, *then* $\Xi_2 \triangleright t_2 \longmapsto \Xi_2' \triangleright t_2'$, *for some* Ξ_2' *and* t_2' *such that* $\Xi_1' \triangleright t_1' \leqslant \Xi_2' \triangleright t_2'$.
(b) If $t_1 = v_1$, *then* $t_2 = v_2$.

DGG$^{\leqslant}$ for GSF. Using Proposition 9.4 we can establish DGG$^{\leqslant}$ for GSF, considering that two GSF terms are related by strict precision iff their elaboration to GSFε are.

Theorem 9.5 (DGG$^{\leqslant}$). *Suppose* $t_1 \leqslant t_2$, $\vdash t_1 : G_1$, *and* $\vdash t_2 : G_2$.

(a) If $t_1 \Downarrow \Xi_1 \triangleright v_1$, *then* $t_2 \Downarrow \Xi_2 \triangleright v_2$, $\cdot \vdash \Xi_1 \triangleright v_1 : G_1 \leqslant \Xi_2 \triangleright v_2 : G_2$ *and* $\Xi_1 \leqslant \Xi_2$, *for some* v_2 *and* Ξ_2.
If $t_1 \Uparrow$ *then* $t_2 \Uparrow$.
(b) If $t_2 \Downarrow \Xi_2 \triangleright v_2$, *then* $t_1 \Downarrow \Xi_1 \triangleright v_1$, $\cdot \vdash \Xi_1 \triangleright v_1 : G_1 \leqslant \Xi_2 \triangleright v_2 : G_2$ *and* $\Xi_1 \leqslant \Xi_2$, *for some* v_1 *and* Ξ_1, *or* $t_1 \Downarrow$ **error**.
If $t_2 \Uparrow$, *then* $t_1 \Uparrow$ *or* $t_1 \Downarrow$ **error**.

Harmless imprecise ascriptions. Finally, we can use the DGG$^{\leqslant}$ to establish that, given a term t of type G, ascribing to a less precise type G' and then back to type G, results in a term semantically equivalent to t:

Lemma 9.6. *Let* $\vdash t : G$, $G \sqsubseteq G'$, *and* $t' = t :: G' :: G$, *then*

- $t \Downarrow \Xi \triangleright v \iff t' \Downarrow \Xi \triangleright v$.
- $t \Downarrow$ **error** $\iff t' \Downarrow$ **error**.

Note in particular that if t produces a value, then t' produces the exact same value. While the above result characterizes an ascription roundtrip through imprecision and back, we can also establish harmlessness results for imprecise ascriptions. Given a term t that reduces to some value, ascribing it to a less precise type also results in a (less strictly precise) value.

Lemma 9.7. *Let* $\vdash t : G$ *such that* $t \Downarrow \Xi \triangleright v$, *and* $G \sqsubseteq G'$. *Then* $t :: G' \Downarrow \Xi \triangleright v'$ *such that* $\vdash \Xi \triangleright v : G \leqslant \Xi \triangleright v' : G'$, *for some* v'.

Likewise, we can characterize ascribing the subterms of elimination forms, such as function application and type application:

Lemma 9.8. *Let* $\vdash t_1 : G_1$ *and* $\vdash t_2 : G_2$ *such that* $\vdash t_1\ t_2 : G$ *and* $t_1\ t_2 \Downarrow \Xi \triangleright v$. *Let* $G_1 \sqsubseteq G_1'$, $G_2 \sqsubseteq G_2'$, *and* $G \sqsubseteq G'$, *such that* $\vdash (t_1 :: G_1')\ (t_2 :: G_2') : G'$. *Then* $(t_1 :: G_1')\ (t_2 :: G_2') \Downarrow \Xi \triangleright v'$ *such that* $\vdash \Xi \triangleright v : G \leqslant \Xi \triangleright v' : G'$, *for some* v'.

Lemma 9.9. *Let* $\vdash t : G_1$ *such that* $\vdash t\ [G_2] : G$ *and* $t\ [G_2] \Downarrow \Xi \triangleright v$. *Let* $G_1 \sqsubseteq G_1'$, $G_2 \leqslant G_2'$, *and* $G \sqsubseteq G'$, *such that* $\vdash (t :: G_1')\ [G_2'] : G'$. *Then* $(t :: G_1')\ [G_2'] \Downarrow \Xi' \triangleright v'$ *such that* $\vdash \Xi \triangleright v : G \leqslant \Xi' \triangleright v' : G'$ *and* $\Xi \leqslant \Xi'$, *for some* v' *and* Ξ'.

Interestingly, Lemma 9.9 is more general than in previous work [Toro et al. 2019], where the instantiated types were restricted to only static types. Similar lemmas can be defined for other eliminations forms, such as projections and n-ary operations.

These results, which embody the motto that external imprecision is harmless in GSF, constitute a valuable compositionality guarantee when embedding fully-static (System F) terms in a gradual world, as will be further illustrated in Section 10.4.

9.4 Syntactic Strict Precision for GSF

For now, strict precision for GSF terms has been defined by appealing to their elaboration to GSFε terms. Unfortunately, with this definition it would be hard for programmers to get an intuition about when two terms are related by strict precision, as it would require understanding the elaborations to GSFε. Here we design a strict precision relation $\leqslant$ for GSF terms syntactically, as a sound approximation of the elaboration-based definition. To define a syntactic strict precision relation for GSF, we start from the GSF to GSFε translation rules, and analyze when two terms yield related elaborations. Let us first look at two crucial cases: ascriptions and type applications.

Ascriptions. For a couple of ascriptions $t_1 :: G_1'$ and $t_2 :: G_2'$ we know that:

$$\text{(Gasct)}\ \frac{t_1 \neq v \qquad \Delta; \Gamma_1 \vdash t_1 \rightsquigarrow t_1' : G_1' \qquad \varepsilon_1 = \mathcal{I}(G_1', G_1)}{\Delta; \Gamma_1 \vdash t_1 :: G_1 \rightsquigarrow \varepsilon_1 t_1' :: G_1 : G_1} \qquad \text{(Gasct)}\ \frac{t_2 \neq v \qquad \Delta; \Gamma_2 \vdash t_2 \rightsquigarrow t_2' : G_2' \qquad \varepsilon_2 = \mathcal{I}(G_2', G_2)}{\Delta; \Gamma_2 \vdash t_2 :: G_2 \rightsquigarrow \varepsilon_2 t_2' :: G_2 : G_2}$$

If $\Omega \vdash \cdot \triangleright \varepsilon_1 t_1' :: G_1 : G_1 \leqslant \cdot \triangleright \varepsilon_2 t_2' :: G_2 : G_2$, then by ($\leqslant$asc$_\varepsilon$), it must be the case that $\varepsilon_1 \leqslant \varepsilon_2$, $\Omega \vdash \cdot \triangleright t_1' : G_1' \leqslant \cdot \triangleright t_2' : G_2'$ and $G_1 \sqsubseteq G_2$, where Ω is well-formed with respect to Γ_1 and Γ_2, i.e., $\Omega \vdash \Gamma_1 \sqsubseteq \Gamma_2$ (Figure 12). As $\mathcal{I}(G_1', G_1) = \langle G_1' \sqcap G_1, G_1' \sqcap G_1 \rangle$ and $\mathcal{I}(G_2', G_2) = \langle G_2' \sqcap G_2, G_2' \sqcap G_2 \rangle$, then we require that $G_1' \sqcap G_1 \leqslant G_2' \sqcap G_2$, which leads to the following strict precision rule for ascriptions on GSF:

$$\frac{\Omega \vdash t_1 : G_1' \leqslant t_2 : G_2' \qquad G_1' \sqcap G_1 \leqslant G_2' \sqcap G_2 \qquad G_1 \sqsubseteq G_2 \qquad t_1, t_2 \neq v}{\Omega \vdash t_1 :: G_1 : G_1 \leqslant t_2 :: G_2 : G_2}$$

Type applications. For a couple of type applications $t_1[G_1']$ and $t_2[G_2']$, we know from the elaboration rules that:

$$\text{(GappG)}\ \frac{\Delta; \Gamma_1 \vdash t_1 \rightsquigarrow t_1' : G_1 \qquad \Delta \vdash G_1' \qquad G_1 \twoheadrightarrow \forall X.G_1'' \qquad \varepsilon_1 = \mathcal{I}(G_1, \forall X.G_1'')}{\Delta; \Gamma_1 \vdash (\varepsilon_1 t_1 :: \forall X.G_1'')\ [G_1'] \rightsquigarrow t_1''\ [G_1'] : G_1''[G_1'/X]}$$

$$\text{(GappG)}\ \frac{\Delta; \Gamma_2 \vdash t_2 \rightsquigarrow t_2' : G_2 \qquad \Delta \vdash G_2' \qquad G_2 \twoheadrightarrow \forall X.G_2'' \qquad \varepsilon_2 = \mathcal{I}(G_2, \forall X.G_2'')}{\Delta; \Gamma_2 \vdash (\varepsilon_2 t_2 :: \forall X.G_2'')\ [G_2'] \rightsquigarrow t_2''\ [G_2'] : G_2''[G_2'/X]}$$

$\boxed{\Omega \vdash v : G \leqslant_v v : G}$ **Syntactic strict value precision**

$$(\leqslant b)\frac{ty(b) = B}{\Omega \vdash b : B \leqslant_v b : B} \qquad (\leqslant \lambda)\frac{\Omega, x : G_1 \sqsubseteq G_2 \vdash t_1 : G_1' \leqslant t_2 : G_2' \quad G_1 \sqsubseteq G_2}{\Omega \vdash (\lambda x : G_1.t_1) : G_1 \to G_1' \leqslant_v (\lambda x : G_2.t_2) : G_2 \to G_2'}$$

$$(\leqslant \lambda)\frac{\Omega, x : G_1 \sqsubseteq G_2 \vdash t_1 : G_1' \leqslant t_2 : G_2' \quad G_1 \sqsubseteq G_2}{\Omega \vdash (\lambda x : G_1.t_1) : G_1 \to G_1' \leqslant_v (\lambda x : G_2.t_2) : G_2 \to G_2'}$$

$$(\leqslant \times)\frac{\Omega \vdash v_1 : G_1 \leqslant v_2 : G_2 \quad \Omega \vdash v_1' : G_1' \leqslant v_2' : G_2'}{\Omega \vdash \langle v_1, v_1' \rangle : G_1 \times G_1' \leqslant_v \langle v_2, v_2' \rangle : G_2 \times G_2'}$$

$$(\leqslant \Lambda)\frac{\Omega \vdash t_1 : G_1 \leqslant t_2 : G_2}{\Omega \vdash (\Lambda X.t_1) : \forall X.G_1 \leqslant_v (\Lambda X.t_2) : \forall X.G_2}$$

$\boxed{\Omega \vdash t : G \leqslant t : G}$ **Syntactic strict term precision**

$$(\leqslant \text{x})\frac{x : G_1 \sqsubseteq G_2 \in \Omega}{\Omega \vdash x : G_1 \leqslant x : G_2} \qquad (\leqslant \text{v})\frac{\Omega \vdash v_1 : G_1 \leqslant_v v_2 : G_2 \quad G_1 \leqslant G_2}{\Omega \vdash v_1 : G_1 \leqslant v_2 : G_2}$$

$$(\leqslant \text{ascv})\frac{\Omega \vdash v_1 : G_1' \leqslant_v v_2 : G_2' \quad G_1' \sqcap G_1 \leqslant G_2' \sqcap G_2 \quad G_1 \sqsubseteq G_2}{\Omega \vdash v_1 :: G_1 : G_1 \leqslant v_2 :: G_2 : G_2}$$

$$(\leqslant \text{asct})\frac{\Omega \vdash t_1 : G_1' \leqslant t_2 : G_2' \quad G_1' \sqcap G_1 \leqslant G_2' \sqcap G_2 \quad G_1 \sqsubseteq G_2 \quad t_1, t_2 \neq v}{\Omega \vdash t_1 :: G_1 : G_1 \leqslant t_2 :: G_2 : G_2}$$

$$(\leqslant \text{op})\frac{\Omega \vdash t_1 : \overline{G_1} \leqslant t_2 : \overline{G_2} \quad ty(op) = \overline{G} \to G' \quad \overline{G} \sqcap \overline{G_1} \leqslant \overline{G} \sqcap \overline{G_2}}{\Omega \vdash op(\overline{t_1}) : G_1' \leqslant op(\overline{t_2}) : G_2'}$$

$$(\leqslant \text{app})\frac{\begin{array}{c}\Omega \vdash t_1 : G_1 \leqslant t_2 : G_2 \qquad \Omega \vdash t_1' : G_1' \leqslant t_2' : G_2' \\ G_1 \rightarrowtriangle G_{11} \to G_{12} \quad G_2 \rightarrowtriangle G_{21} \to G_{22} \quad G_1' \sqcap G_{11} \leqslant G_2' \sqcap G_{21}\end{array}}{\Omega \vdash t_1\ t_1' : G_{12} \leqslant t_2\ t_2' : G_{22}}$$

$$(\leqslant \text{pairt})\frac{(t_1 \neq v_1 \vee t_2 \neq v_2) \quad \Omega \vdash t_1 : G_1 \leqslant t_2 : G_2 \quad \Omega \vdash t_1' : G_1' \leqslant t_2' : G_2'}{\Omega \vdash \langle t_1, t_1' \rangle : G_1 \times G_1' \leqslant \langle t_2, t_2' \rangle : G_2 \times G_2'}$$

$$(\leqslant \text{appG})\frac{\Omega \vdash t_1 : G_1 \leqslant t_2 : G_2 \quad G_1 \rightarrowtriangle \forall X.G_1'' \quad G_2 \rightarrowtriangle \forall X.G_2'' \quad G_1' \leqslant G_2'}{\Omega \vdash t_1\ [G_1'] : G_1''[G_1'/X] \leqslant t_2\ [G_2'] : G_2''[G_2'/X]}$$

$$(\leqslant \text{pair}i)\frac{\Omega \vdash t_1 : G_1 \leqslant t_2 : G_2 \quad G_1 \rightarrowtriangle G_{11} \times G_{21} \quad G_2 \rightarrowtriangle G_{12} \times G_{22}}{\Omega \vdash \pi_i(t_1) : G_{i1} \leqslant \pi_i(t_2) : G_{i2}}$$

$\boxed{\Omega \vdash \Gamma \sqsubseteq \Gamma}$ **Well-formedness** Ω

$$\frac{}{\cdot \vdash \cdot \sqsubseteq \cdot} \qquad \frac{\Omega \vdash \Gamma_1 \sqsubseteq \Gamma_2 \quad G_1 \sqsubseteq G_2}{\Omega, x : G_1 \sqsubseteq G_2 \vdash \Gamma_1, x : G_1 \sqsubseteq \Gamma_2, x : G_2}$$

Fig. 12. GSF: Syntactic strict term precision.

Let us suppose that $\Omega \vdash \cdot \triangleright (\varepsilon_1 t_1 :: \forall X.G_1'')\ [G_1'] : G_1''[G_1'/X] \leqslant \cdot \triangleright (\varepsilon_2 t_2 :: \forall X.G_2'')\ [G_2'] : G_2''[G_2'/X]$, where $\Omega \vdash \Gamma_1 \sqsubseteq \Gamma_2$. Then by $(\leqslant \text{appG}_\varepsilon)$, we know that $\Omega \vdash \cdot \triangleright (\varepsilon_1 t_1 :: \forall X.G_1'')\ [G_1'] : \forall X.G_1'' \leqslant \cdot \triangleright (\varepsilon_2 t_2 :: \forall X.G_2'')\ [G_2'] : \forall X.G_2''$ and $G_1' \leqslant G_2'$. Note that $\mathcal{I}(G_i, \forall X.G_i'') = \mathcal{I}(\forall X.G_i'', \forall X.G_i'') = \langle \forall X.G_i'', \forall X.G_i'' \rangle$, for $i \in \{1, 2\}$. By $(\leqslant \text{Masc}_\varepsilon)$, it must be the case that $\forall X.G_1'' \sqsubseteq \forall X.G_2''$ and

$\Omega \vdash \cdot \triangleright t_1' : G_1 \leqslant \cdot \triangleright t_2' : G_2$. Finally, the strongest requirements yield the following strict precision rule for type applications:

$$\frac{\Omega \vdash t_1 : G_1 \leqslant t_2 : G_2 \qquad G_1' \leqslant G_2'}{\Omega \vdash t_1\ [G_1'] : G_1''[G_1'/X] \leqslant t_2\ [G_2'] : G_2''[G_2'/X]}$$

Syntactic strict precision. Figure 12 defines syntactic strict precision for GSF terms, which soundly reflects strict precision for GSFε and can account for the translation of GSF terms to GSFε. Judgment $\Omega \vdash t_1 : G_1 \leqslant t_2 : G_2$ denotes that term t_1 of type G_1 is more strictly precise to t_2 of type G_2, under precision relation environment Ω. Note that contrary to GSFε, we do not require type stores because source terms only exist prior to evaluation, and hence do not contain type names. Most of the rules are straightforward and derived following the reasoning explained above for ascriptions and type applications. We use metavariable v in GSF to range over constants, functions and type abstractions, and use $\leqslant_v$ to relate them. We make such distinction between precision on values and terms, because a pair of values such as $\Lambda X.\lambda x : X.x :: X$ and $\Lambda X.\lambda x : ?.x :: X$ should not be related ($\leqslant$v), but their ascriptions to $\forall X.X \to X$ should ($\leqslant$ascv).

Rule ($\leqslant$v) demands that the internal types of the values be related in $\leqslant$ because we do not know in which context the value is going to be used. In contrast, rule ($\leqslant$ascv) is more permissive, establishing that the internal types can be in $\sqsubseteq$—but only if the values have ascriptions such that their meet (i.e., initial evidences) are in $\leqslant$ (as explained on how we derive ($\leqslant$asct)). This allows capturing some internal losses of precision, whenever the surrounding type information ensures that the associated evidence will be related by $\leqslant$. For instance, $(\Lambda X.\lambda x : X.x :: X) :: \forall X.X \to X \leqslant (\Lambda X.\lambda x : ?.x :: X) :: \forall X.X \to X$ at the corresponding type.

Rule ($\leqslant$asct) uses the same technique to be as permissive as possible: it only requires $G_1 \sqsubseteq G_2$, but requires the meets of the types involved in the ascriptions to be related by $\leqslant$ as explained before. Likewise, Rule ($\leqslant$app) requires the meets of the function argument types and the actual argument types to be related by $\leqslant$. Note that during translation from GSF to GSFε, the arguments t_1' and t_2' will be ascribed to $dom^\sharp(G_1)$ and $dom^\sharp(G_2)$, respectively. To account for strict precision over the evidence of the ascriptions $\mathcal{I}(G_1', dom^\sharp(G_1)) = \langle G_1' \sqcap dom^\sharp(G_1), G_1' \sqcap dom^\sharp(G_1)\rangle$ and $\mathcal{I}(G_2', dom^\sharp(G_2)) = \langle G_2' \sqcap dom^\sharp(G_2), G_2' \sqcap dom^\sharp(G_2)\rangle$, we require that $G_1' \sqcap dom^\sharp(G_1) \leqslant G_2' \sqcap dom^\sharp(G_2)$. Rule ($\leqslant$appG) follows the GSF$\varepsilon$ precision rule for type instantiation and uses $\leqslant$ to relate the instantiation types.

Soundness of syntactic strict precision. Finally, syntactic strict term precision for GSF is sound with respect to strict term precision of the translated terms in GSFε:

Proposition 9.10. *Suppose t_1 and t_2 GSF terms such that $\cdot \vdash t_1 : G_1 \leqslant t_2 : G_2$, and their elaborations $\cdot \vdash t_1 \rightsquigarrow t_{\varepsilon_1} : G_1$ and $\cdot \vdash t_2 \rightsquigarrow t_{\varepsilon_2} : G_2$. Then $\cdot \vdash \cdot \triangleright t_{\varepsilon_1} : G_1 \leqslant \cdot \triangleright t_{\varepsilon_2} : G_2$.*

10 GRADUAL PARAMETRICITY FOR GSF

In this section, we first discuss two different notions of parametricity for gradual languages that have been developed in the literature (Section 10.1), in order to situate the notion of gradual parametricity for GSF (Section 10.2). Then, we show in Section 10.3 that this notion of gradual parametricity for GSF is incompatible with the DGG. This tension is established solely driven by the definition of parametricity, and not by monotonicity of consistent transitivity (Section 9.1). This suggests that the incompatibility is shared by other languages with essentially the same notion of gradual parametricity, for which the dynamic gradual guarantee has so far been left as an open question. Finally, we explore gradual free theorems in GSF based on examples discussed

in the literature, using both gradual parametricity and the DGG$^{\leqslant}$ in order to establish such results (Section 10.4).

10.1 On Gradual Parametricities

We first review the standard technique to state and prove parametricity. The notion of parametricity established by Reynolds [1983] is usually defined by interpreting types as *binary logical relations*. The fundamental property of such a relation, also known as the *abstraction theorem*, states that a well-typed term is related to itself at its type. Consequently, polymorphic terms must behave uniformly at all possible type instantiations.

Preliminaries. The definition of parametricity for the statically-typed polymorphic lambda calculus is standard and uncontroversial. Notationally, we follow Ahmed et al. [2017] in all the technical development hereafter. The chosen notations scale smoothly to describe gradual parametricity, both in other work and ours. The relational interpretation of types is presented using *atoms* of the form $(t_1, t_2) \in \mathrm{Atom}[T_1, T_2]$, denoting that the closed terms t_1 and t_2 have types T_1 and T_2, respectively. Formally:

$$\mathrm{Atom}[T_1, T_2] = \{(t_1, t_2) \mid \cdot \vdash t_1 : T_1 \wedge \cdot \vdash t_2 : T_2\}$$

The logical relation is defined using two mutually-defined interpretations: one for values and one for computations. For simplicity and uniformity, throughout this section we use notation $(v_1, v_2) \in \mathcal{V}_\rho[\![T]\!]$ when v_1 and v_2 are related values at type T under environment ρ, and notation $(t_1, t_2) \in \mathcal{T}_\rho[\![T]\!]$ when terms t_1 and t_2 are related computations at type T under environment ρ. An environment ρ, which maps a type variable to two types and a relation, is used to relate values at abstract types as explained below. For convenience, we introduce the following notation for projections in ρ: if $\rho = \{X \mapsto (T_{11}, T_{12}, R_1), Y \mapsto (T_{21}, T_{22}, R_2), \ldots\}$, then $\rho_1 = \{X \mapsto T_{11}, Y \mapsto T_{21}, \ldots\}$, $\rho_2 = \{X \mapsto T_{12}, Y \mapsto T_{22}, \ldots\}$, and $\rho_R = \{X \mapsto R_1, Y \mapsto R_2, \ldots\}$.

Let us briefly go through the definitions. Two base values (of type B) are related if they are the same:

$$\mathcal{V}_\rho[\![B]\!] = \{(v, v) \in \mathrm{Atom}_\rho[B]\}$$

where $\mathrm{Atom}_\rho[T] = \{(t_1, t_2) \mid (t_1, t_2) \in \mathrm{Atom}[\rho_1(T), \rho_2(T)]\}$. Two functions are related if given two related argument the application yield related computations:

$$\mathcal{V}_\rho[\![T_1 \to T_2]\!] = \{(v_1, v_2) \in \mathrm{Atom}_\rho[T_1 \to T_2] \mid \forall (v_1', v_2') \in \mathcal{V}_\rho[\![T_1]\!].(v_1\ v_1', v_2\ v_2') \in \mathcal{T}_\rho[\![T_2]\!]\}$$

Two type abstractions are related if their instantiations to two arbitrary types yield related computations for any given relation between the instantiated types:

$$\mathcal{V}_\rho[\![\forall X.T]\!] = \{(v_1, v_2) \in \mathrm{Atom}_\rho[\forall X.T] \mid \forall T_1, T_2, \forall R \in \mathsf{Rel}[T_1, T_2].(v_1\ [T_1], v_2\ [T_2]) \in \mathcal{T}_{\rho, X \mapsto (T_1, T_2, R)}[\![T]\!]\}$$

where R relates values of types T_1 and T_2, formally $\mathsf{Rel}[T_1, T_2] = \{R \subseteq \mathrm{Atom}[T_1, T_2]\}$. This relation allows us to relate values at abstract types: two values are related at an abstract type X, if they are in the relation for X:

$$\mathcal{V}_\rho[\![X]\!] = \rho_R(X)$$

Finally, two computations are related if they reduce to two related values ($t \longmapsto^* v$ specifies that term t reduces in zero or more steps to the value v).

$$\mathcal{T}_\rho[\![T]\!] = \{(t_1, t_2) \in \mathrm{Atom}_\rho[T] \mid t_1 \longmapsto^* v_1 \Rightarrow (t_2 \longmapsto^* v_2 \wedge (v_1, v_2) \in \mathcal{V}_\rho[\![T]\!])\}$$

With the above definitions, we can establish the definition of the logical relation between two open terms. Two open terms are related if both are well-typed with the same type, and if we

close them with any ρ and γ in the interpretation of Δ and Γ, respectively, we obtain related computations.

$$\Delta;\Gamma \vdash t_1 \preceq t_2 : T \triangleq \Delta;\Gamma \vdash t_1 : T \wedge \Delta;\Gamma \vdash t_2 : T \wedge \forall \rho, \gamma . \rho \in \mathcal{D}[\![\Delta]\!] \wedge \gamma \in \mathcal{G}_\rho[\![\Gamma]\!] \Rightarrow (\rho(\gamma_1(t_1)), \rho(\gamma_2(t_2))) \in \mathcal{T}_\rho[\![G]\!]$$

The fundamental property of this relation establishes that a well-typed program is related with itself: if $\Delta;\Gamma \vdash t : T$ then $\Delta;\Gamma \vdash t \preceq t : T$. The proof of this property uses compatibility lemmas for each term constructor. For instance, the compatibility lemma for type instantiation states: if $\Delta;\Gamma \vdash t_1 \preceq t_2 : \forall X.T$ and $\Delta \vdash T'$, then $\Delta;\Gamma \vdash t_1\ [T'] \preceq t_2\ [T'] : T[T'/X]$. An important property that is used to demonstrate parametricity (specifically the above compatibility lemma) [Ahmed 2006; Ahmed et al. 2017] is the following (hereafter called *compositionality*):

$$\text{If } \Delta \vdash T', \rho \in \mathcal{D}[\![\Delta]\!] \text{ and } R = \mathcal{V}_\rho[\![T']\!], \text{ then } \mathcal{V}_{\rho, X \mapsto (\rho_1(T'), \rho_2(T'), R)}[\![T]\!] = \mathcal{V}_\rho[\![T[T'/X]]\!]$$

Observe that for compositionality to be satisfied, the relation R can not be any relation; it must be $\mathcal{V}_\rho[\![T']\!]$. For example, if $T = X$ and $T' = \mathsf{Int}$, then $\mathcal{V}_{\rho, X \mapsto (\rho_1(T'), \rho_2(T'), R)}[\![T]\!] = \mathcal{V}_{\rho, X \mapsto (\rho_1(\mathsf{Int}), \rho_2(\mathsf{Int}), R)}[\![X]\!] = \rho_R(X) = R$ and $\mathcal{V}_\rho[\![T[T'/X]]\!] = \mathcal{V}_\rho[\![X[\mathsf{Int}/X]]\!] = \mathcal{V}_\rho[\![\mathsf{Int}]\!]$. Therefore, $R = \mathcal{V}_\rho[\![T']\!] = \mathcal{V}_\rho[\![\mathsf{Int}]\!]$.

Parametricity for gradual languages—or *gradual parametricity*—is a novel concept around which different efforts have been developed, yielding different notions. The subtle differences in interpretation come from the specificities of gradual typing, namely the potential for runtime errors due to type imprecision. Much of it is linked to the mechanism used to enforce type abstraction. Apart from GSF, gradual parametricity has only been proven for λB [Ahmed et al. 2017] and PolyG$^\nu$ [New et al. 2020], under two fairly different interpretations. Technically, both are defined using logical relations that are fairly standard, except for three important cases: polymorphic types, type variables, and of course, the unknown type. We now briefly review and compare both approaches.

Gradual parametricity in λB. We now present the notion of gradual parametricity for λB, illustrating the main differences with the original notion of parametricity of Reynolds [1983]. As we will see later, GSF follows similar ideas and techniques. Building on prior work by Matthews and Ahmed [2008], λB uses runtime type generation to reduce type applications, and a form of automatic (un)sealing is introduced via *conversions* and type names during reduction. A conversion $T_1 \overset{\phi}{\Rightarrow} T_2$ is used to make explicit the conversion between a type name and the type it is bound to in the store. The label ϕ stands for a type name α accompanied by a sign ($-$ or $+$), where $-\alpha$ represents a sealing and $+\alpha$ an unsealing. For instance, $1 : \mathsf{Int} \overset{-\alpha}{\Rightarrow} \alpha$ is a conversion, representing an integer sealed value with type α. The term $(1 : \mathsf{Int} \overset{-\alpha}{\Rightarrow} \alpha) : \alpha \overset{+\alpha}{\Rightarrow} \mathsf{Int}$ is composed of two conversions, reducing to the plain value 1 after unsealing. Conversions are introduced upon type applications, similar to the ε_{out} evidence in GSF. They are also closely related to sealing and unsealing terms in PolyG$^\nu$.

Since gradual types introduce divergence, λB uses a *step-indexed* logical relation to ensure well-foundedness. Technically, this means that atoms in λB are of the form (W, t_1, t_2), where world W describes the set of assumptions under which the pair of expressions t_1 and t_2 are related. Because reduction occurs relative to a type name store, and type names have indefinite dynamic extent, worlds are of the form $(j, \Sigma_1, \Sigma_2, \kappa)$: j corresponds to the step index, Σ_1, and Σ_2 correspond to the type name stores under which the terms are being typechecked and evaluated, and κ is a map from type names to relations R. As worlds carry type instantiation information and relations, environment ρ now maps type variables to type names. For instance, $\rho(X) = (T_1, T_2, R)$ may correspond to $\rho(X) = \alpha$, in a world W such that $W = (j, \{\alpha := T_1\}, \{\alpha := T_2\}, \{\alpha \mapsto R\})$. For simplicity, we use a dot notation to access different components of a world: $W.j, W.\Sigma_1, W.\Sigma_2$, and $W.\kappa$ are used to access the step-index, both type name stores, and the relation store, respectively. Note that

the index $W.j$ specifies the number of available *future* reduction steps, i.e., a single reduction step reduces the index by one.

Two values are related at a type name α if both values are conversions to α and belong to the relation associated with α:

$$\mathcal{V}_\rho[\![\alpha]\!] = \{(W, v_1 : T_1 \overset{-\alpha}{\Rightarrow} \alpha, v_2 : T_2 \overset{-\alpha}{\Rightarrow} \alpha) \in \mathrm{Atom}_\emptyset[\alpha] \mid (\downarrow W, v_1, v_2) \in W.\kappa(\alpha)\}$$

Above, $\downarrow W$ lowers the step-index of the world and the intepretations κ in the world by one; formally $\downarrow W = (j, W.\Sigma_1, W.\Sigma_2, \lfloor W.\kappa \rfloor_j)$ and $j = W.j - 1$, where $\lfloor \kappa \rfloor_j = \{\alpha \mapsto \lfloor R \rfloor_j \mid \kappa(\alpha) = R\}$, and $\lfloor R \rfloor_j = \{(W, t_1, t_2) \in R \mid W.j < j\}$. For instance, $(W, 1 : \mathsf{Int} \overset{-\alpha}{\Rightarrow} \alpha, 2 : \mathsf{Int} \overset{-\alpha}{\Rightarrow} \alpha) \in \mathcal{V}_\rho[\![\alpha]\!] = \mathcal{V}_\rho[\![X]\!]$, when $\rho(X) = \alpha$, $W.\Sigma_1(\alpha) = \mathsf{Int}$, $W.\Sigma_2(\alpha) = \mathsf{Int}$, and $(\downarrow W, 1, 2) \in W.\kappa(\alpha)$.

As sealing and unsealing are introduced automatically at runtime, to reason parametrically about type abstractions, λB does not directly relate type applications as computations. For instance, consider term $\Lambda X.\lambda x : X.x$, which is related with itself $(W, \Lambda X.\lambda x : X.x, \Lambda X.\lambda x : X.x) \in \mathcal{V}_\emptyset[\![\forall X.X \to X]\!]$. If we instantiate these values with Int, choosing relation $\{(1, 2)\}$, as $W.\Sigma_i \triangleright (\Lambda X.\lambda x : X.x)\ [\mathsf{Int}] \longmapsto W.\Sigma_i, \alpha := \mathsf{Int} \triangleright (\lambda x : \alpha.x) : \alpha \to \alpha \overset{+\alpha}{\Rightarrow} \mathsf{Int} \to \mathsf{Int}$, then according to the standard definition of parametricity, the two instantiations have to be related at type $X \to X$. Following the dynamic semantics of λB, the following must hold:

$$(W', (\lambda x : \alpha.x) : \alpha \to \alpha \overset{+\alpha}{\Rightarrow} \mathsf{Int} \to \mathsf{Int}, (\lambda x : \alpha.x) : \alpha \to \alpha \overset{+\alpha}{\Rightarrow} \mathsf{Int} \to \mathsf{Int}) \in \mathcal{V}_\rho[\![X \to X]\!]$$

for a future world W' such that $(W'', 1, 2) \in W'.\kappa(\alpha)$, for some W''. A future world intuitively captures how the world changes upon reduction: while the step-index decreases by one at each step of reduction, the store is extended after each type instantiation. Formally, we say that W' is a future world of W, notation $W' \geq W$, if the step index is lower ($W'.j < W.j$), the type name stores are super sets of the originals ($W'.\Sigma_1 \supseteq W.\Sigma_1$ and $W'.\Sigma_1 \supseteq W.\Sigma_1$), and the $W'.\kappa$ is a future relation store ($W'.\kappa \geq \lfloor W.\kappa \rfloor_{W'.j}$). We say that κ' is a future relation store of κ, notation $\kappa' \geq \kappa$, if $\forall \alpha \in dom(\kappa)$ then $\kappa'(\alpha) = \kappa(\alpha)$. According to the definition of related functions at type $X \to X$, the application of these functions to related values at type X should yield related computations at type X. In particular using $(W', 1 : \mathsf{Int} \overset{-\alpha}{\Rightarrow} \alpha, 2 : \mathsf{Int} \overset{-\alpha}{\Rightarrow} \alpha) \in \mathcal{V}_\rho[\![X]\!]$, then

$$(W', ((\lambda x : \alpha.x) : \alpha \to \alpha \overset{+\alpha}{\Rightarrow} \mathsf{Int} \to \mathsf{Int})\ (1 : \mathsf{Int} \overset{-\alpha}{\Rightarrow} \alpha),$$
$$((\lambda x : \alpha.x) : \alpha \to \alpha \overset{+\alpha}{\Rightarrow} \mathsf{Int} \to \mathsf{Int})\ (2 : \mathsf{Int} \overset{-\alpha}{\Rightarrow} \alpha)) \in \mathcal{T}_\rho[\![X]\!]$$

But these application expressions do not type check! Therefore, instead of relating the two type application expressions as computations, λB relates only the bodies of the type abstractions *after* the type applications have been performed (highlighted in gray):

$$\begin{aligned}\mathcal{V}_\rho[\![\forall X.T]\!] = \{(W, v_1, v_2) \in \mathrm{Atom}_\rho[\forall X.T] \mid \forall T_1, T_2, \forall R \in \mathsf{Rel}[T_1, T_2].\forall W' \geq W.\forall \alpha.\forall t_1, t_2.\\ W'.\Sigma_1 \triangleright v_1\ [T_1] \longmapsto W'.\Sigma_1, \alpha := T_1 \triangleright (t_1 : \rho(T)[\alpha/X] \overset{\alpha-}{\Rightarrow} \rho(T)[T_1/X]) \wedge\\ W'.\Sigma_2 \triangleright v_2\ [T_2] \longmapsto W'.\Sigma_2, \alpha := T_2 \triangleright (t_2 : \rho(T)[\alpha/X] \overset{\alpha-}{\Rightarrow} \rho(T)[T_2/X])) \wedge\\ (W' \boxtimes (\alpha, T_1, T_2, R),\ t_1, t_2\ ) \in \mathcal{T}_{\rho[X \mapsto \alpha]}[\![T]\!]\}\end{aligned}$$

After both type applications take a step, only the inner terms t_1 and t_2 are related in a world extended with α, the two instantiated types T_1 and T_2, and the chosen relation R. World extension $\boxtimes$ is formally defined as $W \boxtimes (\alpha, T_1, T_2, R) = (W.j, (W.\Sigma_1, \alpha := T_1), (W.\Sigma_2, \alpha := T_2), W.\kappa[\alpha \mapsto R])$. Observe how this definition strips out the outermost conversions in charge of sealing and unsealing (this conversion is similar to evidence ε_{out} in GSF). This technique makes it possible to reason about a pair of related functions applied to a pair of already-sealed related values. In the previous example,

we know that if $W.\Sigma_i \triangleright (\Lambda X.\lambda x : X.x)\ [\mathsf{Int}] \longmapsto W.\Sigma_i, \alpha := \mathsf{Int} \triangleright (\lambda x : \alpha.x) : \alpha \to \alpha \overset{+\alpha}{\Rightarrow} \mathsf{Int} \to \mathsf{Int}$ then $(W', \lambda x : \alpha.x, \lambda x : \alpha.x) \in \mathcal{V}_\rho[\![X \to X]\!]$, where W' is the extended future world. Therefore, we can deduce that $(W', (\lambda x : \alpha.x)\ (1 : \mathsf{Int} \overset{-\alpha}{\Rightarrow} \alpha), (\lambda x : \alpha.x)\ (2 : \mathsf{Int} \overset{-\alpha}{\Rightarrow} \alpha)) \in \mathcal{T}_{\rho[X \mapsto \alpha]}[\![X]\!]$.

The problem with this definition is that it does not support directly reasoning about type applications. For instance, in the previous example, from the logical relation we cannot directly deduce that:

$$(W', (\lambda x : \alpha.x) : \alpha \to \alpha \overset{+\alpha}{\Rightarrow} \mathsf{Int} \to \mathsf{Int}, (\lambda x : \alpha.x) : \alpha \to \alpha \overset{+\alpha}{\Rightarrow} \mathsf{Int} \to \mathsf{Int}) \in \mathcal{V}_\rho[\![\mathsf{Int} \to \mathsf{Int}]\!]$$

To reason about related type applications as computations (and not by considering the inner terms only), one needs to use a *conversion* lemma. This lemma relates two values after the unsealing of some type name α. Essentially, such a lemma says that if $(W, v_1, v_2) \in \mathcal{V}_\rho[\![T]\!]$, T and T' are convertible under $+\alpha$, $W.\Sigma_1(\alpha) = W.\Sigma_2(\alpha) = T''$ and $W.\kappa(\alpha) = \mathcal{V}_\rho[\![T'']\!]$, then

$$(W, v_1 : \rho(T) \overset{+\alpha}{\Rightarrow} \rho(T'), v_2 : \rho(T) \overset{+\alpha}{\Rightarrow} \rho(T)) \in \mathcal{T}_\rho[\![T']\!]$$

Observe that to apply this lemma, α must be bound to the same type in both stores ($W.\Sigma_1(\alpha) = W.\Sigma_2(\alpha) = T'$) and to the value relation of that type ($W.\kappa(\alpha) = \mathcal{V}_\rho[\![T']\!]$). When this situation happens, we we say that α is *synchronized* in W. A similar requirement is established for the proof of parametricity for System F, specifically the compositionality lemma described before. The synchronization requirement is needed in gradual parametricity, among other reasons, to prevent the unsealing of unrelated values such as $(W', 1 : \mathsf{Int} \overset{-\alpha}{\Rightarrow} \alpha, 2 : \mathsf{Int} \overset{-\alpha}{\Rightarrow} \alpha) \in \mathcal{V}_\rho[\![\alpha]\!]$. Otherwise, after unsealing, we would have $(W'', 1, 2) \in \mathcal{V}_\rho[\![\mathsf{Int}]\!]$, which is false.

Gradual parametricity in PolyG$^\nu$. New et al. [2020] recently developed another approach to gradual parametricity, which has the benefit of avoiding the convoluted treatment of type applications described above. In doing so, the notion of gradual parametricity they present is more similar to Reynolds's original presentation. Note however, that this comes at a cost: the *syntax* of PolyG$^\nu$ departs importantly from System F, by requiring all sealing and unsealing to happen explicitly in the term syntax, with outward scoping of type variables:

System F $((\Lambda X.\lambda x : X.x)\ [\mathsf{Int}]\ 1)+1$ PolyG$^\nu$ $\mathsf{unseal}_X((\Lambda X.\lambda x : X.x)\ [X = \mathsf{Int}]\ (\mathsf{seal}_X 1))+1$

Technically, gradual parametricity for PolyG$^\nu$ is established by first translating PolyG$^\nu$ to an intermediate language PolyC$^\nu$ and finally to CBPV$_{\mathrm{OSum}}$, a variant of Levy's Call-by-Push-Value [Levy 1999] with open sums to encode the unknown type. The logical relation of parametricity is defined for CBPV$_{\mathrm{OSum}}$, and differs importantly from that of λB. In particular, even though it still uses type names to relate type abstractions, the definition requires type applications (and not some inner terms) to be related as computations, as expected in the standard treatment of parametricity. Crucially, this is possible only because type application never incurs in automatic insertion of conversions to seal/unseal values, as would happen in λB, because in this approach, sealing and unsealing are explicit in the syntax of terms.

Comparing parametricities. The notion of gradual parametricity of PolyG$^\nu$ is stronger than that of λB, as it directly embodies the kind of parametric reasoning that one is used to in static languages. While λB ensures a form of gradual parametricity, this notion is weaker, because given two related type abstractions and two arbitrary (possibly different) types, we cannot directly reason about both corresponding type applications directly: we can only directly reason about the body of the type abstractions after the type applications have reduced.

While the notion of gradual parametricity of PolyG$^\nu$ is superior, as already mentioned, it is enabled by sacrificing the syntax of System F. In this work, we are interested in gradualizing

System F, and studying the properties we can get, rather than in designing a different static source language in order to accommodate the desired reasoning principles. This led us to embrace runtime sealing through type names, as in λB, and consequently, to aspire to a weaker notion of gradual parametricity than that of PolyG$^{\nu}$. We do believe that both approaches are fully valuable and necessary to understand the many ways in which gradual typing can embrace such an advanced typing discipline.

In particular, as illustrated by New et al. [2020], the weaker notion of parametricity adopted in GSF can lead to behavior that breaks the (strong notion of) parametricity enjoyed by PolyG$^{\nu}$. Note that this can however only occur when manipulating values of *imprecise* polymorphic types; for values of static types, the reasoning principles of standard parametricity do apply. They argue that GSF exhibits non-parametric behavior by considering the following example. Consider the below value:

$$v \triangleq (\Lambda X.\lambda x : X.\mathsf{true}) :: \forall X.? \to \mathsf{Bool}$$

Although v is related to itself at type $\forall X.? \to \mathsf{Bool}$, two *different* instantiations (to Int and Bool, respectively) are not related computations, i.e., $(W, v\ [\mathsf{Int}], v\ [\mathsf{Bool}]) \notin \mathcal{T}_{\rho}[\![? \to \mathsf{Bool}]\!]$. Given two related arguments at type ? such as $\varepsilon_{\mathsf{Int}} 3 :: ?$ twice, $v\ [\mathsf{Int}]\ (\varepsilon_{\mathsf{Int}} 3 :: ?)$ reduces to true, whereas $v\ [\mathsf{Bool}]\ (\varepsilon_{\mathsf{Int}} 3 :: ?)$ reduces to an error. As we saw earlier, this happens because GSF does not directly relate type applications as computations. The logical relation only tells us that after instantiation, the internal terms (without the outermost evidences) $\varepsilon_{\alpha^{\mathsf{Int}} \to \mathsf{Bool}}(\lambda x : \alpha.\mathsf{true}) :: ? \to \mathsf{Bool}$ and $\varepsilon_{\alpha^{\mathsf{Bool}} \to \mathsf{Bool}}(\lambda x : \alpha.\mathsf{true}) :: ? \to \mathsf{Bool}$ are indeed related at type $? \to \mathsf{Bool}$. In this case, if we try to apply both functions to $\varepsilon_{\mathsf{Int}} 3 :: ?$, both programs fail. The only arguments that can be passed such that both applications succeed are related sealed values at type ?, such as $(W, \langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle 3 :: ?, \langle \mathsf{Int}, \alpha^{\mathsf{Bool}} \rangle \mathsf{true} :: ?) \in \mathcal{V}_{\rho}[\![?]\!]$ (assuming an appropriate relation for α).

Finally, note that the fact that $v\ [\mathsf{Bool}]\ (1 :: ?)$ reduces to an error in GSF points to a wider point in the design space of gradually-typed languages: how *eagerly* should type constraints be checked? Indeed, $v\ [\mathsf{Bool}]$ is $\lambda x : \mathsf{Bool}.\mathsf{true}$, whose application to an underlying Int value is ill-typed and can legitimately be expected to fail. In that respect, GSF follows GTLC [Siek and Taha 2006; Siek et al. 2015a], in which $(\lambda x : \mathsf{Bool}.\mathsf{true})\ (1 :: ?)$ also fails with a runtime cast error. This eager form of runtime type checking likewise follows from the Abstracting Gradual Typing methodology as formulated by Garcia et al. [2016]. An interesting perspective would be to study a lazy variant of AGT, and whether it recovers properties of alternative approaches [New and Ahmed 2018].

It is interesting to observe that no runtime error is raised in λB for this example, despite the fact that the parametricity logical relation is essentially the same as that of GSF. The difference comes from the runtime semantics of λB: as we have illustrated in Section 3, λB does not track the type instantiations that occur on imprecise types. This means that the underlying typing violation observed by GSF, which manifests as a runtime error, is not noticed in λB. Therefore, this example highlights yet another point of tension in the design space of System F-based gradual languages.

10.2 Gradual Parametricity in GSF

We now turn to the technical details of gradual parametricity in GSF. As explained above, we follow λB [Ahmed et al. 2017] for the formal development of gradual parametricity, due to the use of runtime type name generation for sealing, and the System F syntax that requires automatic insertion of (un)sealing evidences at runtime. We highlight the main differences in the logical relations of GSF with respect to λB, mainly in the value logical relations for types ? and α.

We establish parametricity for GSF by proving parametricity for GSFε. Specifically, we define a *step-indexed* logical relation for GSFε terms, closely following the relation for λB. The relation is defined on atoms (W, t_1, t_2) that denote two related terms t_1, t_2 in a world W. A world is composed

$$\begin{aligned}
\text{Atom}_n[G_1, G_2] &= \{(W, t_1, t_2) \mid W.j < n \wedge W \in \text{World}_n \wedge W.\Xi_1, \cdot, \cdot \vdash t_1 : G_1 \wedge W.\Xi_2, \cdot, \cdot \vdash t_2 : G_2\} \\
\text{Atom}_n^{\text{val}}[G_1, G_2] &= \{(W, v_1, v_2) \in \text{Atom}_n[G_1, G_2]\} \qquad \text{Atom}_\rho[G] = \cup_{n \geq 0}\{(W, t_1, t_2) \in \text{Atom}_n[\rho(G), \rho(G)]\} \\
\text{Atom}_\rho^{=}[G] &= \{(W, v_1, v_2) \in \text{Atom}_\rho[G] \mid \mathit{unlift}(\pi_2(\mathit{ev}(v_1))) = \mathit{unlift}(\pi_2(\mathit{ev}(v_2)))\} \\
\text{World} &= \cup_{n \geq 0} \text{World}_n \\
\text{World}_n &= \{(j, \Xi_1, \Xi_2, \kappa) \in \text{Nat} \times \text{Store} \times \text{Store} \times (\text{TypeName} \to \text{Rel}_j) \mid \\
&\qquad j < n \wedge \vdash \Xi_1 \wedge \vdash \Xi_2 \wedge \forall \alpha \in \mathit{dom}(\kappa).\kappa(\alpha) \in \text{Rel}_j[\Xi_1(\alpha), \Xi_2(\alpha)]\} \\
\text{Rel}_n[G_1, G_2] &= \{R \in \text{Atom}_n^{\text{val}}[G_1, G_2] \mid \forall (W, v_1, v_2) \in R. \forall W' \geq W.(W', v_1, v_2) \in R\} \\
\lfloor R \rfloor_n &= \{\, (W, t_1, t_2) \in R \mid W.j < n \,\} \qquad \lfloor \kappa \rfloor_n = \{\, \alpha \mapsto \lfloor R \rfloor_n \mid \kappa(\alpha) = R \,\} \\
\kappa' \geq \kappa &\triangleq \forall \alpha \in \mathit{dom}(\kappa).\kappa'(\alpha) = \kappa(\alpha) \\
W' \geq W &\triangleq W'.j \leq W.j \wedge W'.\Xi_1 \supseteq W.\Xi_1 \wedge W'.\Xi_2 \supseteq W.\Xi_2 \wedge W'.\kappa \geq \lfloor W.\kappa \rfloor_{W'.j} \wedge W', W \in \text{World} \\
\downarrow W &= (j, W.\Xi_1, W.\Xi_2, \lfloor W.\kappa \rfloor_j) \quad \text{where } j = W.j - 1 \\
W \boxtimes (\alpha, G_1, G_2, R) &= (W.j, (W.\Xi_1, \alpha := G_1), (W.\Xi_2, \alpha := G_2), W.\kappa[\alpha \mapsto R])
\end{aligned}$$

Fig. 13. Logical relation: auxiliary definitions.

of a step index j, two stores Ξ_1 and Ξ_2 used to typecheck and evaluate the related terms, and a mapping κ, which maps type names to relations R, used to relate sealed values. The components of a world are accessed through a dot notation, e.g., $W.j$ for the step index. The interpretations of values, terms, stores, name environments, and type environments are mutually defined, using the auxiliary definitions of Figure 13. As usual, the value and term interpretations are indexed by a type and a type substitution ρ.

Auxiliary definitions. We write $\text{Atom}_\rho[G]$ (Figure 13) to denote a set of terms of the same type after substitution. The $\text{Atom}_\rho^{=}[G]$ is similar to $\text{Atom}_\rho[G]$ but restricts the set to values that have, after substitution, equally precise evidences (the equality is after unlifting because two sealed values may be related under different instantiations). Remember that the unlifting operator, given an evidence type E, returns a gradual type G, forgetting to which types the type names were instantiated. For instance,

$$(W, \langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle 1 :: \alpha, \langle \mathsf{Bool}, \alpha^{\mathsf{Bool}} \rangle \mathsf{true} :: \alpha) \in \text{Atom}_\rho^{=}[X]$$

if $\rho(X) = \alpha$, as $(W, \langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle 1 :: \alpha, \langle \mathsf{Bool}, \alpha^{\mathsf{Bool}} \rangle \mathsf{true} :: \alpha) \in \text{Atom}_\rho[X]$ (assuming an adequate world) and $\mathit{unlift}(\pi_2(\langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle)) = \mathit{unlift}(\pi_2(\langle \mathsf{Bool}, \alpha^{\mathsf{Bool}} \rangle)) = \alpha$. However,

$$(W, \langle \mathsf{Int}, \mathsf{Int} \rangle 1 :: ?, \langle \mathsf{Bool}, \mathsf{Bool} \rangle \mathsf{true} :: ?) \notin \text{Atom}_\rho^{=}[?]$$

since $\mathit{unlift}(\pi_2(\langle \mathsf{Int}, \mathsf{Int} \rangle)) = \mathsf{Int} \neq \mathsf{Bool} = \mathit{unlift}(\pi_2(\langle \mathsf{Bool}, \mathsf{Bool} \rangle))$. We explain this in detail below, when presenting the logical relations for values.

$\text{Rel}_n[G_1, G_2]$ defines the set of relations of values of type G_1 and G_2. Note that if $R \in \text{Rel}_n[G_1, G_2]$, we also require that for all atoms in R, all future versions of that atom should also be present in R. Intuitively, this is because values in a relation R should still be related after lowering the number of steps (reduction). We use $\lfloor R \rfloor_n$ and $\lfloor \kappa \rfloor_n$ to restrict the step index of the worlds to less than n. Finally, $\kappa' \geq \kappa$ specifies that κ' is a future relation mapping of κ (an extension[14]), and similarly $W' \geq W$ expresses that W' is a future world of W. Intuitively, a future relation mapping represents the same relations as the original plus some extra ones that may have been added during reduction. Similarly, a future world represents a world after some steps of reduction, i.e., a world with a smaller (or equal) step index and a future mapping relation.

[14]Note the relation is antisymmetric as we quantify over all $\alpha \in \mathit{dom}(\kappa)$, and this could be false for some $\alpha \in \mathit{dom}(\kappa')$.

Logical relation for terms. Following λB, the logical interpretation of terms (Figure 14) of a given type enforces a "error-sensitive" view of parametricity: if the first term yields a value, the second must produce a related value at that type; if the first term fails, so must the second. The reason behind this is to ensure parametric behavior in the presence of runtime errors. Given a parametric term, if after two different instantiations one of the resulting terms fails and the other terminates to a value, then both instantiations did not behave similarly.[15] Observe that one reduction takes i steps in the definition of the interpretation of terms while the other one takes any arbitrary number of steps. This is because only one index is needed for this definition to be well-founded, and it would be challenging to establish the number of steps for the second reduction.

Logical relations for values. The logical interpretation of values (Figure 14) uses $\mathrm{Atom}^{=}_{\rho}[G]$, which requires the second component of the evidence of each value to have the same precision to enforce such sensitivity. Indeed, if one is allowed to be more precise than the other, then when later combined in the same context, the more precise value may induce failure while the other does not. For instance, if we have evidences $\langle \mathsf{Int} \to \mathsf{Int}, \alpha^{\mathsf{Int}\to\mathsf{Int}} \rangle$ and $\langle \mathsf{Int} \to \mathsf{Int}, \mathsf{Int} \to \mathsf{Int} \rangle$ and combine them through consistent transitivity with the evidence $\langle \mathsf{Int} \to \mathsf{Int}, \mathsf{Int} \to \mathsf{Int} \rangle$, the first combination fails while the second one succeeds, resulting in $\langle \mathsf{Int} \to \mathsf{Int}, \mathsf{Int} \to \mathsf{Int} \rangle$. For this reason, related values are required to have evidence such that their second components are equal using the unlifting operator ($\mathit{unlift}(\pi_2(\langle \mathsf{Int} \to \mathsf{Int}, \alpha^{\mathsf{Int}\to\mathsf{Int}} \rangle)) = \alpha \neq \mathsf{Int} \to \mathsf{Int} = \mathit{unlift}(\pi_2(\langle \mathsf{Int} \to \mathsf{Int}, \mathsf{Int} \to \mathsf{Int} \rangle))$).

Two base values are related if they are equal. Two functions are related if their application to related values yields related results. Note that, unlike λB, the arguments are related at one step down ($\downarrow W$).[16] Otherwise (in combination with the definition of related values at type ?, which also presents some differences), the logical relation would not be well-founded, as we explain below. Two pairs are related if their components are pointwise related. Two type abstractions are related if given any two types and any relation between them, the instantiated terms (without their unsealing evidence) are also related in a world extended ($\boxtimes$) with α, the two instantiation types G_1 and G_2 and the chosen relation R between sealed values (Figure 13). Note that the step index of this extended world is decreased by one, because we take a reduction step.

Two sealed values are related at a type name α if first, after unsealing, the resulting values are in the relation corresponding to α ($W.\kappa(\alpha)$) in a one step lower current world. The first part of the definition is faithful to λB, while the second part is new. We additionally require that for any evidence ε that justifies the judgment between α and any type G, in any store such that W belongs to its interpretation, the values ascribed to the type G and evidence ε remain related. This technical extension is sufficient to prove Lemma 10.5 (formalized at the end of this section), which states that the ascription of two related values yields related terms; this lemma is essential for the proof of parametricity. The necessity of this extension comes from differences between the dynamic semantics of GSF and λB. The dynamic semantics of GSF combine evidence (eager), whereas λB accumulates cast (lazy). For instance, the conversion $(1 : \mathsf{Int} \overset{-\alpha}{\Rightarrow} \alpha) : \alpha \overset{-\beta}{\Rightarrow} \beta$ from λB needs two reduction steps to obtain 1, i.e., the step-index is reduced by two. In GSF, this information is compressed in a single evidence $\langle \mathsf{Int}, \beta^{\alpha^{\mathsf{Int}}} \rangle$, and needs only one reduction step to obtain the similar value, i.e., the step-index is reduced by one. Observe that if ε exists such that $\varepsilon \Vdash \Xi \vdash \alpha \sim G$, since $W \in \mathcal{S}[\![\Xi]\!]$, α must be synchronized (i.e., $W.\Sigma_1(\alpha) = W.\Sigma_2(\alpha) = G'$ and $W.\kappa(\alpha) = \mathcal{V}_\rho[\![G']\!]$). Intuitively, if α is synchronized, then after multiple possible unsealings the resulting values are

[15] Considering a logical relation where the first term may fail and the other terminates leads to a weaker version of parametricity (in terms of different notions of gradual parametricity).

[16] The operator $\downarrow W$ has the same definition as $\blacktriangleright W$ in λB.

$$
\begin{array}{lcl}
\mathcal{V}_\rho[\![B]\!] & = & \{(W, v, v) \in \mathrm{Atom}^{=}_{\rho}[B]\} \\
\mathcal{V}_\rho[\![G_1 \to G_2]\!] & = & \{(W, v_1, v_2) \in \mathrm{Atom}^{=}_{\rho}[G_1 \to G_2] \mid \forall W' \geq W. \forall v_1', v_2'. \\
 & & \quad (\downarrow W', v_1', v_2') \in \mathcal{V}_\rho[\![G_1]\!] \Rightarrow (W', v_1\ v_1', v_2\ v_2') \in \mathcal{T}_\rho[\![G_2]\!]\} \\
\mathcal{V}_\rho[\![G_1 \times G_2]\!] & = & \{(W, v_1, v_2) \in \mathrm{Atom}^{=}_{\rho}[G_1 \times G_2] \mid \\
 & & \quad (W, \pi_1(v_1), \pi_1(v_2)) \in \mathcal{T}_\rho[\![G_1]\!] \wedge (W, \pi_2(v_1), \pi_2(v_2)) \in \mathcal{T}_\rho[\![G_2]\!]\} \\
\mathcal{V}_\rho[\![\forall X.G]\!] & = & \{(W, v_1, v_2) \in \mathrm{Atom}^{=}_{\rho}[\forall X.G] \mid \forall W' \geq W. \forall t_1, t_2, G_1, G_2, \alpha, \varepsilon_1, \varepsilon_2. \\
 & & \quad \forall R \in \mathrm{Rel}_{W'.j}[G_1, G_2]. \\
 & & \qquad (W'.\Xi_1 \vdash G_1 \wedge W'.\Xi_2 \vdash G_2 \wedge \\
 & & \qquad\ W'.\Xi_1 \triangleright v_1[G_1] \longmapsto W'.\Xi_1, \alpha := G_1 \triangleright \varepsilon_1 t_1 :: \rho(G)[G_1/X] \wedge \\
 & & \qquad\ W'.\Xi_2 \triangleright v_2[G_2] \longmapsto W'.\Xi_2, \alpha := G_2 \triangleright \varepsilon_2 t_2 :: \rho(G)[G_2/X]) \Rightarrow \\
 & & \qquad (\downarrow W' \boxtimes (\alpha, G_1, G_2, R), t_1, t_2) \in \mathcal{T}_{\rho[X \mapsto \alpha]}[\![G]\!]\} \\
\mathcal{V}_\rho[\![X]\!] & = & \mathcal{V}_\rho[\![\rho(X)]\!] \\
\mathcal{V}_\rho[\![\alpha]\!] & = & \{(W, \langle E_{11}, \alpha^{E_{12}}\rangle u_1 :: \alpha, \langle E_{21}, \alpha^{E_{22}}\rangle u_2 :: \alpha) \in \mathrm{Atom}^{=}_{\emptyset}[\alpha] \mid \\
 & & \quad (\downarrow W, \langle E_{11}, E_{12}\rangle u_1 :: W.\Xi_1(\alpha), \langle E_{21}, E_{22}\rangle u_2 :: W.\Xi_2(\alpha)) \in W.\kappa(\alpha) \wedge \\
 & & \quad (\forall \Xi, \varepsilon, G. (W \in \mathcal{S}[\![\Xi]\!] \wedge \varepsilon \Vdash \Xi \vdash \alpha \sim G) \Rightarrow \\
 & & \quad (W, \varepsilon(\langle E_{11}, \alpha^{E_{12}}\rangle u_1 :: \alpha) :: G, \varepsilon(\langle E_{21}, \alpha^{E_{22}}\rangle u_2 :: \alpha) :: G) \in \mathcal{T}_\rho[\![G]\!])\} \\
\mathcal{V}_\rho[\![?]\!] & = & \{(W, \varepsilon_1 u_1 :: ?, \varepsilon_2 u_2 :: ?) \in \mathrm{Atom}^{=}_{\emptyset}[?] \mid const(\pi_2(\varepsilon_i)) = G \wedge \\
 & & \quad (W, \varepsilon_1 u_1 :: G, \varepsilon_2 u_2 :: G) \in \mathcal{V}_\rho[\![G]\!]\} \\
\hline
\mathcal{T}_\rho[\![G]\!] & = & \{(W, t_1, t_2) \in \mathrm{Atom}_{\rho}[G] \mid \forall i < W.j, (\forall \Xi_1, v_1.\ W.\Xi_1 \triangleright t_1 \longmapsto^i \Xi_1 \triangleright v_1 \Rightarrow \\
 & & \quad \exists W' \geq W, v_2.\ W.\Xi_2 \triangleright t_2 \longmapsto^* W'.\Xi_2 \triangleright v_2 \wedge W'.j + i = W.j \wedge \\
 & & \quad W'.\Xi_1 = \Xi_1 \wedge (W', v_1, v_2) \in \mathcal{V}_\rho[\![G]\!]) \wedge \\
 & & \quad (\forall \Xi_1. W.\Xi_1 \triangleright t_1 \longmapsto^i \mathbf{error} \Rightarrow \exists \Xi_2. W.\Xi_2 \triangleright t_2 \longmapsto^* \mathbf{error})\} \\
\hline
\mathcal{S}[\![\cdot]\!] & = & \textsc{World} \\
\mathcal{S}[\![\Xi, \alpha := G]\!] & = & \mathcal{S}[\![\Xi]\!] \cap \{W \in \textsc{World} \mid W.\Xi_1(\alpha) = G \wedge W.\Xi_2(\alpha) = G \wedge \\
 & & \quad \vdash W.\Xi_1 \wedge \vdash W.\Xi_2 \wedge W.\kappa(\alpha) = \lfloor \mathcal{V}_\emptyset[\![G]\!] \rfloor_{W.j}\} \\
\hline
\mathcal{D}[\![\cdot]\!] & = & \{\, (W, \emptyset) \mid W \in \textsc{World} \,\} \\
\mathcal{D}[\![\Delta, X]\!] & = & \{\, (W, \rho[X \mapsto \alpha]) \mid (W, \rho) \in \mathcal{D}[\![\Delta]\!] \wedge \alpha \in dom(W.\kappa) \,\} \\
\hline
\mathcal{G}_\rho[\![\cdot]\!] & = & \{\, (W, \emptyset) \mid W \in \textsc{World} \,\} \\
\mathcal{G}_\rho[\![\Gamma, x : G]\!] & = & \{\, (W, \gamma[x \mapsto (v_1, v_2)]) \mid (W, \gamma) \in \mathcal{G}_\rho[\![\Gamma]\!] \wedge (W, v_1, v_2) \in \mathcal{V}_\rho[\![G]\!] \,\} \\
\hline
\Xi; \Delta; \Gamma \vdash t_1 \preceq t_2 : G \triangleq & & \Xi; \Delta; \Gamma \vdash t_1 : G \wedge \Xi; \Delta; \Gamma \vdash t_2 : G \wedge \forall W \in \mathcal{S}[\![\Xi]\!], \rho, \gamma. \\
 & & ((W, \rho) \in \mathcal{D}[\![\Delta]\!] \wedge (W, \gamma) \in \mathcal{G}_\rho[\![\Gamma]\!]) \Rightarrow (W, \rho(\gamma_1(t_1)), \rho(\gamma_2(t_2))) \in \mathcal{T}_\rho[\![G]\!] \\
\Xi; \Delta; \Gamma \vdash t_1 \approx t_2 : G \triangleq & & \Xi; \Delta; \Gamma \vdash t_1 \preceq t_2 : G \wedge \Xi; \Delta; \Gamma \vdash t_2 \preceq t_1 : G
\end{array}
$$

$$
const(B) = B \qquad const(\alpha^G) = \alpha \qquad const(E_1 \to E_2) = ? \to ? \qquad const(\forall X.E) = \forall X.?
$$

$$
const(E_1 \times E_2) = ? \times ?
$$

Fig. 14. Gradual logical relation.

kept related. For instance, if $(W, \langle \mathsf{Int}, \alpha^{\beta^{\mathsf{Int}}}\rangle 1 :: \alpha, \langle \mathsf{Int}, \alpha^{\beta^{\mathsf{Int}}}\rangle 1 :: \alpha) \in \mathcal{V}_\rho[\![\alpha]\!]$ where α is synchronized, then it must be the case that $(W', \langle \mathsf{Int}, \beta^{\mathsf{Int}}\rangle 1 :: \beta, \langle \mathsf{Int}, \beta^{\mathsf{Int}}\rangle 1 :: \beta) \in W.\kappa(\beta)$, and also that $(W'', \langle \mathsf{Int}, \mathsf{Int}\rangle 1 :: \mathsf{Int}, \langle \mathsf{Int}, \mathsf{Int}\rangle 1 :: \mathsf{Int}) \in \mathcal{V}_\rho[\![\mathsf{Int}]\!]$; but if $(W, \langle \mathsf{Int}, \alpha^{\beta^{\mathsf{Int}}}\rangle 1 :: \alpha, \langle \mathsf{Bool}, \alpha^{\beta^{\mathsf{Bool}}}\rangle \mathsf{true} :: \alpha) \in \mathcal{V}_\rho[\![\alpha]\!]$, then it must be the case that $(W', \langle \mathsf{Int}, \beta^{\mathsf{Int}}\rangle 1 :: \mathsf{Int}, \langle \mathsf{Bool}, \beta^{\mathsf{Bool}}\rangle \mathsf{true} :: \mathsf{Bool}) \in W.\kappa(\beta)$.

Finally, two values are related at type ? if they are related at the least-precise type with the same top-level constructor as the second component of the evidence, $const(\pi_2(\varepsilon_i))$. The function *const* extracts the top-level constructor of an evidence type (Figure 14). The intuition is that to be able to relate these unknown values we must take a step towards relating their actual content; evidence necessarily captures at least the top-level constructor (e.g., if a value is a function, the second evidence type is no less precise than $? \to ?$, i.e., $const(E_1 \to E_2)$). Also, we consider the second

component as in $\langle E_1, E_2\rangle \Vdash \Xi; \Delta \vdash G \sim G'$, E_1 and E_2 correspond to the most precise information about G and G', respectively. Here E_2 corresponds to the most precise information about ?, and E_1 could be a totally unrelated type as in $(W, \langle \mathsf{Int}, \alpha^{\mathsf{Int}}\rangle 1 :: ?, \langle \mathsf{Bool}, \alpha^{\mathsf{Bool}}\rangle \mathsf{true} :: ?) \in \mathcal{V}_\rho[\![?]\!]$. Observe that, unlike λB, the definition does not decrease the index of the world W by 1. However, in λB this is done on a case-by-case basis where needed (i.e., function, type name, etc.). We made this technical change to facilitate the parametricity proof.

Well-foundedness. The logical relation is well-founded for three reasons: *(i)* in the ? case, $const(\pi_2(\varepsilon_i))$ cannot itself be ?, as just explained; *(ii)* in each other recursive cases, the step index is lowered: for functions and pairs, the relation is between reducible expressions (applications, projections) that either take a step or fail; for type abstractions, the relation is with respect to a world whose index is lowered; *(iii)* we require in the definition of related functions that arguments must be related at (at least) one step down. For instance, if $(W, \langle ? \to ?, ? \to ?\rangle u_1 :: ?, \langle ? \to ?, ? \to ?\rangle u_2 :: ?) \in \mathcal{V}_\rho[\![?]\!]$, then $(W, \langle ? \to ?, ? \to ?\rangle u_1 :: ? \to ?, \langle ? \to ?, ? \to ?\rangle u_2 :: ? \to ?) \in \mathcal{V}_\rho[\![? \to ?]\!]$, but function arguments related at $\mathcal{V}_\rho[\![?]\!]$ would contain the original $(W, \langle ? \to ?, ? \to ?\rangle u_1 :: ?, \langle ? \to ?, ? \to ?\rangle u_2 :: ?)$ atom.

Logical relations for stores and environments. The unary relation $\mathcal{S}[\![\Xi]\!]$ specifies all the worlds that satisfy Ξ: every α in $dom(\Xi)$ must be synchronized. The interpretation of $\mathcal{D}[\![\Delta]\!]$ specifies all pairs of worlds W and type substitutions ρ, such that all type variables in Δ are mapped to some α in ρ, and α is associated to some relation in W. The relation $\mathcal{G}_\rho[\![\Gamma]\!]$ specifies that the value environment γ satisfies the type environment Γ under world W if, for every variable $x \in dom(\Gamma)$, the mapped values are related in $\mathcal{V}_\rho[\![\Gamma(x)]\!]$ in world W.

For convenience, we introduce the following notation for projections in γ: if $\gamma = \{x \mapsto (v_{11}, v_{12}), y \mapsto (v_{21}, v_{22}), \ldots\}$, then $\gamma_1 = \{x \mapsto v_{11}, y \mapsto v_{21}, \ldots\}$ and $\gamma_2 = \{x \mapsto v_{12}, y \mapsto v_{22}, \ldots\}$. Type variable substitution $\rho_i(s)$ is defined as syntactic sugar for $\rho(W.\Xi_i, s)$, in a context where W is defined, lifting each substituted type name in the process and defined as

$$(\rho, X \mapsto \alpha)(\Xi, s) = \rho(\Xi, s[lift_\Xi(\alpha)/X])$$
$$\cdot(\Xi, s) = s$$

Parametricity. The logical relation approximation $\Xi; \Delta; \Gamma \vdash t_1 \preceq t_2 : G$ says that given a world W that satisfies Ξ, a type substitution ρ and value environment γ that satisfies Δ and Γ, respectively, under world W, then the pair of substituted terms $\rho_1(\gamma_1(t_1)), \rho_2(\gamma_2(t_2))$ are related computations in $\mathcal{T}_\rho[\![G]\!]$. Logical equivalence $\Xi; \Delta; \Gamma \vdash t_1 \approx t_2 : G$ is defined as the symmetric extension of logical approximation. Finally, the fundamental property says that any well-typed GSFε term is related to itself at its type:

Theorem 10.1 (Fundamental Property). *If* $\Xi; \Delta; \Gamma \vdash t : G$ *then* $\Xi; \Delta; \Gamma \vdash t \preceq t : G$.

As standard, the proof of the fundamental property uses compatibility lemmas for each term constructor and the compositionality lemma. The compatibility lemmas related to type abstractions are the following:

Lemma 10.2 (Compatibility-EΛ). *If* $\Xi; \Delta, X \vdash t_1 \preceq t_2 : G$, $\varepsilon \vdash \Xi; \Delta \vdash \forall X.G \sim G'$ *and* $\Xi; \Delta \vdash \Gamma$ *then* $\Xi; \Delta; \Gamma \vdash \varepsilon(\Lambda X.t_1) :: G' \preceq \varepsilon(\Lambda X.t_2) :: G' : G'$.

Lemma 10.3 (Compatibility-EappG). *If* $\Xi; \Delta; \Gamma \vdash t_1 \preceq t_2 : \forall X.G$ *and* $\Xi; \Delta \vdash G'$, *then* $\Xi; \Delta; \Gamma \vdash t_1\ [G'] \preceq t_2\ [G'] : G[G'/X]$.

The compatibility lemma for type abstractions (Lemma 10.2) says that if two terms are related, then the type abstractions (whose bodies are those terms) ascribed to any type G' are also related

at G'. The compatibility lemma for type instantiations (Lemma 10.3) says that if two terms are related to some polymorphic type $\forall X.G$, then the instantiations to some type G' are related at $G[G'/X]$. The remaining compatibility lemmas are defined analogously and can be found in the companion technical report.

In order to prove Lemma 10.3 (Compatibility-EappG), we establish another important lemma to relate terms after a type substitution.

Lemma 10.4 (Compositionality). *If*

- $W.\Xi_i(\alpha) = \rho(G')$ *and* $W.\kappa(\alpha) = \mathcal{V}_\rho[\![G']\!]$,
- $E'_i = lift_{W.\Xi_i}(\rho(G'))$,
- $E_i = lift_{W.\Xi_i}(G_p)$ *for some* $G_p \sqsubseteq \rho(G)$,
- $\rho' = \rho[X \mapsto \alpha]$,
- $\varepsilon_i = \langle E_i[\alpha^{E'_i}/X], E_i[E'_i/X]\rangle$, *such that* $\varepsilon_i \vdash W.\Xi_i \vdash \rho(G[\alpha/X]) \sim \rho(G[G'/X])$, *and*
- ${\varepsilon_i}^{-1} = \langle E_i[E'_i/X], E_i[\alpha^{E'_i}/X]\rangle$, *such that* ${\varepsilon_i}^{-1} \vdash W.\Xi_i \vdash \rho(G[G'/X]) \sim \rho(G[\alpha/X])$, *then*

(1) $(W, v_1, v_2) \in \mathcal{V}_{\rho'}[\![G]\!] \Rightarrow (W, \varepsilon_1 v_1 :: \rho(G[G'/X]), \varepsilon_2 v_2 :: \rho(G[G'/X])) \in \mathcal{T}_\rho[\![G[G'/X]]\!]$
(2) $(W, v_1, v_2) \in \mathcal{V}_\rho[\![G[G'/X]]\!] \Rightarrow (W, {\varepsilon_1}^{-1} v_1 :: \rho'(G), {\varepsilon_2}^{-1} v_2 :: \rho'(G)) \in \mathcal{T}_{\rho'}[\![G]\!]$

Observe that Lemma 10.4 is informally the combination of the compositionality and conversion lemma from Ahmed et al. [2017]. This lemma says that if α is synchronized in W, and X is bound to α, then (1) given two related values at type G, removing variable X by substitution (unsealing) yields related computations at type $G[G'/X]$; and (2) given two related values at type $G[G'/X]$ then substituting G' for X (sealing) yields related computations at type G. Note that the un(sealing) of type names is done via ascriptions, where evidences are constructed as in the reduction rule for type instantiations. Note that in order to respect termination sensitivity, we require that the evidences used to ascribe both values do not introduce new runtime errors. To do so, we build both evidences by lifting store information, similarly to computing the outer evidence ε_{out} (Section 8.2).

Almost all compatibility lemmas and the compositionality lemma rely on the fact that the ascription of two related values yields related terms.

Lemma 10.5 (Ascriptions Preserve Relations). *If* $(W, v_1, v_2) \in \mathcal{V}_\rho[\![G]\!]$, $\varepsilon \Vdash \Xi; \Delta \vdash G \sim G'$, $W \in \mathcal{S}[\![\Xi]\!]$, *and* $(W, \rho) \in \mathcal{D}[\![\Delta]\!]$, *then* $(W, \rho_1(\varepsilon)v_1 :: \rho(G'), \rho_2(\varepsilon)v_2 :: \rho(G')) \in \mathcal{T}_\rho[\![G']\!]$.

10.3 Parametricity vs. the DGG in GSF

We now give a different perspective from that presented in Section 9.1, regarding the violation of the general dynamic gradual guarantee (DGG, stated with respect to $\sqsubseteq$). More precisely, we show that the definition of parametricity for GSF (Section 10) is incompatible with the DGG. To do so, we again prove that there exists two terms in GSF, related by precision, whose behavior violates the DGG, but this time we do so with a proof of the intermediate results that is fully-driven by the definition of parametricity, and not by monotonicity of consistent transitivity. We present the proof sketch of the intermediate results in order to highlight the key properties that imply this incompatibility. This is particularly relevant because these properties also manifest in λB, for which the DGG has not been formally explored yet.

Recall from Section 9.1 that the term that helps us establish the violation of the DGG is $id_?$, a variant of the polymorphic identity function id_X whose term variable x is given the unknown type. This term always fails when fully applied.

Lemma 10.6. *For any* $\vdash v : ?$ *and* $\vdash G$, *we have* $(\Lambda X.\lambda x : ?.x :: X)\ [G]\ v \Downarrow$ **error**.

Let us consider $id_? \triangleq \Lambda X.\lambda x : ?.x :: X$, whose elaboration is $v_a = \langle \forall X.? \to X, \forall X.? \to X\rangle (\Lambda X.\lambda x : ?.\langle X, X\rangle x :: X) :: \forall X.? \to X$, and two different types Int and Bool, such that:

$$\cdot \triangleright v_a\ [\mathsf{Int}] \longmapsto$$
$$\alpha := \mathsf{Int} \triangleright \langle ? \to \alpha^{\mathsf{Int}}, ? \to \mathsf{Int}\rangle\big(\langle ? \to \alpha^{\mathsf{Int}}, ? \to \alpha^{\mathsf{Int}}\rangle(\lambda x : ?.\langle \alpha^{\mathsf{Int}}, \alpha^{\mathsf{Int}}\rangle x :: \alpha) :: ? \to \alpha\big) :: ? \to \mathsf{Int}$$

and

$$\cdot \triangleright v_a\ [\mathsf{Bool}] \longmapsto$$
$$\alpha := \mathsf{Bool} \triangleright \langle ? \to \alpha^{\mathsf{Bool}}, ? \to \mathsf{Bool}\rangle\big(\langle ? \to \alpha^{\mathsf{Bool}}, ? \to \alpha^{\mathsf{Bool}}\rangle(\lambda x : ?.\langle \alpha^{\mathsf{Bool}}, \alpha^{\mathsf{Bool}}\rangle x :: \alpha) :: ? \to \alpha\big) :: ? \to \mathsf{Bool}$$

If we consider the domain of the external evidences:

$$dom(\langle ? \to \alpha^{\mathsf{Int}}, ? \to \mathsf{Int}\rangle) = dom(\langle ? \to \alpha^{\mathsf{Bool}}, ? \to \mathsf{Bool}\rangle) = \langle ?, ?\rangle$$

and any pair of related values at type ?, then their ascription to ? using evidence $\langle ?, ?\rangle$ yields related values at type ?. In particular for $v_b = \langle \mathsf{Int}, \mathsf{Int}\rangle 1 :: ?$, $(W, v_b, v_b) \in \mathcal{V}[\![?]\!]$, as $\langle \mathsf{Int}, \mathsf{Int}\rangle \fatsemi \langle ?, ?\rangle = \langle \mathsf{Int}, \mathsf{Int}\rangle$ $(\langle ?, ?\rangle v_b :: ? \longmapsto v_b)$, we obtain that : $(\downarrow W \boxtimes (\alpha, \mathsf{Int}, \mathsf{Bool}, R), v_b, v_b) \in \mathcal{V}_{X\mapsto\alpha}[\![?]\!]$ for any $R \in \textsc{Rel}_{W.j}[\mathsf{Int}, \mathsf{Bool}]$. This fact is central to the proof of Lemma 10.6, and to prove it, we use a (rather technical) intermediate lemma, which crisply captures this idea in a more general setting using the term $(\Lambda X.\lambda x : ?.t)$, where t can be any term. Intuitively, consider one step of execution of the application of $(\Lambda X.\lambda x : ?.t)$ to two different arbitrary types, and the resulting outermost evidences ε_1 and ε_2. The ascription of any value v to ? using the domain of ε_1 and ε_2 yields two related computations. The intermediate lemma is formalized as follows:

Lemma 10.7. *Let* $\vdash (\Lambda X.\lambda x : ?.t) \rightsquigarrow v_a : \forall X.? \to X$ *and* $\vdash v \rightsquigarrow v_b : ?$. *Let* G_1 *and* G_2, *such that* $const(G_1) \neq const(G_2)$. *If* $\cdot \triangleright v_a\ [G_i] \longmapsto \alpha := G_i \triangleright \varepsilon_i v_i :: ? \to G_i$ *and* $\varepsilon_i \Vdash ? \to \alpha \sim ? \to G_i$, *then* $\forall W \in \mathcal{S}[\![\cdot]\!], \forall R \in \textsc{Rel}_{W.j}[G_1, G_2], (W \boxtimes (\alpha, G_1, G_2, R), dom(\varepsilon_1)v_b :: ?, dom(\varepsilon_2)v_b :: ?) \in \mathcal{T}_{X\mapsto\alpha}[\![?]\!]$.

We now show that instantiating $id_?$ to any arbitrary type such as Int, and applying it to any value of type ? such as $v_b = \langle \mathsf{Int}, \mathsf{Int}\rangle 1 :: ?$ necessarily leads to a runtime error (Lemma 10.6). For simplicity, we omit worlds in the development below. Consider the configuration $\cdot \triangleright v_a\ [\mathsf{Int}]\ v_b$, which steps to:

$$\alpha := \mathsf{Int} \triangleright (\langle ? \to \alpha^{\mathsf{Int}}, ? \to \mathsf{Int}\rangle\big(\langle ? \to \alpha^{\mathsf{Int}}, ? \to \alpha^{\mathsf{Int}}\rangle(\lambda x : ?.\langle \alpha^{\mathsf{Int}}, \alpha^{\mathsf{Int}}\rangle x :: \alpha) :: ? \to \alpha\big) :: ? \to \mathsf{Int})\ v_b$$

By the fundamental property (Theorem 10.1) on $id_?$ and 1, we know that:

(1) v_a is related to itself at type $\forall X.? \to X$, and then choosing $G_1 = \mathsf{Int}, G_2 = \mathsf{Bool}, R = \{(\langle \mathsf{Int}, \mathsf{Int}\rangle 1 :: \mathsf{Int}, \langle \mathsf{Bool}, \mathsf{Bool}\rangle \mathsf{true} :: \mathsf{Bool}\}$ we know that
$$(v_1, v_2) \in \mathcal{V}_{X\mapsto\alpha}[\![? \to \alpha]\!]$$
where $v_1 = \langle ? \to \alpha^{\mathsf{Int}}, ? \to \alpha^{\mathsf{Int}}\rangle(\lambda x : ?.\langle \alpha^{\mathsf{Int}}, \alpha^{\mathsf{Int}}\rangle x :: \alpha) :: ? \to \alpha$, and
$v_2 = \langle ? \to \alpha^{\mathsf{Bool}}, ? \to \alpha^{\mathsf{Bool}}\rangle(\lambda x : ?.\langle \alpha^{\mathsf{Bool}}, \alpha^{\mathsf{Bool}}\rangle x :: \alpha) :: ? \to \alpha$.
(2) $\langle \mathsf{Int}, \mathsf{Int}\rangle 1 :: ?$ is related to itself at type ?.

Then we notice that, by associativity of consistent transitivity, the pending redex is equivalent to:

$$\alpha := \mathsf{Int} \triangleright \langle \alpha^{\mathsf{Int}}, \mathsf{Int}\rangle\big(v_1(\langle ?, ?\rangle v_b :: ?)\big) :: \mathsf{Int}$$

By (2) and Lemma 10.7, $\langle ?, ?\rangle v_b :: ? \longmapsto v_b$ and $(v_b, v_b) \in \mathcal{V}_{X\mapsto\alpha}[\![?]\!]$. We instantiate the result in (1) $(v_1, v_2) \in \mathcal{V}_{X\mapsto\alpha}[\![? \to \alpha]\!]$ with arguments $(v_b, v_b) \in \mathcal{V}_{X\mapsto\alpha}[\![?]\!]$. But notice that $v_2\ v_b$ always fails (as α is instantiated to Bool not Int), therefore $v_1\ v_b$ must also fail and the result holds, otherwise v_1 and v_2 would not be related. Furthermore, let us assume (falsely) that $v_2\ v_b$ reduces to some

value. Then $\alpha := \mathsf{Int} \triangleright v_1\ v_b \longmapsto^* \Xi \triangleright \langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle 1 :: \alpha$, and $\alpha := \mathsf{Int} \triangleright v_2\ v_b \longmapsto^* \Xi \triangleright \langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle 1 :: \alpha$, then we would have to prove that $(\langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle 1 :: \alpha, \langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle 1 :: \alpha) \in \mathcal{V}_{X \mapsto \alpha}[\![\alpha]\!]$, i.e., $(\langle \mathsf{Int}, \mathsf{Int} \rangle 1 :: \mathsf{Int}, \langle \mathsf{Int}, \mathsf{Int} \rangle 1 :: \mathsf{Int}) \in R$ which is false as $R = \{(\langle \mathsf{Int}, \mathsf{Int} \rangle 1 :: \mathsf{Int}, \langle \mathsf{Bool}, \mathsf{Bool} \rangle \mathsf{true} :: \mathsf{Bool})\}$. Therefore, $v_1\ v_b$ ought to fail.

As a consequence of Lemma 10.6, the dynamic gradual guarantee is violated in GSF.

Corollary 10.8. *There exist* $\vdash t_1 : G$ *and* t_2, *such that* $t_1 \sqsubseteq t_2$, $t_1 \Downarrow v$ *and* $t_2 \Downarrow$ **error**.

Proof. Let $id_X \triangleq \Lambda X.\lambda x : X.x :: X$, and $id_? \triangleq \Lambda X.\lambda x : ?.x :: X$. By definition of precision, we have $id_X \sqsubseteq id_?$. Let $\vdash v : G$ and $\vdash v' : ?$, such that $v \sqsubseteq v'$. Pose $t_1 \triangleq id_X\ [G]\ v$ and $t_2 \triangleq id_?\ [G]\ v'$. By definition of precision, we have $t_1 \sqsubseteq t_2$. By evaluation, $t_1 \Downarrow v$. But by Lemma 10.6, $t_2 \Downarrow$ **error**. □

Interestingly, Lemma 10.6 holds irrespective of the actual choices for representing evidence in GSFε. The key reason is the logical interpretation of $\forall X.G$. Therefore, the incompatibility described here does not apply only to GSF but to other gradual languages that use similar logical relations such as λB: in fact, we have been able to prove that Lemmas 10.7 and 10.6 also hold in λB (by using conversions instead of evidences and using λB logical relations), so we conjecture that the reason of the incompatibility is the same as in GSF.

10.4 Gradual Free Theorems in GSF

The parametricity logical relation (Section 10) allows us to define notions of logical approximation ($\preceq$) and equivalence ($\approx$) that are sound with respect to contextual approximation ($\preceq^{ctx}$) and equivalence ($\approx^{ctx}$), and hence can be used to derive free theorems about well-typed GSF terms [Ahmed et al. 2017; Wadler 1989]. The definitions of contextual approximation and equivalence, and the soundness of the logical relation, are fairly standard. As shown by Ahmed et al. [2017], in a gradual setting, the free theorems that hold for System F are weaker, as they have to be understood "modulo errors and divergence". Ahmed et al. [2017] prove two such free theorems in λB. However, these free theorems only concern *fully static* type signatures. This leaves unanswered the question of what *imprecise* free theorems are enabled by gradual parametricity. To the best of our knowledge, this topic has not been formally developed in the literature so far, despite several claims about expected theorems, exposed hereafter.

Igarashi et al. [2017a] report that the System F polymorphic identity function, if allowed to be cast to $\forall X.? \to X$, would always trigger a runtime error when applied, suggesting that functions of type $\forall X.? \to X$ are always failing. Consequently, System F_G rejects such a cast by adjusting the precision relation (Section 3). But the corresponding free theorem—i.e., that applying any function of type $\forall X.? \to X$ either diverges or fails—is not proven. Also, Ahmed et al. [2011] declare that parametricity dictates that any value of type $\forall X.X \to ?$ is either constant or always failing or diverging (p.7). This gradual free theorem is not proven either. In fact, in both an older system [Ahmed et al. 2009b] and its newest version [Ahmed et al. 2017], as well as in System F_G, casting the identity function to $\forall X.X \to ?$ yields a function that returns *without errors*, though the returned value is still sealed, and as such is unusable (Section 3). The parametricity relation in GSF does not impose such behavior: it only imposes uniformity of behavior, including failure, of polymorphic terms, which leaves some freedom regarding when to fail. As we saw earlier, the unknown type can stand for any type, including any type variable. Consequently, in GSF, a function of type $\forall X.? \to X$ could behave like the identity function with type $\forall X.X \to X$, or as a function of type $\forall X.\mathsf{Int} \to X$ that always fails when applied, or a function that given a pair returns its first or second component with type $\forall X.(X \times X) \to X$, etc. In particular, we show next that ascribing the System F identity function to $\forall X.? \to X$ yields a function that behaves exactly as the identity function (and hence never fails).

The DGG$^{\leqslant}$-related Lemmas 9.4 and 9.7 help us prove that in GSF types $\forall X.? \to X$ and $\forall X.X \to ?$ are inhabited by non-constant, non-failing, parametricity-preserving terms. Observe that this result is a consequence of the fact that imprecise ascriptions are harmless in GSF.

In particular, both types admit the ascribed System F identity function, among many others (for instance, the polymorphic term $\Lambda X.\lambda x : X.\lambda f : X \to X.f\ x$ of type $\forall X.X \to (X \to X) \to X$ can also be ascribed to $\forall X.X \to ?$).

We formalize this using the following corollary:

Corollary 10.9. *Let t and v be static terms such that $\vdash t : \forall X.T$, $\vdash v : T'$, and $t[T']\ v \Downarrow v'$.*

(1) If $\forall X.T \sqsubseteq \forall X.X \to ?$ then $(t :: \forall X.X \to ?)[T']\ v \Downarrow v''$, and $v' \leqslant v''$.
(2) If $\forall X.T \sqsubseteq \forall X.? \to X$ then $(t :: \forall X.? \to X)[T']\ v \Downarrow v''$, and $v' \leqslant v''$.

Cheap Theorems. The intuition of $\forall X.? \to X$ denoting always-failing functions is not entirely misguided: in GSF, this result does hold *for a subset* of the terms of that type. This leads us to observe that we can derive "cheap theorems" with gradual parametricity: obtained not by looking only at the type, but by also considering the head constructors of a term. For instance:

Theorem 10.10. *Let $v \triangleq \Lambda X.\lambda x : ?.t$ for some t, such that $\vdash v : \forall X.? \to X$. Then for any $\vdash v' : G$, we either have $v\ [G]\ v' \Downarrow$ **error** or $v\ [G]\ v' \Uparrow$.*

This result is proven by exploiting the gradual parametricity result (Theorem 10.1). Note that what makes it a "free" theorem is that it holds independently of the body t, therefore *without having to analyze the whole term.* Not as good as a free theorem, but cheap. It is worth noting that although the external loss of precision is harmless, the internal loss of precision may change the expected behavior of a term. For example, the function $\Lambda X.\lambda x :?.t$ from Theorem 10.10 might be the imprecise identity function $\Lambda X.\lambda x :?.x :: X$. Therefore, we could expect that applied to a type and a value of the same type, it returns the same value; however, by Theorem 10.10, it always fails or diverges.

11 EMBEDDING DYNAMIC SEALING IN GSF

A gradual language is expected to cover a spectrum between two typing disciplines, such as simple static typing and dynamic typing. The static end of the spectrum is characterized by the conservative extension results [Siek et al. 2015a], which we have established for GSF with respect to System F (Propositions 6.10 and 8.5). The dynamic end of the spectrum is typically characterized by an *embedding* from the considered dynamic language to the gradual language [Siek et al. 2015a]. For instance, in the case of GTLC [Siek and Taha 2006], the dynamic language is an untyped lambda calculus with primitives.

In this section, we study the "dynamic end" of GSF. Unsurprisingly, GSF can embed an untyped lambda calculus with primitives, called λ_{dyn} (Section 11.1). More interestingly, we highlight the expressive power of the underlying type name generation mechanism of GSF by proving that GSF can faithfully embed an untyped lambda calculus with *dynamic sealing*, λ_{seal} (Section 11.2). This language, also known as the cryptographic lambda calculus, was first studied in a typed version by Pierce and Sumii [2000], and then untyped [Sumii and Pierce 2004]. One of their objectives was to study whether dynamic sealing could be used in order to dynamically impose parametricity via a compiler from System F to λ_{seal}. Recently, Devriese et al. [2018] prove that such a compiler is not fully abstract, i.e., compiled System F equivalent terms are not contextually equivalent in λ_{seal}. Nevertheless, the dynamic sealing mechanism of λ_{seal} to protect abstract data, and its relation to gradual parametricity, is an interesting question. We define an embedding of λ_{seal} terms into GSF (Section 11.3), and prove that this embedding is semantic preserving (Section 11.4).

11.1 Embedding a Dynamically-Typed Language in GSF

The essence of embedding a dynamically-typed language in a gradual language is to ascribe every introduction form with the unknown type [Siek and Taha 2006; Siek et al. 2015a]. For instance, the expression (1 2) from a dynamically-typed language can be embedded as (1 :: ?) (2 :: ?). Observe that not adding the ascriptions would yield an ill-typed term, as per the conservative extension result with respect to the static typing discipline. Let us call λ_{dyn} the dynamically-typed lambda calculus with pairs and primitives. We aim at an embedding of λ_{dyn} that preserves termination, divergence, and failure. We will establish this result formally as a corollary of a stronger result for the extended language with dynamic sealing (see Corollary 11.8).

The embedding of λ_{dyn} terms into GSF is defined as:

$$\begin{array}{rlrl}
\lceil b \rceil &= b :: ? & \lceil x \rceil &= x \\
\lceil \lambda x.t \rceil &= (\lambda x : ?. \lceil t \rceil) :: ? & \lceil t_1\ t_2 \rceil &= \mathsf{let}\ x : ? = \lceil t_1 \rceil\ \mathsf{in}\ \mathsf{let}\ y : ? = \lceil t_2 \rceil\ \mathsf{in}\ x\ y \\
\lceil \langle t_1, t_2 \rangle \rceil &= \langle \lceil t_1 \rceil, \lceil t_2 \rceil \rangle :: ? & \lceil \pi_1(t) \rceil &= \pi_1(\lceil t \rceil) \\
\lceil op(\overline{t}) \rceil &= \mathsf{let}\ \overline{x} : ? = \lceil \overline{t} \rceil\ \mathsf{in}\ op(\overline{x}) :: ? & \lceil \pi_2(t) \rceil &= \pi_2(\lceil t \rceil)
\end{array}$$

The only novelty here with respect to prior work is that the embedding produces application terms in A-normal form in order to ensure that embedded terms behave as expected. For example, the term $(1\ \Omega)$, with $\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$, diverges in the dynamically-typed language. But if we would embed an application $\lceil t_1\ t_2 \rceil$ simply as $\lceil t_1 \rceil\ \lceil t_2 \rceil$, because evidence combination would detect the underlying type error before reducing the application. Note that this precaution is unnecessary for pairs, because there are no typing constraints between both components. To better understand the need to use the A-normal form, we present both translations (with and without A-normal form) of program $(1\ \Omega)$ to GSF and then their translations to GSFε. For simplicity, let us suppose that $\lceil \Omega \rceil = \Omega$ and, 1_ε and Ω_ε are GSFε terms, where $\vdash \Omega \rightsquigarrow \Omega_\varepsilon : ?$, $\vdash (1 :: ?) \rightsquigarrow 1_\varepsilon : ?$ and $1_\varepsilon = \varepsilon_{\mathsf{Int}} 1 :: ?$.

$\lceil t_1\ t_2 \rceil =$	
GSF	$\mathsf{let}\ x : ? = 1 :: ?\ \mathsf{in}\ \mathsf{let}\ y : ? = \Omega\ \mathsf{in}\ x\ y$
GSFε	$\mathsf{let}\ x : ? = 1_\varepsilon\ \mathsf{in}\ \mathsf{let}\ y : ? = \Omega_\varepsilon\ \mathsf{in}\ (\varepsilon_{?\to?} x :: ? \to ?)\ y$
$\longmapsto \cdot \triangleright$	$\mathsf{let}\ y : ? = \Omega_\varepsilon\ \mathsf{in}\ (\varepsilon_{?\to?} 1_\varepsilon :: ? \to ?)\ y$
	diverges reducing Ω_ε

$\lceil t_1\ t_2 \rceil =$	$\lceil t_1 \rceil\ \lceil t_2 \rceil$
GSF	$(1 :: ?)\ \Omega$
GSFε	$(\varepsilon_{?\to?} 1_\varepsilon :: ? \to ?)\ \Omega_\varepsilon$
$\longmapsto \cdot \triangleright$	**error**
	$(\varepsilon_{\mathsf{Int}} \fatsemi \varepsilon_{?\to?})$ fails

11.2 The Cryptographic Lambda Calculus λ_{seal}

The cryptographic lambda calculus λ_{seal} is an extension of λ_{dyn} with primitives for protecting abstract data by sealing [Sumii and Pierce 2004]. Figure 15 presents the syntax and dynamic semantics of the λ_{seal} language we consider here, which is a simplified variant of that of Sumii and Pierce [2004]. In addition to standard terms, which correspond to λ_{dyn}, the λ_{seal} language introduces four new syntactic constructs dedicated to sealing. First, the term $\nu x.t$ generates a fresh key to seal and unseal values, bound to x in the body t. Seals, denoted by the metavariable σ, are values tracked in the set of allocated seals μ. The sealing construct $\{t_1\}_{t_2}$ evaluates t_1 to a value v and t_2 to a seal σ, and seals v with σ. Term $\mathsf{let}\ \{x\}_{t_1} = t_2\ \mathsf{in}\ t$ is for unsealing. At runtime, t_1 should evaluate to a seal σ and t_2 to a sealed value $\{v\}_{\sigma'}$. If $\sigma = \sigma'$, unsealing succeeds and t is evaluated with x bound to v. Otherwise, unsealing fails, producing a runtime sealing error **unseal_error**.[17]

To illustrate, consider the following term:

$$\nu x.\nu y.\lambda b.\mathsf{let}\ \{n\}_x = \{1\}_{(\mathsf{if}\ b\ \mathsf{then}\ x\ \mathsf{else}\ y)}\ \mathsf{in}\ n + 1$$

[17] The original term for unsealing in λ_{seal} has the syntax $\mathsf{let}\ \{x\}_{t_1} = t_2\ \mathsf{in}\ t\ \mathsf{else}\ t_3$; if the unsealing fails, reduction recovers from error evaluating t_3. To be able to encode such a construct, we would need to extend GSF with error handling.

$$x \in \textsc{Var}, \sigma \in \textsc{Seal}, \mu \subset \textsc{Seal}$$

$$\begin{array}{rcll} t & ::= & b \mid \lambda x.t \mid \langle t, t\rangle \mid x \mid t\ t \mid \pi_i(t) \mid op(\overline{t}) \mid \nu x.t \mid \{t\}_t \mid \mathsf{let}\ \{x\}_t = t\ \mathsf{in}\ t \mid \sigma & \text{(terms)} \\ v & ::= & b \mid \lambda x.t \mid \langle v, v\rangle \mid \{v\}_\sigma \mid \sigma & \text{(values)} \end{array}$$

$\boxed{t \parallel \mu \longrightarrow t \parallel \mu}$ **Notion of reduction**

$$(\lambda x.t)\ v \parallel \mu \longrightarrow t[v/x] \parallel \mu \qquad \pi_i(\langle v_1, v_2\rangle) \parallel \mu \longrightarrow v_i \parallel \mu \qquad op(\overline{v}) \parallel \mu \longrightarrow \delta(op, \overline{v}) \parallel \mu$$

$$\nu x.t \parallel \mu \longrightarrow t[\sigma/x] \parallel \mu, \sigma \quad \text{where } \sigma \notin \mu \qquad \{v\}_{v'} \parallel \mu \longrightarrow \textbf{seal\_type\_error} \quad \text{where } \nexists \sigma.\ v' \equiv \sigma$$

$$\mathsf{let}\ \{x\}_\sigma = \{v\}_{\sigma'}\ \mathsf{in}\ t \parallel \mu \longrightarrow \begin{cases} t[v/x] \parallel \mu & \sigma \equiv \sigma' \\ \textbf{unseal\_error} & \sigma \not\equiv \sigma' \end{cases}$$

$$\mathsf{let}\ \{x\}_\sigma = v\ \mathsf{in}\ t \parallel \mu \longrightarrow \textbf{seal\_type\_error} \quad \text{where } \nexists \sigma', v'.\ v \equiv \{v'\}_{\sigma'}$$

$$\mathsf{let}\ \{x\}_v = v'\ \mathsf{in}\ t \parallel \mu \longrightarrow \textbf{seal\_type\_error} \quad \text{where } \nexists \sigma.\ v \equiv \sigma$$

$\boxed{t \parallel \mu \longmapsto t \parallel \mu}$ **Evaluation frames and reduction**

$$\begin{array}{rcll} f & ::= & \square\ t \mid v\ \square \mid \langle \square, t\rangle \mid \langle v, \square\rangle \mid \pi_i(\square) \mid \{\square\}_t \mid \{v\}_\square \mid op(\overline{v}, \square, \overline{t}) & \text{(term frames)} \\ & & \mid \mathsf{let}\ \{x\}_\square = t\ \mathsf{in}\ t \mid \mathsf{let}\ \{x\}_v = \square\ \mathsf{in}\ t & \end{array}$$

$$\frac{t \parallel \mu \longrightarrow t' \parallel \mu'}{t \parallel \mu \longmapsto t' \parallel \mu'} \qquad \frac{t \parallel \mu \longmapsto t' \parallel \mu'}{f[t] \parallel \mu \longmapsto f[t'] \parallel \mu'}$$

$$\frac{t \parallel \mu \longrightarrow \textbf{error}}{t \parallel \mu \longmapsto \textbf{error}} \qquad \frac{t \parallel \mu \longmapsto \textbf{error}}{f[t] \parallel \mu \longmapsto \textbf{error}}$$

Fig. 15. λ_{seal}: Untyped lambda calculus with sealing.

This term first generates two fresh seals x and y, and then defines a function that receives a boolean b and attempts to unseal a sealed value. The value 1 is sealed using either x or y, depending on b, and unsealed with x. If the function is applied to true, unsealing succeeds because the seals coincide, and the function returns 2. Otherwise, unsealing fails, and an **unseal_error** is raised.

Overall, we can distinguish three kinds of runtime errors in λ_{seal} (**error**): in addition to unsealing errors, **unseal_error**, there are two kinds of runtime *type* errors, hereafter called **type_error**—omitted in Figure 15—and **seal_type_error**. The former corresponds to standard runtime type errors such as applying a non-function, and can happen in λ_{dyn}. The latter is specific to λ_{seal}, and corresponds to expressions that do not produce seals when expected, such as $\{1\}_2$.

11.3 Embedding λ_{seal} in GSF

We now present a semantic-preserving embedding of λ_{seal} terms in GSF. The embedding relies on a general seal/unseal generator, expressed as a GSF term. This term, called *su* hereafter, is a polymorphic pair of two functions, of type $\forall X.(X \to ?) \times (? \to X)$, instantiated at the unknown type, and ascribed to the unknown type:

$$su \equiv (\Lambda X.\langle(\lambda x : X.x :: ?), (\lambda x : ?.x :: X)\rangle)\ [?] :: ?$$

When evaluated, the type application generates a fresh type name, simulating the seal generation of λ_{seal}'s term $\nu x.t$. Then the first component of the pair represents a sealing function, while the second component represents an unsealing function, which can only successfully be applied to values sealed with the first component. We write su^σ to denote a particular value resulting from the evaluation of the term *su*, where the type name σ is generated and stored in Ξ. Crucially, a

$$\lceil b \rceil = b :: ? \qquad \lceil x \rceil = x$$
$$\lceil \lambda x.t \rceil = (\lambda x : ?.\lceil t \rceil) :: ? \qquad \lceil t_1\ t_2 \rceil = \text{let } x : ? = \lceil t_1 \rceil \text{ in let } y : ? = \lceil t_2 \rceil \text{ in } x\ y$$
$$\lceil \langle t_1, t_2 \rangle \rceil = \langle \lceil t_1 \rceil, \lceil t_2 \rceil \rangle :: ? \qquad \lceil \pi_1(t) \rceil = \pi_1(\lceil t \rceil)$$
$$\lceil op(\overline{t}) \rceil = \text{let } \overline{x} : ? = \lceil \overline{t} \rceil \text{ in } op(\overline{x}) :: ? \qquad \lceil \pi_2(t) \rceil = \pi_2(\lceil t \rceil)$$

$$\lceil \nu x.t \rceil = \text{let } x : ? = su \text{ in } \lceil t \rceil$$
$$\lceil \{t_1\}_{t_2} \rceil = \text{let } x : ? = \lceil t_1 \rceil \text{ in let } y : ? = \lceil t_2 \rceil \text{ in } \pi_1(y)\ x$$
$$\lceil \text{let } \{z\}_{t_1} = t_2 \text{ in } t_3 \rceil = \text{let } x : ? = \lceil t_1 \rceil \text{ in let } y : ? = \lceil t_2 \rceil \text{ in let } z : ? = \pi_2(x)\ y \text{ in } \lceil t_3 \rceil$$
$$\text{where } su \equiv ((\Lambda X.\langle(\lambda x : X.x :: ?), (\lambda x : ?.x :: X)\rangle)\ [?]) :: ?$$

Fig. 16. Embedding λ_{seal} in GSF.

value that passed through $\pi_1(su^\sigma)$ is sealed, and can only be observed after passing through the unsealing function $\pi_2(su^\sigma)$. Trying to unseal it with a different function results in a runtime error.

Embedding Translation. Figure 16 defines the embedding from λ_{seal} to GSF. The cases unrelated to sealing are as presented in Section 11.1. The crux of the embedding is in the use of the term *su*. A seal generation term $\nu x.t$ is embedded into GSF by let-binding the variable x to the term *su*, whose value su^σ will be substituted in the translation of t. Recall that the first component of the pair su^σ is used for sealing, and the second one for unsealing. Therefore, the sealing operation $\{t_1\}_{t_2}$ is embedded by let-binding the translations of t_1 and t_2 to fresh variables x and y, and applying the first component of y (the sealing function) to x (the value to be sealed). Likewise, an unsealing $\text{let } \{z\}_{t_1} = t_2 \text{ in } t_3$ is embedded by binding the translation of t_1 and t_2 to fresh variables x and y, then unsealing y using the second component of x (the unsealing function), and binding the result to z, for use in the translation of the term t_3. The use of A-normal forms in the embedding of sealing and unsealing is required because both are eventually interpreted as function applications, so the precaution discussed in Section 11.1 applies. Finally, note that because seals σ cannot appear in source text, so the translation need not consider them.

Illustration. As an example, the embedding of the λ_{seal} term $\nu x.\nu y.\text{let } \{n\}_x = \{1\}_x \text{ in } n + 1$ is the following GSF term:

$$\begin{array}{l}\text{let } x : ? = su \text{ in}\\ \text{let } y : ? = su \text{ in}\\ \text{let } u : ? = x \text{ in}\\ \text{let } z : ? = (\text{let } n_1 : ? = 1 \text{ in let } s : ? = x \text{ in } \pi_1(s)\ n_1) \text{ in}\\ \text{let } n : ? = \pi_2(u)\ z \text{ in } n + 1\end{array}$$

The following reduction trace shows the most critical steps of the program above. We define su_ε as the translation of *su* to GSFε, and su_ε^σ is the value of su_ε, where a fresh seal σ is generated. Note that we omit some trivial evidences and type annotations for readability. This program generates two fresh type names (σ and σ'), reducing the first *su* to su_ε^σ and the second one to $su_\varepsilon^{\sigma'}$. Then, after a few substitution steps, the first component of su_ε^σ is applied to 1, sealing the value, and then applies the second component of su_ε^σ, unsealing the sealed value. The whole program reduces to 2.

$$\begin{array}{lll}
\cdot & \triangleright\ \text{let } x = su_\varepsilon \text{ in let } y = su_\varepsilon \text{ in } \cdots & \text{initial program}\\
\longmapsto^*\ \sigma := ?, \sigma' := ? & \triangleright\ \text{let } u = su_\varepsilon^\sigma \text{ in } \cdots \text{ let } s = su_\varepsilon^\sigma \text{ in } \cdots & \sigma \text{ and } \sigma' \text{ are generated}\\
\longmapsto^*\ \sigma := ?, \sigma' := ? & \triangleright\ \text{let } z = \pi_1(su_\varepsilon^\sigma)(\varepsilon_{\text{Int}} 1 :: ?) \text{ in } \cdots & \text{substitution steps}\\
\longmapsto^*\ \sigma := ?, \sigma' := ? & \triangleright\ \text{let } n = \pi_2(su_\varepsilon^\sigma)(\langle \text{Int}, \sigma^{\text{Int}} \rangle 1 :: ?) \text{ in } n + 1 & \text{argument is sealed by } \sigma\\
\longmapsto^*\ \sigma := ?, \sigma' := ? & \triangleright\ \text{let } n = \varepsilon_{\text{Int}} 1 :: ? \text{ in } n + 1 & \text{unsealing eliminates } \sigma\\
\longmapsto^*\ \sigma := ?, \sigma' := ? & \triangleright\ \varepsilon_{\text{Int}} 2 :: ? &
\end{array}$$

If we slightly modify the previous λ_{seal} program by $vx.vy.\text{let } \{n\}_y = \{1\}_x \text{ in } n+1$, then unsealing fails with **unseal_error** because it uses a different seal to unseal than the one used to seal. The embedding of this λ_{seal} term in GSF is very similar to the previous one; the only difference is that, now, u is bound to y. The following reduction trace illustrates where the embedding of the λ_{seal} term fails. Note that the resulting value of $\pi_2(su_\varepsilon^{\sigma'})$ is $\langle ? \to \sigma'^{?}, ? \to ? \rangle(\lambda x : ?.\langle \sigma'^{?}, \sigma'^{?} \rangle x :: \sigma') :: ? \to ?$, where σ' is a type name and $\sigma'^{?}$ is an evidence. Then, the sealed value $\langle \mathsf{Int}, \sigma^{\mathsf{Int}} \rangle 1 :: ?$ is substituted in the body of the function, failing in the consistent transitivity $\langle \mathsf{Int}, \sigma^{\mathsf{Int}} \rangle \fatsemi \langle \sigma'^{?}, \sigma'^{?} \rangle$.

	$\cdot$	$\triangleright$	$\text{let } x = su_\varepsilon \text{ in let } y = su_\varepsilon \text{ in } \cdots$	initial program
$\longmapsto^*$	$\sigma := ?, \sigma' := ?$	$\triangleright$	$\text{let } u = su_\varepsilon^{\sigma'} \text{ in } \cdots \text{ let } s = su_\varepsilon^{\sigma} \text{ in } \cdots$	σ and σ' are generated
$\longmapsto^*$	$\sigma := ?, \sigma' := ?$	$\triangleright$	$\text{let } z = \pi_1(su_\varepsilon^{\sigma})(\varepsilon_{\mathsf{Int}} 1 :: ?) \text{ in } \cdots$	substitution steps
$\longmapsto^*$	$\sigma := ?, \sigma' := ?$	$\triangleright$	$\text{let } n = \pi_2(su_\varepsilon^{\sigma'})(\langle \mathsf{Int}, \sigma^{\mathsf{Int}} \rangle 1 :: ?) \text{ in } n+1$	argument is sealed by σ
$\longmapsto^*$			**error**	**error** unsealing by σ'

11.4 Semantic Preservation of the λ_{seal} Embedding in GSF

We now prove that the embedding of λ_{seal} into GSF is correct, namely that a λ_{seal} term and its translation to GSF behave similarly: either they both terminate to a value, both diverge, or both yield an error. Note that the semantic preservation theorem below only accounts for what we call *valid* λ_{seal} terms, i.e., terms that do not produce runtime type errors related to sealing, i.e., **seal_type_error**. We come back to this point at the end of this section. We write $t \Downarrow$ or $t \Downarrow v \parallel \mu$ if $t \parallel \cdot \longmapsto^* v \parallel \mu$, for some v and μ. We write $t \Uparrow$ if t diverges, and $t \Downarrow \textbf{error}$ if $t \parallel \cdot \longmapsto^* \textbf{error}$, where $\textbf{error} \triangleq \textbf{type\_error}$ or **unseal_error**. As before, we write $t \Downarrow$ if $\vdash t \rightsquigarrow t_\varepsilon : ?$ and $\cdot \triangleright t_\varepsilon \longmapsto^* \Xi \triangleright v$, for some v and Ξ.

Theorem 11.1 (Embedding of λ_{seal}). *Let t be a valid closed λ_{seal} term.*

a. $\vdash \lceil t \rceil : ?$
b. $t \Downarrow$ *implies* $\lceil t \rceil \Downarrow$
c. $t \Uparrow$ *implies* $\lceil t \rceil \Uparrow$
d. $t \Downarrow \textbf{error}$ *implies* $\lceil t \rceil \Downarrow \textbf{error}$

To prove Theorem 11.1, we use a simulation relation $\approx$ between λ_{seal} and GSFε, defined in Figure 17. The simulation relation $\mu; \Xi; \Gamma \vdash t \approx t_\varepsilon : ?$ uses a set of allocated seals μ by the reduction of the λ_{seal} term t. The GSFε term t_ε typechecks in the typing environment Γ where all variables have type unknown, and it typechecks and it is evaluated in the store Ξ with all its type names instantiated, also, to the unknown type. In all the rules of the simulation, we implicitly assume that μ and Ξ are synchronized, i.e., if $\sigma \in \mu$ then $\sigma := ? \in \Xi$. Rules whose names begin with (TR) relate a λ_{seal} term and its translation in GSFε, i.e., embedding first the λ_{seal} term into GSF, and then translating the resulting GSF term to a GSFε term. For instance, Rule (TRb) relates the λ_{seal} value 1 with the GSFε value $\varepsilon_{\mathsf{Int}} 1 :: ?$. Note that Rule (TRp) uses metavariable D to denote the possible types of GSFε raw values (u), obtained by the embedding: either a base type B, an unknown function type $? \to ?$, or a pair of raw values $D \times D$. Rule (TRsG) relates the seal generation term $vx.t$ with the GSFε term that let-binds the variable x to the term su_ε to be substituted in t'; it requires that the bodies of the seal generation and let-binding be related. The remaining rules, whose names begin with (R), help us keep terms related as they reduce. One of the most important rules is (Rsed2), which relates a λ_{seal} sealed value with a GSF value that has sealing evidence, where σ is a type name and σ^{E_2} is an evidence type. Rule (Rsed1) relates a sealed value $\{v_1\}_{v_2}$ with a GSFε term that takes the first component of v_2' (expected to be a su_ε^{σ} value related to the seal v_2), and applies it to v_1' related to v_1. Dually, Rule (Runs) relates a term for unsealing with a GSFε term that takes the second component of v_1' (expected to be a su_ε^{σ} value related to the seal v_1), and applies it

$$\text{(TRx)}\ \frac{x : ? \in \Gamma}{\mu; \Xi; \Gamma \vdash x \approx x : ?} \qquad \text{(TRb)}\ \frac{ty(b) = B}{\mu; \Xi; \Gamma \vdash b \approx \varepsilon_B b :: ? : ?} \qquad \text{(TRs)}\ \frac{\sigma \in \mu \qquad \sigma := ? \in \Xi}{\mu; \Xi; \Gamma \vdash \sigma \approx su_\varepsilon^\sigma : ?}$$

$$\text{(TRp)}\ \frac{\mu; \Xi; \Gamma \vdash v_1 \approx \varepsilon_{D_1} u_1 :: ? : ? \qquad \mu; \Xi; \Gamma \vdash v_2 \approx \varepsilon_{D_2} u_2 :: ? : ?}{\mu; \Xi; \Gamma \vdash \langle v_1, v_2 \rangle \approx \varepsilon_{D_1 \times D_2} \langle u_1, u_2 \rangle :: ? : ?}$$

$$\text{(TR}\lambda\text{)}\ \frac{\mu; \Xi; \Gamma, x : ? \vdash t_1 \approx t_2 : ?}{\mu; \Xi; \Gamma \vdash (\lambda x.t_1) \approx \varepsilon_{?\to?}(\lambda x.t_2) :: ? : ?} \qquad \text{(TRpt)}\ \frac{\mu; \Xi; \Gamma \vdash t_1 \approx t_1' : ? \qquad \mu; \Xi; \Gamma \vdash t_2 \approx t_2' : ?}{\mu; \Xi; \Gamma \vdash \langle t_1, t_2 \rangle \approx \varepsilon_{?\times?} \langle t_1', t_2' \rangle :: ? : ?}$$

$$\text{(R?)}\ \frac{\mu; \Xi; \Gamma \vdash t \approx t' : ?}{\mu; \Xi; \Gamma \vdash t \approx \varepsilon_? t' :: ? : ?} \qquad \text{(Rapp)}\ \frac{\mu; \Xi; \Gamma \vdash v_1 \approx v_1' : ? \qquad \mu; \Xi; \Gamma \vdash v_2 \approx v_2' : ?}{\mu; \Xi; \Gamma \vdash v_1\ v_2 \approx (\varepsilon_{?\to?} v_1' :: ? \to ?)\ v_2' : ?}$$

$$\text{(TRappL)}\ \frac{\mu; \Xi; \Gamma \vdash t_1 \approx t_1' : ? \qquad \mu; \Xi; \Gamma \vdash t_2 \approx t_2' : ?}{\mu; \Xi; \Gamma \vdash t_1\ t_2 \approx \text{let } x = t_1' \text{ in let } y = t_2' \text{ in } (\varepsilon_{?\to?} x :: ? \to ?)\ y : ?}$$

$$\text{(RappR)}\ \frac{\mu; \Xi; \Gamma \vdash v_1 \approx v_1' : ? \qquad \mu; \Xi; \Gamma \vdash t_2 \approx t_2' : ?}{\mu; \Xi; \Gamma \vdash v_1\ t_2 \approx \text{let } y = t_2' \text{ in } (\varepsilon_{?\to?} v_1' :: ? \to ?)\ y : ?}$$

$$\text{(TRpi)}\ \frac{\mu; \Xi; \Gamma \vdash t \approx t' : ?}{\mu; \Xi; \Gamma \vdash \pi_i(t) \approx \pi_i(\varepsilon_{?\times?} t' :: ? \times ?) : ?} \qquad \text{(TRsG)}\ \frac{\mu; \Xi; \Gamma, x : ? \vdash t \approx t' : ?}{\mu; \Xi; \Gamma \vdash \nu x.t \approx \text{let } x = su_\varepsilon \text{ in } t' : ?}$$

$$\text{(Rsed1)}\ \frac{\mu; \Xi; \Gamma \vdash v_1 \approx v_1' : ? \qquad \mu; \Xi; \Gamma \vdash v_2 \approx v_2' : ?}{\mu; \Xi; \Gamma \vdash \{v_1\}_{v_2} \approx (\varepsilon_{?\to?} \pi_1(\varepsilon_{?\times?} v_2' :: ? \times ?) :: ? \to ?)\ v_1' : ?}$$

$$\text{(TRsed1L)}\ \frac{\mu; \Xi; \Gamma \vdash t_1 \approx t_1' : ? \qquad \mu; \Xi; \Gamma \vdash t_2 \approx t_2' : ?}{\mu; \Xi; \Gamma \vdash \{t_1\}_{t_2} \approx \text{let } x = t_1' \text{ in let } y = t_2' \text{ in } (\varepsilon_{?\to?} \pi_1(\varepsilon_{?\times?} y :: ? \times ?) :: ? \to ?)\ x : ?}$$

$$\text{(Rsed1R)}\ \frac{\mu; \Xi; \Gamma \vdash v_1 \approx v_1' : ? \qquad \mu; \Xi; \Gamma \vdash t_2 \approx t_2' : ?}{\mu; \Xi; \Gamma \vdash \{v_1\}_{t_2} \approx \text{let } y = t_2' \text{ in } (\varepsilon_{?\to?} \pi_1(\varepsilon_{?\times?} y :: ? \times ?) :: ? \to ?)\ v_1' : ?}$$

$$\text{(Rsed2)}\ \frac{\mu; \Xi; \Gamma \vdash v \approx \langle E_1, E_2 \rangle u :: ? : ? \qquad \sigma \in \mu \qquad \sigma := ? \in \Xi}{\mu; \Xi; \Gamma \vdash \{v\}_\sigma \approx \langle E_1, \sigma^{E_2} \rangle u :: ? : ?}$$

$$\text{(Runs)}\ \frac{\mu; \Xi; \Gamma \vdash v_1 \approx v_1' : ? \qquad \mu; \Xi; \Gamma \vdash v_2 \approx v_2' : ? \qquad \mu; \Xi; \Gamma, z : ? \vdash t_3 \approx t_3' : ?}{\mu; \Xi; \Gamma \vdash \text{let } \{z\}_{v_1} = v_2 \text{ in } t_3 \approx \text{let } z = (\varepsilon_{?\to?} \pi_2(\varepsilon_{?\times?} v_1' :: ? \times ?) :: ? \to ?)\ v_2' \text{ in } t_3' : ?}$$

$$\text{(TRunsL)}\ \frac{\mu; \Xi; \Gamma \vdash t_1 \approx t_1' : ? \qquad \mu; \Xi; \Gamma \vdash t_2 \approx t_2' : ? \qquad \mu; \Xi; \Gamma, z : ? \vdash t_3 \approx t_3' : ?}{\mu; \Xi; \Gamma \vdash \text{let } \{z\}_{t_1} = t_2 \text{ in } t_3 \approx \text{let } x = t_1' \text{ in let } y = t_2' \text{ in let } z = (\varepsilon_{?\to?} \pi_2(\varepsilon_{?\times?} x :: ? \times ?) :: ? \to ?)\ y \text{ in } t_3' : ?}$$

$$\text{(RunsR)}\ \frac{\mu; \Xi; \Gamma \vdash v_1 \approx v_1' : ? \qquad \mu; \Xi; \Gamma \vdash t_2 \approx t_2' : ? \qquad \mu; \Xi; \Gamma, z : ? \vdash t_3 \approx t_3' : ?}{\mu; \Xi; \Gamma \vdash \text{let } \{z\}_{v_1} = t_2 \text{ in } t_3 \approx \text{let } y = t_2' \text{ in let } z = (\varepsilon_{?\to?} \pi_2(\varepsilon_{?\times?} v_1' :: ? \times ?) :: ? \to ?)\ y \text{ in } t_3' : ?}$$

Fig. 17. Simulation relation between λ_{seal} and GSFε terms.

to v_2' related to v_2 (expected to be a sealed value). Also, the rule requires the bodies t_3 and t_3 to be related.

We first establish a number of useful lemmas. First, all GSFε terms that are in the relation have type unknown, simulating the fact that they are related to untyped λ_{seal} terms.

Lemma 11.2. *If* $\mu; \Xi; \Gamma \vdash t \approx t_\varepsilon : ?$ *then* $\Xi; \Gamma \vdash t_\varepsilon : ?$.

Also, the relation $\approx$ guarantees that if we have a λ_{seal} value related to a GSFε term, then the latter reduces to a related value.

Lemma 11.3. *If* $\mu; \Xi \vdash v \approx t_\varepsilon : ?$ *, then there exists* v_ε *s.t.* $\Xi \triangleright t_\varepsilon \longmapsto^* \Xi \triangleright v_\varepsilon$*, and* $\mu; \Xi \vdash v \approx v_\varepsilon : ?$.

For example, we know by Rule (Rsed1L) that

$$\sigma; \sigma := ? \vdash \{1\}_\sigma \approx \text{let } x = \varepsilon_{\mathsf{Int}} 1 :: ? \text{ in let } y = su^\sigma_\varepsilon \text{ in } (\varepsilon_{?\to?}\pi_1(\varepsilon_{?\times?} y :: ? \times ?) :: ? \to ?)\ x : ?$$

Thus, we know that the GSFε term reduces to a value, in this case, $\langle \mathsf{Int}, \sigma^{\mathsf{Int}} \rangle 1 :: ?$.

Lemma 11.4 establishes substituting related values in related terms yields related terms.

Lemma 11.4. *If* $\mu; \Xi; \Gamma, x : ? \vdash t \approx t_\varepsilon : ?$ *and* $\mu; \Xi; \Gamma \vdash v \approx v_\varepsilon : ?$*, then* $\mu; \Xi; \Gamma \vdash t[v/x] \approx t_\varepsilon[v_\varepsilon/x] : ?$.

Lemma 11.5 shows that the relation $\approx$ simulates both the notions of reduction $\longrightarrow$ and $\longrightarrow$, and the reduction relations $\longmapsto$ and $\longmapsto$, including error cases. Note that a single step of reduction in λ_{seal} can be simulated by several reduction steps in GSFε, hence the use of $\longmapsto^*$ in the conclusions of the lemma cases. For example, we have $\mu; \Xi \vdash \pi_1(\langle 1, 2\rangle) \approx \pi_1(\varepsilon_{?\times?}(\varepsilon_{\mathsf{Int}\times\mathsf{Int}}\langle 1, 2\rangle :: ?) :: ? \times ?) : ?$, and the GSF$\varepsilon$ term needs to reduce inside the frame $\pi_1(\square)$ before eliminating the projection like the λ_{seal} term.

Lemma 11.5. *Suppose that* t *is a term of* λ_{seal}*,* t_ε *is a term from GSF*ε *and* $\mu; \Xi \vdash t \approx t_\varepsilon : ?$.

(a) If $t \parallel \mu \longrightarrow t' \parallel \mu'$*, then there exists* t'_ε *s.t.* $\Xi \triangleright t_\varepsilon \longmapsto^* \Xi' \triangleright t'_\varepsilon$ *and* $\mu'; \Xi' \vdash t' \approx t'_\varepsilon : ?$
(b) If $t \parallel \mu \longrightarrow$ **error**, *then* $\Xi \triangleright t_\varepsilon \longmapsto^*$ **error**
(c) If $t \parallel \mu \longmapsto t' \parallel \mu'$*, then there exists* t'_ε *s.t.* $\Xi \triangleright t_\varepsilon \longmapsto^* \Xi' \triangleright t'_\varepsilon$ *and* $\mu'; \Xi' \vdash t' \approx t'_\varepsilon : ?$
(d) If $t \parallel \mu \longmapsto$ **error**, *then* $\Xi \triangleright t_\varepsilon \longmapsto^*$ **error**

Proof. The proof is by induction on $\mu; \Xi \vdash t \approx t_\varepsilon : ?$ and by analysis of the different cases.

Case ((a)). Most of the cases use Lemma 11.3, Lemma 11.4, and the consistent transitivity relation.

Case ((b)). Most of the cases use Lemma 11.3 and the consistent transitivity relation.

Case ((c)). The proof follows by cases analysis on $t \parallel \mu \longmapsto t' \parallel \mu'$and from Case ((a)).

Case ((d)). The proof follows by cases analysis on $t \parallel \mu \longmapsto$ **error** and from Case ((b)). □

The main property of the relation $\approx$ is that related terms behave similarly:

Lemma 11.6. *If* $\vdash t \approx t_\varepsilon : ?$ *then*

- $t \Downarrow v \parallel \mu$ *implies* $\cdot \triangleright t_\varepsilon \longmapsto^* \Xi \triangleright v_\varepsilon$*, where* $\mu; \Xi \vdash v \approx v_\varepsilon : ?$.
- $t \Uparrow$ *implies* t_ε *diverges.*
- $t \Downarrow$ **error** *implies* $\cdot \triangleright t_\varepsilon \longmapsto^*$ **error**.

Proof. The proof is by case analysis on the reduction of t.

- Suppose $t \Downarrow v \parallel \mu$. Then $\cdot \triangleright t_\varepsilon \longmapsto^* \Xi \triangleright v_\varepsilon$ and $\mu; \Xi \vdash v \approx v_\varepsilon : ?$ by Lemmas 11.3 and 11.5((c)).
- Suppose $t \Uparrow$. Then t_ε diverges by Lemma 11.5((c)).
- Suppose $t \Downarrow$ **error**, then $\cdot \triangleright t_\varepsilon \longmapsto^*$ **error** by Lemma 11.5((c) and (d)). □

Finally, a λ_{seal} term and its embedding into GSFε are related.

Lemma 11.7. *If* $\vdash \lceil t \rceil \rightsquigarrow t_\varepsilon : ?$*, then* $\vdash t \approx t_\varepsilon : ?$.

Semantics preservation (Theorem 11.1) follows from Lemmas 11.6 and 11.7.

Leaking the Encoding. As mentioned earlier, the semantic preservation result does not account for λ_{seal} terms that can raise runtime seal type errors, **seal_type_error**. The reason is that, without further caution, the encoding of seals as pairs of functions could be abused. For instance, the

term $\mathsf{let}\ \{y\}_{\langle \lambda x.x, \lambda x.x \rangle} = 1\ \mathsf{in}\ y$ raises a **seal_type_error** in λ_{seal}, because the expression that is supposed to produce a seal produces a pair of functions. Nevertheless, the embedding of this term in GSF reduces to 1. To properly deal with such cases—and therefore obtain a semantic preservation statement with equivalences instead of implications—would require introducing a primitive way of distinguishing "proper seals" produced by the translation from standard pairs of functions. A direct solution would be to exploit the data abstraction capabilities of System F, and hence GSF. Of course, in a statically-typed version of λ_{seal} [Pierce and Sumii 2000], this problem is sidestepped because a **seal_type_error** can never occur at runtime.

Embedding of the Dynamically-Typed Language. Finally, a direct consequence of the semantics preservation theorem is that the embedding of λ_{dyn} is also correct; in fact the embedding result holds as stated by Siek et al. [2015a] (Theorem 2), with equivalences instead of implications:

Corollary 11.8 (Embedding of λ_{dyn}). *Let t be a closed λ_{dyn} term.*

(a) $\vdash \lceil t \rceil : ?$
(b) $t \Downarrow$ *if and only if* $\lceil t \rceil \Downarrow$
(c) $t \Uparrow$ *if and only if* $\lceil t \rceil \Uparrow$

This result follows from Theorem 11.1 combined with the fact that a **seal_type_error** simply cannot occur in λ_{dyn}, which has no sealing-related terms.

12 GRADUAL EXISTENTIAL TYPES IN GSF

Existential types are the foundation of data abstraction and information hiding: concrete representations of abstract data types are elements of existential types [Mitchell and Plotkin 1988; Pierce 2002]. It is well known that existential types can be encoded in terms of universal types [Pierce 2002]. However, several polymorphic languages [Ahmed 2006; Ahmed et al. 2009a; Neis et al. 2009] include both universal and existential types primitively, instead of relying on the encoding. The reason is that proving certain properties, such as representation independence results, is much simpler with direct support for existential types.

Although some efforts have already been developed to protect data abstraction in a dynamically-typed language [Abadi et al. 1995; Rossberg 2003; Sumii and Pierce 2004; Wadler 2017], prior work on gradual parametric polymorphism leaves the treatment of existential types as future work [Ahmed et al. 2017; Toro et al. 2019]. In this section, we present an extension of GSF with existential types, dubbed GSF$^{\exists}$. We first briefly review existential types (Section 12.1) and why a direct treatment is preferable to an encoding (Section 12.2). We then informally introduce gradual existential types in action (Section 12.3) before formally developing GSF$^{\exists}$ (Section 12.4). Finally, we discuss the metatheory of GSF$^{\exists}$ (Section 12.5).

12.1 Existential Types in a Nutshell

An ***abstract data type*** (**ADT** for short) guarantees that a client can neither guess nor depend on its implementation [Mitchell and Plotkin 1988; Reynolds 1983]. Formally, an ADT consists of a type name A, a concrete representation type T, implementations of some operations for creating, querying and manipulating values of type T, and an abstraction boundary enclosing the representation and operations [Pierce 2002]. Thus, an ADT provides a public name to a type but hides its representation. The *representation independence* property for an ADT establishes that we can change its representation without affecting clients. This property is a particularly useful application of relational parametricity [Reynolds 1983]; we can show that two different implementations of an ADT are contextually equivalent so long as there exists a relation between their concrete type representations that is preserved by their operations.

Data abstraction is formalized by extending System F with existential types, of the form $\exists X.T$. Elements of an existential type are usually called packages, written $\mathsf{pack}\langle T', t\rangle$ as $\exists X.T$, where T' is the hidden representation type and the term component t has type $T[T'/X]$. The existential elimination construct $\mathsf{unpack}\langle X, x\rangle = t_1 \text{ in } t_2$ allows the components of the package to be accessed by a client, keeping the actual representation type hidden. Packages with different hidden representation types can inhabit the same existential type. Thus, we can implement an ADT in different ways, creating different existential packages.

For instance, consider a semaphore ADT with three operations: *bit* to create a semaphore, *flip* to produce a semaphore in the inverted state, and *read* to consult the state of the semaphore, as a Bool. We can encode such an ADT as an existential type with a triple:

$$\mathit{Sem} \equiv \exists X.X \times (X \to X) \times (X \to \mathsf{Bool})$$

Alternatively, for readability, we can use a hypothetical record syntax:

$$\mathit{Sem} \equiv \exists X.\{\mathit{bit} : X, \mathit{flip} : X \to X, \mathit{read} : X \to \mathsf{Bool}\}$$

Below are two equivalent implementations of this *Sem* ADT:

$$\begin{array}{ll} s_1 \equiv \mathsf{pack}\langle \mathsf{Bool}, v_1\rangle \text{ as } \mathit{Sem} & \mathit{where}\ v_1 \equiv \{\mathit{bit} = \mathsf{true}, \mathit{flip} = (\lambda x : \mathsf{Bool}.\neg\, x), \mathit{read} = (\lambda x : \mathsf{Bool}.x)\} \\ s_2 \equiv \mathsf{pack}\langle \mathsf{Int}, v_2\rangle \text{ as } \mathit{Sem} & \mathit{where}\ v_2 \equiv \{\mathit{bit} = 1,\ \mathit{flip} = (\lambda x : \mathsf{Int}.1 - x),\ \mathit{read} = (\lambda x : \mathsf{Int}.0 < x)\} \end{array}$$

In the first implementation, the concrete representation type is Bool, and in the second it is Int. The representation and operations of the *Sem* ADT are abstract to a client, in the sense that the representation of *bit* is hidden, and it can only be manipulated and queried by the operations *flip* and *read* . For instance, if we have the expression $\mathsf{unpack}\langle X, x\rangle = s \text{ in } t$, where s is an implementation of *Sem*, we can do $(x.\mathit{read}\ (x.\mathit{flip}\ x.\mathit{bit}))$ in the expression t, but $(x.\mathit{read}\ (x.\mathit{flip}\ \mathsf{true}))$ or $x.\mathit{bit} == \mathsf{true}$ are invalid programs that do not typecheck. Note that untyped versions of these programs would run normally with s_1, but they would crash with s_2.

12.2 Existential Types: Primitive or Encoded?

Existential types are closely connected with universal types, and in fact they can simply be *encoded* in terms of universal types, using the following encoding [Harper 2012]:

$$\begin{array}{rcl} \exists X.T & \equiv & \forall Y.(\forall X.T \to Y) \to Y \\ \mathsf{pack}\langle T', t\rangle \text{ as } \exists X.T & \equiv & \Lambda Y.\lambda f : (\forall X.T \to Y).f\ [T']\ t \\ \mathsf{unpack}\langle X, x\rangle = t_1 \text{ in } t_2 & \equiv & t_1\ [T_2]\ (\Lambda X.\lambda x : T.t_2) \ \text{ where } \ \Delta; \Gamma \vdash t_1 : \exists X.T \ \text{ and } \ \Delta, X; \Gamma, x : T \vdash t_2 : T_2 \end{array}$$

The intuition behind this encoding is that an existential type is viewed as a universal type taking the overall result type Y, followed by a polymorphic function representing the client with result type Y, and yielding a value of type Y as result. A package is a polymorphic function taking the client as argument, and unpacking corresponds to applying this polymorphic function.

Therefore, to study gradual existential types in GSF, one could simply adopt this encoding. However, if we want to reason about interesting properties such as representation independence and free theorems, it is preferable to give meaning to existential types directly.

The benefit of a direct treatment of existential types can already be appreciated in the fully-static setting, with the simple examples of packages s_1 and s_2 above. Suppose we want to show that s_1 and s_2 are contextually equivalent, i.e., indistinguishable by any context. To prove this equivalence, it is sufficient to show that the packages are related according to a parametricity logical relation that is sound with respect to contextual equivalence [Reynolds 1983]. Using the direct interpretation of

existential types, such a proof is considerably easier and more intuitive than using their universal encodings.[18]

The additional complexity of reasoning about existential types via their universal encoding hardly scales to more involved examples. For instance, Ahmed et al. [2009a] prove challenging cases of equivalences in the presence of abstract data types and mutable references, where the encoding would have been a liability; hence their choice of supporting existential types directly. Considering that the GSF logical relation also involves a number of technicalities (evidence, worlds, etc.), providing direct support for existentials is all the more appealing.

12.3 Gradual Existential Types in GSF$^{\exists}$

In this section, we show some illustrative examples of gradual existential types in action, highlighting their benefits and expected properties when type imprecision is involved. In particular, we want to dynamically preserve the information hiding property presumed for abstract data types.

Typed-Untyped Interoperability. Gradual existential types allow programmers to embed an untyped implementation of a library as a static ADT, by picking the unknown type as the hidden representation type. For instance, if v_3 is an untyped record, then s_3 below is a gradually well-typed implementation of the *Sem* ADT. The translation $\lceil\cdot\rceil$ embeds untyped terms in the gradual language, basically by introducing ? on all binders and constants [Siek and Taha 2006].

$$\begin{array}{l}\text{let } v_3 = \{bit = 1, flip = (\lambda x.1 - x), read = (\lambda x.0 < x)\} \text{ in}\\ \text{let } s_3 = \mathsf{pack}\langle ?, \lceil v_3 \rceil\rangle \text{ as } Sem \text{ in } C[s_3]\\ \text{where } C \equiv \mathsf{unpack}\langle X, x\rangle = \square \text{ in } (x.read\ (x.flip\ x.bit))\end{array}$$

The package s_3 is essentially a version of the package s_2 where types have been erased (replaced with the unknown type). As illustrated later (Section 12.5), one can prove in GSF$^{\exists}$ that s_3 is contextually equivalent to s_2 (and hence to s_1 as well), using a direct interpretation of gradual existential types. The static client or context C, given a package implementation of the *Sem* ADT, changes the state of the semaphore and then reads the state. The whole example runs without error, producing false as the final result.

Of course, we could have associated a package implementation that does not respect the ADT signature. For instance, we define v_3' as a variant of v_3, where *flip* has type ? $\to$ Bool. We obtain the package s_3', which is still gradually well-typed. However, using the package with client C results in a runtime type error. The runtime error happens when the $\neg$ operator is applied to $x.bit$, because $\neg$ expects a Bool argument, but dynamically *bit* is an Int.

$$\begin{array}{l}\text{let } v_3' = \{bit = 1, flip = (\lambda x.\neg\ x), read = (\lambda x.0 < x)\} \text{ in}\\ \text{let } s_3' = \mathsf{pack}\langle ?, \lceil v_3' \rceil\rangle \text{ as } Sem \text{ in } C[s_3']\end{array}$$

The dual case of typed/untyped interoperability is that of a static package being used in dynamic code. The following example defines the untyped function g, which take as arguments the function f and an expression x to be applied to f. The function g is applied to the typed components of the package s_2, reducing the whole program without error to true.

$$\text{let } g = (\lambda f.\lambda x.f\ x) \text{ in } \mathsf{unpack}\langle X, x\rangle = s_2 \text{ in } ((g\ x.read)\ x.bit)$$

[18] In the companion technical report we provide sketches of these two proof techniques in System F.

Taking the same example, but changing $x.bit$ to the expression $(1 :: ?)$ yields a runtime error, because the function $x.read$ is expecting a sealed value, but instead it receives an unsealed Int.

$$\text{let } g = (\lambda f.\lambda x.f\ x) \text{ in unpack}\langle X, x\rangle = s_2 \text{ in } ((g\ x.read)\ (1 :: ?))$$

Optimistic Type Checking. The following example shows how the optimistic gradual type checker accepts programs that run without errors, which would be rejected with a static type checker.

$$\begin{array}{l}\text{unpack}\langle X, x\rangle = s_2 \text{ in}\\ \text{let } f = \lambda z.\text{if}(z) \text{ then } (x.flip :: ?) \text{ else } ((\lambda x : \text{Int}.1 - x) :: ?) \text{ in}\\ \text{let } v_2' = \{bit = x.bit,\ flip = f \text{ true},\ read = x.read\} \text{ in}\\ \text{let } s_2' = \text{pack}\langle X, v_2'\rangle \text{ as } Sem \text{ in}\\ \text{unpack}\langle Y, y\rangle = s_2' \text{ in } (y.read\ (y.flip\ y.bit))\end{array}$$

The package s_2' is essentially the same as s_2—in fact they are equivalent. The function f receives a Bool argument to decide whether to return the (hidden) *flip* function from package s_2, or a literal (not hidden) function. This program is gradually well-typed because of the ascriptions to the unknown type in the branches of the conditional. In contrast, a static type system would reject this program (without the ? ascriptions in the conditional branches) because the then branch would have type $X \to X$, while the else branch would have type $\text{Int} \to \text{Int}$. The gradual program runs properly, yielding false as a result.

Note that if the definition of *flip* in v_2' would be f false, then a runtime error would be raised. The error would be produced during the evaluation of the definition of s_2' because v_2' ought to have type $X \times (X \to X) \times (X \to \text{Bool})$, but instead it would have type $X \times (\text{Int} \to \text{Int}) \times (X \to \text{Bool})$.

Internal vs. External Imprecision. Another point to take into account is the nature of the imprecision of a term of existential type. As discussed previously regarding universal types (Section 9.2), the imprecision for existential types can be either internal or external, and this has an impact on runtime behavior. The following program is fully static except for the imprecise ascription of s_2 to the type $Sem_1 \equiv \exists X.X \times (X \to ?) \times (X \to \text{Bool})$. Observe that $Sem \sqsubseteq Sem_1$.

$$\text{unpack}\langle X, x\rangle = s_2 :: Sem_1 \text{ in } (x.read\ (x.flip\ (x.flip\ x.bit)))$$

Here we are in the presence of an ascribed imprecision (i.e., external), preserving the GSF property that if we ascribe a static closed term to a less precise type, its behavior is preserved: this program runs without error, and evaluates to true. Indeed, we will later show that $\text{GSF}^{\exists}$ satisfies the weak dynamic gradual guarantee $\text{DGG}^{\leqslant}$ (Section 12.5).

Conversely, in the following example, the imprecision is now internal, due to the imprecise signature Sem_1 of the package.

$$\text{unpack}\langle X, x\rangle = \text{pack}\langle \text{Int}, v_2\rangle \text{ as } Sem_1 \text{ in } (x.flip\ (x.flip\ x.bit)) + 10$$

This program is accepted statically, but fails at runtime because according to type-driven sealing, it would otherwise reveal hidden information, namely, the fact that the supposedly-hidden representation type is Int. The function $x.flip$ has type $X \to ?$, which specifies that it has to be applied to a sealed value and could return another sealed value, or in this case, an Int value. The application $(x.flip\ x.bit)$ has type ?, and it is used as the argument of $x.flip$ again, optimistically treated as an abstract type. Then, the result of the second application of $x.flip$ is added to 10, being optimistic

$$x \in \textsc{Var}, X \in \textsc{TypeVar}, \alpha \in \textsc{TypeName} \quad \Sigma \in \textsc{TypeName} \xrightarrow{\text{fin}} \textsc{Type}, \Delta \subset \textsc{TypeVar}, \Gamma \in \textsc{Var} \xrightarrow{\text{fin}} \textsc{Type}$$

$$\begin{array}{rcll} T & ::= & \cdots \mid \exists X.T & \text{(types)} \\ t & ::= & \cdots \mid \mathsf{pack}\langle T, t\rangle \text{ as } \exists X.T \mid \mathsf{unpack}\langle X, x\rangle = t \text{ in } t & \text{(terms)} \\ v & ::= & \cdots \mid \mathsf{pack}\langle T, v\rangle \text{ as } \exists X.T & \text{(values)} \end{array}$$

$\boxed{\Sigma;\Delta;\Gamma \vdash t : T}$ **Well-typed terms**

$$\text{(Tpack)}\frac{\Sigma;\Delta;\Gamma \vdash t : T_1 \qquad \Sigma;\Delta \vdash T_1 = T[T'/X] \qquad \Sigma;\Delta \vdash T'}{\Sigma;\Delta;\Gamma \vdash \mathsf{pack}\langle T', t\rangle \text{ as } \exists X.T : \exists X.T}$$

$$\text{(Tunpack)}\frac{\Sigma;\Delta;\Gamma \vdash t_1 : T_1 \qquad \Sigma;\Delta, X;\Gamma, x : \mathit{schm}_e(T_1) \vdash t_2 : T_2 \qquad \Sigma;\Delta \vdash T_2}{\Sigma;\Delta;\Gamma \vdash \mathsf{unpack}\langle X, x\rangle = t_1 \text{ in } t_2 : T_2}$$

$$\begin{array}{l} \mathit{schm}_e : \textsc{Type} \rightharpoonup \textsc{Type} \\ \mathit{schm}_e(\exists X.T) = T \\ \mathit{schm}_e(T) \text{ undefined o/w} \end{array}$$

$\boxed{\Sigma;\Delta \vdash T = T}$ **Type equality**

$$\frac{\Sigma;\Delta, X \vdash T_1 = T_2}{\Sigma;\Delta \vdash \exists X.T_1 = \exists X.T_2}$$

$\boxed{\Sigma \triangleright t \longrightarrow \Sigma \triangleright t}$ **Notion of reduction**

$$\Sigma \triangleright (\mathsf{unpack}\langle X, x\rangle = \mathsf{pack}\langle T', v\rangle \text{ as } \exists X.T \text{ in } t) \longrightarrow \Sigma, \alpha := T' \triangleright t[\alpha/X][v/x] \quad \text{where } \alpha \notin \mathit{dom}(\Sigma)$$

$\boxed{\Sigma \triangleright t \longmapsto \Sigma \triangleright t}$ **Evaluation frames and reduction**

$$f \quad ::= \quad \cdots \mid \mathsf{pack}\langle T, \square\rangle \text{ as } \exists X.T \mid \mathsf{unpack}\langle X, x\rangle = \square \text{ in } t \quad \text{(term frames)}$$

Fig. 18. $\text{SF}^{\exists}$: Syntax, static and dynamic semantics (extends Figure 1).

again with the result of the function $x.\mathit{flip}$, but this time at type Int. The program fails at runtime because of the attempt to use $x.\mathit{flip}$ with both types $X \to X$ and $X \to \text{Int}$. Note that if we allow both behaviors of the function $x.\mathit{flip}$, returning 11, then we would be revealing that the hidden representation type is Int. Thus, we admit at runtime the first application of $x.\mathit{flip}$, accepted only with the type $X \to \text{Int}$, but it fails in the second application because it receives an Int instead of a sealed value.

12.4 Semantics of $\text{GSF}^{\exists}$

In this section, we formally present the design and semantics of $\text{GSF}^{\exists}$, an extension of GSF with existential types that exhibits the behaviors illustrated above. First, we introduce the static language $\text{SF}^{\exists}$, which is the starting point to apply AGT. Actually, we only apply AGT to the new features in $\text{SF}^{\exists}$ since the others have already been gradualized. Then, we focus on $\text{GSF}^{\exists}$, the static and dynamic semantics derived by AGT. Finally, we show the principal properties that $\text{GSF}^{\exists}$ fulfills.

The Static Language $\text{SF}^{\exists}$. We derive $\text{GSF}^{\exists}$ by applying AGT to SF extended with existential types, called $\text{SF}^{\exists}$ (Figure 18). We extend SF statics with the rules (Tpack) and (Tunpack) for a package and its elimination form, which are standard. We augment the definition of type equality to deal with existential types, and use the schm_e function to extract the schema of an existential type.

The dynamic semantics of the unpack constructor is very similar to the type application; also a fresh type name α is generated and bound to the representation type T' in the global type name store Σ. Then, we substitute α (instead of the representation type) and the term component, for the

variables X and x in the body of the unpack. Like SF, $\text{SF}^{\exists}$ is also type safe, and all well-typed terms are parametric. As usual, in $\text{SF}^{\exists}$, two packages are related if their term components are related under some relations between their concrete type representations. We can define the interpretation of existential types in $\text{SF}^{\exists}$ using the same auxiliary definitions for the logical relation of GSF. Some definitions, such as $\text{Atom}^{=}_{\rho}[\exists X.G]$, become simpler because they do not need to deal with the evidence.

$$\begin{aligned}\mathcal{V}_{\rho}[\![\exists X.G]\!] = \quad & \{(W, \text{pack}\langle T_1, v_1\rangle \text{ as } \exists X.\rho(G), \text{pack}\langle G_2, v_2\rangle \text{ as } \exists X.\rho(G)) \in \text{Atom}^{=}_{\rho}[\exists X.G] \mid \\ & \forall W' \geq W, \alpha.\exists R \in \text{Rel}_{W'.j}[G_1, G_2].(W' \boxtimes (\alpha, G_1, G_2, R), v_1, v_2) \in \mathcal{V}_{\rho[X\mapsto\alpha]}[\![G]\!]\}\end{aligned}$$

$\text{GSF}^{\exists}$: Statics. We derive the statics of $\text{GSF}^{\exists}$ following AGT. As in Section 5, we first define the syntax of gradual typing, and we give them meaning through the concretization function. Then, we lift the static semantics of the static language to gradual settings using the corresponding abstraction function, which forms a Galois connection. Being consistent with the above, we extend the syntactic category of gradual types $G \in \text{GType}$ with existential types:

$$G ::= B \mid G \to G \mid \forall X.G \mid G \times G \mid X \mid \alpha \mid ? \mid \exists X.G$$

As usual, the unknown type represents any type, including existential types. We naturally extend the concretization function C and abstraction function A to existential types, preserving the Galois connection established earlier (Proposition 6.3):

$$C(\exists X.G) = \{\exists X.T \mid T \in C(G)\} \qquad A(\{\overline{\exists X.T_i}\}) = \exists X.A(\{\overline{T_i}\})$$

We define in Figure 19 the inductive definition of type precision, which is equivalent to Definition 6.1 (Proposition 12.1). As a result, $\exists X.?$ denotes any existential type, is more precise than the unknown type and less precise than $\exists X.X \to X$.

With the meaning of gradual types, the $\text{GSF}^{\exists}$ static semantics follow as usual with AGT. In this case, we need to define the gradual counterpart of the type equality predicate, whose lifting is type consistency. Following Definition 6.6, we can find in Figure 19 an equivalent inductive characterization of type consistency (Proposition 12.2). Then, we lift functions using abstraction, concretization and Definition 6.8. Our only new function in $\text{SF}^{\exists}$ is $schm_e$, whose lifting $schm_e^{\sharp}$ is defined in Figure 19 as expected.

The gradual typing rules of $\text{GSF}^{\exists}$ (Figure 19) extend those of GSF. The new rules are obtained by replacing type predicates and functions with their corresponding consistent liftings in the static typing rules. Observe that Rule (Gpack) uses type consistency instead of type equality so that the implementation term can be of a type that is distinct from, but consistent with the package type (after substituting for the representation type). For example:

$$\text{pack}\langle \text{Bool}, v_1\rangle \text{ as } Sem_3 \qquad \text{where } Sem_3 \equiv \exists X.X \times (X \to X) \times (X \to ?)$$

Here, the type of v_1 is $\text{Bool} \times (\text{Bool} \to \text{Bool}) \times (\text{Bool} \to \text{Bool})$, which is more precise than Sem_3 [Bool].

Rule (Gunpack) uses the consistent existential schema function $schm_e^{\sharp}$, which allows a term of unknown type to be optimistically treated as a package, and therefore unpacked.

$\text{GSF}^{\exists}$: Dynamics. We now turn to the dynamic semantics of $\text{GSF}^{\exists}$. As we did before, we give the dynamic semantics of $\text{GSF}^{\exists}$ in terms of a more informative variant called $\text{GSF}^{\exists}_{\varepsilon}$. In $\text{GSF}^{\exists}_{\varepsilon}$, all values are ascribed, and ascriptions carry evidence.

$$x \in \text{Var}, X \in \text{TypeVar}, \alpha \in \text{TypeName} \quad \Xi \in \text{TypeName} \xrightarrow{\text{fin}} \text{GType}, \Delta \subset \text{TypeVar}, \Gamma \in \text{Var} \xrightarrow{\text{fin}} \text{GType}$$

$$\begin{array}{rcll} G & ::= & \cdots \mid \exists X.G & \text{(gradual types)} \\ t & ::= & \cdots \mid \mathsf{pack}\langle G, t\rangle \text{ as } \exists X.G \mid \mathsf{unpack}\langle X, x\rangle = t \text{ in } t & \text{(gradual terms)} \end{array}$$

$\boxed{\Xi;\Delta;\Gamma \vdash t : G}$ **Well-typed terms**

$$\text{(Gpack)}\ \frac{\Xi;\Delta;\Gamma \vdash t : G_1 \qquad \Xi;\Delta \vdash G_1 \sim G[G'/X] \qquad \Xi;\Delta \vdash G'}{\Xi;\Delta;\Gamma \vdash \mathsf{pack}\langle G', t\rangle \text{ as } \exists X.G : \exists X.G}$$

$$\text{(Gunpack)}\ \frac{\Xi;\Delta;\Gamma \vdash t_1 : G_1 \qquad \Xi;\Delta, X;\Gamma, x : \mathit{schm}^{\sharp}_{e}(G_1) \vdash t_2 : G_2 \qquad \Xi;\Delta \vdash G_2}{\Xi;\Delta;\Gamma \vdash \mathsf{unpack}\langle X, x\rangle = t_1 \text{ in } t_2 : G_2}$$

$$\begin{array}{l} \mathit{schm}^{\sharp}_{e} : \text{GType} \rightharpoonup \text{GType} \\ \mathit{schm}^{\sharp}_{e}(\exists X.G) = G \\ \mathit{schm}^{\sharp}_{e}(?) = ? \\ \mathit{schm}^{\sharp}_{e}(G) \text{ undefined o/w} \end{array}$$

$\boxed{\Xi;\Delta \vdash G \sim G}$ **Type consistency**

$$\frac{\Xi;\Delta, X \vdash G_1 \sim G_2}{\Xi;\Delta \vdash \exists X.G_1 \sim \exists X.G_2}$$

$\boxed{G \sqsubseteq G}$ **Type precision**

$$\frac{G_1 \sqsubseteq G_2}{\exists X.G_1 \sqsubseteq \exists X.G_2}$$

Fig. 19. GSF$^{\exists}$: Syntax and static semantics (extends Figure 3).

Figure 20 presents the syntax, static and dynamics semantics of GSF$^{\exists}_{\varepsilon}$; essentially those of GSFε naturally extended with existential types. It is worth noting that we introduce the syntactic form packu$\langle G', v\rangle$ as $\exists X.G$ for raw existential values. The reduction rule (*R*pack) reduces the term pack$\langle G', v\rangle$ as $\exists X.G$ to the value $\varepsilon_{\exists X.G}$packu$\langle G', v\rangle$ as $\exists X.G :: \exists X.G$, inserting the evidence $\varepsilon_{\exists X.G}$ (evidence of the reflexive judgment $\exists X.G \sim \exists X.G$), the ascription to $\exists X.G$ and changing the syntax of the package by packu. The reduction rule (*R*unpack) specifies the reduction of an unpack expression: we substitute a fresh type name α for X in the body of the unpack, as well as a (carefully ascribed) package implementation for x. In particular, this rule combines the evidence from the actual implementation term $\varepsilon_1 u :: G_1$ with the evidence of the package, substituting the representation type on the left G' and the fresh type name α on the right for the type variable X. Note that the evidence ε justifies that the static type of the package declared by the keyword "as" is consistent with $\exists X.G$. Thus, $\varepsilon[\hat{G'}, \hat{\alpha}]$ justifies that the static type after the substitution by G' is consistent with $G[\alpha/X]$. Formally, $\varepsilon[\hat{G'}, \hat{\alpha}] = \langle p_1(\varepsilon)\ [\hat{G'}], p_2(\varepsilon)\ [\hat{\alpha}]\rangle$, where $\hat{G'} = \mathit{lift}_{\Xi'}(G')$ and $\hat{\alpha} = \mathit{lift}_{\Xi'}(\alpha)$. Consequently, the resulting evidence of $\varepsilon_1 \fatsemi \varepsilon[\hat{G'}, \hat{\alpha}]$ justifies that the type of the implementation term is consistent with $G[\alpha/X]$. Failure to justify this judgment produces an error, specifying that the implementation term is not appropriate. This evidence plays a key role in making the implementation term abstract, i.e., ensuring information hiding.

To support the dynamic semantics for existential types, we need to extend the representation of evidence types E in GSF$^{\exists}_{\varepsilon}$, adding $\exists X.E$ for existential evidence types. Additionally, we extend the definitions of consistent transitivity naturally: consistent transitivity between evidences with existential types simply relies on the underlying schemes:

$$\begin{array}{lcll} t & ::= & \cdots \mid \mathsf{pack}\langle G, t\rangle \text{ as } \exists X.G \mid \mathsf{unpack}\langle X, x\rangle = t \text{ in } t & \text{(terms)} \\ u & ::= & \cdots \mid \mathsf{packu}\langle G', v\rangle \text{ as } \exists X.G & \text{(raw values)} \end{array}$$

$\boxed{\Xi;\Delta;\Gamma \vdash s : G}$ **Well-typed terms**

$$(\text{Epacku})\frac{\Xi;\Delta;\Gamma \vdash v : G[G'/X] \qquad \Xi;\Delta \vdash G'}{\Xi;\Delta;\Gamma \vdash \mathsf{packu}\langle G', v\rangle \text{ as } \exists X.G : \exists X.G} \qquad (\text{Epack})\frac{\Xi;\Delta;\Gamma \vdash t : G[G'/X] \qquad \Xi;\Delta \vdash G'}{\Xi;\Delta;\Gamma \vdash \mathsf{pack}\langle G', t\rangle \text{ as } \exists X.G : \exists X.G}$$

$$(\text{Eunpack})\frac{\Xi;\Delta;\Gamma \vdash t_1 : \exists X.G_1 \qquad \Xi;\Delta, X;\Gamma, x : G_1 \vdash t_2 : G_2 \qquad \Xi;\Delta \vdash G_2}{\Xi;\Delta;\Gamma \vdash \mathsf{unpack}\langle X, x\rangle = t_1 \text{ in } t_2 : G_2}$$

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

$$(R\text{pack})\ \Xi \triangleright \mathsf{pack}\langle G', v\rangle \text{ as } \exists X.G \longrightarrow \Xi \triangleright \varepsilon_{\exists X.G}\mathsf{packu}\langle G', v\rangle \text{ as } \exists X.G :: \exists X.G$$

$$(R\text{unpack})\ \Xi \triangleright \mathsf{unpack}\langle X, x\rangle = \varepsilon\mathsf{packu}\langle G', \varepsilon_1 u :: G_1\rangle :: \exists X.G \text{ in } t \longrightarrow \begin{cases} \Xi' \triangleright t[\hat{\alpha}/X][((\varepsilon_1 \fatsemi \varepsilon[\hat{G'}, \hat{\alpha}])u :: G[\alpha/X])/x] \\ \quad \text{where } \Xi' \triangleq \Xi, \alpha := G' \text{ for some } \alpha \notin dom(\Xi) \\ \quad \hat{G'} = lift_{\Xi'}(G') \text{ and } \hat{\alpha} = lift_{\Xi'}(\alpha) \\ \mathbf{error} \qquad \text{if not defined} \end{cases}$$

$\boxed{\Xi \triangleright t \longmapsto \Xi \triangleright t \text{ or } \mathbf{error}}$ **Evaluation frames and reduction**

$$f \quad ::= \quad \cdots \mid \mathsf{pack}\langle G, \square\rangle \mid \mathsf{unpack}\langle X, x\rangle = \square \text{ in } t$$

Fig. 20. $\mathrm{GSF}^{\exists}_{\varepsilon}$: Syntax, static and dynamic semantics (extends Figure 4).

$$(\text{ex})\frac{\langle E_1, E_2\rangle \fatsemi \langle E_3, E_4\rangle = \langle E_1', E_2'\rangle}{\langle \exists X.E_1, \exists X.E_2\rangle \fatsemi \langle \exists X.E_3, \exists X.E_4\rangle = \langle \exists X.E_1', \exists X.E_2'\rangle}$$

Illustration. We now return to the gradual semaphore implementation s_3^*, which is the translation of the term s_3 from $\mathrm{GSF}^{\exists}$ to $\mathrm{GSF}^{\exists}_{\varepsilon}$. Remember that all base values in $\mathrm{GSF}^{\exists}_{\varepsilon}$ are ascribed to their base types, but for simplicity below, we omit trivial evidences. The following reduction trace illustrates all the important aspects of reduction in $\mathrm{GSF}^{\exists}_{\varepsilon}$:

$$\begin{array}{lll} & \mathsf{unpack}\langle X, x\rangle = \varepsilon_{Sem}\mathsf{packu}\langle ?, v_3^*\rangle :: Sem \text{ in } (x.read\ (x.flip\ x.bit)) & \text{initial evidence} \\ (R\text{unpack}) \longmapsto^* & (\langle ? \to \mathsf{Bool}, \alpha^? \to \mathsf{Bool}\rangle(\lambda x.0 < x) :: \alpha \to \mathsf{Bool}) & \\ (R\text{proj}i) & ((\langle ? \to \mathsf{Int}, \alpha^? \to \alpha^{\mathsf{Int}}\rangle(\lambda x.1 - x) :: \alpha \to \alpha)\ (\langle \mathsf{Int}, \alpha^{\mathsf{Int}}\rangle 1 :: \alpha)) & \text{consistent transitivity} \\ (R\text{app}) \longmapsto & (\langle ? \to \mathsf{Bool}, \alpha^? \to \mathsf{Bool}\rangle(\lambda x.0 < x) :: \alpha \to \mathsf{Bool})\ (\langle \mathsf{Int}, \alpha^{\mathsf{Int}}\rangle(1 - 1) :: \alpha) & \text{unsealing eliminates } \alpha \\ (R\text{op},R\text{asc}) \longmapsto^* & (\langle ? \to \mathsf{Bool}, \alpha^? \to \mathsf{Bool}\rangle(\lambda x.0 < x) :: \alpha \to \mathsf{Bool})\ (\langle \mathsf{Int}, \alpha^{\mathsf{Int}}\rangle 0 :: \alpha) & \text{the return is sealed} \\ (R\text{app}) \longmapsto & \varepsilon_{\mathsf{Bool}}(0 < 0) :: \mathsf{Bool} & \text{unsealing eliminates } \alpha \\ (R\text{op},R\text{asc}) \longmapsto^* & \varepsilon_{\mathsf{Bool}}\mathsf{false} :: \mathsf{Bool} & \end{array}$$

In this example, the initial evidence of the package is fully static. We omit some steps in the reduction, but it is crucial to show in the rule (Runpack) how the evidence $\varepsilon_{Sem}[?, \alpha^?]$ is calculated:

$$\varepsilon_{Sem}[?, \alpha^?] \equiv \langle ? \times (? \to ?) \times (? \to \mathsf{Bool}), \alpha^? \times (\alpha^? \to \alpha^?) \times (\alpha^? \to \mathsf{Bool})\rangle$$

After some application of the rule (Rproj$i$), the term component is protected by the type name α. The application step (Rapp) then gives rise to unsealing evidence to interact with the implementation and sealing evidence to protect the implementation.

12.5 Properties of $\mathrm{GSF}^{\exists}$

In this section, we summarize the main properties, statics and dynamics, concerning $\mathrm{GSF}^{\exists}$. We cover the refined criteria for gradual typing and parametricity.

Static Properties. We can show that the GSF$^{\exists}$ meet the same static properties as GSF.

Proposition 12.1 (GSF$^{\exists}$: Precision, Inductively). *The inductive definition of type precision given in Figure 19 is equivalent to Definition 6.1.*

Proposition 12.2 (GSF$^{\exists}$: Consistency, Inductively). *The inductive definition of type consistency given in Figure 19 is equivalent to Definition 6.6.*

The type system of GSF$^{\exists}$ is equivalent to the SF$^{\exists}$ type system on fully-static terms (Proposition 12.3), where $\vdash_S$ denote the typing judgment of SF$^{\exists}$.

Proposition 12.3 (GSF$^{\exists}$: Static Equivalence for Static Terms). *Let t be a static term and G a static type ($G = T$). We have $\vdash_S t : T$ if and only if $\vdash t : T$.*

The static semantics of GSF$^{\exists}$ satisfy the static gradual guarantee (Proposition 12.4), where type precision (Definition 6.1) extends naturally to *term* precision.

Proposition 12.4 (GSF$^{\exists}$: Static Gradual Guarantee). *Let t and t' be closed GSF$^{\exists}$ terms such that $t \sqsubseteq t'$ and $\vdash t : G$. Then $\vdash t' : G'$ and $G \sqsubseteq G'$.*

Dynamic Gradual Guarantees. Not surprisingly, GSF$^{\exists}$ does not satisfy the dynamic gradual guarantee (Section 9) with respect to precision $\sqsubseteq$ for existential types. Let us return to the semaphore implementation s_1. Note that $s_1 \sqsubseteq s_4$, where $s_4 = \mathsf{pack}\langle \mathsf{Bool}, v_1\rangle$ as $\exists X.X \times (X \to X) \times (? \to \mathsf{Bool})$. If we use these terms in the same context as follows, we will obtain that

$$\mathsf{unpack}\langle X, x\rangle = s_1 \text{ in } (x.\mathit{read}\ (x.\mathit{flip}\ x.\mathit{bit})) \sqsubseteq \mathsf{unpack}\langle X, x\rangle = s_4 \text{ in } (x.\mathit{read}\ (x.\mathit{flip}\ x.\mathit{bit}))$$

However, the term on the left reduces to 2, while the (less precise) term on the right produces a runtime error because of the attempt to apply the function *read* (in this case of type $? \to \mathsf{Bool}$) to a sealed value. On the other hand, with a simple extension of strict precision to existential types, GSF$^{\exists}$ does satisfy the weaker dynamic gradual guarantee DGG$^{\leqslant}$ (Theorem 9.5). Figure 21 defines the strict type and term precision for both GSF$^{\exists}_{\varepsilon}$ and GSF$^{\exists}$.

Parametricity. We establish parametricity for GSF$^{\exists}$ by proving parametricity for GSF$^{\exists}_{\varepsilon}$. We extend the step-indexed logical relation for GSFε (Figure 14), adding the interpretation of existential types. Usually, two packages are related if their term components are related under some conditions [Ahmed 2006; Neis et al. 2009]. But in gradual settings, the definition of $\mathcal{V}_\rho[\![\exists X.G]\!]$ is more complex. We start with the classical interpretation of the existential types adapted to our previous logical relation (which is not adequate for the interpretation of gradual existential types):

$$\begin{aligned}\mathcal{V}_\rho[\![\exists X.G]\!] = \quad & \{(W, \varepsilon_1 \mathsf{packu}\langle G_1, v_1\rangle :: \exists X.\rho(G), \varepsilon_2 \mathsf{packu}\langle G_2, v_2\rangle :: \exists X.\rho(G)) \in \mathrm{Atom}^{=}_{\rho}[\exists X.G] \mid \\ & \forall W' \geq W, \alpha.\exists R \in \mathrm{Rel}_{W'.j}[G_1, G_2].(W' \boxtimes (\alpha, G_1, G_2, R), v_1, v_2) \in \mathcal{V}_{\rho[X\mapsto\alpha]}[\![G]\!]\}\end{aligned}$$

Let us focus on some simple and not very interesting programs, but useful to explain our existential types interpretation. For example, if we relate these two package, $\varepsilon_{\exists X.X}\mathsf{packu}\langle \mathsf{Int}, \varepsilon_{\mathsf{Int}}1 :: \mathsf{Int}\rangle :: \exists X.X$ and $\varepsilon_{\exists X.X}\mathsf{packu}\langle \mathsf{Bool}, \varepsilon_{\mathsf{Bool}}\mathsf{true} :: \mathsf{Bool}\rangle :: \exists X.X$, under the above definition, then we would have to prove that their component terms, $\varepsilon_{\mathsf{Int}}1 :: \mathsf{Int}$ and $\varepsilon_{\mathsf{Bool}}\mathsf{true} :: \mathsf{Bool}$, are related in $\mathcal{V}_{\rho[X\to\alpha]}[\![X]\!]$, which is not true. Keep in mind that for two terms to be related in our logical relationship they must have the same type, and they are related in a type variable if they are related in the type variable substituted by its associated type name.

Taking the above into account, we change the logical interpretation of existential types slightly. It is worth pointing out that this definition is not enough to interpret existential types.

$\boxed{G \leqslant G}$ **Strict type precision**

$$\frac{G_1 \leqslant G_2}{\exists X.G_1 \leqslant \exists X.G_2}$$

$\boxed{\Omega \vdash \Xi_1 \triangleright s : G \leqslant \Xi_2 \triangleright s : G}$ **Strict term precision** (for conciseness, s ranges over both t and u)

$$(\leqslant \mathsf{packu}_\varepsilon)\frac{G_1' \leqslant G_2' \quad \Omega \vdash \Xi_1 \triangleright v_1 : G_1[G_1'/X] \leqslant \Xi_2 \triangleright v_2 : G_2[G_2'/X] \quad \exists X.G_1 \sqsubseteq \exists X.G_2}{\Omega \vdash \Xi_1 \triangleright \mathsf{packu}\langle G_1', v_1\rangle \text{ as } \exists X.G_1 : \exists X.G_1 \leqslant \Xi_2 \triangleright \mathsf{packu}\langle G_2', v_2\rangle \text{ as } \exists X.G_2 : \exists X.G_2}$$

$$(\leqslant \mathsf{pack}_\varepsilon)\frac{G_1' \leqslant G_2' \quad \Omega \vdash \Xi_1 \triangleright t_1 : G_1[G_1'/X] \leqslant \Xi_2 \triangleright t_2 : G_2[G_2'/X] \quad \exists X.G_1 \leqslant \exists X.G_2}{\Omega \vdash \Xi_1 \triangleright \mathsf{pack}\langle G_1', t_1\rangle \text{ as } \exists X.G_1 : \exists X.G_1 \leqslant \Xi_2 \triangleright \mathsf{pack}\langle G_2', t_2\rangle \text{ as } \exists X.G_2 : \exists X.G_2}$$

$$(\leqslant \mathsf{unpack}_\varepsilon)\frac{\Omega \vdash \Xi_1 \triangleright t_1 : \exists X.G_1 \leqslant \Xi_2 \triangleright t_2 : \exists X.G_2 \quad \Omega, x : G_1 \sqsubseteq G_2 \vdash \Xi_1 \triangleright t_1' : G_1' \leqslant \Xi_2 \triangleright t_2' : G_2'}{\Omega \vdash \Xi_1 \triangleright \mathsf{unpack}\langle X, x\rangle = t_1 \text{ in } t_1' : G_1' \leqslant \Xi_2 \triangleright \mathsf{unpack}\langle X, x\rangle = t_2 \text{ in } t_2' : G_2'}$$

$\boxed{G \rightarrowtriangle G}$ **Type matching**

$$\exists X.G \rightarrowtriangle \exists X.G \qquad\qquad ? \rightarrowtriangle \exists X.?$$

$\boxed{\Omega \vdash v : G \leqslant_v v : G}$ **Strict value precision**

$$(\leqslant \mathsf{packu})\frac{G_1' \leqslant G_2' \quad \Omega \vdash v_1 : G_1'' \leqslant_v v_2 : G_2'' \quad \exists X.G_1 \sqsubseteq \exists X.G_2 \quad G_1'' \sqcap G_1[G_1'/X] \leqslant G_2'' \sqcap G_2[G_2'/X]}{\Omega \vdash \mathsf{pack}\langle G_1', v_1\rangle \text{ as } \exists X.G_1 : \exists X.G_1 \leqslant_v \mathsf{pack}\langle G_2', v_2\rangle \text{ as } \exists X.G_2 : \exists X.G_2}$$

$\boxed{\Omega \vdash \Xi_1 \triangleright t : G \leqslant \Xi_2 \triangleright t : G}$ **Strict term precision**

$$(\leqslant \mathsf{pack})\frac{G_1' \leqslant G_2' \quad \Omega \vdash t_1 : G_1'' \leqslant t_2 : G_2'' \quad \exists X.G_1 \leqslant \exists X.G_2 \quad G_1'' \sqcap G_1[G_1'/X] \leqslant G_2'' \sqcap G_2[G_2'/X]}{\Omega \vdash \mathsf{pack}\langle G_1', t_1\rangle \text{ as } \exists X.G_1 : \exists X.G_1 \leqslant \mathsf{pack}\langle G_2', t_2\rangle \text{ as } \exists X.G_2 : \exists X.G_2}$$

$$(\leqslant \mathsf{unpack})\frac{\Omega \vdash t_1 : G_1 \leqslant t_2 : G_2 \quad G_1 \rightarrowtriangle \exists X.G_1'' \quad G_2 \rightarrowtriangle \exists X.G_2'' \quad \Omega, x : G_1'' \sqsubseteq G_2'' \vdash t_1' : G_1' \leqslant t_2' : G_2'}{\Omega \vdash \mathsf{unpack}\langle X, x\rangle = t_1 \text{ in } t_1' : G_1' \leqslant \mathsf{unpack}\langle X, x\rangle = t_2 \text{ in } t_2' : G_2'}$$

Fig. 21. $\mathrm{GSF}^{\exists}_{\varepsilon}$ and $\mathrm{GSF}^{\exists}$: Extensions for strict precision.

$$\begin{aligned}\mathcal{V}_\rho[\![\exists X.G]\!] = \quad & \{(W, \varepsilon_1 \mathsf{packu}\langle G_1, v_1\rangle :: \exists X.\rho(G), \varepsilon_2 \mathsf{packu}\langle G_2, v_2\rangle :: \exists X.\rho(G)) \in \mathrm{Atom}^{=}_{\rho}[\exists X.G] \mid \\ & \forall W' \geq W, \alpha.\exists R \in \mathrm{Rel}_{W',j}[G_1, G_2].(W' \boxtimes (\alpha, G_1, G_2, R), \\ & \varepsilon_1[\hat{G_1}, \hat{\alpha}]\, v_1 :: \rho(G)[\alpha/X]\,,\ \varepsilon_2[\hat{G_2}, \hat{\alpha}]\, v_2 :: \rho(G)[\alpha/X]\,) \in \mathcal{T}_{\rho[X \mapsto \alpha]}[\![G]\!]\ \}\end{aligned}$$

First, we establish that two packages are related if their term components ascribed to the existential type body, substituting the fresh type name α by X, are related. Second, since we ascribed term components to other types, we need evidence justifying this. More specifically, we need two evidences that justify $\rho(G)[G_1/X]$ is consistent with $\rho(G)[\alpha/X]$ and $\rho(G)[G_2/X]$ is consistent with $\rho(G)[\alpha/X]$, respectively. In this sense, we use evidences $\varepsilon_1[\hat{G_1}, \hat{\alpha}]$ and $\varepsilon_2[\hat{G_2}, \hat{\alpha}]$; they are just ε_1 and ε_2, substituting representation types in the left and the fresh type name α in the right, by X. Note that the combination of these evidences with the internal evidences of the package (term component evidences) through transitivity can fail.

This interpretation of existential types is pretty complete but is not enough. Now, suppose that we have the packages $\varepsilon_{\exists X.?}\mathsf{packu}\langle \mathsf{Int}, \varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int}\rangle :: \exists X.?$ and $\varepsilon_{\exists X.?}\mathsf{packu}\langle \mathsf{Bool}, \varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int}\rangle :: \exists X.?$. These two packages are very similar; the only difference consists in their representation type. They are related under the above interpretation of existential types, due the fact that we can relate $\varepsilon_?(\varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int})$ and $\varepsilon_?(\varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int})$ under the unknown type. But we do not want to relate these

packages. First, it is easy to show that the encodings to universal types of these two packages are not related because they do not behave in the same way. Second, we could use the packages in the same context (e.g., if we ascribe them by the type $\exists X.X$) with different behaviors, losing the property that says if two packages are related then they are contextually equivalents. Therefore, we need to be more strict in the definition of when two packages are related.

Finally, we define the interpretation of existential types as follows:

$$\begin{aligned}\mathcal{V}_\rho[\![\exists X.G]\!] = \quad & \{(W, \varepsilon_1 \mathsf{packu}\langle G_1, v_1\rangle :: \exists X.\rho(G), \varepsilon_2 \mathsf{packu}\langle G_2, v_2\rangle :: \exists X.\rho(G)) \in \mathrm{Atom}^{=}_{\rho}[\exists X.G] \mid \\ & \forall W' \geq W, \alpha.\exists R \in \mathrm{Rel}_{W'.j}[G_1, G_2].\ \forall \Xi, \varepsilon \Vdash \Xi; dom(\rho) \vdash \exists X.G \sim \exists X.G, W' \in \mathcal{S}[\![\Xi]\!]. \\ & (\ (\varepsilon_1 \fatsemi \rho_1(\varepsilon)) \wedge (\varepsilon_2 \fatsemi \rho_2(\varepsilon))\ ) \Rightarrow (W' \boxtimes (\alpha, G_1, G_2, R), \\ & (\varepsilon_1 \fatsemi \rho_1(\varepsilon)\ )[\hat{G_1}, \hat{\alpha}]v_1 :: \rho(G)[\alpha/X], (\varepsilon_2 \fatsemi \rho_2(\varepsilon)\ )[\hat{G_2}, \hat{\alpha}]v_2 :: \rho(G)[\alpha/X]) \in \mathcal{T}_{\rho[X\mapsto\alpha]}[\![G]\!]\}\end{aligned}$$

The representation type of a package in gradual settings act as a pending substitution, which has to make sense for all possible (more precise) existential types. In a static world, we do not have to deal with this problem, because evidence never gains precision, and the initial type checking ensures that the program never fails. For this reason, we extend the interpretation of existential types by quantifying over all evidences that justify that $\exists X.G \sim \exists X.G$. Doing so ensures that the representation type behaves correctly for any existential type that is more precise than $\exists X.G$. Note that studying the encoding of existential into universal types leads us to justify the same definition.

Representation Independence and Gradual Free Theorems. We prove the soundness of the logical relation extended with existential types with respect to contextual equivalence.

Proposition 12.5. *If* $\Xi; \Delta; \Gamma \vdash t_1 \approx t_2 : G$, *then* $\Xi; \Delta; \Gamma \vdash t_1 \approx^{ctx} t_2 : G$.

With this result, we can return to the semaphore example and show the representation independence for the two different implementations s_1 and s_3. Let us recall the definition of these packages, where the former uses Bool as representation type, while the latter uses the unknown type:

$$\begin{aligned} s_1 &\equiv \mathsf{pack}\langle \mathrm{Bool}, v_1\rangle \text{ as } \mathit{Sem} \quad \textit{where } v_1 \equiv \{\mathit{bit} = \mathrm{true}, \mathit{flip} = (\lambda x : \mathrm{Bool}.\neg\, x), \mathit{read} = (\lambda x : \mathrm{Bool}.x)\} \\ s_3 &\equiv \mathsf{pack}\langle ?, \lceil v_3 \rceil\rangle \text{ as } \mathit{Sem} \quad \textit{where } v_3 \equiv \{\mathit{bit} = 1, \mathit{flip} = (\lambda x.1 - x), \mathit{read} = (\lambda x.0 < x)\}\end{aligned}$$

To prove that these two packages are contextually equivalent (Proposition 12.6), it suffices by Proposition 12.5 to show that each logically approximates the other. (Note that to proceed with the proof below, we deal with the tuple-based representation of *Sem*, since GSF has no records.) We prove only one direction, namely $s_1 \preceq s_3 : \mathit{Sem}$; the other is proven analogously. Therefore, we are required to show that $s_1^* \preceq s_3^* : \mathit{Sem}$, where s_1^* and s_3^* are the translation of s_1 and s_3 from $\mathrm{GSF}^{\exists}$ to $\mathrm{GSF}^{\exists}_{\varepsilon}$, respectively.

Proposition 12.6. $s_1 \approx^{ctx} s_3 : \mathit{Sem}$

To prove $s_1^* \preceq s_3^* : \mathit{Sem}$, we are required to show that for all W, $(W, s_1^*, s_3^*) \in \mathcal{T}_\emptyset[\![\mathit{Sem}]\!]$. Therefore, we have to prove that $\vdash s_i^* : \mathit{Sem}$ (but this is already proven) and $(W, s_1^*, s_3^*) \in \mathcal{V}_\emptyset[\![\mathit{Sem}]\!]$ (since s_i^* are already values). Expanding the definition of $\mathcal{V}_\emptyset[\![\mathit{Sem}]\!]$, we need to show that $\forall W' \geq W$ and α, $\exists R \in \mathrm{Rel}_{W'.j}[\mathrm{Bool}, ?]$, such that $\forall \varepsilon \Vdash \cdot; \cdot \vdash \mathit{Sem} \sim \mathit{Sem}$:

$$(W'', (\varepsilon_{\mathit{Sem}} \fatsemi \varepsilon)[\mathrm{Bool}, \hat{\alpha}]v_1^* :: G[\alpha/X], (\varepsilon_{\mathit{Sem}} \fatsemi \varepsilon)[?, \hat{\alpha}]v_3^* :: G[\alpha/X]) \in \mathcal{T}_{[X\mapsto\alpha]}[\![G]\!]$$

where $W'' = W' \boxtimes (\alpha, \mathrm{Bool}, ?, R)$, $G = \mathit{schm}^{\sharp}_{e}(\mathit{Sem}) = X \times (X \to X) \times (X \to \mathrm{Bool})$, $s_1^* = \varepsilon_{\mathit{Sem}} \mathsf{packu}\langle \mathrm{Bool}, v_1^*\rangle :: \mathit{Sem}$ and $s_2^* = \varepsilon_{\mathit{Sem}} \mathsf{packu}\langle ?, v_2^*\rangle :: \mathit{Sem}$. Since $\varepsilon_{\mathit{Sem}}$ is a static evidence, it

cannot gain precision and so $(\varepsilon_{Sem} \fatsemi \varepsilon) = \varepsilon_{Sem}$. Therefore, now we are required to show

$$(\downarrow W'', v_1', v_3') \in \mathcal{V}_{[X \mapsto \alpha]}[\![G]\!]$$

where

$$\begin{array}{ll} v_1' = & \langle \mathsf{Bool} \times (\mathsf{Bool} \to \mathsf{Bool}) \times (\mathsf{Bool} \to \mathsf{Bool}), \alpha^{\mathsf{Bool}} \times (\alpha^{\mathsf{Bool}} \to \alpha^{\mathsf{Bool}}) \times (\alpha^{\mathsf{Bool}} \to \mathsf{Bool}) \rangle \\ & \langle \mathsf{true}, \langle (\lambda x : \mathsf{Bool}. \neg\, x), (\lambda x : \mathsf{Bool}. x) \rangle \rangle :: \alpha \times (\alpha \to \alpha) \times (\alpha \to \mathsf{Bool}) \\ v_3' = & \langle \mathsf{Int} \times (? \to \mathsf{Int}) \times (? \to \mathsf{Bool}), \alpha^{\mathsf{Int}} \times (\alpha^{?} \to \alpha^{\mathsf{Int}}) \times (\alpha^{?} \to \mathsf{Bool}) \rangle \\ & \langle 1, \langle (\lambda x. 1 - x), (\lambda x. 0 < x) \rangle \rangle :: \alpha \times (\alpha \to \alpha) \times (\alpha \to \mathsf{Bool}) \end{array}$$

Taking $R = \{(W^*, \varepsilon_{\mathsf{Bool}}\mathsf{true} :: \mathsf{Bool}, \varepsilon_{\mathsf{Int}} 1 :: ?), (W^*, \varepsilon_{\mathsf{Bool}}\mathsf{false} :: \mathsf{Bool}, \varepsilon_{\mathsf{Int}} 0 :: ?) \mid W^* \geq W'\}$, it is easy to show that

$$\begin{array}{l} - (\downarrow^2 W'', \langle \mathsf{Bool}, \alpha^{\mathsf{Bool}} \rangle \mathsf{true} :: \alpha, \langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle 1 :: \alpha) \in \mathcal{V}_{[X \mapsto \alpha]}[\![X]\!] \\ - (\downarrow^2 W'', \langle \mathsf{Bool} \to \mathsf{Bool}, \alpha^{\mathsf{Bool}} \to \alpha^{\mathsf{Bool}} \rangle (\lambda x : \mathsf{Bool}. \neg\, x) :: \alpha \to \alpha, \\ \qquad \langle ? \to \mathsf{Int}, \alpha^{?} \to \alpha^{\mathsf{Int}} \rangle (\lambda x. 1 - x) :: \alpha \to \alpha) \in \mathcal{V}_{[X \mapsto \alpha]}[\![X \to X]\!] \\ - (\downarrow^2 W'', \langle \mathsf{Bool} \to \mathsf{Bool}, \alpha^{\mathsf{Bool}} \to \mathsf{Bool} \rangle (\lambda x : \mathsf{Bool}. x) :: \alpha \to \mathsf{Bool}, \\ \qquad \langle ? \to \mathsf{Bool}, \alpha^{?} \to \mathsf{Bool} \rangle (\lambda x. 0 < x) :: \alpha \to \mathsf{Bool}) \in \mathcal{V}_{[X \mapsto \alpha]}[\![X \to \mathsf{Bool}]\!] \end{array}$$

Note that $\downarrow^2 W'' \geq W'$. Thus, the result follows immediately.

13 RELATED WORK

We have already discussed at length related work on gradual parametricity [Ahmed et al. 2017; Igarashi et al. 2017a; New et al. 2020; Xie et al. 2018], highlighting the different design choices, properties, and limitations of each. Hopefully our discussions adequately reflect the many subtleties involved in designing a gradual parametric language. A key lesson of this work is that the tension between graduality and parametricity comes from the early commitment to seal values based on type information. Very recently, Labrada et al. [2022] proposed plausible sealing as a new intermediate language mechanism that allows postponing such decisions to runtime. The resulting intermediate language satisfies both graduality and parametricity, but the formal treatment so far only accounts for a restricted form of polymorphism. Their approach uses lexically-scoped sealing instead of global seals used in prior work, including this work. As a result, their language cannot embed the cryptographic lambda calculus; indeed, such an embedding crucially relies on the unrestricted scope of dynamic seals (Section 11).

The relation between parametric polymorphism in general and dynamic typing greatly predates the work on gradual typing. Abadi et al. [1991] first note that without further precaution, type abstraction might be violated. Subsequent work explored different approaches to protect parametricity, especially **runtime-type generation (RTG)** [Abadi et al. 1995; Leroy and Mauny 1991; Rossberg 2003]. Sumii and Pierce [2004] and Guha et al. [2007] use dynamic sealing, originally proposed by Morris [1973], in order to dynamically enforce type abstraction. Matthews and Ahmed [2008] also use RTG in order to protect polymorphic functions in an integration of Scheme and ML. This line of work eventually led to the polymorphic blame calculus [Ahmed et al. 2011] and its most recent version with the proof of parametricity by Ahmed et al. [2017]. We adapt their logical relation to the evidence-based semantics of GSF.

Hou et al. [2016] prove the correctness of compiling polymorphism to dynamic typing with embeddings and partial projections; the compilation setting however differs significantly from gradual typing. New and Ahmed [2018] use embedding-projection pairs to formulate a semantic account of the dynamic gradual guarantee, coined graduality, in a language with explicit casts.

Inspired by the work of Neis et al. [2009] on parametricity in a non-parametric language, they extended their approach to gradual parametricity, yielding the PolyG$^{\nu}$ language design with explicit sealing [New et al. 2020], discussed at length in this article.

Devriese et al. [2018] disprove a conjecture by Pierce and Sumii [2000] according to which the compilation of System F to a language with dynamic sealing primitives is fully abstract, i.e., preserves contextual equivalences. They show that, for similar reasons, the embedding of System F in a polymorphic blame calculus like λB is not fully abstract; their observation also applies to GSF. Full abstraction might be too strong a criteria for gradual typing: already in the simply-typed setting, embedding typed terms in gradual contexts is not fully abstract, because gradual types admit non-terminating terms. Nevertheless, whenever the observable effects of the static and gradual languages are the same, then full abstraction is a powerful criteria to ensure the preservation of static reasoning principles [Jacobs et al. 2021]. For instance, Jacobs et al. [2021] show the fully abstract embedding criterion for a gradual version of the simply-typed lambda calculus extended with recursive types and sums. In previous work [Toro et al. 2019], we show that GSF satisfies an *imprecise termination* property, which is a weaker yet useful result that sheds light on gradual free theorems about imprecise type signatures. Here, we extend this result by introducing a strict notion of precision relative to which the dynamic gradual guarantee is satisfied.

The embedding of a cryptographic lambda calculus in GSF directly relates to unpublished work by Siek and Wadler [2016], which studies the connection between a polymorphic cast calculus [Ahmed et al. 2011] and a cryptographic lambda calculus based on that of Pierce and Sumii [2000]. Their approach is similar to our own, though the technicalities differ: their translation targets a cast calculus, more akin to GSFε, while we define the embedding directly at the level of GSF. Therefore, our result is the first to relate the dynamic end of a gradual parametric *source* language and dynamic sealing. Also, our embedding is inspired by the type $Univ = \exists Y.\forall X.(X \to Y) \times (Y \to X)$ put forth by Devriese et al. [2018] in their analysis of full abstraction mentioned above. The type *Univ* can be interpreted as stating the existence of a universal type Y, i.e., a type that all other types can be embedded into and extracted from. As they show by a parametricity argument, *Univ* cannot be inhabited in System F. In GSF, however, the unknown type ? plays this role of a universal type. Interestingly, Siek and Wadler [2016] discuss an alternative embedding that resembles ours, although they discard it as it does not align well with their treatment of blame. A systematic treatment of blame within the Abstracting Gradual Typing framework is an active research topic.

This work is generally related to gradualization of advanced typing disciplines, in particular to gradual information-flow security typing [Disney and Flanagan 2011; Fennell and Thiemann 2013, 2016; Garcia and Tanter 2015; Toro et al. 2018]. In these systems, one aims at preserving *noninterference*, i.e., that private values do not affect public outputs. Both parametricity and noninterference are 2-safety properties, expressed as a relation of two program executions. While Garcia and Tanter [2015] show that one can derive a pure security language with AGT that satisfies both noninterference and the dynamic gradual guarantee, Toro et al. [2018] find that in the presence of mutable references, one can have either the dynamic gradual guarantee, or noninterference, but not both. Also similarly to this work, AGT for security typing Toro et al. [2018] needs a more precise abstraction for evidence types (based on security *label intervals*) in order to enforce noninterference. Together, these results suggest that type-based approaches to gradual typing are in tension with semantically-rich typing disciplines. Solutions might come from restricting the considered *syntax*, as in PolyG$^{\nu}$ in the context of parametricity, or the *range* of graduality, as recently established by Azevedo de Amorim et al. [2020] in the context of noninterference, where the dynamic end of the spectrum is not fully untyped security-wise.

14 CONCLUSION

GSF is a gradual parametric language that bridges between System F and an untyped language with dynamic sealing primitives. The spectrum between both extremes is fairly continuous, even if not perfectly: the implicit type-driven resolution of sealing in its runtime semantics, which appears necessary in order to respect the syntax of System F, implies that GSF violates the dynamic gradual guarantee in certain cases. We precisely characterize the weaker continuity that GSF supports, along with all its other properties. In particular, we prove that GSF can faithfully embed dynamic sealing. Additionally, we extend GSF with support for existential types for gradual data abstraction. The design of GSF is largely driven by the Abstracting Gradual Typing (AGT) methodology. We find that AGT greatly streamlines the static semantics of GSF, but does not yield a language that respects parametricity by default; non-trivial exploration was necessary to uncover how to strengthen the structure and treatment of runtime evidence in order to recover a notion of gradual parametricity. In turn, this strengthening breaks the dynamic gradual guarantee when loss of precision interferes with type-driven sealing.

This work focuses on the semantics and metatheoretical properties of GSF, without explicitly taking into account efficiency considerations such as pay-as-you-go [Igarashi et al. 2017a; Siek and Taha 2006], space efficiency [Herman et al. 2010; Siek and Wadler 2010], cast elimination [Rastogi et al. 2012], and so on. Optimizing the dynamic semantics of GSF is left for future work. Likewise, blame tracking has not been considered [Wadler and Findler 2009]. The use of explicit polymorphism in the design of GSF hampers certain interoperability scenarios. We are exploring a resolution of this tension based on a flexible runtime mechanism, whose metatheory is on-going work.

As extensively discussed, gradual parametricity is subtle, and there are many scenarios when the decision of failing or not is open to debate and various considerations. This work contributes to this discussion by proposing several practical principles, such as ensuring that fully-static terms can be embedded in gradual contexts and made imprecise externally without affecting their behavior. There are two main trends in the design of gradual parametric languages: those based on System F, like λB, CSA, System F_G and GSF, and those that depart from that syntax, like PolyG$^\nu$. GSF contributes to the System F trend. Also, we have argued that while PolyG$^\nu$ enjoys a stronger metatheory than languages from the other trend, several limitations regarding modularity and abstraction caused by its use of explicit sealing are not benign. The question remains open of whether there is a third way, embracing both System F and a fully satisfying metatheory. Plausible sealing [Labrada et al. 2022] seems promising in that regard, but there are still open issues for the technique to fully cover System F.

ACKNOWLEDGMENTS

We thank Amal Ahmed, Dominique Devriese, Kenji Maillard, Max New, Gabriel Scherer, the attendees of various oral presentations of this work, and the anonymous reviewers of both POPL 2019 and JACM for useful feedback and suggestions that drastically improved both the presentation and our understanding of this work.

REFERENCES

Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. 1991. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems* 13, 2 (April 1991), 237–268.

Martin Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. 1995. Dynamic typing in polymorphic languages. *Journal of Functional Programming* 5, 1 (1995), 111–130.

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)*, Peter Sestoft (Ed.), Vol. 3924. Springer-Verlag, Vienna, Austria, 69–83.

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009a. State-dependent representation independence. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009)*. ACM Press, Savannah, GA, USA, 340–353.

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

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011. Blame for all. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*. ACM Press, Austin, Texas, USA, 201–214.

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.

Arthur Azevedo de Amorim, Matt Fredrikson, and Limin Jia. 2020. Reconciling noninterference and gradual typing. In *Proceedings of the 2020 Symposium on Logic in Computer Science (LICS 2020)*.

Johannes Bader, Jonathan Aldrich, and Éric Tanter. 2018. Gradual program verification. In *Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2018) (Lecture Notes in Computer Science)*, Işil Dillig and Jens Palsberg (Eds.), Vol. 10747. Springer-Verlag, Los Angeles, CA, USA, 25–46.

Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2014. A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN Conference on Functional Programming (ICFP 2014)*. ACM Press, Gothenburg, Sweden, 283–295.

Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2016. Gradual type-and-effect systems. *Journal of Functional Programming* 26 (Sept. 2016), 19:1–19:69.

Gavin Bierman, Erik Meijer, and Mads Torgersen. 2010. Adding dynamic types to C$^{\#}$. In *Proceedings of the 24th European Conference on Object-oriented Programming (ECOOP 2010) (Lecture Notes in Computer Science)*, Theo D'Hondt (Ed.). Springer-Verlag, Maribor, Slovenia, 76–100.

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.

Robert Cartwright and Mike Fagan. 1991. Soft typing. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. Toronto, Ontario, Canada, 278–292.

Giuseppe Castagna and Victor Lanvin. 2017. Gradual typing with union and intersection types. See [ICFP 2017 2017], 41:1–41:28.

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

Haskell B. Curry, J. Roger Hindley, and J. P. Seldin. 1972. *Combinatory Logic, Volume II*. Studies in logic and the foundations of mathematics, Vol. 65. North-Holland Pub. Co.

Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1982)*. ACM Press, New York, NY, USA, 207–212. https://doi.org/10.1145/582153.582176

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.

Tim Disney and Cormac Flanagan. 2011. Gradual information flow typing. In *International Workshop on Scripts to Programs*.

Joseph Eremondi, Éric Tanter, and Ronald Garcia. 2019. Approximate normalization for gradual dependent types. *Proceedings of the ACM on Programming Languages* 3, ICFP (Aug. 2019), 88:1–88:30.

Luminous Fennell and Peter Thiemann. 2013. Gradual security typing with references. In *Proceedings of the 26th Computer Security Foundations Symposium (CSF)*. 224–239.

Luminous Fennell and Peter Thiemann. 2016. LJGS: Gradual security types for object-oriented languages. In *Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Rome, Italy, 9:1–9:26.

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.

Ronald Garcia and Éric Tanter. 2015. Deriving a Simple Gradual Security Language. eprint arXiv:1511.01399.

Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of typestate-oriented programming. *ACM Transactions on Programming Languages and Systems* 36, 4, Article 12 (Oct. 2014), 12:1–12:44 pages.

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.

Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. 2007. Relationally-parametric polymorphic contracts. In *Proceedings of the ACM Dynamic Languages Symposium (DLS 2007)*. ACM Press, Montreal, Canada, 29–40.

Robert Harper. 2012. *Practical Foundations for Programming Languages*. Cambridge University Press.

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

Kuen-Bang Hou, Nick Benton, and Robert Harper. 2016. Correctness of compiling polymorphism to dynamic typing. *Journal of Functional Programming* 27 (2016), 1:1–1:24.

ICFP 2017 2017.

Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. 2017b. Gradual session types. See [ICFP 2017 2017], 38:1–38:28.

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017a. On polymorphic gradual typing. See [ICFP 2017 2017], 40:1–40:29.

Lintaro Ina and Atsushi Igarashi. 2011. Gradual typing for generics. In *Proceedings of the 26th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2011)*. ACM Press, Portland, Oregon, USA, 609–624.

Koen Jacobs, Amin Timany, and Dominique Devriese. 2021. Fully abstract from static to gradual. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 7:1–7:30.

Elizabeth Labrada, Matías Toro, Éric Tanter, and Dominique Devriese. 2022. Plausible sealing for gradual parametricity. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (April 2022), 70:1–70:28.

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.

Xavier Leroy and Michel Mauny. 1991. Dynamics in ML. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA 1991) (Lecture Notes in Computer Science)*, Vol. 523. Springer-Verlag, 406–426.

Paul Blain Levy. 1999. Call-by-push-value: A subsuming paradigm. In *4th International Conference on Typed Lambda Calculi and Applications (TLCA'99) (Lecture Notes in Computer Science)*, Jean-Yves Girard (Ed.), Vol. 1581. Springer-Verlag, 228–242.

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)*, Sophia Drossopoulou (Ed.), Vol. 4960. Springer-Verlag, Budapest, Hungary, 16–31.

Jacob Matthews and Robert Bruce Findler. 2007. Operational semantics for multi-language programs. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007)*. ACM Press, Nice, France, 3–10.

John C. Mitchell. 1988. Polymorphic type inference and containment. *Information and Computation* 76, 2-3 (Feb. 1988), 211–249.

John C. Mitchell and Gordon D. Plotkin. 1988. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems* 10, 3 (July 1988), 470–502. https://doi.org/10.1145/44501.45065

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

Georg Neis, Derek Dryer, and Andreas Rossberg. 2009. Non-parametric parametricity. In *Proceedings of the 14th ACM SIGPLAN Conference on Functional Programming (ICFP 2009)*. ACM Press, Edinburgh, Scotland, UK, 135–148.

Max S. New and Amal Ahmed. 2018. Graduality from Embedding-Projection Pairs. 73:1–73:30 pages.

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.

Martin Odersky and Konstantin Läufer. 1996. Putting type annotations to work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 96)*. ACM Press, St. Petersburg Beach, Florida, USA, 54–67.

Oxford. 2021. *Oxford Advanced Learner's Dictionary* (10th ed.). Oxford University Press. Evidence.

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

Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA.

Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. 2012. The ins and outs of gradual type inference. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*. ACM Press, Philadelphia, USA, 481–494.

John C. Reynolds. 1974. Towards a theory of type structure. In *Proceedings 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.

Andreas Rossberg. 2003. Generativity and dynamic opacity for abstract types. In *Proceedings of the 5th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2003)*. 241–252.

Ilya Sergey and Dave Clarke. 2012. Gradual ownership types. In *Proceedings of the 21st European Symposium on Programming Languages and Systems (ESOP 2012) (Lecture Notes in Computer Science)*, Helmut Seidl (Ed.), Vol. 7211. Springer-Verlag, Tallinn, Estonia, 579–599.

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 Walid Taha. 2007. Gradual typing for objects. In *Proceedings of the 21st European Conference on Object-oriented Programming (ECOOP 2007) (Lecture Notes in Computer Science)*, Erik Ernst (Ed.). Springer-Verlag, Berlin, Germany, 2–27.

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. 2015a. 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.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015b. Monotonic references for efficient gradual typing. In *Proceedings of the 24th European Symposium on Programming Languages and Systems (ESOP 2015) (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer-Verlag, London, UK, 432–456.

Jeremy G. Siek and Philip Wadler. 2016. The Key to Blame: Gradual Typing Meets Cryptography. Unpublished manuscript.

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.

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage migration: From scripts to programs. In *Proceedings of the ACM Dynamic Languages Symposium (DLS 2006)*. ACM Press, Portland, Oregon, USA, 964–974.

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. 2017. A gradual interpretation of union types. In *Proceedings of the 24th Static Analysis Symposium (SAS 2017) (Lecture Notes in Computer Science)*, Vol. 10422. Springer-Verlag, New York City, NY, USA, 382–404.

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

Philip Wadler. 1989. Theorems for free!. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*. ACM, London, United Kingdom, 347–359.

Philip Wadler. 2017. Abstract data types without the types. *Journal of Universal Computer Science* 23, 1 (2017), 5–20.

Philip Wadler and Robert Bruce Findler. 2009. Well-typed programs can't be blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems (ESOP 2009) (Lecture Notes in Computer Science)*, Giuseppe Castagna (Ed.), Vol. 5502. Springer-Verlag, York, UK, 1–16.

Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. 2020. Gradual verification of recursive heap data structures. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 228:1–228:28.

Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. 2011. Gradual typestate. In *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP 2011) (Lecture Notes in Computer Science)*, Mira Mezini (Ed.), Vol. 6813. Springer-Verlag, Lancaster, UK, 459–483.

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)*, Amal Ahmed (Ed.), Vol. 10801. Springer-Verlag, Thessaloniki, Greece, 3–30.

Received 28 May 2020; revised 8 June 2022; accepted 22 June 2022