Gradual Parametricity, Revisited

MATÍAS TORO, University of Chile, Chile
ELIZABETH LABRADA, University of Chile, Chile
ÉRIC TANTER, University of Chile, Chile and Inria Paris, France

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 were recently proposed. Among other issues, these proposals can however signal parametricity errors in unexpected situations, and improperly handle type instantiations when imprecise types are involved. These observations further suggest that existing polymorphic cast calculi are not well suited for supporting a gradual counterpart of System F. Consequently, we revisit the challenge of designing a gradual language with explicit parametric polymorphism, exploring the extent to which the Abstracting Gradual Typing methodology helps us derive such a language, GSF. We present the design and metatheory of GSF, and provide a reference implementation. In addition to avoiding the uncovered semantic issues, GSF satisfies all the expected properties of a gradual parametric language, save for one property: the dynamic gradual guarantee, which was left as conjecture in all prior work, is here proven to be simply incompatible with parametricity. We nevertheless establish a weaker property that allows us to disprove several claims about gradual free theorems, clarifying the kind of reasoning supported by gradual parametricity.

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

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

ACM Reference Format:
Matías Toro, Elizabeth Labrada, and Éric Tanter. 2019. Gradual Parametricity, Revisited. *Proc. ACM Program. Lang.* 3, POPL, Article 17 (January 2019), 30 pages. https://doi.org/10.1145/3290330

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 does a best effort statically, treating imprecision optimistically. The runtime semantics of the gradual language detects at runtime any invalidation of optimistic static assumptions. 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, in particular that it conservatively extends a corresponding statically-typed language, that it can

[*]This work is partially funded by CONICYT/FONDECYT Regular/1150017, CONICYT/Doctorado Nacional/2015-21150510 & 21151566, and by the European Research Council under ERC Starting Grant SECOMP (715753).

Authors' addresses: Matías Toro, University of Chile, Computer Science Department (DCC), Beauchef 851, Santiago, Chile; Elizabeth Labrada, University of Chile, Computer Science Department (DCC), Beauchef 851, Santiago, Chile; Éric Tanter, University of Chile, Computer Science Department (DCC), Beauchef 851, Santiago, Chile , Inria Paris, France.

2475-1421/2019/1-ART17
https://doi.org/10.1145/3290330

faithfully embed dynamically-typed terms, and that the static-to-dynamic transition is smooth, a property formally captured as the (static and dynamic) gradual guarantees [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 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. 2018a], session types [Igarashi et al. 2017b], refinements [Lehmann and Tanter 2017], set-theoretic types [Castagna and Lanvin 2017], Hoare logic [Bader et al. 2018] and parametric polymorphism [Ahmed et al. 2011, 2017; Igarashi et al. 2017a; Ina and Igarashi 2011; 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. The first gradual language to come with a proof of parametricity is the cast calculus λB [Ahmed et al. 2017], recently used as a target language by Xie et al. [2018]. Another recent effort is System F_G, an actual gradual source language (*i.e.* without explicit casts), which is compiled to a cast calculus akin to λB, called System F_C [Igarashi et al. 2017a].

Contributions. This work starts from the novel identification of design issues in existing proposals, especially in their dynamic semantics. In short, parametricity errors can be raised in unexpected situations, and type instantiations are ignored when imprecise types are involved. Consequently, we argue that neither λB nor System F_C are adequate targets for an explicitly-parametric gradual language (§2).

To this end, we introduce GSF, a gradual counterpart of System F that addresses the design issues identified in prior work and satisfies parametricity (§8). We explicitly lay out the design principles, goals and non-goals of GSF and introduce the language briefly through examples (§3). We then explain how we derive GSF from a variant of System F called SF (§4), by following the Abstracting Gradual Typing methodology (AGT) [Garcia et al. 2016]. While the statics of GSF follow naturally from SF using AGT (§5), the dynamic semantics are more challenging (§6/§7). GSF satisfies the expected properties of gradual languages (§5/§7), except the dynamic gradual guarantee. This property was left open as a conjecture in prior work; here we prove that it is in fact *incompatible* with parametricity (§9). We uncover a novel, weaker property that GSF satisfies, which allows us to disprove several claims related to gradual free theorems for imprecise type signatures (§10).

Complete definitions and proofs of the main results can be found in the extended version [Toro et al. 2018b]. Additionally, GSF is implemented as an interactive prototype that exhibits both typing derivations and reduction traces. All the examples mentioned in this paper, as well as others, are readily available in the online demo: https://pleiad.cl/gsf.

2 THE NEED TO REVISIT GRADUAL PARAMETRICITY

We start with a quick introduction to parametric polymorphism and parametricity, before motivating gradual parametricity through examples and finally exposing different issues in both the static and dynamic semantics of existing languages.

2.1 Background: Parametric Polymorphism

Parametric polymorphism allows 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 $\forall X.X \rightarrow X$ 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 $\forall X.X \rightarrow X$. 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 intrinsic, 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. Technically, implicit polymorphism induces a notion of subtyping that relates polymorphic types to their instantiations [Mitchell 1988; Odersky and Läufer 1996]; *e.g.* $\forall X.X \rightarrow X <: \mathsf{Int} \rightarrow \mathsf{Int}$. Implicitly-polymorphic languages generally use an explicitly-polymorphic language underneath (*e.g.* GHC Core), providing the convenience of implicitness through an inference phase that produces an explicitly-annotated program. In essence, the use of the subtyping judgment $\forall X.X \rightarrow X <: \mathsf{Int} \rightarrow \mathsf{Int}$ is materialized in terms by introducing an explicit instantiation [Int], and vice-versa, the use of the judgment $\mathsf{Int} \rightarrow \mathsf{Int} <: \forall X.\mathsf{Int} \rightarrow \mathsf{Int}$ 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 $\forall X.X \rightarrow X$ 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.[1]

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 list permutation function 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 $\forall X.X \rightarrow X$, 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; Xie et al. 2018].

2.2 Gradual Parametricity in a Nutshell

Basics Gradual parametricity supports imprecise typing information, yet ensures that assumptions about parametricity are enforced at runtime whenever they are not provable statically. In the following program, function `f` expects a function `g` of type $\forall X.X \rightarrow X$ as argument. It is applied

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

to an argument `h` of the unknown type. By consistency, this program is well-typed; however the compliance of `h` with respect to its assumed parametric signature is unknown statically.

```
let f = λg:(∀X.X → X).g [Int] 10 in let h : ? = ... in f h
```

By parametricity, function `f` can deduce that `g` behaves like the identity function (§2.1). In presence of gradual types—as in any variant of System F with errors and non-termination—this conclusion should be relaxed: gradual simple types admit both non-termination [Siek and Taha 2006] and runtime type errors. Therefore, as a consequence of parametricity, we can prove that if the program above terminates, it should either produce 10, or fail with a runtime error, possibly denoting that `h` was in fact not a proper identity function.

Let us consider three possible implementations of `h`:

```
h1 = ΛX.λx:X.x      h2 = ΛX.λx:?.x      h3 = ΛX.λx:?.x+1
```

Function `h1` is the standard System F identity function, and function `h2` is a less precise version, which behaves identically. Therefore, using either of these functions in the program above produces the result 10. Conversely, function `h3` is not a proper identity function. Note that the function is well-typed, because `x` has type ? in the body. Also, using `h3` in the program above is type safe, because `f` happens to instantiate its argument at type Int, so execution could proceed safely without errors and yield 11; this would however be a violation of parametricity, so an error should be raised.

State of the Art. While the basics of gradual parametricity are well understood, the details are tricky. In particular, 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. 2009, 2011]; only very recent work on a variant of that calculus, λB, has achieved this result [Ahmed et al. 2017]. In fact, λB is a cast calculus, not a gradual source language, meaning that the program written above would not be valid; explicit casts should be sprinkled in different places to achieve the same result. Igarashi *et al.* recently developed a gradual source language, System F_G, which does support the intended lightweight, cast-free syntax of gradual languages. Following the early tradition of gradual typing [Siek and Taha 2006], the semantics of System F_G are given by translation to a cast calculus, System F_C, which is a close cousin of λB. Igarashi *et al.* in fact do not prove parametricity, but conjecture that due to the similarity between System F_G and λB, parametricity should hold. Xie et al. [2018] develop a language with implicit polymorphism (here referred to as CSA), which compiles to λB and therefore satisfies parametricity.

On the metatheoretic front, beyond parametricity, there are other important properties that are relevant for gradual languages, most notably the conservative extension and the gradual guarantees [Siek et al. 2015a]. The former states that, on fully static programs, a gradual language should behave exactly like its static counterpart. The latter states that making types less precise does not introduce static or dynamic type errors. λB is *not* a conservative extension of System F (§2.3), and the gradual guarantees are left as an open question. System F_G is a conservative extension of System F, and CSA of an implicit variant of System F. Both System F_G and CSA satisfy the static gradual guarantee, although System F_G uses an ad hoc notion of precision tuned to that effect (§2.3). The dynamic gradual guarantee for both System F_G and CSA are still open questions.

Finally, gradual free theorems about *imprecise* type signatures have not been formally studied, beyond a number of claims that we mention below and disprove in §10.

2.3 Static Semantics Issues

While the static semantics of simple gradual languages are uncontroversial, devising the static semantics of gradual polymorphic languages has proven to be fairly challenging, yielding systems that are arguably hard to grasp. We highlight the most salient issues with λB and System F_G below, and then relate to CSA, which addresses them to some extent.

Mixing Explicit and Implicit Polymorphism. Both λB and System F_G are languages with *explicit* polymorphism, *i.e.* with explicit type abstraction and type application terms. However, instead of focusing on explicit polymorphism only, both languages accommodate some form of implicitness, but with different flavors. Consider the type of a polymorphic identity function, $\forall X.X \rightarrow X$. In λB this type is compatible with $\mathsf{Int} \rightarrow \mathsf{Int}$, which is a defining feature of *implicit* polymorphism. More surprisingly, this type is also compatible with $\mathsf{Int} \rightarrow \mathsf{Bool}$. (Runtime errors will account for the obvious mistake.) This means in particular that λB is not a proper conservative extension of System F, as both type systems disagree on some fully static terms. Technically, instead of the traditional consistency relation, λB introduces two close but distinct relations on types, called convertibility and compatibility, in order to orchestrate these non-trivial semantics. Conversely, System F_G relies on a notion of consistency, and is a proper conservative extension of System F. As an explicitly polymorphic language, System F_G does not relate $\forall X.X \rightarrow X$ with any of its static instantiations. However, it *does* relate that type with $? \rightarrow ?$, considered to be *quasi-polymorphic*, on the basis that using the unknown type should bring some of the flexibility of implicit polymorphism.

Ad-hoc Precision. Conversely to System F_G, λB has no notion of type precision, and does not discuss any of the gradual guarantees. The precision relation of System F_G features some constraints that might be surprising to programmers. Specifically, System F_G allows loss of precision only in non-parametric positions of a polymorphic type. For instance, $\forall X.X \rightarrow \mathsf{Int}$ is considered more precise than $\forall X.X \rightarrow ?$, but unrelated to $\forall X.? \rightarrow \mathsf{Int}$. Because precision induces consistency, it means that $\forall X.X \rightarrow \mathsf{Int}$ and $\forall X.? \rightarrow \mathsf{Int}$ are inconsistent with each other. This choice is motivated by the desire to avoid a counterexample of the gradual guarantee: they claim that a function of type $\forall X.? \rightarrow X$ must fail on all inputs in order to respect parametricity (we disprove this claim in §10), so accepting that this type is less precise than $\forall X.X \rightarrow X$ breaks the dynamic gradual guarantee.

But tailoring the precision relation to avoid a class of counterexamples is not benign. First, changing the definition of precision to accommodate a theorem does not necessarily result in a programmer's expectations being adjusted. Let us recall that the gradual guarantees were introduced by Siek et al. [2015a] in order to formally capture the expectations of programmers using gradual languages. The restriction on precision imposed by System F_G breaks the intuition of programmers that, starting program from a well-typed program, removing static type information yields a program that is *by definition* less precise—and should also be well-typed.

Second, the restricted rule excludes instances of precision that are harmless for the dynamic gradual guarantee. For instance, in System F_G, $\forall X.X \rightarrow X$ is not more precise than $\forall X.X \rightarrow ?$, despite the fact that a function of type $\forall X.X \rightarrow ?$ can be a proper identity function (§10).

Third, Igarashi et al. [2017a] only prove the static guarantee based on this ad hoc precision, and leave the dynamic guarantee as a conjecture, so it is unclear whether the restriction on precision imposed by System F_G is indeed sufficient.

Separating Concerns. Recently, Xie et al. [2018] raise similar concerns about the static semantics of λB and System F_G, in particular regarding the mixing of explicit and implicit polymorphism. In response, they clearly separate the subtyping relation induced by implicit polymorphism from the consistency relation induced by gradual types. Their notion of consistent subtyping extends the notion of Siek and Taha [2007]. As a result, CSA features intuitive and straightforward definitions of precision and consistency, while accommodating the flexibility of implicit polymorphism in full.

We fully concur with the necessity to untangle implicitness from consistency in order to achieve a principled design. Xie *et al.* leave open the question of designing an *explicitly*-polymorphic gradual language. Additionally, Xie *et al.* do not deal with the dynamic semantics of their language beyond a translation to λB. Therefore CSA inherits both the virtues of λB, such as parametricity, and its issues, uncovered next.[2]

2.4 Dynamic Semantics Issues

In the design of gradually-typed languages, cast calculi are typically used as target languages to give runtime semantics to gradual programs. However, as observed by Garcia et al. [2016], there is little justification or guidance available to design or choose a cast calculus for interpreting a given gradual source language. To this date, only the Abstracting Gradual Typing methodology (AGT) provides a systematic approach to derive the dynamic semantics of gradual languages by directly giving meaning to gradual typing derivations [Garcia et al. 2016].

Since the early work on the polymorphic blame calculus [Ahmed et al. 2009, 2011], all existing work has built upon variants of that cast calculus. While a cast language like λB can be used as a source language [Ahmed et al. 2017], λB has been used in recent work as the target language of choice for gradual source languages [Igarashi et al. 2017a; Xie et al. 2018]. In this section, we identify two questionable design decisions in both λB and System F_C that arguably make them inadequate as internal languages of a gradual version of System F.

Excess of Failure. Consider the following example, written in System F_G (the λB and System F_C versions are more verbose because of explicit casts):

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

What would the programmer expect out of this program? While the annotated return type of `f` is left unknown, the function itself is the System F identity function. Therefore, one might expect that instantiating the function to Int, passing 1 and adding 1, should yield 2 as a result.

However, in both λB and System F_C, the above program fails with a runtime error. The reason is that the result of `f [Int] 1` is sealed, and therefore unusable directly. Ahmed et al. [2011] justify this behavior (already present in early work [Ahmed et al. 2009]), or the alternative of always failing before returning, based on a claim about gradual free theorems. Intuitively, this can be surprising because the underlying value is the System F identity function, which *does* behave parametrically; it is therefore unclear what parametricity violation is being reported. As we will see later, this failing behavior is in fact not formally demanded by parametricity (§10).

Lack of Failure. A major interest of gradual types is that they *soundly* augment the expressiveness of the original static type system. Let us illustrate first in a simply-typed setting (STLC refers to the simply-typed lambda calculus with base types):

- Consider the STLC term $t = \lambda x : \_.x$, which behaves as the identity function. t is incomplete because the type annotation on x is missing so far.
- 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 $\mathsf{Int} \rightarrow \mathsf{Int}$, $\mathsf{Bool} \rightarrow \mathsf{Bool}$, etc.
- 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 $\forall X.X \rightarrow X$ to precisely specify that t can be applied with any argument type and return the same type.
- With a gradual variant of STLC, we can give term t the imprecise type $? \rightarrow ?$ to statically capture the fact that t is definitely a function, without committing to specific domain and codomain types.
- This lack of precision is soundly backed by runtime enforcement, such that the term $(t\ 3)\ 1$ evaluates to a runtime type error.

[2]The implicit polymorphism of Xie et al. [2018] faces other challenges, most notably the lack of coherence of the runtime semantics. This issue is entirely related to implicit polymorphism and is therefore not addressed here.

The exact same line of reasoning should apply when starting from System F, as follows:

- Consider the System F term $t = \lambda x : \_.(x\ [\mathsf{Int}])$, which behaves as an instantiation function to Int. t is incomplete because the type annotation on x is missing so far.
- t is operationally valid at different types, but cannot be given a general type in System F. It 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.
- 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.
- 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.
- 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.

However, the runtime semantics of λB and System F_C suffer from a fundamental issue that breaks the argument above: they do not respect type instantiations that involve the unknown type, and consequently do not fail as expected.[3] Below is another simple example in System F_G in which the polymorphic identity function is instantiated to Int and passed a Bool value:

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

This System F_G program (and its translation to λB) returns `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. Consequently, current polymorphic cast calculi such as λB and System F_C are inadequate to serve as the runtime support of a gradual variant of System F.

3 GSF, INFORMALLY

This paper presents the design, semantics and metatheory of GSF, a gradual counterpart of System F that addresses the issues raised above. This section focuses on the informal aspects of GSF: design principles and methodology, as well as some illustrative examples of GSF in action.

3.1 Design Principles, Goals and Non-Goals

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:

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 must embody the complexity of dynamically enforcing parametricity solely in its dynamic semantics; its static semantics should be as straightforward as possible.
Natural precision: Precision is intended to capture the level of static typing information of a gradual type, with ? as the least precise and static types as the most precise [Siek et al. 2015a]. GSF should preserve this simple intuition.

The mandatory goals for GSF, *i.e.* the properties that it should definitely satisfy, are:

[3]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.

Type safety: GSF should be type safe, meaning all programs should either evaluate to a value, halt with a runtime error, or diverge. Well-typed GSF terms should not get stuck.
Conservative extension: GSF should be a conservative extension of System F: both languages should coincide in their static and dynamic semantics for fully static programs.
Faithful instantiations: GSF should respect type instantiations. In particular, explicit instantiations of imprecise types should be enforced.
Parametricity: GSF should enforce the notion of parametricity understood for gradual programs [Ahmed et al. 2017]. In particular, a polymorphic function should behave uniformly across all its instantiations—*i.e.* always take related inputs to related outputs, or always fail or diverge.
Static gradual guarantee: By virtue of the simple statics principle stated above, GSF should satisfy the static gradual guarantee, *i.e.* typeability should be monotonic with respect to the natural notion of precision.

Similarly important are the explicit *non-goals* that we adopt when designing GSF:

Dynamic gradual guarantee: While GSF should strive to satisfy the dynamic gradual guarantee, this should *not* be at the expense of any of the above-stated principles and goals. In other words, the dynamic gradual guarantee is the first candidate property to abandon (or weaken) if need be.
Implicit polymorphism: While implicit polymorphism is certainly a desirable feature for usability, the integration of implicit polymorphism in GSF is future work.
Blame tracking: Tracking blame in order to report more informative error messages is valuable, but most important is to properly *identify* error cases. As discussed in §2.4, λB and System F_G both miss important errors and raise errors in unexpected situations.
Performance: We focus on the semantics and meta-theoretical 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], etc. Optimizing the dynamic semantics of GSF is left for future work.

3.2 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. 2018a]. 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.

3.3 GSF in Action

Recall the example from §2.2, in which a function `f` defined as:

```
let f = λg:(∀X.X → X).g [Int] 10
```

is applied to a function h of unknown type. GSF behaves exactly as described with each of the three variant implementations of h, namely:

```
let h : ? = ΛX.λx:X.x in f h    ----> 10
let h : ? = ΛX.λx:?.x in f h    ----> 10
let h : ? = ΛX.λx:?.x+1 in f h  ----> error
```

In the last case, the runtime error is raised when the body of the function attempts to perform an addition, since this type-specific operation is a violation of parametricity.

The fact that GSF adopts explicit polymorphism *à la* System F means that a polymorphic type is not consistent with any of its instantiations. In practice, this means that:

```
let h : ? = λx:?.x in f h         ----> error
```

The runtime error occurs when the body of f performs the type application, because the value bound to g is not of the appropriate constructor (Λ). If changing the definition of h to include the Λ constructor is not an option, one can perform this adaptation explicitly upon application of f:

```
let h : ? = λx:?.x in f (ΛX.h)  ----> 10
```

Finally, GSF does not report spurious parametricity violations, and enforces type instantiations even when applied to an imprecisely-typed value:

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

Hence GSF addresses the issues in the dynamic semantics of λB and System F_C, and soundly augments the expressiveness of System F (§2.4). Other illustrative examples are available online.

4 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 $\pi_i(t)$,[4] 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 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*: upon type application, a fresh type name α is generated and bound to the instantiation type T in a global type name store Σ. 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 1. SF: Simple Static Polymorphic Language with Runtime Type Generation

$$\begin{aligned}
C &: \mathrm{GT{\scriptsize YPE}} \rightarrow \mathcal{P}^*(\mathrm{T{\scriptsize YPE}})\\
C(B) &= \{\, B \,\}\\
C(G_1 \rightarrow G_2) &= \{T_1 \rightarrow T_2 \mid T_1 \in C(G_1), T_2 \in C(G_2)\}\\
C(G_1 \times G_2) &= \{T_1 \times T_2 \mid T_1 \in C(G_1), T_2 \in C(G_2)\\
C(X) &= \{\, X \,\}\\
C(\alpha) &= \{\, \alpha \,\}\\
C(\forall X.G) &= \{\forall X.T \mid T \in C(G)\}\\
C(?) &= \mathrm{T{\scriptsize YPE}}
\end{aligned}$$

$$\begin{aligned}
A &: \mathcal{P}^*(\mathrm{T{\scriptsize YPE}}) \rightarrow \mathrm{GT{\scriptsize YPE}}\\
A(\{\, B \,\}) &= B\\
A(\{\, \overline{T_{i1} \rightarrow T_{i2}} \,\}) &= A(\{\, \overline{T_{i1}} \,\}) \rightarrow A(\{\, \overline{T_{i2}} \,\})\\
A(\{\, \overline{T_{i1} \times T_{i2}} \,\}) &= A(\{\, \overline{T_{i1}} \,\}) \times A(\{\, \overline{T_{i2}} \,\})\\
A(\{\, X \,\}) &= X\\
A(\{\, \alpha \,\}) &= \alpha\\
A(\{\, \overline{\forall X.T_i} \,\}) &= \forall X.A(\{\, \overline{T_i} \,\})\\
A(\{\, \overline{T_i} \,\}) &= ? \;\; \textit{otherwise}
\end{aligned}$$

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

type names is necessary to derive equalities [Igarashi et al. 2017a]. For instance, in the reduction of the well-typed program (id [Int $\rightarrow$ Int]) (id [Int]), where id is the polymorphic identity function, the equality $\alpha := \mathsf{Int} \rightarrow \mathsf{Int}, \beta := \mathsf{Int}; \Delta \vdash \alpha = \beta \rightarrow \beta$ 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 well-formed types. A type environment Γ binds term variables to types, and is well-formed ($\Sigma; \Delta \vdash \Gamma$) if all types are well-formed.

The decision of using type names instead of the traditional substitution semantics is in anticipation of gradualization: indeed, prior work has shown that runtime type generation is crucial in order to be able to 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.

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.

5 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.

5.1 Syntax and Syntactic Meaning of Gradual Types

We introduce the syntactic category of gradual types $G \in \mathrm{GT{\scriptsize YPE}}$, by admitting the unknown type in any position, namely:

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

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). In other words, Int $\rightarrow$? 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 (§5.2).

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

Proposition 5.2 (Precision, inductively). *The inductive definition of type precision given in Figure 3 is equivalent to Definition 5.1.*

Observe that both $\forall X.X \rightarrow ?$ and $\forall X.? \rightarrow X$ are more precise than $\forall X.? \rightarrow ?$, and less precise than $\forall X.X \rightarrow X$, thereby reflecting the original intuition about precision [Siek et al. 2015a]. Also $\forall X.? \rightarrow ?$ and $? \rightarrow ?$ 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. The abstraction function A is direct (Figure 2): it preserves type constructors and falls back on the unknown type whenever an 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 5.3 (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 $A(C(G)) \sqsubseteq G$.

5.2 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 5.4 (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:

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

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 5.5 (Consistency, inductively). *The inductive definition of type consistency given in Figure 3 is equivalent to Definition 5.4.*

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 $? \rightarrow ?$ is not consistent with $\forall X.? \rightarrow ?$. However, in line with gradual typing, both are consistent with ?, as expected. Therefore GSF does not treat $? \rightarrow ?$ 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]:

$$
\begin{array}{lll}
x \in \textsc{Var}, X \in \textsc{TypeVar}, \alpha \in \textsc{TypeName} & \Xi \in \textsc{TypeName} \overset{\text{fin}}{\rightharpoonup} \textsc{GType}, \Delta \subset \textsc{TypeVar}, \Gamma \in \textsc{Var} \overset{\text{fin}}{\rightharpoonup} \textsc{GType} & \\
G \quad ::= \quad B \mid G \to G \mid \forall X.G \mid G \times G \mid X \mid \alpha \mid ? & & \text{(gradual types)} \\
t \quad ::= \quad b \mid \lambda x : G.t \mid \Lambda X.t \mid \langle t, t \rangle \mid x \mid t :: G \mid op(\overline{t}) \mid t\ t \mid t\ [G] \mid \pi_i(t) & & \text{(gradual terms)}
\end{array}
$$

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

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

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

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

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

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

$$
\begin{array}{llll}
dom^{\sharp} : \textsc{GType} \rightharpoonup \textsc{GType} & cod^{\sharp} : \textsc{GType} \rightharpoonup \textsc{GType} & inst^{\sharp} : \textsc{GType}^2 \rightharpoonup \textsc{GType} & proj_i^{\sharp} : \textsc{GType} \rightharpoonup \textsc{GType} \\
dom^{\sharp}(G_1 \to G_2) = G_1 & cod^{\sharp}(G_1 \to G_2) = G_2 & inst^{\sharp}(\forall X.G, G') = G[G'/X] & proj_i^{\sharp}(G_1 \times G_2) = G_i \\
dom^{\sharp}(?) = ? & cod^{\sharp}(?) = ? & inst^{\sharp}(?, G') = ? & proj_i^{\sharp}(?) = ? \\
dom^{\sharp}(G) \text{ undefined o/w} & cod^{\sharp}(G) \text{ undefined o/w} & inst^{\sharp}(G, G') \text{ undefined o/w} & proj_i^{\sharp}(G) \text{ undefined o/w}
\end{array}
$$

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

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

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

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

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

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

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

Fig. 3. GSF: Syntax and Static Semantics

Definition 5.6 (Consistent lifting of functions). Let F_n be a function of type $\text{Type}^n \longrightarrow \text{Type}$. Its consistent lifting $F_n^\sharp$, of type $\text{GType}^n \longrightarrow \text{GType}$, 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 5.7 (Consistent type functions). *The definitions of $dom^\sharp$, $cod^\sharp$, $inst^\sharp$, and $proj_i^\sharp$ given in Fig. 3 are consistent liftings, as per Def. 5.6, of the corresponding functions from Fig. 1.*

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.* $inst^\sharp(? \longrightarrow ?, G')$ is undefined).

5.3 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.[5]

Proposition 5.8 (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 (Def. 5.1) 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'. 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 5.9 (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 $G \sqsubseteq G'$.*

6 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 §7 to the details of the representation and operations of evidence that enable GSF to satisfy parametricity while adequately tracking type instantiations.

6.1 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,

[5]As usual, the propositions here are stated over closed terms, but are proven as corollaries of statements over open terms.

combining evidences through steps of *consistent transitivity*, which either yield more precise evidence, or fail if the evidences to combine are incompatible. 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 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].[6] Formally:

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

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. However, type equality in SF is more subtle (§4), so the general presentation of evidence as pairs is required.

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* $\circ$, determines whether two evidences support the transitivity: here, $\varepsilon_2 \circ \varepsilon_1 = \langle \mathsf{Bool}, \mathsf{Bool} \rangle \circ \langle \mathsf{Int}, \mathsf{Int} \rangle$ 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. 2018a].

6.2 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 that 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 §6.1 above, $(\lambda x : ?.x + 1)$ 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 ε_1 and ε_2 are the ones from §6.1.[7]

[6] We use blue color for evidence ε to enhance readability of the structure of terms in the next section and beyond.
[7] Such initial evidences are computed by means of an *interior* function, given by the abstract interpretation framework [Garcia et al. 2016]. The definition of interior and the type-preserving translation are direct.

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 4. GSFε: Syntax, Static and Dynamic Semantics

Despite this translation, we do preserve 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, we can present a uniform syntax and greatly simplify 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.

Because the translation from GSF to GSFε introduces explicit ascriptions everywhere consistency is used, the only remaining use of consistency in the typing rules of GSFε is in the rule (Easc). 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 (Rasc) 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, if defined, justifies that $G_u \sim G_2$; if undefined, reduction steps to **error**. Rule (Rop) 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 (Rapp) 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 §6.1, produces **error**. The return value is ascribed to the expected return type, using the codomain evidence of the function $cod(\varepsilon_1)$. Rule (Rpair) 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 (Rproj$i$) extracts a component of a pair and ascribes it to the projected type, using the corresponding evidence obtained with $p_i(\varepsilon)$.[8]

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

- 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 *outer* ascription asserts that $G[\alpha/X]$ is consistent with $G[G'/X]$, witnessed by evidence ε_{out}. This evidence plays a key role in avoiding unjustified failures as described in §2.4. We define ε_{out} in §7.2 below, once the representation of evidence is introduced.

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 operations, $lift_\Xi$: GType $\longrightarrow$ EType and *unlift* : EType $\longrightarrow$ GType.[9] Because type names have meaning related to a store, the lifting is parameterized by the store Ξ. Term substitution 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.

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

[8]We use $p_i(\varepsilon)$ to avoid confusion with $\pi_i(\varepsilon)$, which refers to the first projection of evidence (itself a metalanguage pair).
[9]In standard AGT [Garcia et al. 2016] the lifting is simply the identity, *i.e.* EType = GType.

7 EVIDENCE FOR GRADUAL PARAMETRICITY

We now turn to the actual representation of evidence. We first explain in §7.1 why the standard representation of evidence as pair of gradual types is insufficient for gradual parametricity. We then introduce the refined representation of evidence to enforce parametricity (§7.2), and basic properties of the language. Richer properties of GSF are discussed in §8, §9 and §10.

7.1 Simple Evidence, and Why It Fails

In standard AGT [Garcia et al. 2016], evidence is simply represented as a pair of gradual types, *i.e.* EType = GType. Consistent transitivity is defined through the abstract interpretation framework. The definition for simple types is as follows ($\varepsilon \Vdash J$ means ε supports the consistent judgment J):

Definition 7.1 (Consistent transitivity). 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 $(\varepsilon_{ab} \circ \varepsilon_{bc}) \Vdash G_a \sim G_b$, where:

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

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}). 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. The third property is key for establishing the dynamic gradual guarantee [Garcia et al. 2016].

Lemma 7.2. *(Properties of consistent transitivity).*
(a) Associativity. $(\varepsilon_1 \circ \varepsilon_2) \circ \varepsilon_3 = \varepsilon_1 \circ (\varepsilon_2 \circ \varepsilon_3)$*, or both are undefined.*
(b) Optimality. If $\varepsilon = \varepsilon_1 \circ \varepsilon_2$ *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* $\varepsilon_1 \circ \varepsilon_2$ *is defined, then* $\varepsilon_1 \circ \varepsilon_2 \sqsubseteq \varepsilon_1' \circ \varepsilon_2'$.

Unfortunately, adopting gradual types for evidence types and simply extending the consistent transitivity definition to deal with GSF types and consistency judgments yields a gradual language that breaks parametricity.[10] To illustrate, consider this simple program:

```
(ΛX.(λx:X.let y:? = x in let z:? = y in z + 1)) [Int] 1
```

The function is not parametric because it ends up adding 1 to its argument, although it does so after two intermediate bindings, typed as ?. Without further precaution, the parametricity violation of this program would not be detected at runtime. Assume that the type application generates the fresh name α, bound to Int in the store. For justifying that x can flow to y (the let-binding is equivalent to a function application), we need evidence for $\mathsf{Int} \sim ?$ by consistent transitivity between the evidences $\langle \mathsf{Int}, \alpha\rangle$, which justifies $\mathsf{Int} \sim \alpha$,[11] and $\langle \alpha, \alpha\rangle$, which justifies $\alpha \sim ?$.[12] Using the definition of consistent transitivity (Def. 7.1), $\langle \mathsf{Int}, \alpha\rangle \circ \langle \alpha, \alpha\rangle = \langle \mathsf{Int}, \alpha\rangle$. Similarly, for justifying the flow of y to z, the previous evidence must be combined with $\langle ?, ?\rangle$, which justifies $? \sim ?$. By Def. 7.1, $\langle \mathsf{Int}, \alpha\rangle \circ \langle ?, ?\rangle = A^2(\{\langle \mathsf{Int}, \mathsf{Int}\rangle, \langle \mathsf{Int}, \alpha\rangle\}) = \langle \mathsf{Int}, ?\rangle$. This evidence can subsequently be used to produce evidence to justify that the addition is well-typed, since $\langle \mathsf{Int}, ?\rangle \circ \langle \mathsf{Int}, \mathsf{Int}\rangle = \langle \mathsf{Int}, \mathsf{Int}\rangle$. Therefore the program produces 2, without errors: parametricity is violated.

[10] 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. Exploring this perspective is future work.
[11] Note that conversely to the simply-typed setting, both components of evidence are not necessarily equal, as in this case.
[12] This evidence is obtained by substituting α for X in the initial evidence $\langle X, X\rangle$ for $X \sim ?$.

$$\text{(unsl)}\;\frac{\langle E_1, E_2\rangle \circ \langle E_3, E_4\rangle = \langle E_1', E_2'\rangle}{\langle E_1, \alpha^{E_2}\rangle \circ \langle \alpha^{E_3}, E_4\rangle = \langle E_1', E_2'\rangle} \qquad \text{(idL)}\;\frac{}{\langle E, E\rangle \circ \langle ?, ?\rangle = \langle E, E\rangle} \qquad \text{(sealL)}\;\frac{\langle E_1, E_2\rangle \circ \langle E_3, E_4\rangle = \langle E_1', E_2'\rangle}{\langle E_1, E_2\rangle \circ \langle E_3, \alpha^{E_4}\rangle = \langle E_1', \alpha^{E_2'}\rangle}$$

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

$$\text{(func?L)}\;\frac{\langle E_1 \to E_2, E_3 \to E_4\rangle \circ \langle ? \to ?, ? \to ?\rangle = \langle E_1' \to E_2', E_3' \to E_4'\rangle}{\langle E_1 \to E_2, E_3 \to E_4\rangle \circ \langle ?, ?\rangle = \langle E_1' \to E_2', E_3' \to E_4'\rangle}$$

Fig. 5. Consistent Transitivity (selected rules)

7.2 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. We define *evidence types*, $E \in$ ETYPE, to be an enriched version of gradual types:

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

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 therefore 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 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}. Formally, a type name is enriched with its transitive bindings in the store, $\mathit{lift}_\Xi(\alpha) = \alpha^{\mathit{lift}_\Xi(\Xi(\alpha))}$. Unlifting simply forgets the additional information: $\mathit{unlift}_\Xi(\alpha^E) = \alpha$. In all other cases, both operations recur structurally.

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.

Consistent Transitivity. With this syntactic enrichment, consistent transitivity can be strengthened to account for sealing and unsealing, ensuring parametricity. Consistent transitivity is defined inductively; selected rules are presented in Figure 5.

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, $\langle \mathsf{Int}, \alpha^{\mathsf{Int}}\rangle \circ \langle \alpha^?, ?\rangle = \langle \mathsf{Int}, \mathsf{Int}\rangle \circ \langle ?, ?\rangle = \langle \mathsf{Int}, \mathsf{Int}\rangle$.

As shown in §7.1, it is important for consistent transitivity to not lose precision when combining an evidence with an unknown evidence. To this end, rule (identL) in Fig. 5 preserves the left evidence. Going back to the example of §7.1, we now have $\langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle \circ \langle ?, ? \rangle = \langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle$, instead of $\langle \mathsf{Int}, ? \rangle$. Because $\langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle \circ \langle \mathsf{Int}, \mathsf{Int} \rangle$ 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.* $\langle \mathsf{Int}, \mathsf{Int} \rangle \circ \langle ?, \alpha^{?} \rangle = \langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle$.

Figure 5 only shows one structurally-recursive rule, corresponding to the function case (func); consistent transitivity is computed recursively with the domain and codomain evidences. To combine a function evidence with unknown evidence, the unknown evidence is first "expanded" to match the type constructor (func?L). There are similar rules for the other type constructors. Also, there are symmetric variants of the above rules—such as (identR) and (sealR)—in which every evidence and every evidence type is swapped.

Importantly, this refined definition of consistent transitivity preserves associativity and optimality, based on a natural notion of precision for evidence types. It does however break monotonicity,[13] and hence the dynamic gradual guarantee. In §9, we give a semantic argument establishing that the dynamic gradual guarantee is fundamentally incompatible with parametricity anyway, independently of this refinement.

Outer Evidence. The reduction rule of a type application ($R$app$G$) produces an outer evidence ε_{out} that justifies that $G[\alpha/X]$ is consistent with $G[G'/X]$. The precise definition of this evidence is delicate, addressing a subtle tension between the precision required for justifying unsealing when possible, and the imprecision required for parametricity.

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

In this definition, ε, α, G', Ξ, and Ξ' come from rule ($R$app$G$). Determining E_* is the key challenge. The second evidence type of ε refines $\forall X.G$ by exploiting the fact that the underlying polymorphic value $\Lambda X.t$ is consistent with it; this extra precision is crucial for unsealing. The roundtrip unlift/lift "resets" the sealing information of evidence type names to that contained in the store; this relaxation is crucial for parametricity (to prove the compositionality lemma—§8).

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.

Illustration. The following reduction trace illustrates all the important aspects of reduction:

	$(\varepsilon_{\forall X.X\to X}(\Lambda X.\lambda x : X.x) :: \forall X.X \to ?)\ [\mathsf{Int}]\ (\varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int})$	initial evidence
($R$app$G$) $\longmapsto$	$(\langle \alpha^{\mathsf{Int}} \to \alpha^{\mathsf{Int}}, \mathsf{Int} \to \mathsf{Int} \rangle\ (\varepsilon_{\alpha\to\alpha}(\lambda x : \alpha.x) :: \alpha \to ?) :: \mathsf{Int} \to ?)\ (\varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int})$	note the precision of ε_{out}
(Rasc) $\longmapsto$	$(\langle \alpha^{\mathsf{Int}} \to \alpha^{\mathsf{Int}}, \mathsf{Int} \to \mathsf{Int} \rangle\ (\lambda x : \alpha.x) :: \mathsf{Int} \to ?)\ (\varepsilon_{\mathsf{Int}} 1 :: \mathsf{Int})$	consistent transitivity
(Rapp) $\longmapsto$	$\langle \alpha^{\mathsf{Int}}, \mathsf{Int} \rangle\ (\langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle\ 1 :: \alpha) :: ?$	argument is sealed
(Rasc) $\longmapsto$	$\langle \mathsf{Int}, \mathsf{Int} \rangle\ 1 :: ?$	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 (Rapp) 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 (Rasc). As a result, the ascribed identity function yields usable values, because the outer evidence subsequently takes care of unsealing. This addresses the excess of failure reported with λB and System F_C in §2.4. Note that if the function were not intrinsically precise on its return type, *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.

[13] 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, $\langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle \circ \langle \alpha^{\mathsf{Int}}, \mathsf{Int} \rangle = \langle \mathsf{Int}, \mathsf{Int} \rangle$ (rule unsl), and $\langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle \circ \langle ?, ? \rangle = \langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle$ (rule idL), but $\langle \mathsf{Int}, \mathsf{Int} \rangle \not\sqsubseteq \langle \mathsf{Int}, \alpha^{\mathsf{Int}} \rangle$.

7.3 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. $t \Downarrow \mathbf{error}$) if $\vdash t \rightsquigarrow t_\varepsilon : G$ and $\cdot \triangleright t_\varepsilon \longmapsto^* \Xi \triangleright v$ (resp. $\cdot \triangleright t_\varepsilon \longmapsto^* \Xi \triangleright \mathbf{error}$) 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 7.3 (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 5.8 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 7.4 (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. As we will see in §10, a static term is also guaranteed to terminate.

8 GSF: PARAMETRICITY

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 [Ahmed et al. 2017]. In the following, we only go briefly over the definition of the relation (Figure 6), and focus on the few differences with the λB relation, essentially dealing with evidences.

The relation is defined on tuples (W, t_1, t_2) that denote two related terms t_1, t_2 in a world W. A world is composed 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 at the bottom of Figure 6. As usual, the value and term interpretations are indexed by a type and a type substitution ρ. We use $\text{Atom}_n[G_1, G_2]$ to denote a set of pair of terms of type G_1 and G_2, and worlds with a step index less than n. We write $\text{Atom}_n^{\text{val}}[G_1, G_2]$ to restrict that set to values, and $\text{Atom}_\rho[G]$ to denote a set of terms of the same type after substitution. The $\text{Atom}_\rho^{=}[G]$ variant is similar to $\text{Atom}_n^{\text{val}}[G_1, G_2]$ 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). $\text{Rel}_n[G_1, G_2]$ defines the set of relations of values of type G_1 and G_2. 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 (and extension), and similarly $W' \geq W$ expresses that W' is a future world of W. The $\downarrow$ operator lowers the step index of a world by 1.

The logical interpretation of terms of a given type enforces a "termination-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. Note that $\text{Atom}_\rho^{=}[G]$ requires the second component of the evidence of each value to have the same precision in order 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.

Two base values are related if they are equal. Two functions are related if their application to related values yields related results. Two type abstractions are related if given any two types and any relation between them, the instantiated terms (without their unsealing evidence) are also

$$
\begin{array}{lcl}
\mathcal{V}_\rho\llbracket B\rrbracket & = & \{(W,v,v)\in \mathrm{Atom}^{=}_{\rho}[B]\}\\
\mathcal{V}_\rho\llbracket G_1\rightarrow G_2\rrbracket & = & \{(W,v_1,v_2)\in \mathrm{Atom}^{=}_{\rho}[G_1\rightarrow G_2] \mid \forall W'\succeq W.\forall v_1',v_2'.\\
 & & \quad (W',v_1',v_2')\in \mathcal{V}_\rho\llbracket G_1\rrbracket \Rightarrow (W',v_1\ v_1',v_2\ v_2')\in \mathcal{T}_\rho\llbracket G_2\rrbracket\}\\
\mathcal{V}_\rho\llbracket G_1\times G_2\rrbracket & = & \{(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\llbracket G_1\rrbracket \wedge (W,\pi_2(v_1),\pi_2(v_2))\in \mathcal{T}_\rho\llbracket G_2\rrbracket\}\\
\mathcal{V}_\rho\llbracket \forall X.G\rrbracket & = & \{(W,v_1,v_2)\in \mathrm{Atom}^{=}_{\rho}[\forall X.G] \mid \forall W'\succeq W.\forall t_1,t_2,G_1,G_2,\alpha,\varepsilon_1,\varepsilon_2.\\
 & & \quad \forall R\in \mathrm{R{\small EL}}_{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]}\llbracket G\rrbracket\}\\
\mathcal{V}_\rho\llbracket X\rrbracket & = & \mathcal{V}_\rho\llbracket \rho(X)\rrbracket\\
\mathcal{V}_\rho\llbracket \alpha\rrbracket & = & \{(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 (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)\}\\
\mathcal{V}_\rho\llbracket ?\rrbracket & = & \{(W,\varepsilon_1 u_1 :: ?,\varepsilon_2 u_2 :: ?)\in \mathrm{Atom}^{=}_{\emptyset}[?] \mid \mathit{const}(\pi_2(\varepsilon_i)) = G \wedge\\
 & & \quad (W,\varepsilon_1 u_1 :: G,\varepsilon_2 u_2 :: G)\in \mathcal{V}_\rho\llbracket G\rrbracket\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{T}_\rho\llbracket G\rrbracket & = & \{(W,t_1,t_2)\in \mathrm{Atom}_\rho[G] \mid \forall i< W.j, (\forall \Xi_1,v_1.\ W.\Xi_1\triangleright t_1 \longmapsto^{i} \Xi_1\triangleright v_1 \Rightarrow\\
 & & \quad \exists W'\succeq W, v_2.\ W.\Xi_2\triangleright t_2 \longmapsto^{*} W'.\Xi_2\triangleright v_2 \wedge W'.j+i = W.j \wedge\\
 & & \quad W'.\Xi_1 = \Xi_1 \wedge (W',v_1,v_2)\in \mathcal{V}_\rho\llbracket G\rrbracket) \wedge\\
 & & \quad (\forall \Xi_1.W.\Xi_1\triangleright t_1 \longmapsto^{i} \mathbf{error} \Rightarrow \exists \Xi_2.W.\Xi_2\triangleright t_2 \longmapsto^{*} \mathbf{error})\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{S}\llbracket \cdot\rrbracket & = & \mathrm{W{\small ORLD}}\\
\mathcal{S}\llbracket \Xi,\alpha:=G\rrbracket & = & \mathcal{S}\llbracket \Xi\rrbracket \cap \{W\in \mathrm{W{\small ORLD}} \mid W.\Xi_1(\alpha)=G \wedge W.\Xi_2(\alpha) = G\wedge\\
 & & \vdash W.\Xi_1 \wedge \vdash W.\Xi_2 \wedge W.\kappa(\alpha) = \lfloor \mathcal{V}_\emptyset\llbracket G\rrbracket\rfloor_{W.j}\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{D}\llbracket \cdot\rrbracket & = & \{(W,\emptyset)\mid W\in \mathrm{W{\small ORLD}}\}\\
\mathcal{D}\llbracket \Delta,X\rrbracket & = & \{(W,\rho[X\mapsto\alpha]) \mid (W,\rho)\in \mathcal{D}\llbracket\Delta\rrbracket \wedge \alpha\in \mathit{dom}(W.\kappa)\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{G}_\rho\llbracket \cdot\rrbracket & = & \{(W,\emptyset)\mid W\in \mathrm{W{\small ORLD}}\}\\
\mathcal{G}_\rho\llbracket \Gamma,x:G\rrbracket & = & \{(W,\gamma[x\mapsto(v_1,v_2)]) \mid (W,\gamma)\in \mathcal{G}_\rho\llbracket\Gamma\rrbracket \wedge (W,v_1,v_2)\in \mathcal{V}_\rho\llbracket G\rrbracket\}
\end{array}
$$

$$
\begin{array}{lcl}
\Xi;\Delta;\Gamma\vdash t_1 \preceq t_2 : G & \triangleq & \Xi;\Delta;\Gamma\vdash t_1 : G \wedge \Xi;\Delta;\Gamma\vdash t_2 : G \wedge \forall W\in \mathcal{S}\llbracket\Xi\rrbracket,\rho,\gamma.\\
 & & ((W,\rho)\in \mathcal{D}\llbracket\Delta\rrbracket \wedge (W,\gamma)\in \mathcal{G}_\rho\llbracket\Gamma\rrbracket) \Rightarrow (W,\rho(\gamma_1(t_1)),\rho(\gamma_2(t_2)))\in \mathcal{T}_\rho\llbracket G\rrbracket\\
\Xi;\Delta;\Gamma\vdash t_1 \approx t_2 : G & \triangleq & \Xi;\Delta;\Gamma\vdash t_1 \preceq t_2 : G \wedge \Xi;\Delta;\Gamma\vdash t_2 \preceq t_1 : G
\end{array}
$$

$$
\begin{array}{rl}
\mathrm{Atom}_n[G_1,G_2] = & \{(W,t_1,t_2)\mid W.j<n \wedge W\in \mathrm{W{\small ORLD}} \wedge W.\Xi_1,\cdot,\cdot\vdash t_1 : G_1 \wedge W.\Xi_2,\cdot,\cdot\vdash t_2 : G_2\}\\
\mathrm{Atom}^{\mathrm{val}}_n[G_1,G_2] = & \{(W,v_1,v_2)\in \mathrm{Atom}_n[G_1,G_2]\} \qquad \mathrm{Atom}_\rho[G] = \cup_{n\geq 0}\{(W,t_1,t_2)\in \mathrm{Atom}_n[\rho(G),\rho(G)]\}\\
\mathrm{Atom}^{=}_{\rho}[G] = & \{(W,v_1,v_2)\in \mathrm{Atom}_\rho[G] \mid \mathit{unlift}(\pi_2(\mathit{ev}(v_1))) = \mathit{unlift}(\pi_2(\mathit{ev}(v_2)))\}\\
\mathrm{W{\small ORLD}} = & \cup_{n\geq 0}\, \mathrm{W{\small ORLD}}_n\\
\mathrm{W{\small ORLD}}_n = & \{(j,\Xi_1,\Xi_2,\kappa)\in \mathrm{N{\small AT}}\times\mathrm{S{\small TORE}}\times\mathrm{S{\small TORE}}\times(\mathrm{T{\small YPE}N{\small AME}}\rightarrow \mathrm{R{\small EL}}_j)\mid\\
 & \quad j<n \wedge \vdash \Xi_1 \wedge \vdash \Xi_2 \wedge \forall\alpha\in \mathit{dom}(\kappa).\kappa(\alpha)\in \mathrm{R{\small EL}}_j[\Xi_1(\alpha),\Xi_2(\alpha)]\}\\
\mathrm{R{\small EL}}_n[G_1,G_2] = & \{R\in \mathrm{Atom}^{\mathrm{val}}_n[G_1,G_2]\mid \forall(W,v_1,v_2)\in R.\forall W'\succeq W.(W',v_1,v_2)\in R\}\\
\lfloor R\rfloor_n = & \{(W,e_1,e_2)\in R\mid W.j\leq n\} \qquad \lfloor\kappa\rfloor_n = \{\alpha\mapsto\lfloor R\rfloor_n \mid \kappa(\alpha)=R\}\\
\kappa'\succeq\kappa \triangleq & \forall\alpha\in \mathit{dom}(\kappa).\kappa'(\alpha) = \kappa(\alpha)\\
W'\succeq W \triangleq & W'.j\leq W.j \wedge W'.\Xi_1 \supseteq W.\Xi_1 \wedge W'.\Xi_2\supseteq W.\Xi_2 \wedge W'.\kappa\succeq \lfloor W.\kappa\rfloor_{W'.j} \wedge W',W\in \mathrm{W{\small ORLD}}\\
\downarrow W = & (j,W.\Xi_1,W.\Xi_2,\lfloor W.\kappa\rfloor_j) \quad \text{where } j = W.j-1
\end{array}
$$

Fig. 6. Gradual logical relation and auxiliary definitions

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. Note that the step index of this extended world is decreased by one, because we take a reduction step. Two pairs are related if their components are pointwise related. Two sealed values are related at a type name α if, after unsealing, the resulting values are in the relation corresponding to α in the current world, $W.\kappa(\alpha)$.

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, $const(\pi_2(\varepsilon_i))$.[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.* $const(E_1 \to E_2)$).

The logical relation is well-founded for two 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 indexed is lowered.

The interpretations of stores, type name environments and type environments are straightforward (Figure 6). The logical relation allows us to define logical approximation, whose symmetric extension is logical equivalence. Any well-typed GSFε term is related to itself at its type:

Theorem 8.1 (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. Almost every compatibility lemma relies on the fact that the ascription of two related values yield related terms.

Lemma 8.2 (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* $(W, \rho_1(\varepsilon)v_1 :: \rho(G'), \rho_2(\varepsilon)v_2 :: \rho(G')) \in \mathcal{T}_\rho[\![G']\!]$.

Note that type substitution on evidences takes as parameter the corresponding store: $\rho_i(\varepsilon)$ is syntactic sugar for $\rho(W.\Xi_i, \varepsilon)$, lifting each substituted type name in the process, *e.g.* if $\rho(X) = \alpha$, $W.\Xi_1(\alpha) = \mathsf{Int}$, and $W.\Xi_2(\alpha) = \mathsf{Bool}$, then $\rho_1(\langle X, X\rangle) = \langle \alpha^{\mathsf{Int}}, \alpha^{\mathsf{Int}}\rangle$, and $\rho_2(\langle X, X\rangle) = \langle \alpha^{\mathsf{Bool}}, \alpha^{\mathsf{Bool}}\rangle$.

9 PARAMETRICITY VS. DYNAMIC GRADUAL GUARANTEE

We now turn to the dynamic gradual guarantee [Siek et al. 2015a]. 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 $v \sqsubseteq v'$. We show that parametricity as defined in §8 is however incompatible with this guarantee. First, we can prove the following lemma:

Lemma 9.1. *For any* $\vdash v : ?$ *and* $\vdash G$, *we have* $(\Lambda X.\lambda x : ?.x :: X)\ [G]\ v \Downarrow \mathbf{error}$.

Proof. Let $v' = (\Lambda X.\lambda x : ?.x :: X)$, $\vdash v' \rightsquigarrow v_\forall : \forall X.? \to X$, and v s.t. $\vdash v \rightsquigarrow v_? : ?$.

By the fundamental property (Th. 8.1), $\vdash v_\forall \preceq v_\forall : \forall X.? \to X$ so for any $W_0 \in \mathcal{S}[\![\cdot]\!]$, $(W_0, v_\forall, v_\forall) \in \mathcal{T}_\emptyset[\![\forall X.? \to X]\!]$. Because $v_\forall$ is a value, $(W_0, v_\forall, v_\forall) \in \mathcal{V}_\emptyset[\![\forall X.? \to X]\!]$. By reduction, $\cdot \triangleright v_\forall\ [G_i] \longmapsto^* \Xi'_i \triangleright \varepsilon'_i v_i :: ? \to G_i$ for some ε'_i, ε_i and $\varepsilon_{i\alpha}$, where $\Xi'_i = \{\alpha = G_i\}$ and $v_i = \varepsilon_i(\lambda x : ?.(\varepsilon_{i\alpha} x :: \alpha)) :: ? \to \alpha$. We can instantiate the definition of $\mathcal{V}_\emptyset[\![\forall X.? \to X]\!]$ with W_0, $G_1 = G$ and G_2 structurally different (and different from ?), some $R \in \textsc{Rel}_{W_0.j}[G_1, G_2]$, v_1, v_2, ε'_1 and ε'_2, then we have that $(W_1, v_1, v_2) \in \mathcal{T}_{X\mapsto\alpha}[\![? \to X]\!]$, where $W_1 = (\downarrow (W_0 \boxtimes (\alpha, G_1, G_2, R))$. As v_1 and v_2 are values, $(W_1, v_1, v_2) \in \mathcal{V}_{X\mapsto\alpha}[\![? \to X]\!]$. Also, by associativity of consistent transitivity, the reduction of $\Xi'_i \triangleright (\varepsilon'_i v_i :: ? \to G_i)\ v_?$ is equivalent to that of $\Xi'_i \triangleright cod(\varepsilon'_i)(v_i\ (dom(\varepsilon'_i)v_? :: ?)) :: G_i$.

By the fundamental property (Th. 8.1) we know that $\vdash v_? \preceq v_? : ?$; we can instantiate this definition with some $W_2 \geq W_1$, and we have that $(W_2, v_?, v_?) \in \mathcal{T}_\emptyset[\![?]\!]$. Since $v_?$ is a value, $(W_2, v_?, v_?) \in$

[14] *const* extracts the top-level constructor of an evidence type, *e.g.* $const(E_1 \to E_2) = ? \to ?$ and $const(\forall X.E) = \forall X.?$.

$\mathcal{V}_{X\mapsto\alpha}[\![?]\!]$. By the ascription lemma (8.2), $(W_2, dom(\varepsilon'_1)v_? :: ?, dom(\varepsilon'_2)v_? :: ?) \in \mathcal{T}_\rho[\![?]\!]$. If $dom(\varepsilon'_1)v_? :: ?$ reduces to **error** then the result follows immediately. Otherwise, $\Xi'_i \triangleright dom(\varepsilon'_1)v_? :: ? \longmapsto^* \Xi'_i \triangleright v''_i$, and $(W_3, v''_1, v''_2) \in \mathcal{V}_\rho[\![?]\!]$, where $W_3 =\downarrow W_2$, and some v''_1 and v''_2. We can instantiate the definition of $\mathcal{V}_{X\mapsto\alpha}[\![? \to X]\!]$ with W_3, v''_1 and v''_2, obtaining that $(W_3, v_1\ v''_1, v_2\ v''_2) \in \mathcal{T}_{X\mapsto\alpha}[\![X]\!]$. We then proceed by contradiction. Suppose that $\Xi'_i \triangleright v_i\ v''_i \longmapsto^* \Xi''_i \triangleright v'_i$ (for a big-enough step index). If $v''_i = \varepsilon''_{iv}u :: ?$, then by evaluation $v'_i = \varepsilon'_{iv}u :: \alpha$, for some ε'_{iv}. But by definition of $\mathcal{V}_{X\mapsto\alpha}[\![X]\!]$, it must be the case that for some $W_4 \geq W_3$, $(W_4, \varepsilon'_{1v}u :: G_1, \varepsilon'_{2v}u :: G_2) \in R$, which is impossible because u cannot be ascribed to structurally different types G_1 and G_2. Therefore $v_1\ v''_1$ cannot reduce to a value, and hence the term $v_\forall\ [G]\ v_?$ cannot reduce to a value either. Because $v_\forall$ is non-diverging, its application must produce **error**. □

Consequently, the dynamic gradual guarantee is violated:

Corollary 9.2. *There exist $\vdash t_1 : G$ and $t_2 \sqsupseteq t_1$ such that $t_1 \Downarrow v$ 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 9.1, $t_2 \Downarrow$ **error**. □

Interestingly, Lemma 9.1 holds irrespective of the actual choices for representing evidence in GSFε. The key element is the (standard) logical interpretation of $\forall X.G$. Therefore the incompatibility described here does not apply only to GSF: in fact, we have been able to prove that Lemma 9.1 also holds in λB [Ahmed et al. 2017], whose notion of parametricity is essentially the same as GSF.

By sticking to this standard notion of parametricity, one way to accommodate the dynamic gradual guarantee is to change the definition of precision, as done by Igarashi et al. [2017a] (denying that $t_1 \sqsubseteq t_2$ in the proof of Corollary 9.2). We believe this is questionable, because precision is a syntactic and intuitive notion describing "how static a type is", and replacing parts of a type with ? is clearly making it "less static" (recall §2.3). 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]. Currently, it seems that the incompatibility of the dynamic gradual guarantee with parametricity has to be understood, in conjunction with a similar observation regarding noninterference [Toro et al. 2018a], as hinting towards further refined criteria for semantically-rich gradual typing. In particular, weaker forms of the dynamic gradual guarantee might still be useful, as explored next.

10 GRADUAL FREE THEOREMS IN GSF

The parametricity logical relation (§8) 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

$$
\begin{aligned}
\mathcal{N}_{\rho}^{\Sigma}[\![B \sqsubseteq G]\!] &= \{v = \varepsilon b :: G' \mid \Sigma; \cdot; \cdot \vdash v : G\} \\
\mathcal{N}_{\rho}^{\Sigma}[\![T_1 \rightarrow T_2 \sqsubseteq G]\!] &= \{v = \varepsilon u :: G' \mid v \in \mathrm{ImpSV}_{\rho}^{\Sigma}[T_1 \rightarrow T_2 \sqsubseteq G] \wedge \\
&\qquad \forall v' \in \mathcal{N}_{\rho}^{\Sigma}[\![T_1 \sqsubseteq dom^{\sharp}(G)]\!], (\varepsilon u :: dom^{\sharp}(G') \rightarrow cod^{\sharp}(G'))\; v' \in \mathcal{C}_{\rho}^{\Sigma}[\![T_2 \sqsubseteq cod^{\sharp}(G)]\!]\} \\
\mathcal{N}_{\rho}^{\Sigma}[\![\forall X.T \sqsubseteq G]\!] &= \{v = \varepsilon u :: G' \mid v \in \mathrm{ImpSV}_{\rho}^{\Sigma}[\forall X.T \sqsubseteq G] \wedge (\forall T', \Sigma \vdash T', \Sigma \triangleright (\varepsilon u :: \forall X.schm^{\sharp}(G'))[T'] \\
&\qquad \longmapsto \Sigma, \alpha := T' \triangleright \varepsilon' t' :: G' \wedge t' \in \mathcal{C}_{\rho[X \mapsto \alpha]}^{\Sigma, \alpha := T'}[\![T \sqsubseteq schm^{\sharp}(G)]\!])\} \\
\mathcal{N}_{\rho}^{\Sigma}[\![T_1 \times T_2 \sqsubseteq G]\!] &= \{v = \varepsilon u :: G' \mid v \in \mathrm{ImpSV}_{\rho}^{\Sigma}[G_1 \times G_2 \sqsubseteq G] \wedge \\
&\qquad (p_i(\varepsilon)\pi_i(u) :: proj_i^{\sharp}(G')) \in \mathcal{N}_{\rho}^{\Sigma}[\![T_i \sqsubseteq proj_i^{\sharp}(G)]\!]\} \\
\mathcal{N}_{\rho}^{\Sigma}[\![X \sqsubseteq G]\!] &= \mathcal{N}_{\rho}^{\Sigma}[\![\rho(X) \sqsubseteq \rho(G)]\!] \\
\mathcal{N}_{\rho}^{\Sigma}[\![\alpha \sqsubseteq G]\!] &= \{v = \varepsilon u :: G' \mid v \in \mathrm{ImpSV}_{\rho}^{\Sigma}[\alpha \sqsubseteq G] \wedge \Sigma(\alpha) = T \wedge \\
&\qquad \forall G', T \sqsubseteq G', (\langle \pi_1(\varepsilon), lift_{\Sigma}(T) \rangle u :: G \in \mathcal{C}_{\rho}^{\Sigma}[\![T \sqsubseteq G']\!]\} \\
\mathcal{C}_{\rho}^{\Sigma}[\![T \sqsubseteq G]\!] &= \{t \mid \Sigma; \cdot; \cdot \vdash t : \Sigma(\rho(G)) \wedge \Sigma \triangleright t \longmapsto^{*} \Sigma' \triangleright v \wedge v \in \mathcal{N}_{\rho}^{\Sigma'}[\![T \sqsubseteq G]\!]\} \\
\mathrm{ImpSV}_{\rho}^{\Sigma}[T \sqsubseteq G] &= \{v = \varepsilon u :: G' \mid static(u) \wedge \pi_2(\varepsilon) = lift_{\Sigma}(\rho(T)) \wedge \Sigma; \cdot; \cdot \vdash v : \rho(G)\} \\
\Sigma; \Delta; \Gamma \models t : T \sqsubseteq G &\triangleq \Sigma; \Delta; \Gamma \vdash t : G \wedge \forall \rho \in \mathcal{D}^{\Sigma}[\![\Delta]\!], \forall \gamma \in \mathcal{G}_{\rho}^{\Sigma}[\![\Gamma]\!], \rho(\gamma(t)) \in \mathcal{C}_{\rho}^{\Sigma}[\![T \sqsubseteq G]\!]
\end{aligned}
$$

Fig. 7. Imprecise termination logical predicate.

of type $\forall X.? \rightarrow X$ are always failing. Consequently, System F_G rejects such a cast by tweaking the precision relation (§2.3). But the corresponding free theorem is not proven. Also, Ahmed et al. [2011] declare that parametricity dictates that any value of type $\forall X.X \rightarrow ?$ 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. 2009] and its newest version [Ahmed et al. 2017], as well as in System F_G, casting the identity function to $\forall X.X \rightarrow ?$ yields a function that returns *without errors*, though the returned value is still sealed, and as such unusable (§2.4). Considering that the underlying function is intrinsically parametric, why shall we expect it to fail or return unusable values? In fact, while the specific choice of runtime semantics may decree failure, such behavior is *not* imposed by the parametricity relation *per se*. Parametricity only imposes uniformity of behavior, including failure, of polymorphic terms, which leaves some freedom regarding when to fail.

Disproving Gradual Free Claims. We uncover a novel property of GSF: it preserves the strong normalization property of System F terms *even as they are ascribed to less precise types*, as long as they are used with similarly-terminating terms, and instantiated at static types.

We establish this result using a logical predicate, named *imprecise termination* (Figure 7[15]), whose statement $\models t : T \sqsubseteq G$ expresses that t is a static term of type T that has been ascribed a less precise type G. As usual, the predicates for values and terms carry a type environment and type name store; we do not need step indexing because the logical relation is defined inductively on the structure of T (not G). At the function type, the predicate specifies that when applied to an imprecisely-terminating argument, the application terminates and yields an imprecisely-terminating result. For type application, only static type instantiations are considered. The predicate $\mathrm{ImpSV}_{\rho}^{\Sigma}[T \sqsubseteq G]$ characterizes imprecisely-ascribed static values. The rest of the definitions are essentially administrative ascriptions to align types as required by GSFε.

Static terms satisfy the imprecise termination predicate, and are hence hereditarily terminating:

[15] $schm^{\sharp}$ (consistently) extracts the schema of a gradual type, *i.e.* $schm^{\sharp}(\forall X.G) = G$, $schm^{\sharp}(?) = ?$, undefined o/w.

Lemma 10.1. *Let t be a static term. If $\vdash t : T$ and $T \sqsubseteq G$, then $\vdash (t :: G) \rightsquigarrow t' : G$ and $\models t' : T \sqsubseteq G$.*

This property is related to, but weaker than the dynamic gradual guarantee. Nevertheless, it is powerful enough to disprove the claims from the literature about $\forall X.? \to X$ and $\forall X.X \to ?$: both types admit the ascribed System F identity function, among many others,[16] as a non-constant, non-failing, parametricity-preserving inhabitant. We believe this result constitutes a valuable compositionality guarantee when embedding fully-static (System F) terms in a gradual world. Another corollary is that closed static terms always terminate (by $\models t : T \sqsubseteq T$), hence superseding Proposition 7.4.

Cheap Theorems. The intuition of $\forall X.? \to X$ denoting always-failing functions is not entirely misguided: 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 10.2. *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 holds independently of the body t, therefore *without having to analyze the whole term.* Not as good as a free theorem, but cheap.

11 RELATED WORK

We have already discussed at length related work on gradual parametricity, especially the most recent developments [Ahmed et al. 2017; Igarashi et al. 2017a; Xie et al. 2018]. In addition to static semantics issues in λB and System F_G, all theses languages suffer from dynamic semantics that do not accurately track type instantiations (§2.4). Note that, conversely to λB, GSF does not impose any syntactic value restriction on polymorphic terms; such a restriction might be necessary when exploring the extension of GSF with implicit polymorphism. Finally, instead of leaving the dynamic gradual guarantee as a conjecture, we show that it is incompatible with parametricity, at least given the standard definitions of both notions. Note that some language features are also known to break the dynamic gradual guarantee, such as structural type tests and object identity [Siek et al. 2015a], as well as method overloading and extension methods [Muehlboeck and Tate 2017].

The relation between parametric polymorphism in general and dynamic typing much 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]. Pierce and Sumii [2000] 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. It would be interesting to extend their simply-typed setting to parametric polymorphism, and study the interplay of parametricity and graduality when casts, and possibly seals, are explicit as in the work of Neis et al. [2009] on parametricity in a non-parametric language.

[16] *e.g.* $\Lambda X.\lambda x : X.\lambda f : X \to X.f\ x$ of type $\forall X.X \to (X \to X) \to X$ can also be ascribed to $\forall X.X \to ?$.

Devriese et al. [2018] disprove a conjecture by Pierce and Sumii [2000] according to which the compilation of System F to an untyped language with dynamic sealing is fully abstract, *i.e.* preserves contextual equivalences. They show that, for similar reasons, the embedding of System F in current polymorphic blame calculi 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. Imprecise termination (§10) is a weaker, yet useful result that sheds light on gradual free theorems about imprecise type signatures. It should be possible to generalize this result to account for the harmless content of imprecise ascriptions; we leave this perspective for future work.

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. 2018a]. In these systems, one aims at preserving *noninterference*, *i.e.* that private values dot 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. [2018a] find that in 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 needs a more precise abstraction for evidence types (based on security *label intervals*) in order to enforce noninterference. Together, these results suggest that new criteria are needed to characterize the spectrum of type-based reasoning that gradual typing supports when applied to semantically-rich disciplines.

12 CONCLUSION

We uncover design flaws in prior work on gradual parametric languages that enforce relational parametricity. We exploit the Abstracting Gradual Typing (AGT) methodology to design a new gradual language with explicit parametric polymorphism, GSF. 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 parametricity. We show that parametricity is, like noninterference [Toro et al. 2018a], incompatible with the dynamic gradual guarantee laid forth by Siek et al. [2015a]. We nevertheless establish a novel, weaker property of GSF regarding the embedding of System F terms at less precise types, which allows us to disprove some claims from the literature about gradual free theorems.

Future work also includes extending GSF and its associated reasoning with existential types, both in terms of their encoding, and as primitives in the language. We shall also study the integration of implicit polymorphism on top of GSF, most likely following the approach of Xie et al. [2018]. Finally, it would be interesting to understand whether the evidence-based runtime semantics presented here can be used to derive a cast calculus akin to λB, and then address efficiency considerations.

ACKNOWLEDGMENTS

We thank Amal Ahmed, Dominique Devriese, Kenji Maillard, Gabriel Scherer, the attendees of various oral presentations of this work, and the anonymous reviewers, for useful feedback and suggestions that 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, Robert Bruce Findler, Jacob Matthews, and Philip Wadler. 2009. 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.

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.

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.

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.

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.*

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. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016).* ACM Press, St Petersburg, FL, USA, 429–442.

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.

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Sympolic 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.

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.

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.

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

Fabian Muehlboeck and Ross Tate. 2017. Sound gradual typing is nominally alive and well. , 56:1–56:30 pages.

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.

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.

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 *Porceedings of the Programming Symposium (Lecture Notes in Computer Science)*, Vol. 19. Springer-Verlag, 408–423.

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

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.

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. 2018a. 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. 2018b. Gradual Parametricity, Revisited (with Appendix). arXiv:1807.04596 [cs.PL].

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.

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.

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.