Pragmatic Gradual Polymorphism with References

Wenjia Ye and Bruno C. d. S. Oliveira

The University of Hong Kong, Hong Kong SAR, China
{wjye,bruno}@cs.hku.hk

Abstract. Gradualizing System F has been widely discussed. A big challenge is to preserve relational parametricity and/or the gradual guarantee. Most past work has focused on the preservation of parametricity, but often without the gradual guarantee. A few recent works satisfy both properties by giving up System F syntax, or with some restrictions and the introduction of sophisticated mechanisms in the dynamic semantics.

While parametricity is important for polymorphic languages, most mainstream languages typically do not satisfy it, for a variety of different reasons. In this paper, we explore the design space of polymorphic languages that satisfy the gradual guarantee, but do not preserve parametricity. When parametricity is not a goal, the design of polymorphic gradual languages can be considerably simplified. Moreover, it becomes easy to add features that are of practical importance, such as mutable references. We present a new gradually typed polymorphic calculus, called λ^G_{gpr}, with mutable references and with an easy proof of the gradual guarantee. In addition, compared to other gradual polymorphism work, λ^G_{gpr} is defined using a Type-Directed Operational Semantics (TDOS), which allows the dynamic semantics to be defined directly instead of elaborating to a target cast language. λ^G_{gpr} and all the proofs in this paper are formalized in Coq.

Keywords: Gradual Typing · Type System · Polymorphism.

1 Introduction

Statically typed languages can statically detect potential errors in programs, but must necessarily be conservative and reject some well-behaved programs. With dynamically typed languages, all programs are accepted, which offers a great amount of flexibility. However, the accepted dynamic programs include programs with type errors, making it harder to detect programs that are ill-behaved because of type errors. Considering the weaknesses and advantages of static and dynamic type systems, many approaches have proposed to integrate these two spectrums [1, 7, 8, 22, 35]. *Gradual typing* [31, 35] provides a smooth integration of the two styles and has been under active research in the programming languages community. In addition to the type soundness property, a gradual language should behave as a static language if it is fully annotated. Conversely, it should behave as a dynamic language for fully dynamic programs. Importantly, the *gradual guarantee* [32] has been proposed to ensure a smooth transition between static and dynamic typing.

The importance of System F as a foundation for programming languages with polymorphism naturally leads to the question of whether it is possible to gradualize it. Various researchers have explored this question. In this line of research, a long-standing goal

has been how to preserve *relational parametricity* [28]. Parametricity ensures a uniform behavior for all instantiations of polymorphic functions, and is an important property of System F. In addition it is also desirable to preserve the *gradual guarantee* [32], which is recognized as an important property for gradual languages. Unlike System F, where no dynamic mechanism is needed to ensure parametricity, with gradualized versions of System F this is no longer the case. Ahmed *et al.* [3] showed that parametricity can be enforced using a *dynamic sealing* mechanism at runtime. They prove parametricity, but the gradual guarantee is not discussed. Igarashi *et al.* [17] improved on the dynamic sealing approach and proposed a more efficient mechanism. While the gradual guarantee has been discussed, it was left as a conjecture. Toro *et al.* [37] even proved that gradual guarantee and parametricity are incompatible. By giving up the traditional System F syntax, New *et al.* [24] proved the gradual guarantee and parametricity by using user-provided sealing annotations, but this requires resorting to syntax that is not based on System F. Finally, Labrada *et al.* [20] proved the gradual guarantee and parametricity by inserting sealing with some restrictions. For instance, only base and variable types can be used to instantiate type applications.

While parametricity is highly valued and it is guaranteed in practice in some functional languages, many mainstream programming languages – such as Java, TypeScript or Flow – do not have parametricity. In mainstream languages the value of parametric polymorphism, and its ability to express a whole family of functions in a reusable and type-safe manner is certainly recognized. However, such languages are imperative and come with a variety of programming language features (such as unrestricted forms of mutable state, exceptions, parallelism and concurrency mechanisms, reflection, etc.) that make it hard to apply reasoning principles known in functional programming. In particular, most of those features are known to be highly challenging to deal with in the presence of parametricity [2, 18, 23]. This makes it non-obvious how to design a language with all those features, while preserving parametricity, in the first place. Moreover, preserving parametricity may require extra dynamic checks at runtime, which for implementations where performance is a critical factor may discourage implementers from doing such checks. Therefore all the aforementioned programming languages support System F like mechanisms to deal with polymorphism and benefit from the reuse afforded by polymorphism. However, the reasoning principles that arise from polymorphism, such as parametricity is discarded, and parametricity is not enforced.

In particular, programming languages such as TypeScript or Flow, which support some form of gradual/optional typing, and are widely used in practice, do not support parametricity. Figure 1 encodes an example from Ahmed *et al.*'s work [3], which was used to illustrate the parametricity challenge in gradual typing, in TypeScript and Flow. In this program, the polymorphic function `Ks` has a polymorphic type: $(X \to Y \to Y)$, where X and Y are type variables. In a calculus with parametricity, we know that a function with such type should always return the *second* argument or, in the presence of runtime casts, return an error. In the program, `Ks` is as a function that casts a dynamic constant function (`K`) that returns the *first* argument, which violates parametricity. When the TypeScript and Flow programs are run the first argument 2 is returned, illustrating that both languages do not enforce parametricity. In a gradual language with parametricity the result that we would expect is an error. Furthermore, even if we turn to Typed

```
function K(x:any, y:any): any {
    return x;
}

function Ks<X, Y>(x: X, y: Y): Y {
    let CAST = (K as any) as ((x:
  X, y: Y) ⇒ Y);
    return CAST(x, y);
}

function run() {
    console.log(Ks<number,
  number>(2,3));
}
```

(a) TypeScript code.

```
function K(x:any, y:any): any {
      return x;
}

function Ks<X, Y>(x: X, y: Y): Y {
    let CAST = ((K : any) : ((x:
  X, y: Y) ⇒ Y));
    return CAST(x, y);
}

function run() {
    console.log(Ks (2,3));

}
```

(b) Flow code.

Fig. 1: Ahmed *et al.* [3] program for illustrating parametricity in TypeScript and Flow.

Racket [36], which is a well-established gradual language used in both gradual typing research and in practice, the result is similar and 2 is returned:

```
(: K Any)
(define K ( λ (x) ( λ (y) x)))
(define Ks
  (cast K (All (X Y) (→ X (→ Y Y)))))
((Ks 2) 3)
```

Therefore Typed Racket does not enforce parametricity either.

In this paper, we explore the more pragmatic design space of polymorphic gradual languages with the gradual guarantee, but no parametricity. We believe that such designs are relevant because many practical language designs do not support parametricity, but support various other programming features instead. Dropping the requirement for parametricity enables us to explore language designs with many relevant practical features, while being in line with current designs for existing practical gradually typed languages. In particular, this paper studies the combination of parametric polymorphism, gradual typing and references. We show that, when parametricity is not a goal, the design of gradually polymorphic languages can be simplified, making it easier to add features such as references. Moreover, the gradual guarantee, which has shown to be quite problematic in all existing calculi with gradual polymorphism, is simple to achieve. We present a standard static calculus with polymorphism and mutable references called λ_{gpr}. Then we introduce the gradual counterpart, called λ^{G}_{gpr}.

The approach that we follow to give the dynamic semantics to λ^{G}_{gpr} is to use the recently proposed Type-Directed Operational Semantics TDOS [16, 42]. In contrast, traditionally the semantics of a gradually typed language is defined by elaboration to a target *cast calculus* such as the *blame calculus* [39]. In other words, the dynamic semantics of the gradual source language is given *indirectly* by translating to the target

language. As Ye *et al.* [42] shows, TDOS avoids such indirection and uses bidirectional typing and type annotations to enforce both *implicit* and *explicit* casting at runtime in gradually typed languages.

In summary, we make the following contributions in this paper:

- **The λ^G_{gpr} calculus:** A gradual calculus with polymorphism and mutable references. λ^G_{gpr} calculus is the gradual counterpart of the λ_{gpr} calculus. Both λ^G_{gpr} and λ_{gpr} are shown to be *type sound* and *deterministic*.
- **Gradual guarantee for λ^G_{gpr}.** We prove the *gradual guarantee* for λ^G_{gpr}. The proof is easy and quite simple, in contrast to previous work in gradual polymorphism, where the gradual guarantee was a major obstacle.
- **A TDOS extension.** TDOS has been applied to gradual typing before [42]. However, the previous work on TDOS for gradual typing only works in a purely functional, simply typed calculus. Our work shows that the TDOS approach can incorporate other features, including polymorphism and references.
- **A mechanical formalization in the Coq theorem prover.** All the calculi and proofs in this paper have been mechanically formalized in the Coq theorem prover. The Coq formalization can be found in the supplementary materials of this paper:

`https://www.zenodo.org/badge/latestdoi/581421930`

2 Overview

This section provides a background for gradual polymorphic calculi, calculi with gradual references and the key ideas of our static system (λ_{gpr}) with polymorphism and references and its gradual counterpart (λ^G_{gpr}).

2.1 Background

Gradual References. Mutable references read or write content into a memory cell. A common set of operations is: allocating a memory cell (ref e); updating references ($e_1 := e_2$) and reading the content from a reference ($!e$). Locations (o) point to the memory cell. For a reference value ref 1, a new location (o) is generated and value 1 is stored in the cell at the location o. If 2 is assigned to this location $o := 2$, the cell value is updated to 2. Later, when we read this cell ($!o$), 2 is returned. Siek *et al.* [31] defined an *invariant* consistency relation for reference types. Reference types are only consistent with themselves. For example:

$$(\lambda x.\,(x := 2) : \text{Ref} \star \to \text{Ref} \star)\,(\text{ref } 1) \qquad \text{– Rejected!} \quad \text{Ref Int} \not\sim \text{Ref} \star$$

Although the type Int is consistent with $\star$, it does not mean that Ref Int is consistent with Ref $\star$. Therefore, the argument type is not consistent with the function input, and the program is rejected. Herman *et al.* [14] proposed a gradually typed lambda source language with references, which defines the dynamic semantics by elaborating to a coercion calculus. The above program is allowed in their calculus. They define *variant* consistency where if A is consistent with B then Ref A is consistent with Ref B. In their

calculus, casts are combined to achieve space-efficiency. Furthermore, Siek *et al.* [33] explored monotonic references with variant consistency. Their main consideration is space efficiency. No runtime overhead is imposed in the statically typed part of programs. All the above works have not considered the gradual guarantee.

Toro and Tanter [38] showed how to employ the Abstracting Gradual Typing (AGT) [12] methodology to design a gradually typed calculus with mutable references ($\lambda_{\widetilde{REF}}$). Their dynamic semantics of the source language is defined by translating to an evidence base calculus. They prove a bisimulation with the coercion calculus by Herman et al. [14]. $\lambda_{\widetilde{REF}}$ is proved to satisfy the gradual guarantee. The consistency of $\lambda_{\widetilde{REF}}$ is also variant.

Gradual Polymorphism. Gradual polymorphism is a popular topic. Researchers have been working in this area for a long time. Prior work has focused on two key properties: *relational parametricity* [28] and the *gradual guarantee* [32]. Relational parametricity ensures that all instantiations to a polymorphic value behave uniformly. The gradual guarantee ensures that less dynamic programs behave the same as more static programs.

Satisfying these two properties at once has shown to be problematic. Ahmed *et al.* [3] showed that a naive combination of the unknown type $\star$ and type substitution breaks relational parametricity. They show the problem using a simple expression with two casts. To simplify the presentation, we ignore blame labels. Suppose that $K^\star = \lceil \lambda x.\lambda y.x \rceil$, the dynamically typed constant function, is cast to a polymorphic type:

$$K^\star : \star \Rightarrow \forall X.\, \forall Y.\, X \to Y \to X \qquad\qquad K^\star : \star \Rightarrow \forall X.\, \forall Y.\, X \to Y \to Y$$

The notation $e : A \Rightarrow B$, borrowed from the blame calculus [29], means cast expression e from type A to type B. The constant function $K^\star$ returns the first argument. Considering relational parametricity, a value of type $\forall X.\, \forall Y.\, X \to Y \to X$ should be a constant value which always returns the first argument. While a value of type $\forall X.\, \forall Y.\, X \to Y \to Y$ should return the second argument. Therefore, the first cast succeeds and the second cast should fail. However, if these two casts are applied to the arguments in the usual way employing type substitutions, then we obtain the following:

$$\begin{aligned}
&(K^\star : \star \Rightarrow \forall X.\, \forall Y.\, X \to Y \to X)\ \mathsf{Int}\ \mathsf{Int}\ 2\ 3 \\
\hookrightarrow^* &\ (K^\star : \star \Rightarrow \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int})\ 2\ 3 \\
\hookrightarrow^* &\ 2\ 3 \\
&(K^\star : \star \Rightarrow \forall X.\, \forall Y.\, X \to Y \to Y)\ \mathsf{Int}\ \mathsf{Int}\ 2\ 3 \\
\hookrightarrow^* &\ (K^\star : \star \Rightarrow \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int})\ 2\ 3 \\
\hookrightarrow^* &\ 2\ 3
\end{aligned}$$

The second cast succeeds and returns the first argument, which breaks parametricity. The reason for this behavior is that, after the type substitution, the polymorphic information is lost. Note that, as we have seen in Section 1, this is exactly how various practical languages (TypeScript, Flow and Typed Racket) behave.

Much of the work on gradual polymorphism aims at addressing the above problem. That is, for the second cast we would like to obtain blame instead of 2, so that parametricity is not violated. While the preservation of parametricity is a worthy goal,

it typically requires substantial changes to a calculus to ensure its preservation, since naive direct type substitutions do not work. Furthermore this also affects proofs, which can become significantly more complicated due to the changes in the calculus. To address this problem a well-known approach, originally proposed by Ahmed et al. [3], is to employ dynamic sealing. With dynamic sealing we do not do the substitution directly but record a fresh variable binding. However, even calculi that satisfy parametricity have to compromise on the important gradual guarantee property, or System F syntax, or be equiped with heavy forms of runtime evidence [20, 37]. A thorough discussion of various approaches is given in Section 6.

2.2 Key Ideas

Our key design decision is to give up support for parametricity in exchange for a simpler calculus that is also easier to extend with other important practical features. In particular, in our work we illustrate how to obtain a polymorphic gradually typed calculus, with gradual references and with the gradual guarantee. In contrast, none of the existing gradually polymorphic calculi supports references and the gradual guarantee is only supported with restrictions [20]; or major modifications in the syntax and semantics of the language [24]; or not supported/proved at all [3, 17, 37].

A direct semantics with a TDOS. Our gradually typed calculus λ^G_{gpr} has a direct semantics by using a (TDOS) [15] approach. In λ^G_{gpr}, type annotations are *operationally relevant* and they basically play a role similar to casts. Nevertheless, implicit casts should also be enforced for a gradual calculus at runtime. Most previous work makes the implicit casts explicit via the elaboration process. That is the reason why dynamic semantics is not defined directly. We resort to bidirectional typing with inferred ($\Rightarrow$) and checked ($\Leftarrow$) modes. Using the checking mode of bidirectional typing, the consistency ($\sim$) between values and the checked type is checked and enforced via an implicit cast. At compile time, the flexible consistency relation allows more programs to be accepted, while the checking mode signals casts that are needed at runtime. For example, in the typing rule for applications.

$$\frac{\Sigma; \Gamma \vdash e_1 \Rightarrow A_1 \to A_2 \qquad \Sigma; \Gamma \vdash e_2 \Leftarrow A_1}{\Sigma; \Gamma \vdash e_1\, e_2 \Rightarrow A_2}\ \textsc{Typ-app}$$

The checking mode signals an implicit cast for the argument. The argument e_2 is checked to be consistent with the type A_1 using the bidirectional subsumption rule:

$$\frac{\Sigma; \Gamma \vdash e \Rightarrow B \qquad \Gamma \vdash B \sim A}{\Sigma; \Gamma \vdash e \Leftarrow A}\ \textsc{Typ-sim}$$

For instance, $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int})\,(\mathsf{True} : \star)$ type-checks, but at run-time the invalid cast to the value argument $(\mathsf{True} : \star)$ is detected and an error is reported.

Conservativity, no parametricity and direct substitutions. The λ^G_{gpr} calculus is a conservative extension of its static counterpart. Notably, our λ^G_{gpr} is a simple polymorphic

calculus, without using mechanisms such as dynamic sealing and evidences. Instead, since parametricity is not a goal, we can simply use direct type substitutions during reduction as follows:

$$\overline{((\Lambda X.\, e : A) : \forall X.\, B)\; C \hookrightarrow e[X \mapsto C] : A[X \mapsto C] : B[X \mapsto C]}$$

Our type application rule substitutes type directly unlike in previous work with dynamic sealing where a fresh type name variable is generated and stored in a global or local context. Dynamic sealing takes extra time and space. With a large enough number of type applications, the space consumption may go unbounded.

Gradual guarantee and references. Furthermore, λ^G_{gpr} is mechanically formalized and shown to have the gradual guarantee. Our application of the eager semantics and the choice of value forms for λ^G_{gpr} simplify the gradual guarantee. To prove the gradual guarantee we need a precision ($\sqsubseteq$) relation. The gradual guarantee theorem needs to ensure that if the more static program does not go wrong, then the less static program should not go wrong as well. The precision relation is used to relate two programs, which have different type information. Type precision compares the amount of static type information for programs and types. A type is more precise than another if it is more static. The unknown type ($\star$) is the least precise type, since we do not have any static information about that type. Let's consider two programs:

$$\lambda x.\, 1 : \mathsf{Int} \to \mathsf{Int}$$
$$\lambda x.\, 1 : \star \to \star$$

The first one is more precise than the second one because the second program is fully dynamic. The value forms of λ^G_{gpr} are annotated and include terms such as $i : \mathsf{Int}$ and $(\lambda x.\, e : A \to B) : C$. The simplicity of the proof of the gradual guarantee is greatly related to the choice of representation of values. In λ^G_{gpr}, the gradual guarantee theorem can be formalized in a simple way with a lemma similar to a lemma proposed by Garcia et al. [12]. The lemma states that if e_1 is more precise than e_2 and e_1 takes a step to e'_1 then e_2 takes a step to e'_2 and e'_1 is more precise than e'_2. With this lemma, we can infer that two expressions related by precision have the same behavior. Thus, this lemma is enough to obtain the dynamic gradual guarantee. Notably, λ^G_{gpr} is extended with mutable references using a form of variant consistency [14, 38]. This is in contrast to the previously discussed gradually polymorphic calculi where references are not supported.

3 The λ_{gpr} Calculus: Syntax, Typing and Semantics

In this section, we will introduce the λ_{gpr} calculus, which is a calculus with references and polymorphism. λ_{gpr} calculus is an extended version of System F with references and is the static calculus that serves as a foundation for the gradual calculus in Section 4.

3.1 Syntax

The syntax of the λ_{gpr} calculus is shown in Figure 2.

Syntax

Types	$A, B ::= \mathsf{Int} \mid A \to B \mid X \mid \forall X.\, A \mid \mathsf{Unit} \mid \mathrm{Ref}\ A$
Expressions	$e ::= x \mid i \mid \lambda x : A.\, e \mid e : A \mid e_1\ e_2 \mid \Lambda X.\, e \mid e\ A \mid !e \mid e_1 := e_2 \mid \mathrm{ref}\ e \mid \mathsf{unit} \mid o$
Values	$v ::= i \mid \Lambda X.\, e \mid \lambda x : A.\, e \mid \mathsf{unit} \mid o$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, X$
Stores	$\mu ::= \cdot \mid \mu, o = v$
Locations	$\Sigma ::= \cdot \mid \Sigma, o : A$
Frame	$F ::= v\ \square \mid \square\ e \mid \square\ A \mid !\square \mid v_1 := \square \mid \square := e_2 \mid \mathrm{ref}\ \square \mid \square : A$

Fig. 2: λ_{gpr} syntax

Types. Meta-variables A, B range over types. Types include base types (Int), function types ($A \to B$), type variables (X), polymorphic types ($\forall X.\, A$), the unit type Unit and reference types $\mathrm{Ref}\ A$, which denotes a reference with type A.

Expressions. Meta-variables e range over expressions. Most of the expressions are standard: variables (x), integers (i), annotations ($e : A$), applications ($e_1\ e_2$), type applications ($e\ A$), dereferences ($!e$), assignments $e_1 := e_2$, references ($\mathrm{ref}\ e$), unit (unit), locations o, lambda abstractions ($\lambda x : A.\, e$) (which are annotated with input type A), and type abstractions ($\Lambda X.\, e$).

Values. Meta-variables v range over values. A raw value is either an integer (i), a type abstraction ($\Lambda X.\, e$), a lambda abstraction ($\lambda x : A.\, e$), a unit (unit) or a location (o).

Contexts, stores, locations and frames. The type context Γ tracks the bound variables x with their types and the bound type variables X. Typing location Σ tracks the bound locations o with their types, while the store μ tracks locations with their stored values during the reduction process. Frames (F) include applications, type applications, dereferences, assignments and references.

3.2 Type System

Before introducing the type system, we show the well-formedness of types at the top of Figure 3. The well-formedness of types ensures that there are no free type variables and that each type variable is bound in the contexts.

Typing relation. The typing relation of λ_{gpr} is shown at the bottom of Figure 3. The type system essentially includes the usual System F rules, except that they also propagate the location typing context (Σ). Reference locations o are stored in the location typing context Σ (rule STYP-LOC). The bound type of locations indicates the type of stored values. For instance, o points to 1 stored in a memory cell. The integer type for 1 is tracked by the location o in the location typing context Σ. Other rules related to references such as assignments (rule STYP-ASSIGN), references (rule STYP-REF) and dereferences (rule STYP-DEREF) are standard. Annotation expressions ($e : A$) are not necessary for the static

$\boxed{\Gamma \vdash A}$ *(Well-formedness of types)*

$$\text{TW-INT}\ \frac{}{\Gamma \vdash \mathsf{Int}} \qquad \text{TW-UNIT}\ \frac{}{\Gamma \vdash \mathsf{Unit}} \qquad \text{TW-VAR}\ \frac{X \in \Gamma}{\Gamma \vdash X} \qquad \text{TW-ARR}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B}$$

$$\text{TW-ALL}\ \frac{\Gamma, X \vdash A}{\Gamma \vdash \forall X.\, A} \qquad \text{TW-REF}\ \frac{\Gamma \vdash A}{\Gamma \vdash \mathsf{Ref}\ A}$$

$\boxed{\Sigma; \Gamma \vdash_s e : A}$ *(Typing rules for expressions)*

$$\text{STYP-LIT}\ \frac{}{\Sigma; \Gamma \vdash_s i : \mathsf{Int}} \qquad \text{STYP-UNIT}\ \frac{}{\Sigma; \Gamma \vdash_s \mathsf{unit} : \mathsf{Unit}} \qquad \text{STYP-VAR}\ \frac{x : A \in \Gamma}{\Sigma; \Gamma \vdash_s x : A} \qquad \text{STYP-LOC}\ \frac{o : A \in \Sigma}{\Sigma; \Gamma \vdash_s o : \mathsf{Ref}\ A}$$

$$\text{STYP-REF}\ \frac{\Sigma; \Gamma \vdash_s e : A}{\Sigma; \Gamma \vdash_s \mathsf{ref}\ e : \mathsf{Ref}\ A} \qquad \text{STYP-DEREF}\ \frac{\Sigma; \Gamma \vdash_s e : \mathsf{Ref}\ A}{\Sigma; \Gamma \vdash_s !e : A} \qquad \text{STYP-ASSIGN}\ \frac{\Sigma; \Gamma \vdash_s e_1 : \mathsf{Ref}\ A \qquad \Sigma; \Gamma \vdash_s e_2 : A}{\Sigma; \Gamma \vdash_s e_1 := e_2 : \mathsf{Unit}}$$

$$\text{STYP-ABS}\ \frac{\Sigma; \Gamma, x : A \vdash_s e : B}{\Sigma; \Gamma \vdash_s \lambda x : A.\, e : A \to B} \qquad \text{STYP-APP}\ \frac{\Sigma; \Gamma \vdash_s e_1 : A_1 \to A_2 \qquad \Sigma; \Gamma \vdash_s e_2 : A_1}{\Sigma; \Gamma \vdash_s e_1\, e_2 : A_2} \qquad \text{STYP-ANNO}\ \frac{\Sigma; \Gamma \vdash_s e : A}{\Sigma; \Gamma \vdash_s (e : A) : A}$$

$$\text{STYP-TABS}\ \frac{\Sigma; \Gamma, X \vdash_s e : A}{\Sigma; \Gamma \vdash_s \Lambda X.\, e : \forall X.\, A} \qquad \text{STYP-TAPP}\ \frac{\Gamma \vdash A \qquad \Sigma; \Gamma \vdash_s e : \forall X.\, B}{\Sigma; \Gamma \vdash_s e\, A : B[X \mapsto A]}$$

Fig. 3: The type system of λ_{gpr} calculus.

system where the annotated types are syntactically equal (rule STYP-ANNO), but they will play an important role in the gradual system and are included here.
Definition 1 defines well-formed stores (μ) with respect to the typing locations Σ, using the typing relation:

Definition 1 (Well-formedness of the store with respect to Σ).

$$\Sigma \vdash \mu \equiv if\ dom(\mu) = dom(\Sigma)\ and\ \Sigma; \cdot \vdash \mu(o) : \Sigma(o),\ for\ every\ o \in \mu$$

A store is well-formed with the typing location if the store and the typing location contain the same domains. For each location, which is in the store, the bounded value $\mu(o)$ can be inferred with the type bound in the typing location ($\Sigma(o)$).

3.3 Dynamic Semantics

The operational semantics for the λ_{gpr} calculus is shown in Figure 4 (we ignore the gray parts for now). $\mu; e \hookrightarrow \mu'; e'$ represents the reduction rules, which states that e with store μ reduces to e' with the updated store μ'. The reduction rules of λ_{gpr} are

$\boxed{\mu; e \hookrightarrow_s \mu'; e'}$ *(Operational semantics)*

STEP-EVAL
$$\frac{\mu; e \hookrightarrow_s \mu'; e'}{\mu; F[e] \hookrightarrow_s \mu'; F[e']}$$

STEP-ANNOV
$$\frac{}{\mu; v : A \ : A \hookrightarrow_s \mu; v \ : A}$$

STEP-ASSIGN
$$\frac{}{\mu; o := v \hookrightarrow_s \mu[o \mapsto v]; \mathsf{unit}}$$

STEP-TAP
$$\frac{}{\mu; ((\Lambda X.\, e) \ : \forall X.A\ )\ A \hookrightarrow_s \mu; (e[X \mapsto A]) \ : (A[X \mapsto A])}$$

STEP-DEREF
$$\frac{o = v \in \mu}{\mu; !o \hookrightarrow_s \mu; v : A}$$

STEP-BETA
$$\frac{}{\mu; ((\lambda x : A.\, e) \ : A \to B\ )\ v \hookrightarrow_s \mu; e[x \mapsto v] \ : B \ \ : B}$$

STEP-REFV
$$\frac{o \notin \mu}{\mu; \mathrm{ref}\ v \hookrightarrow_s \mu, o = v; o}$$

Fig. 4: Reduction rules for λ_{gpr}.

straightforward. A reference value is bound in the store by a fresh location as shown in rule STEP-REFV. The dereference rule extracts the bound value of the location in the store (rule STEP-DEREF). Rule STEP-EVAL evaluates the frames. Let's see how the example $o_1 := (\Lambda X.\, (\lambda x : X.\, x)\, !o_2)\ \mathsf{Int}$ with the existing store $o_1 = 1, o_2 = 2$ reduces. 2 is read from store $o_1 = 1, o_2 = 2$. After the type substitution, 2 is substituted into the lambda. Then 2 is used to update the store pointed by o_1. Finally, the store becomes $o_1 = 2, o_2 = 2$. The detailed steps are as follows:

$$\begin{aligned}
& o_1 = 1, o_2 = 2; o_1 := (\Lambda X.\, (\lambda x : X.\, x)\, !o_2)\ \mathsf{Int} \\
\hookrightarrow\ & \{\text{by rule STEP-EVAL, rule STEP-DEREF }\} \\
& o_1 = 1, o_2 = 2; o_1 := (\Lambda X.\, (\lambda x : X.\, x)\, 2)\ \mathsf{Int} \\
\hookrightarrow\ & \{\text{by rule STEP-TAP }\} \\
& o_1 = 1, o_2 = 2; o_1 := (\lambda x : \mathsf{Int}.\, x)\, 2 \\
\hookrightarrow\ & \{\text{by rule STEP-BETA}\} \\
& o_1 = 1, o_2 = 2; o_1 := 2 \\
\hookrightarrow\ & \{\text{by rule STEP-ASSIGN}\} \\
& o_1 = 2, o_2 = 2; \mathsf{unit}
\end{aligned}$$

Theorem 1 shows that the λ_{gpr} calculus is deterministic:

Theorem 1 (Determinism of λ_{gpr}). *If $\Sigma; \cdot \vdash_s e : A$, $\Sigma \vdash \mu$, $\mu; e \hookrightarrow_s \mu_1; e_1$ and $\mu; e \hookrightarrow_s \mu_2; e_2$ then $e_1 = e_2$ and $\mu_1 = \mu_2$.*

Furthermore, the preservation Theorem 2 and progress Theorem 3 of λ_{gpr} calculus are shown below:

Theorem 2 (Type Preservation of λ_{gpr}). *If $\Sigma; \cdot \vdash_s e : A$, $\Sigma \vdash \mu$ and $\mu; e \hookrightarrow_s \mu'; e'$ then $\Sigma'; \cdot \vdash_s e' : A$, $\Sigma' \vdash \mu'$ and $\Sigma' \supseteq \Sigma$.*

Theorem 3 (Progress of λ_{gpr}). *If $\Sigma; \cdot \vdash_s e : A$ then e is a value or $\exists e' \mu'$, $\mu; e \hookrightarrow_s \mu'; e'$.*

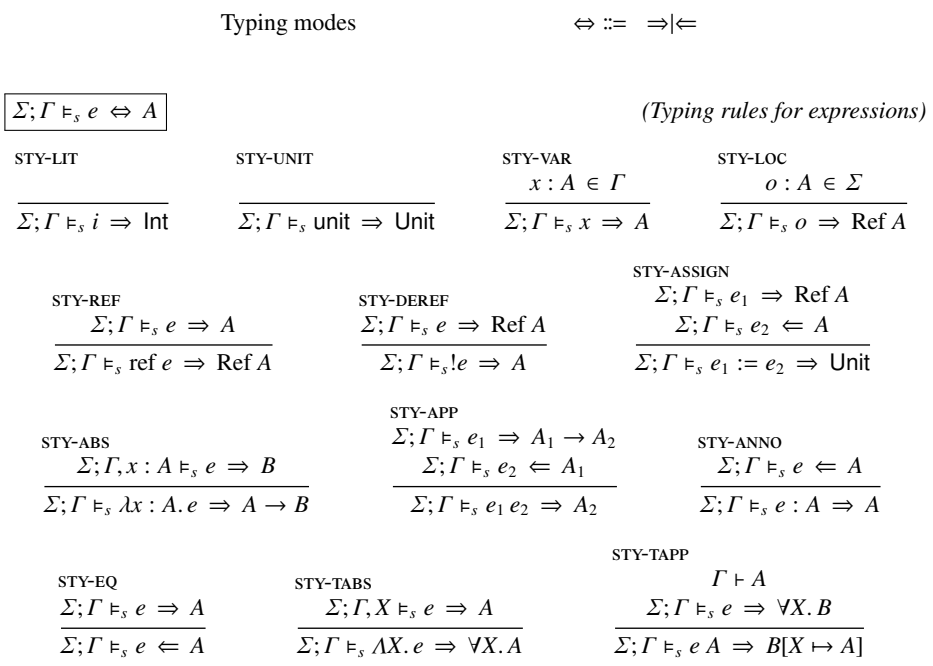

Fig. 5: Bidirectional typing for the λ_{gpr} calculus.

3.4 Bidirectional Typing

We also present a set of bidirectional typing rules (shown in Figure 5) for λ_{gpr}. Although bidirectional typing is not essential for λ_{gpr}, it is used later for the gradual typing criteria proofs. The typing judgment is represented as $\Sigma;\Gamma \vdash e \Leftrightarrow A$. The expression e is inferred ($\Rightarrow$) or checked ($\Leftarrow$) by type A under the typing context Γ and location typing context Σ. Typing modes ($\Leftrightarrow$) contain the inference mode ($\Rightarrow$) and checking mode ($\Leftarrow$), which are shown at the top of Figure 5. One extra rule is rule STY-EQ, which switches modes. We proved that the two type systems are equivalent:

Lemma 1 (Typing Equivalence for λ_{gpr}). *$\Sigma;\Gamma \vdash_s e : A$ iff $\Sigma;\Gamma \vDash_s e \Leftrightarrow A$.*

4 The λ^G_{gpr} Calculus

This section introduces the λ^G_{gpr} calculus, which gradualizes the λ_{gpr} calculus. Normally, a gradually typed lambda calculus (GTLC) does not define the operational semantics directly, but is elaborated to a cast calculus. λ^G_{gpr} instead defines the dynamic semantics directly using the TDOS approach [15]. λ^G_{gpr} is proved to be type sound and it has a gradual guarantee. The calculus does not have parametricity, enabling simplifications

Syntax

Types	$A, B ::= \mathsf{Int} \mid A \to B \mid X \mid \forall X.\, A \mid \mathsf{Unit} \mid \mathsf{Ref}\ A \mid \star$
Expressions	$e ::= x \mid i \mid e : A \mid e_1\ e_2 \mid e\ A \mid !e \mid e_1 := e_2 \mid \mathsf{ref}\ e \mid \mathsf{unit} \mid o \mid \Lambda X.\, e : A \mid \lambda x.\, e : A \to B$
Results	$r ::= e \mid \mathsf{blame}$
Raw Values	$u ::= i \mid \Lambda X.\, e : A \mid \lambda x.\, e : A \to B \mid \mathsf{unit} \mid o$
Values	$v ::= u : A$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, X$
Stores	$\mu ::= \cdot \mid \mu, o = v$
Location	$\Sigma ::= \cdot \mid \Sigma, o : A$
Frame	$F ::= v\ \square \mid \square\ e \mid \square\ A \mid !\square \mid v_1 := \square \mid \square := e_2 \mid \mathsf{ref}\ \square$

$\boxed{\Gamma \vdash A \sim B}$ *(Consistency)*

$$\text{S-UNIT}\ \frac{}{\Gamma \vdash \mathsf{Unit} \sim \mathsf{Unit}} \qquad \text{S-VAR}\ \frac{\Gamma \vdash X}{\Gamma \vdash X \sim X} \qquad \text{S-Z}\ \frac{}{\Gamma \vdash \mathsf{Int} \sim \mathsf{Int}} \qquad \text{S-DYNL}\ \frac{\Gamma \vdash A}{\Gamma \vdash \star \sim A} \qquad \text{S-DYNR}\ \frac{\Gamma \vdash A}{\Gamma \vdash A \sim \star}$$

$$\text{S-ARR}\ \frac{\Gamma \vdash A_1 \sim B_1 \quad \Gamma \vdash A_2 \sim B_2}{\Gamma \vdash A_1 \to A_2 \sim B_1 \to B_2} \qquad \text{S-FORALL}\ \frac{\Gamma, X \vdash A \sim B}{\Gamma \vdash \forall X.\, A \sim \forall X.\, B} \qquad \text{S-REF}\ \frac{\Gamma \vdash A \sim B}{\Gamma \vdash \mathsf{Ref}\ A \sim \mathsf{Ref}\ B}$$

Fig. 6: λ^G_{gpr} syntax and consistency.

in the calculus, and the addition of features such as gradual references, which none of the previous gradual calculi with polymorphism support.

4.1 Static Semantics

Syntax, type well-formedness and consistency. Figure 6 shows the syntax and consistency of the λ^G_{gpr} calculus. The gray parts are the same as λ_{gpr}. The λ^G_{gpr} calculus extends types with the unknown type $\star$ with respect to λ_{gpr}. Because of the power of the unknown type $\star$, dynamic type checking is required and run-time errors may be raised. Therefore, in addition to expressions, λ^G_{gpr} has the run-time error blame. Because of the run-time checking requirement for the gradual typing system, we need annotations for type abstractions and lambda abstractions. Furthermore, due to the imprecision of the unknown type $\star$, values are also annotated. Otherwise, examples such as $1 : \star$ are troublesome. Because of the value forms, annotations are not included in frames, unlike in the λ_{gpr} calculus. We will explain the details later.

Well-formed types are extended with the following rule for the unknown type $\star$:

$$\frac{}{\Gamma \vdash \star}$$

Notably, instead of syntactic equality, a more general relation called consistency ($\Gamma \vdash A \sim B$) is defined in λ^G_{gpr}. Every well-formed type is consistent with itself. The unknown

$\boxed{\Sigma; \Gamma \vdash e \Leftrightarrow A}$ *(Typing rules for expressions)*

$$\textsc{Typ-lit}\ \frac{}{\Sigma; \Gamma \vdash i \Rightarrow \mathsf{Int}} \qquad \textsc{Typ-unit}\ \frac{}{\Sigma; \Gamma \vdash \mathsf{unit} \Rightarrow \mathsf{Unit}} \qquad \textsc{Typ-var}\ \frac{x : A \in \Gamma}{\Sigma; \Gamma \vdash x \Rightarrow A} \qquad \textsc{Typ-loc}\ \frac{o : A \in \Sigma}{\Sigma; \Gamma \vdash o \Rightarrow \mathsf{Ref}\ A}$$

$$\textsc{Typ-ref}\ \frac{\Sigma; \Gamma \vdash e \Rightarrow A}{\Sigma; \Gamma \vdash \mathsf{ref}\ e \Rightarrow \mathsf{Ref}\ A} \qquad \textsc{Typ-deref}\ \frac{A_1 \triangleright \mathsf{Ref}\ A \quad \Sigma; \Gamma \vdash e \Rightarrow A_1}{\Sigma; \Gamma \vdash !e \Rightarrow A} \qquad \textsc{Typ-assign}\ \frac{A_1 \triangleright \mathsf{Ref}\ A \quad \Sigma; \Gamma \vdash e_1 \Rightarrow A_1 \quad \Sigma; \Gamma \vdash e_2 \Leftarrow A}{\Sigma; \Gamma \vdash e_1 := e_2 \Rightarrow \mathsf{Unit}}$$

$$\textsc{Typ-abs}\ \frac{\Sigma; \Gamma, x : A \vdash e \Leftarrow B}{\Sigma; \Gamma \vdash \lambda x.e : A \to B \Rightarrow A \to B} \qquad \textsc{Typ-app}\ \frac{A \triangleright A_1 \to A_2 \quad \Sigma; \Gamma \vdash e_1 \Rightarrow A \quad \Sigma; \Gamma \vdash e_2 \Leftarrow A_1}{\Sigma; \Gamma \vdash e_1\, e_2 \Rightarrow A_2} \qquad \textsc{Typ-anno}\ \frac{\Sigma; \Gamma \vdash e \Leftarrow A}{\Sigma; \Gamma \vdash e : A \Rightarrow A}$$

$$\textsc{Typ-sim}\ \frac{\Gamma \vdash A \sim B \quad \Sigma; \Gamma \vdash e \Rightarrow A}{\Sigma; \Gamma \vdash e \Leftarrow B} \qquad \textsc{Typ-tabs}\ \frac{\Sigma; \Gamma, X \vdash e \Leftarrow A}{\Sigma; \Gamma \vdash \Lambda X.e : A \Rightarrow \forall X.A} \qquad \textsc{Typ-tapp}\ \frac{\Gamma \vdash A \quad \Sigma; \Gamma \vdash e \Rightarrow A_1 \quad A_1 \triangleright \forall X.B}{\Sigma; \Gamma \vdash e\, A \Rightarrow B[X \mapsto A]}$$

$A \triangleright A_1 \to A_2$	$A \triangleright \forall X.A_1$	$A \triangleright \mathsf{Ref}\ A_1$
$A \to B \triangleright A \to B$	$\forall X.A \triangleright \forall X.A$	$\mathsf{Ref}\ A \triangleright \mathsf{Ref}\ A$
$\star \triangleright \star \to \star$	$\star \triangleright \forall X.\star$	$\star \triangleright \mathsf{Ref}\ \star$

Fig. 7: The type system for the λ^G_{gpr} calculus.

type is consistent with any other well-formed type. Structural types such as functions, references and polymorphic types are consistent if their type sub-components are consistent. Note that for two reference types, consistency is variant: if A and B are consistent then Ref A and Ref B are consistent. Unlike invariant consistency [31], type A and B do not have to be the same. As usual, consistency is reflexive and symmetric, but not transitive. We use the following abbreviation for consistency: $A \sim B \equiv \cdot \vdash A \sim B$.

Typing relation. Bidirectional typing is used to design the type system. Bidirectional typing is not essential for λ_{gpr} but it is necessary for λ^G_{gpr}. Annotation expressions $(e : A)$ and the checking mode $(\Leftarrow)$ signal the use of casts (explicitly or implicitly) at run-time.

The typing rules of the λ^G_{gpr} calculus are shown in Figure 7. They are almost the same as λ_{gpr}'s type system. For rule Typ-app, rule Typ-tapp, rule Typ-assign and rule Typ-deref, the unknown type $\star$ can be matched with, respectively, a dynamic function type $(\star \to \star)$, a dynamic polymorphic type $(\forall X.\star)$ and a dynamic reference type (Ref $\star$). In a system with gradual typing and the unknown type $\star$ we always have to consider

cases where the type may be unknown. For instance in an application $e_1\ e_2$, e_1 can infer a function type as usual, but it can also infer type $\star$ and still be well-typed. So, a matching function ($A \triangleright B$) is needed to account for both possibilities. The table at the bottom of Figure 7 shows the definition of the matching functions $A \triangleright B$. Note that we overload the notation, but there are 3 different matching functions, in each column of the table, that are employed by the rules correspondingly. For example, rule Typ-deref employs the matching function in the third column of the table. The first row in the table depicts the form of the matching function, while the other two rows give its definition.

The checking mode rule Typ-sim is generalized to check if the inferred type A and checked type B are consistent. Note that rule Typ-sim is the only rule in the checked mode and, as such, does not overlap with anything else. Moreover, all the rules in the inference mode are syntax directed. Therefore, the rules are basically directly implementable, as usual for bidirectional type-checking rules. Note that in λ^G_{gpr} annotation expressions combined with consistency play an important role, where more programs are allowed. For instance, $(\lambda x.((x : \star)\ 1) : \mathsf{Bool} \to \star)\ \mathsf{True}$ is accepted, but raises a blame error at run-time. Note that dynamically typed lambdas $\lambda x.e$ are syntactic sugar for $\lambda x.e : \star \to \star$. The use of this syntactic sugar enables us to encode the dynamically typed lambda calculus (DTLC) [4] easily in λ^G_{gpr}.

Definition 2 shows dynamic type checking for raw and annotated values, which is done at run-time. Dynamic type checking for values exploits the annotations that are present at run-time, and does not make use of the typing relation. Dynamic type checking is essentially a constant time operation, with little cost (note that the function is not recursive).

Definition 2 (Dynamic type). *$|u|_\mu = A$ and $|v|_\mu = A$ denote the dynamic type of the raw and annotated values.*

$$
\begin{aligned}
|i|_\mu &= \mathsf{Int} \\
|(\lambda x.\, e : A \to B)|_\mu &= A \to B \\
|(\Lambda X.\, e : A)|_\mu &= \forall X.\, A \\
|\mathsf{unit}|_\mu &= \mathsf{Unit} \\
|o|_\mu &= \mathit{Ref}\ |v|_\mu \quad \mathit{when}\ o = v \in \mu \\
|(u : A)|_\mu &= A
\end{aligned}
$$

$|u|_\mu = A$ states that the dynamic type of the raw value u is A under store μ. Notably, for locations o, the dynamic type is defined by the dynamic type of the bounded values in the store. Other rules are straightforward. Lemma 2 shows that if a raw value can be inferred with type A, then its dynamic type is type A as well.

Lemma 2 (Synthesis of Dynamic Types). *For any raw value u, if $\Sigma \vdash \mu$ and $\Sigma; \cdot \vdash u \Rightarrow A$ then $|u|_\mu = A$.*

As in λ_{gpr}, a term typed using the inference mode is guaranteed to infer a unique type. In addition, Lemma 3 shows that each well-typed term can be checked.

Lemma 3 (Synthesis principality). *If $\Sigma; \Gamma \vdash e \Rightarrow A$ then exists B, $\Sigma; \Gamma \vdash e \Leftarrow B$ and $\Gamma \vdash A \sim B$.*

$\boxed{\mu; v \hookrightarrow_A \mu'; r}$ *(Casting for values)*

$$\textsc{casting-sim}\ \frac{|u|_\mu \sim B}{\mu; u : A \hookrightarrow_B \mu; u : B} \qquad \textsc{casting-nsim}\ \frac{\neg|u|_\mu \sim B}{\mu; u : A \hookrightarrow_B \mu; \mathsf{blame}}$$

$\boxed{\mu; v \hookrightarrow_{B,A} \mu'; r}$ *(Double casting)*

$$\textsc{TLists-baseb}\ \frac{\mu; v \hookrightarrow_A \mu; \mathsf{blame}}{\mu; v \hookrightarrow_{B,A} \mu; \mathsf{blame}} \qquad \textsc{TLists-cons}\ \frac{\mu; v \hookrightarrow_A \mu; v' \quad \mu; v' \hookrightarrow_B \mu; r}{\mu; v \hookrightarrow_{B,A} \mu; r}$$

Fig. 8: Casting for values

4.2 Dynamic Semantics

The dynamic semantics contains two parts. The first part is casting, which casts a value to another value with a target type. In casting the dynamic type of the value is the source type. The second part is the reduction rules.

Casting. Figure 8 shows the casting rules of the λ^G_{gpr} calculus. $\mu; v \hookrightarrow_A \mu; r$ represents casting values v by type A under store μ. The dynamic type of the raw values u is checked to be consistent with type A or not. If two types are consistent, then the intermediate type can be removed and the raw values are annotated with target types. Otherwise, a run-time error is raised. For example when $1 : \star$ is cast by type Bool, the dynamic type of 1 is Int, which is not consistent with Bool, and blame is raised. While in $1 : \star$ cast by type Int, the type Int is consistent with type Int. Thus, type $\star$ is erased and 1 is annotated with type Int. Since a location o is a raw value, if we want to obtain the dynamic type of the location, we should obtain it from the store μ. Therefore, casting uses the store. Casting by two types is shown at the bottom of Figure 8. It simply casts the types one by one, using the basic casting relation.

Reduction. The reduction rules of λ^G_{gpr} calculus are shown in Figure 9. Raw values are reduced to become values, which are annotated by the dynamic type of the raw values with rule STEP-U. Due to this rule, annotations are not included in the frame. Annotated expressions are further dealt by rule STEP-ANNO and rule STEP-ANNOP. From the typing rules of rules TYP-APP, TYP-TAPP, TYP-ASSIGN, and TYP-DEREF, type $\star$ is allowed to match, respectively, a dynamic function, a polymorphic function or a reference type. Moreover, we know that $\star$ is consistent with any type. Therefore, we should check whether the internal values cannot match with the wanted type structure. For example, ill-formed applications $((1 : \star)\, 2)$ where the internal value (1) is not an lambda abstraction. There are similar examples for type applications and assignments: $(1 : \star)\ \mathsf{Bool}$ and $(\mathsf{True} : \star) := 2$ where 1 is not a type abstraction and True is not a location. Using

$\boxed{\mu; e \hookrightarrow \mu'; r}$ *(Operational semantics)*

VSTEP-EVAL
$$\frac{\mu; e \hookrightarrow \mu'; e'}{\mu; F[e] \hookrightarrow \mu'; F[e']}$$

VSTEP-BLAME
$$\frac{\mu; e \hookrightarrow \mu'; \mathsf{blame}}{\mu; F[e] \hookrightarrow \mu'; \mathsf{blame}}$$

VSTEP-ANNOP
$$\frac{\mu; e \hookrightarrow \mu'; \mathsf{blame}}{\mu; e : A \hookrightarrow \mu'; \mathsf{blame}}$$

VSTEP-BETA
$$\frac{A \rhd A_2 \to B_2 \qquad \mu; v \hookrightarrow_{A_1, A_2} \mu; v'}{\mu; ((\lambda x.\, e : A_1 \to B_1) : A)\, v \hookrightarrow \mu; e[x \mapsto v'] : B_1 : B_2}$$

VSTEP-ASSIGND
$$\frac{\mu; v_1 \hookrightarrow_{\mathrm{Ref}\ \star} \mu; \mathsf{blame}}{\mu; v_1 := v_2 \hookrightarrow \mu; \mathsf{blame}}$$

VSTEP-ANNOV
$$\frac{\mu; v \hookrightarrow_A \mu; r}{\mu; v : A \hookrightarrow \mu; r}$$

VSTEP-BETAP
$$\frac{A \rhd A_2 \to B_2 \qquad \mu; v \hookrightarrow_{A_1, A_2} \mu; \mathsf{blame}}{\mu; ((\lambda x.\, e : A_1 \to B_1) : A)\, v \hookrightarrow \mu; \mathsf{blame}}$$

VSTEP-BETAD
$$\frac{\mu; v_1 \hookrightarrow_{\star \to \star} \mu; \mathsf{blame}}{\mu; v_1\, v_2 \hookrightarrow \mu; \mathsf{blame}}$$

VSTEP-TAP
$$\frac{B \rhd \forall X.\, B_2}{\mu; ((\Lambda X.\, e : A) : B)\, C \hookrightarrow \mu; e[X \mapsto C] : A[X \mapsto C] : B_2[X \mapsto C]}$$

VSTEP-TAPD
$$\frac{\mu; v \hookrightarrow_{\forall X.\, \star} \mu; \mathsf{blame}}{\mu; v\, B \hookrightarrow \mu; \mathsf{blame}}$$

VSTEP-REFV
$$\frac{o \notin \mu}{\mu; \mathrm{ref}\ v \hookrightarrow \mu, o = v; o}$$

VSTEP-DEREF
$$\frac{o = v \in \mu \qquad A_1 \rhd \mathrm{Ref}\ A}{\mu; !(o : A_1) \hookrightarrow \mu; v : A}$$

VSTEP-DEREFP
$$\frac{\mu; v \hookrightarrow_{\mathrm{Ref}\ \star} \mu; \mathsf{blame}}{\mu; !v \hookrightarrow \mu; \mathsf{blame}}$$

VSTEP-ASSIGN
$$\frac{|o|_\mu = A_1 \qquad A \rhd \mathrm{Ref}\ A_2 \qquad \mu; v_2 \hookrightarrow_{A_1, A_2} \mu; v_2'}{\mu; (o : A) := v_2 \hookrightarrow \mu[o \mapsto v_2']; \mathsf{unit}}$$

VSTEP-ASSIGNP
$$\frac{|o|_\mu = A_1 \qquad A \rhd \mathrm{Ref}\ A_2 \qquad \mu; v_2 \hookrightarrow_{A_1, A_2} \mu; \mathsf{blame}}{\mu; (o : A) := v_2 \hookrightarrow \mu; \mathsf{blame}}$$

VSTEP-U
$$\frac{|u|_\mu = A}{\mu; u \hookrightarrow \mu; u : A}$$

VSTEP-ANNO
$$\frac{\neg \mathit{value}\ e : A \qquad \mu; e \hookrightarrow \mu'; e'}{\mu; e : A \hookrightarrow \mu'; e' : A}$$

Fig. 9: Reduction rules for λ^G_{gpr}.

rules VSTEP-BETAD, VSTEP-TAPD, VSTEP-DEREFP, and VSTEP-ASSIGND, we cast the value to the corresponding dynamic types and filter out programs with errors. To apply a value to a functional value (rules VSTEP-BETA and VSTEP-BETAP), the argument type must be consistent with function input types A_2. Moreover, the expected substituted value type is A_1. Thus, the argument value should be cast by A_2 and A_1, which may return a blame error. To preserve the type, the substituted body is annotated with B_1 and B_2. When a value v is annotated with a type A, the type of the value must be consistent with type A, and run-time checking is needed to validate consistency (rule VSTEP-ANNOV). A reference value ref v is bound in the store with a fresh location o (rule VSTEP-REFV). To obtain a value from the store by the location, from the last expression we use rule VSTEP-DEREF.

Note that in the typing rule for references:

$$\frac{\Sigma; \cdot \vdash o : A_1 \Rightarrow A_1 \qquad A_1 \rhd \mathsf{Ref}\, A}{\Sigma; \cdot \vdash !(o : A_1) \Rightarrow A}\ \textsc{Typ-deref}$$

The expected type is A but the bound value type is consistent with A. Thus we annotate v using type A. When assigning a value to replace the bound value in the reference using rules VSTEP-ASSIGN and VSTEP-ASSIGNP :

$$\frac{A \rhd \mathsf{Ref}\, A_2 \qquad \Sigma; \cdot \vdash o : A \Rightarrow A \qquad \Sigma; \cdot \vdash v_2 \Leftarrow A}{\Sigma; \cdot \vdash (o : A) := v_2 \Rightarrow \mathsf{Unit}}\ \textsc{Typ-assign}$$

The bound value by location o has type A_1, while the type of v_2 is consistent with type A_2 and A_2 is consistent with A_1. The expected type to be replaced is type A_1, therefore v_2 is cast by type A_1 and A_2. Note that the cast result can be blamed. If a type is applied to a polymorphic value, from the last expression (rule VSTEP-TAP):

$$\frac{B \rhd \forall X.\, B_2 \qquad \Sigma; \cdot \vdash (\Lambda X.\, e : A) : B \Rightarrow B}{\Sigma; \cdot \vdash ((\Lambda X.\, e : A) : B)\, C \Rightarrow B_2[X \mapsto C]}\ \textsc{Typ-tapp}$$

The expected type is $(B_2[X \mapsto C])$ but the substituted expression $(e[X \mapsto C] : A[X \mapsto C])$ has type $(A[X \mapsto C])$, so it is annotated with type $(B_2[X \mapsto C])$.

Properties of λ^G_{gpr}. λ^G_{gpr} is deterministic (Theorem 4) and type sound (Theorem 5 and Theorem 6).

Theorem 4 (Determinism of λ^G_{gpr}). *If $\Sigma; \cdot \vdash e \Leftrightarrow A$, $\mu; e \hookrightarrow \mu_1; r_1$ and $\mu; e \hookrightarrow \mu_2; r_2$ then $r_1 = r_2$ and $\mu_1 = \mu_2$.*

Theorem 5 (Type Preservation of λ^G_{gpr}). *If $\Sigma; \cdot \vdash e \Leftrightarrow A$, $\Sigma \vdash \mu$, and $\mu; e \hookrightarrow \mu'; e'$ then $\Sigma'; \cdot \vdash e' \Leftrightarrow A$, $\Sigma' \vdash \mu'$ and $\Sigma' \supseteq \Sigma$.*

Theorem 6 (Progress of λ^G_{gpr}). *If $\Sigma; \cdot \vdash e \Leftrightarrow A$ then e is a value or $\exists r\, \mu', \mu; e \hookrightarrow \mu'; r$.*

4.3 Gradual Typing Criteria

Siek *et al.* [31,32] proposed a set of criteria for gradual typing system. At the end of the spectrum, a fully annotated gradually typed program should behave as a statically typed program. Conversely, a gradually typed program without annotations should behave as a dynamic program. Siek *et al.* proposed the gradual guarantee, which states that having annotations that are more/less precise should not change the behavior of the programs. Here we show that λ^G_{gpr} has the gradual guarantee.

To prove the gradual guarantee, we define the precision for types, expressions and stores. At the top of Figure 10 is type precision $A \sqsubseteq B$, which states that type A is more precise than B. The unknown type $\star$ is less precise than any other types. Each type is more precise than itself. The precision of functions, polymorphic functions and

$\boxed{A \sqsubseteq B}$ *(Type Precision)*

$$\textsc{TP-UNIT}\ \frac{}{\mathsf{Unit} \sqsubseteq \mathsf{Unit}} \qquad \textsc{TP-VAR}\ \frac{}{X \sqsubseteq X} \qquad \textsc{TP-Z}\ \frac{}{\mathsf{Int} \sqsubseteq \mathsf{Int}} \qquad \textsc{TP-DYN}\ \frac{}{A \sqsubseteq \star} \qquad \textsc{TP-ARR}\ \frac{A_1 \sqsubseteq B_1 \qquad A_2 \sqsubseteq B_2}{A_1 \to A_2 \sqsubseteq B_1 \to B_2}$$

$$\textsc{TP-FORALL}\ \frac{A \sqsubseteq B}{\forall X.\, A \sqsubseteq \forall X.\, B} \qquad \textsc{TP-REF}\ \frac{A \sqsubseteq B}{\mathrm{Ref}\ A \sqsubseteq \mathrm{Ref}\ B}$$

$\boxed{e_1 \sqsubseteq e_2}$ *(Expression Precision)*

$$\textsc{EP-LIT}\ \frac{}{i \sqsubseteq i} \qquad \textsc{EP-VAR}\ \frac{}{x \sqsubseteq x} \qquad \textsc{EP-UNIT}\ \frac{}{\mathsf{unit} \sqsubseteq \mathsf{unit}} \qquad \textsc{EP-O}\ \frac{}{o \sqsubseteq o} \qquad \textsc{EP-REF}\ \frac{e_1 \sqsubseteq e_2}{\mathrm{ref}\ e_1 \sqsubseteq \mathrm{ref}\ e_2} \qquad \textsc{EP-DEREF}\ \frac{e_1 \sqsubseteq e_2}{!e_1 \sqsubseteq !e_2}$$

$$\textsc{EP-ABS}\ \frac{e_1 \sqsubseteq e_2 \qquad A_1 \sqsubseteq A_2 \qquad B_1 \sqsubseteq B_2}{\lambda x.\, e_1 : A_1 \to B_1 \sqsubseteq \lambda x.\, e_2 : A_2 \to B_2} \qquad \textsc{EP-APP}\ \frac{e_1 \sqsubseteq e_3 \qquad e_2 \sqsubseteq e_4}{e_1\ e_2 \sqsubseteq e_3\ e_4} \qquad \textsc{EP-ASSIGN}\ \frac{e_1 \sqsubseteq e_3 \qquad e_2 \sqsubseteq e_4}{e_1 := e_2 \sqsubseteq e_3 := e_4}$$

$$\textsc{EP-ANNO}\ \frac{e_1 \sqsubseteq e_2 \qquad A_1 \sqsubseteq A_2}{e_1 : A_1 \sqsubseteq e_2 : A_2} \qquad \textsc{EP-TABS}\ \frac{e_1 \sqsubseteq e_2 \qquad A_1 \sqsubseteq A_2}{\Lambda X.\, e_1 : A_1 \sqsubseteq \Lambda X.\, e_2 : A_2} \qquad \textsc{EP-TAPP}\ \frac{e_1 \sqsubseteq e_2 \qquad A_1 \sqsubseteq A_2}{e_1\ A_1 \sqsubseteq e_2\ A_2}$$

$\boxed{\mu_1 \sqsubseteq \mu_2}$ *(Store Precision)*

$$\textsc{SP-NIL}\ \frac{}{\cdot \sqsubseteq \cdot} \qquad \textsc{SP-EMPTY}\ \frac{\mu_1 \sqsubseteq \mu_2 \qquad v_1 \sqsubseteq v_2}{\mu_1, o = v_1 \sqsubseteq \mu_2, o = v_2}$$

Fig. 10: Precision Relation.

reference types holds, if the precision of their sub-components holds. Note that the precision of function types is "covariant" in the argument types since to compare the precision of the two programs:

$$\lambda x.\, 1 : \mathsf{Int} \to \mathsf{Int}$$
$$\lambda x.\, 1 : \star \to \mathsf{Int}$$

we should just say that the first one is more precise than the second one because the input type of the second one is fully dynamic. Expression precision is shown in the middle of Figure 10. The rules can mostly be derived from the type precision. Each expression is in a precision relation with itself. Structural expressions are in a precision relation if their sub-expressions are related. Lastly, store precision, shown at the bottom of Figure 10, shows that precision holds if the precision of values in the store holds.

$\boxed{\mu; e \hookrightarrow_{s*} \mu'; e'}$ *(Operational semantics)*

$$\textsc{Step-eval}\ \frac{\mu; e \hookrightarrow_{s*} \mu'; e'}{\mu; F[e] \hookrightarrow_{s*} \mu'; F[e']} \qquad \textsc{Step-annov}\ \frac{}{\mu; u : A : A \hookrightarrow_{s*} \mu; u : A} \qquad \textsc{Step-assign}\ \frac{}{\mu; o := v \hookrightarrow_{s*} \mu[o \mapsto v]; \mathsf{unit}}$$

$$\textsc{Step-tap}\ \frac{}{\mu; ((\Lambda X.\, e : A) : \forall X.\, A)\ A \hookrightarrow_{s*} \mu; e[X \mapsto A] : A[X \mapsto A]} \qquad \textsc{Step-deref}\ \frac{o = v \in \mu}{\mu; !o \hookrightarrow_{s*} \mu; v : A}$$

$$\textsc{Step-beta}\ \frac{}{\mu; ((\lambda x.\, e : A \to B) : A \to B)\ v \hookrightarrow_{s*} \mu; e[x \mapsto v] : B : B} \qquad \textsc{Step-refv}\ \frac{o \notin \mu}{\mu; \mathrm{ref}\ v \hookrightarrow_{s*} \mu, o = v; o}$$

$$\textsc{Step-u}\ \frac{|u|_\mu = A}{\mu; u \hookrightarrow_{s*} \mu; u : A} \qquad \textsc{Step-anno}\ \frac{\neg \mathit{value}\ e : A \quad \mu; e \hookrightarrow_{s*} \mu'; e'}{\mu; e : A \hookrightarrow_{s*} \mu'; e' : A}$$

Fig. 11: Reduction rules for λ_{gpr}.

Static criteria. We show that the full static type system of λ^G_{gpr} is equivalent to the λ_{gpr} calculus (Theorem 7). We use s to denote a relation from the static system in case of ambiguity. Theorem 8 shows the static gradual guarantee of λ^G_{gpr}. If a more precise program is well-typed then a less precise program should be well-typed with a less precise type.

Theorem 7 (Equivalence for λ_{gpr} (statics)). *If* $\cdot; \cdot \vDash_s e \Leftrightarrow A$ *if and only if* $\cdot; \cdot \vdash e \Leftrightarrow A$.

Theorem 8 (Static Gradual Guarantee). *If* $e_1 \sqsubseteq e_2$, $\cdot; \cdot \vdash e_1 \Leftrightarrow A$ *then* $\cdot; \cdot \vdash e_2 \Leftrightarrow B$ *and* $A \sqsubseteq B$.

Dynamic criteria. Theorem 9 says that fully static programs of λ^G_{gpr} calculus behaves in the same as the λ_{gpr} at run-time. To make the proofs easier, the reduction rules of λ_{gpr} calculus have extra annotations to follow λ^G_{gpr} (we denoted as $s*$). It means that there are extra identical annotations, as shown in the gray parts of Figure 4. However, these annotations are identical and they can be removed without affecting the final reduction result. In addition, as in λ^G_{gpr}: values have annotations; raw values should step to be annotated values; and annotations are not included in Frames. This requires a few extra rules, which are shown in Figure 11.

Notably, λ^G_{gpr} has the dynamic gradual guarantee (Theorem 10). The proof is simple in comparison to the original proof by Siek et al. [32]. This simple theorem is formalized following the work of Garcia *et al.* [12]. It says that if a more precise program with a more precise store can reduce, then the less precise program with a less precise store can also reduce. Furthermore, their resulting programs and stores should keep the precision relation.

Theorem 9 (Equivalence for λ_{gpr} (dynamic)). $\forall\ \cdot;\cdot \vDash_s e\ \Leftrightarrow\ A$,

- *If* $\mu; e \hookrightarrow_{s*} \mu'; e'$ *then* $\mu; e \hookrightarrow \mu'; e'$.
- *If* $\mu; e \hookrightarrow \mu'; e'$ *then* $\mu; e \hookrightarrow_{s*} \mu'; e'$.

Theorem 10 (Dynamic Gradual Guarantee). *If* $e_1 \sqsubseteq e_2$, $\mu_1 \sqsubseteq \mu_2$, $\cdot;\cdot \vdash e_1\ \Leftrightarrow\ A$, $\cdot;\cdot \vdash e_2\ \Leftrightarrow\ B$ *and* $\mu_1; e_1 \hookrightarrow \mu_1'; e_1'$ *then there exists* e_2' *and* μ_2' *such that* $\mu_2; e_2 \hookrightarrow \mu_2'; e_2'$, $e_1' \sqsubseteq e_2'$ *and* $\mu_1' \sqsubseteq \mu_2'$.

5 Discussion

In this section, we briefly discuss alternative designs and possible extensions.

Preserving relational parametricity. An alternative design is to have a directed semantics gradual polymorphism calculi, which preserves parametricity. We employ the eager semantics similar to the AGT methodology, which is applied in the GSF calculus. Toro *et al.* [37] analyzed the following example to show how parametricity is broken by the naive use of the dynamic sealing in the eager semantics:

$$(\Lambda X.(\lambda x : X.\mathsf{let}\ y : \star = x\ \mathsf{in}\ \mathsf{let}\ z : \star = y\ \mathsf{in}\ z + 1))\ \mathsf{Int}\ 1$$

The polymorphic function with type $(\forall X.\, X \to \star)$ breaks parametricity, which should be detected at run-time and raise an error. However, the application of the function reduces to 2. A fresh name variable α is generated and is bounded to the type Int. Variable x to y is flowing from type Int to type α; y to z is flowing from type $\star$ to type $\star$; and x to z is flowing from Int to $\star$. Any of these type flows are safe. Thus the reason for the loss of parametricity is related to the loss of precise type information. Consequently, dynamic sealing is not enough to enforce relational parametricity. For the above example, GSF detects the error by the refining evidences such as $(\langle \alpha^{E_1}, \alpha^{E_2} \rangle)$. Importantly in the type flow from y to z, more precise types (Int and α^{Int}) instead of $\star$ and $\star$ are obtained, so when moving from x to z the type changes from Int to α^{Int}. When doing the addition, the run-time error is detected since the flow from α^{Int} to Int is not defined. A potential approach for us is to use tracked types ($A^{<B_1,B_2>}$), which are similar to the refined evidences in the GSF calculus. Because λ^{G}_{gpr} is a source language, we do not have evidences, thus a possible approach is to record information in types. For the above example, tracked types can track the unknown type with more precise types from y to z to be Int and α^{Int} which is $\star^{(\mathsf{Int},\alpha^{Int})}$ and then from x to z to be $\star^{(\mathsf{Int},\alpha^{Int})}$ as the refined evidences and a run-time error is detected when doing the addition.

A space-efficient gradual polymorphic calculus. Ozaki *et al.* [27] explored the space efficiency problem in the gradual polymorphic calculus. They extended the coercion calculus (λC) [29] with parametric polymorphism (called $\lambda C^{\forall}$). Dynamic sealing was applied in $\lambda C^{\forall}$ to enforce relational parametricity. Consequently, a sequence of coercions is allowed and they showed that it cannot be normalized to a smaller coercion. In other words, the size of sequences is unbounded. Notably, they stated and proved that $\lambda C^{\forall}$ cannot be space-efficient when dynamic sealing is supported. Furthermore,

they conjectured that the gradual polymorphic calculus with dynamic sealing cannot become space-efficient. Our λ^G_{gpr} calculus substitutes types directly, as the traditional semantics without employing dynamic sealing. Moreover, the eager semantics is applied. Thus we believe that it is possible for our λ^G_{gpr} calculus to be a space-efficient gradual polymorphic calculus. Two tentative and promising rules are as follows:

$$\frac{A \sim C}{e : A : B : C \hookrightarrow e : A : C} \qquad \frac{\neg A \sim C}{e : A : B : C \hookrightarrow \mathsf{blame}}$$

With the above two rules, annotations are removed or an error is raised, to achieve the space-efficient goal. Surprisingly, with these two rules, it seems possible to have a space-efficient gradual references calculus naturally. We intend to explore this in the future.

Implicit polymorphic references. Implicit (higher-rank) polymorphism [10, 19, 26] is pervasive in theoretic and practical programming languages. Existing gradual polymorphic calculi are mainly explicitly polymorphic. One exception is the work of Xie *et al.* [41]. Explicit polymorphism means that polymorphic types are not related to any of its instantiated types but in implicit polymorphism, they are related. Xie *et al.* [41] designed a source gradual implicit polymorphism calculus with consistent subtyping but their dynamic semantics is defined by translating to the well-known polymorphic blame calculus ($\lambda B^{\forall}$) [3] without the proof of the dynamic gradual guarantee. A possible extension of Xie et al.'s work is to support implicit polymorphism with a direct dynamic semantics, and to explore the dynamic gradual guarantee and parametricity properties. However, it is well-known that a naive combination of implicit polymorphism and references lead to an unsound language. A possible solution is to limit polymorphism to syntactic let-bound values as adopted by Standard ML [40].

Alternative forms of values. In our calculus, all values are annotated, such as $1 : \mathsf{Int}$ or $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) : \mathsf{Int} \to \mathsf{Int}$. This introduces some overhead as some annotations are redundant. We can have an alternative and workable form of values as follows:

$$v ::= u \mid u : \star \mid (\Lambda X.\, e : A) : \forall X.\, B \mid (\lambda x.\, e : A_1 \to B_1) : A_2 \to B_2$$

The above value form removes redundant annotations such as integers ($1 : \mathsf{Int}$). This is good for performance, but it would make the proof of dynamic gradual guarantee harder. However, the resulting calculus with fewer annotations should have an equivalent semantics to our calculus, and would be a better candidate for guiding an implementation.

6 Related Work

Gradual typing. Gradual typing is a term coined by Siek *et al.* [31]. The unknown type ?, which we represent as $\star$, is the new notion introduced to a gradual type system to integrate dynamic and static typing. By using the unknown type $\star$, equality on types is lifted to consistency. Any type is consistent with type $\star$. Therefore, run-time type

checking is needed for a gradually typed lambda calculus. Traditionally, the dynamic semantics of a gradual language is defined by elaborating to a target language, which includes cast calculi [3, 11, 29, 34, 39] and coercion calculi [13, 14, 27, 29, 30].

Garcia *et al.* [12] proposed the abstracting gradual typing (AGT) approach, which allows for deriving a gradual type system by lifting the static type system. They argue about the weakness of elaborating to a target language, and did not resort to a target language in their calculus by using intrinsic terms. Our λ^{G}_{gpr} defines the dynamic semantics directly without using intrinsic terms, but employing instead an approach based on type-directed operational semantics (TDOS). Type directed operational semantics (TDOS) was proposed by Huang *et al.* [15] to design calculi with the merge operator and intersection types. Ye *et al.* [42] explored the use of the TDOS in gradual typing. In TDOS, type annotations are relevant at runtime and can affect the semantics, unlike many traditional calculi where types are not runtime relevant. With a TDOS we can design a gradually typed calculus without elaboration to a cast calculus, since the semantics can be given directly. Our λ^{G}_{gpr} employs the eager semantics for higher-order values following an approach similar to AGT. Ye *et al.* only consider a TDOS for a simply typed, purely functional language. Our work shows that the TDOS approach can be extended to important features, such as polymorphism and references.

Gradual typing with references. Many languages with static and dynamic typing, employing some form of optional typing, support references. These include Flow [8], Dart [6] and TypeScript [5]. However for optional typing, the run-time checking is not performed for fully dynamic programs, leading to unsoundness with respect to the static type system. In the work of Siek *et al.* [31], he already considered mutable references, but in a very simple setting without annotation expressions. Furthermore, the gradually typed lambda calculus is elaborated to a target language to define the dynamic semantics. Herman *et al.* [14] designed a coercion calculus with references, which is space efficient. A gradualizer, introduced by Cimini and Siek [9], can derive a gradual static type system and cast insertion with references systematically. Toro *et al.* [38] designed source gradual typing system with references $\lambda_{\widetilde{REF}}$ and a corresponding target language $\lambda^{\epsilon}_{\widetilde{REF}}$ using the Abstracting Gradual Typing (AGT) methodology. They designed the $\lambda^{\epsilon}_{\widetilde{REF}}$ as a space-efficient calculus and proved the gradual guarantee. Our λ^{G}_{gpr} is the first polymorphic gradually typed language with references.

Existing gradual polymorphic calculi. In the following we summarize some of the solutions to the problem of preserving parametricity and gradual guarantee in gradual polymorphic calculi and the changes that these solutions entail.

Dynamic sealing. Ahmed et al. [3] solved the problem in Section 2 by using dynamic sealing, inspired by the work of Matthews *et al.* [21]. They proposed the polymorphic blame calculus [3] (we present it as $\lambda B^{\forall}$), which is a widely used cast calculus with dynamic sealing. The most interesting construct of $\lambda B^{\forall}$ is the named type binding $\nu X := A.t$, which is introduced to record the instantiated type of a type variable. The programs

in Section 2 behave as expected in $\lambda B^{\forall}$:

$$\begin{aligned}
&(K^{\star} : \star \Rightarrow \forall X.\, \forall Y.\, X \to Y \to X)\ \mathsf{Int}\ \mathsf{Int}\ 2\ 3\\
\hookrightarrow^{*}\ &\nu Y := \mathsf{Int}.\nu X := \mathsf{Int}.(2 : X \Rightarrow \star : \star \Rightarrow X)\\
\hookrightarrow^{*}\ &2\\
&(K^{\star} : \star \Rightarrow \forall X.\, \forall Y.\, X \to Y \to Y)\ \mathsf{Int}\ \mathsf{Int}\ 2\ 3\\
\hookrightarrow^{*}\ &\nu Y := \mathsf{Int}.\nu X := \mathsf{Int}.(2 : X \Rightarrow \star : \star \Rightarrow Y)\\
\hookrightarrow^{*}\ &\mathit{blame}
\end{aligned}$$

The first program succeeds and returns the first argument. While the second program fails, since the polymorphic information is recorded as $X := \mathsf{Int}$ and $Y := \mathsf{Int}$ in type bindings and the original type variable names are preserved in the casts. Notably, for higher-order values, $\lambda B^{\forall}$ follows the lazy semantics as the blame calculus [29,39]. That is, for a function value, the checking is delayed until an argument value is applied. This, unfortunately results in unbounded space consumption for higher-order casts [13, 14].

As Xie *et al.* [41] pointed out, the compatibility relation of $\lambda B^{\forall}$ mixes explicit and implicit polymorphism to some extent, since they employ the following rule:

$$\frac{A[X \mapsto \star] \prec B}{\forall X.\, A \prec B}$$

This compatibility rule of $\lambda B^{\forall}$ allows $\forall X.\, X \to X$ to be compatible with any static instantiated types such as $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Bool} \to \mathsf{Bool}$. These types are not related in System F so $\lambda B^{\forall}$ is not a conservative extension of System F. The gradual guarantee has not been discussed in $\lambda B^{\forall}$, but they show the parametricity property.

The F_G and F_C calculi. Igarashi *et al.* [17] improved on $\lambda B^{\forall}$. They designed a source calculus (F_G) and a target calculus (F_C), which is a conservative extension of System F. The dynamic semantics of F_G is indirect and defined by translation to F_C. F_G does not relate $\forall X.\, X \to X$ with static instantiations, but only with the dynamic instantiation $\star \to \star$. The type $\star \to \star$ is called quasi-polymorphic, since it is an instantiation of $\forall X.\, X \to X$ similarly to what happens with implicit polymorphism. However, a type such as $\mathsf{Int} \to \mathsf{Int}$ is not quasi-polymorphic. Instead of binding types locally by ($\nu X := A.t$), they made the type bindings global. Their reduction form $\Sigma \triangleright f \hookrightarrow \Sigma' \triangleright f'$ is augmented with a store, which records the bounded type variables $X := A$. The above example reduces in F_C as follows.

$$\begin{aligned}
&\Sigma \triangleright (K^{\star} : \star \Rightarrow \forall X.\, \forall Y.\, X \to Y \to X)\ \mathsf{Int}\ \mathsf{Int}\ 2\ 3\\
\hookrightarrow^{*}\ &\Sigma \triangleright (\Lambda X.\Lambda Y.K^{\star} : \star \Rightarrow X \to Y \to X)\ \mathsf{Int}\ \mathsf{Int}\ 2\ 3\\
\hookrightarrow^{*}\ &\Sigma, X := \mathsf{Int}, Y := \mathsf{Int} \triangleright (K^{\star} : \star \Rightarrow X \to Y \to X)\ \mathsf{Int}\ 2\ 3\\
\hookrightarrow^{*}\ &2
\end{aligned}$$

Furthermore, they argue that type bindings generated locally lead to run-time overheads. Their observation is that type bindings are not required for every substitution, but only

for casts with the dynamic type ($\star$). Therefore they employ two kinds of type variables, which are distinguished by labels. One kind is static type variables (X::S) and the other kind is gradual type variables (X::G). Type application for static type abstraction does not generate type bindings, which are only generated for gradual type abstractions. Parametricity and the static gradual guarantee are proved, although the proofs are not mechanized. However, the dynamic gradual guarantee is left as conjecture. In addition their static gradual guarantee is proved with some constraints in the type precision relation. In their precision, $\forall X.\, X \to X$ is more precise than $\forall X.\, X \to \star$ but not $\forall X.\, \star \to X$.

The GSF calculus. Toro *et al.* [37] presented the gradual polymorphic calculus (named GSF), which employs the Abstracting Gradual Typing (AGT) methodology. In AGT, casting of higher-order values is eager compared to $\lambda B^{\forall}$ and F_C. This avoids the problem of space consumption although, as New *et al.* [25] pointed out, the η principle (which ensures $V \equiv \lambda x.Vx$ in the call-by-value languages) is broken. To preserve parametricity, global dynamic sealing, which does not distinguish between static and gradual variables, is used. They also refine the presentation of evidence, which witnesses the consistency judgement, ensuring that it holds. Instead of simple evidences such as $(\langle \alpha, Int \rangle)$, they employ sealing evidences $(\langle \alpha^E, Int \rangle)$. GSF satisfies parametricity but not the gradual guarantee. Importantly, they proved that the gradual guarantee is incompatible with parametricity.

Parametricity with the Gradual Guarantee. To achieve both parametricity and the gradual guarantee, New *et al.* [24] designed $PolyG^{\nu}$ calculus which gave up the syntax of System F and the users are required to provide different sealing options. They introduced the sealed syntax as $seal_X\ M$ which explicitly seals terms. With the user-defined syntax, the gradual guarantee and parametricity are proved. More recently, Labrada *et al.* [20] improve on GSF. They do not change the syntax of System F but insert plausible sealing forms during the elaboration from a gradual source language which is named Funk to a target cast calculus. They proved the gradual guarantee and parametricity for the target language, but for the source language (Funk), the gradual guarantee comes with a restriction for type applications, which can only be instantiated with base and variable types. Some of the main theorems are proved in Agda.

Summary. In order to keep parametricity we need several compromises. For instance, we need to use a dynamic sealing mechanism instead of direct type substitution causing extra space and time consumption. In many of the earlier calculi, the gradual guarantee is not obtained. In the later calculi, the gradual guarantee is either restricted or we need to give up the syntax of System F. Traditionally, many works on gradual typing are based on two different calculi: a source gradually typed language, and a target cast/-coercion calculus where casts/coercions are explicit. The dynamic semantics is defined by elaborating the source language to the target calculus. In other words, the semantics of the gradually typed language is given indirectly via a second, target language. All previously discussed works follow this indirect way to give the semantics to a gradually typed source language.

Furthermore, none of the gradually typed polymorphic calculi supports references. However, even for a static polymorphic calculus extended with mutable references ob-

	$\lambda B^{\forall}$	F_G	GSF	$PolyG^{\nu}$	Funk	λ^G_{gpr}
	2011	2017	2019	2020	2022	present work
Direct Substitution	×	×	×	×	×	✓
System F extension	×	✓	✓	×	✓	✓
Direct Semantics	×	×	×	×	×	✓
Parametricity	✓	✓	✓	✓	✓	×
Gradual Guarantee	×	×	×	✓	✓́	✓
Semantics	Lazy	Lazy	Eager	Lazy	Eager	Eager
Mechanized Proofs	×	×	×	×	✓́	✓
References	×	×	×	×	×	✓

Table 1: Comparison among gradual polymorphism calculi. A × denotes no. A ✓ denotes yes while ✓́ denotes partial yes.

taining parametricity is highly non-trivial. As Ahmed *et al.* [2] stated: "*combing mutable references with polymorphism can be extremely tricky*." From the analysis of Jaber and Tzevelekos [18], we know that naively moving from a polymorphic calculus to incorporate with mutable references, breaks parametricity. The reason is that common references can be instantiated with differently typed variables. Therefore, extending a gradual polymorphic calculus with the mutable references is non-trivial, and none of the existing gradual languages with polymorphism support references.

Table 1 summarizes several features and differences in existing gradually polymorphic calculi.

7 Conclusion

In this paper, we design a static system λ_{gpr} with polymorphism and references and its gradual counterpart λ^G_{gpr}. λ^G_{gpr} has a direct semantics without resorting to a cast calculi. In λ^G_{gpr}, the gradual guarantee is proved but we give up parametricity. In exchange, our calculus can be simplified, since sophisticated mechanisms such as dynamic sealing are not needed. Our calculus follows the original semantics of System F, based on direct type substitutions, avoiding extra space and time complexity that is necessary by mechanisms such as dynamic sealing. In the future, we could try to find out if there is a way to keep both gradual guarantee and relational parametricity for the source language, or explore more efficient formulations of λ^G_{gpr}.

Acknowledgements We are grateful to anonymous reviewers and our colleagues at the HKU PL group. This work has been sponsored by Hong Kong Research Grants Council projects number 17209520 and 17209821.

References

1. Abadi, M., Cardelli, L., Pierce, B.C., Plotkin, G.D.: Dynamic typing in a statically-typed language. In: POPL '89 (1989)
2. Ahmed, A., Appel, A.W., Virga, R.: An indexed model of impredicative polymorphism and mutable references (2003)
3. Ahmed, A., Findler, R.B., Siek, J.G., Wadler, P.: Blame for all. In: Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 201–214 (2011)
4. Barendregt, H.P., Church, A.: The impact of the lambda calculus (2014)
5. Bierman, G., Abadi, M., Torgersen, M.: Understanding typescript. In: European Conference on Object-Oriented Programming. pp. 257–281. Springer (2014)
6. Bracha, G.: The dart programming language. Addison-Wesley Professional (2015)
7. Cartwright, R., Fagan, M.: Soft typing. In: PLDI '91 (1991)
8. Chaudhuri, A.: Flow: a static type checker for javascript. SPLASH-I In Systems, Programming, Languages and Applications: Software for Humanity (2015)
9. Cimini, M., Siek, J.G.: The gradualizer: a methodology and algorithm for generating gradual type systems. Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2016)
10. Dunfield, J., Krishnaswami, N.R.: Complete and easy bidirectional typechecking for higher-rank polymorphism. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. p. 429–442. ICFP '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2500365.2500582, https://doi.org/10.1145/2500365.2500582
11. Garcia, R.: Calculating threesomes, with blame. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming. pp. 417–428 (2013)
12. Garcia, R., Clark, A.M., Tanter, E.: Abstracting gradual typing. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 429–442. POPL '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837670, https://doi.org/10.1145/2837614.2837670
13. Herman, D., Tomb, A., Flanagan, C.: Space-efficient gradual typing. In: In Trends in Functional Programming (TFP (2007)
14. Herman, D., Tomb, A., Flanagan, C.: Space-efficient gradual typing. Higher-Order and Symbolic Computation **23**(2), 167 (2010)
15. Huang, X., Oliveira, B.C.d.S.: A Type-Directed Operational Semantics For a Calculus with a Merge Operator. In: Hirschfeld, R., Pape, T. (eds.) 34th European Conference on Object-Oriented Programming (ECOOP 2020). Leibniz International Proceedings in Informatics (LIPIcs), vol. 166, pp. 26:1–26:32. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.26, https://drops.dagstuhl.de/opus/volltexte/2020/13183
16. Huang, X., Zhao, J., Oliveira, B.C.d.S.: Taming the merge operator. Journal of Functional Programming **31**, e28 (2021)
17. Igarashi, Y., Sekiyama, T., Igarashi, A.: On polymorphic gradual typing. Proc. ACM Program. Lang. **1**(ICFP) (aug 2017). https://doi.org/10.1145/3110284, https://doi.org/10.1145/3110284
18. Jaber, G., Tzevelekos, N.: Trace semantics for polymorphic references*. 2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) pp. 1–10 (2016)
19. Jones, S.L.P., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. Journal of Functional Programming **17**, 1 – 82 (2007)

20. Labrada, E., Toro, M., Tanter, E., Devriese, D.: Plausible sealing for gradual parametricity. Proc. ACM Program. Lang. **6**(OOPSLA1) (apr 2022). https://doi.org/10.1145/3527314, https://doi.org/10.1145/3527314
21. Matthews, J., Ahmed, A.: Parametric polymorphism through run-time sealing or, theorems for low, low prices! In: Drossopoulou, S. (ed.) Programming Languages and Systems. pp. 16–31 (2008)
22. Matthews, J., Findler, R.B.: Operational semantics for multi-language programs. ACM Trans. Program. Lang. Syst. **31**(3) (apr 2009). https://doi.org/10.1145/1498926.1498930, https://doi.org/10.1145/1498926.1498930
23. Møgelberg, R.E., Simpson, A.K.: Relational parametricity for computational effects. Logical Methods in Computer Science **5**, 1–31 (2009)
24. New, M.S., Jamner, D., Ahmed, A.: Graduality and parametricity: together again for the first time. Proceedings of the ACM on Programming Languages **4**, 1 – 32 (2020)
25. New, M.S., Licata, D.R., Ahmed, A.: Gradual type theory. Proc. ACM Program. Lang. **3**(POPL) (jan 2019). https://doi.org/10.1145/3290328, https://doi.org/10.1145/3290328
26. Odersky, M., Läufer, K.: Putting type annotations to work. In: POPL '96 (1996)
27. Ozaki, S., Sekiyama, T., Igarashi, A.: Is space-efficient polymorphic gradual typing possible? (2021)
28. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: IFIP Congress (1983)
29. Siek, J., Thiemann, P., Wadler, P.: Blame and coercion: Together again for the first time. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 425–435. PLDI '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2737968, https://doi.org/10.1145/2737924.2737968
30. Siek, J.G., Garcia, R., Taha, W.: Exploring the design space of higher-order casts. In: ESOP (2009)
31. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop. vol. 6, pp. 81–92 (2006)
32. Siek, J.G., Vitousek, M.M., Cimini, M., Boyland, J.T.: Refined criteria for gradual typing. In: 1st Summit on Advances in Programming Languages (SNAPL 2015). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015)
33. Siek, J.G., Vitousek, M.M., Cimini, M., Tobin-Hochstadt, S., Garcia, R.: Monotonic references for efficient gradual typing. In: ESOP (2015)
34. Siek, J.G., Wadler, P.: Threesomes, with and without blame. In: Proceedings for the 1st Workshop on Script to Program Evolution. p. 34–46. STOP '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1570506.1570511, https://doi.org/10.1145/1570506.1570511
35. Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: from scripts to programs. In: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications. pp. 964–974 (2006)
36. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed scheme. In: POPL '08 (2008)
37. Toro, M., Labrada, E., Tanter, É.: Gradual parametricity, revisited. Proceedings of the ACM on Programming Languages **3**, 1 – 30 (2019)
38. Toro, M., Tanter, É.: Abstracting gradual references. Sci. Comput. Program. **197**, 102496 (2020)
39. Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: European Symposium on Programming. pp. 1–16. Springer (2009)
40. Wright, A.K.: Simple imperative polymorphism. LISP and Symbolic Computation **8**, 343–355 (1995)

41. Xie, N., Bi, X., d. S. Oliveira, B.C.: Consistent subtyping for all. ACM Transactions on Programming Languages and Systems (TOPLAS) **42**, 1 – 79 (2018)
42. Ye, W., Oliveira, B.C.d.S., Huang, X.: Type-directed operational semantics for gradual typing. In: 35th European Conference on Object-Oriented Programming (ECOOP 2021). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2021)