Consistent Subtyping for All

Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira

The University of Hong Kong
{nnxie,xbi,bruno}@cs.hku.hk

Abstract. Consistent subtyping is employed in some gradual type systems to validate type conversions. The original definition by Siek and Taha serves as a guideline for designing gradual type systems with subtyping. Polymorphic types à la System F also induce a subtyping relation that relates polymorphic types to their instantiations. However Siek and Taha's definition is not adequate for polymorphic subtyping. The first goal of this paper is to propose a generalization of consistent subtyping that is adequate for polymorphic subtyping, and subsumes the original definition by Siek and Taha. The new definition of consistent subtyping provides novel insights with respect to previous polymorphic gradual type systems, which did not employ consistent subtyping. The second goal of this paper is to present a gradually typed calculus for implicit (higher-rank) polymorphism that uses our new notion of consistent subtyping. We develop both declarative and (bidirectional) algorithmic versions for the type system. We prove that the new calculus satisfies all static aspects of the refined criteria for gradual typing, which are mechanically formalized using the Coq proof assistant.

1 Introduction

Gradual typing [20] is an increasingly popular topic in both programming language practice and theory. On the practical side there is a growing number of programming languages adopting gradual typing. Those languages include Clojure [6], Python [26], TypeScript [5], Hack [25], and the addition of Dynamic to C# [4], to cite a few. On the theoretical side, recent years have seen a large body of research that defines the foundations of gradual typing [12, 8, 9], explores their use for both functional and object-oriented programming [20, 21], as well as its applications to many other areas [23, 3].

A key concept in gradual type systems is *consistency* [20]. Consistency weakens type equality to allow for the presence of *unknown* types. In some gradual type systems with subtyping, consistency is combined with subtyping to give rise to the notion of *consistent subtyping* [21]. Consistent subtyping is employed by gradual type systems to validate type conversions arising from conventional subtyping. One nice feature of consistent subtyping is that it is derivable from the more primitive notions of *consistency* and *subtyping*. As Siek and Taha [21] put it this shows that "*gradual typing and subtyping are orthogonal and can be combined in a principled fashion*". Thus consistent subtyping is often used as a guideline for designing gradual type systems with subtyping.

Unfortunately, as noted by Garcia et al. [12], notions of consistency and/or consistent subtyping "*become more difficult to adapt as type systems get more complex*". In particular, for the case of type systems with subtyping, certain kinds of subtyping do not fit well with the original definition of consistent subtyping by Siek and Taha [21]. One important case where such mismatch happens is in type systems supporting implicit (higher-rank) polymorphism [17, 10]. It is well-known that polymorphic types à la System F induce a subtyping relation that relates polymorphic types to their instantiations [16, 15]. However Siek and Taha's definition is not adequate for this kind of subtyping. Moreover the current framework for *Abstracting Gradual Typing* (AGT) [12] also does not account for polymorphism, with the authors acknowledging that this is one of the interesting avenues for future work.

Existing work on gradual type systems with polymorphism does not use consistent subtyping. The Polymorphic Blame Calculus ($\lambda\mathsf{B}$) [1] is an *explicitly* polymorphic calculus with explicit casts, which is often used as a target language for gradual type systems with polymorphism. In $\lambda\mathsf{B}$ a notion of *compatibility* is employed to validate conversions allowed by casts. Interestingly $\lambda\mathsf{B}$ *allows conversions from polymorphic types to their instantiations.* For example, it is possible to cast a value with type $\forall a.a \to a$ into $\mathsf{Int} \to \mathsf{Int}$. Thus an important remark here is that while $\lambda\mathsf{B}$ is explicitly polymorphic, casting and conversions are closer to *implicit* polymorphism. That is, in a conventional explicitly polymorphic calculus (such as System F), the primary notion is type equality, where instantiation is not taken into account. Thus the types $\forall a.a \to a$ and $\mathsf{Int} \to \mathsf{Int}$ are deemed *incompatible.* However in *implicitly* polymorphic calculi [17, 10] $\forall a.a \to a$ and $\mathsf{Int} \to \mathsf{Int}$ are deemed *compatible*, since the latter type is an instantiation of the former. Therefore $\lambda\mathsf{B}$ is in a sense a hybrid between implicit and explicit polymorphism, utilizing type equality (à la System F) for validating applications, and *compatibility* for validating casts.

An alternative approach to polymorphism has recently been proposed by Igarashi et al. [13]. Like $\lambda\mathsf{B}$ their calculus is explicitly polymorphic. However, in that work they employ type consistency to validate cast conversions, and forbid conversions from $\forall a.a \to a$ to $\mathsf{Int} \to \mathsf{Int}$. This makes their casts closer to explicit polymorphism, in contrast to $\lambda\mathsf{B}$. Nonetheless, there is still same flavour of implicit polymorphism in their calculus when it comes to interactions between dynamically typed and polymorphically typed code. For example, in their calculus type consistency allows types such as $\forall a.a \to \mathsf{Int}$ to be related to $\star \to \mathsf{Int}$, which can be viewed as a form of subtyping.

The first goal of this paper is to study the gradually typed subtyping and consistent subtyping relations for *predicative implicit polymorphism.* To accomplish this, we first show how to reconcile consistent subtyping with polymorphism by generalizing the original consistent subtyping definition by Siek and Taha. The new definition of consistent subtyping can deal with polymorphism, and preserves the orthogonality between consistency and subtyping. To slightly rephrase Siek and Taha, the motto of our paper is that:

Gradual typing and **polymorphism** *are orthogonal and can be combined in a principled fashion.*[1]

With the insights gained from our work, we argue that, for implicit polymorphism, Ahmed et al.'s notion of compatibility is too permissive (i.e. too many programs are allowed to type-check), and that Igarashi et al.'s notion of type consistency is too conservative. As a step towards an algorithmic version of consistent subtyping, we present a syntax-directed version of consistent subtyping that is sound and complete with respect to our formal definition of consistent subtyping. The syntax-directed version of consistent subtyping is remarkably simple and well-behaved, without the ad-hoc *restriction* operator [21]. Moreover, to further illustrate the generality of our consistent subtyping definition, we show that it can also account for *top types*, which cannot be dealt with by Siek and Taha's definition either.

The second goal of this paper is to present a (source-level) gradually typed calculus for (predicative) implicit higher-rank polymorphism that uses our new notion of consistent subtyping. As far as we are aware, there is no work on bridging the gap between implicit higher-rank polymorphism and gradual typing, which is interesting for two reasons. On one hand, from a practitioner's point of view, modern functional languages (such as Haskell) employ sophisticated type-inference algorithms that, aided by type annotations, can deal with implicit higher-rank polymorphism. So a natural question is how gradual typing can be integrated in such languages. On the other hand, there is several existing work on integrating *explicit* polymorphism into gradual typing [1, 13]. Yet no work investigates how to move such expressive power into a source language via implicit polymorphism. Therefore as a step towards gradualizing such type systems, this paper develops both declarative and algorithmic versions for a gradual type system with implicit higher-rank polymorphism. The new calculus brings the expressive power of full implicit higher-rank polymorphic into a gradually typed source language. We prove that the new calculus satisfies all of the *static* aspects of the refined criteria for gradual typing proposed by Siek et al. [24].

In summary, the contributions of this paper are:

- We define a framework for consistent subtyping with:
  - a new definition of consistent subtyping that subsumes and generalizes that of Siek and Taha. This new definition can deal with polymorphism and top types.
  - a syntax-directed version of consistent subtyping that is sound and complete with respect to our definition of consistent subtyping, but still guesses polymorphic instantiations.
- Based on consistent subtyping, we present a declarative gradual type system with predicative implicit higher-rank polymorphism. We prove that our calculus satisfies the static aspects of the refined criteria for gradual typing [24],

[1] Note here that we borrow Siek and Taha's motto mostly to talk about the static semantics. As Ahmed et al. [1] show there are several non-trivial interactions between polymorphism and casts at the level of the dynamic semantics.

$$\boxed{A <: B}$$

$$\mathsf{Int} <: \mathsf{Int} \qquad \mathsf{Bool} <: \mathsf{Bool} \qquad \mathsf{Float} <: \mathsf{Float} \qquad \mathsf{Int} <: \mathsf{Float}$$

$$\frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2} \qquad [l_i : A_i^{i \in 1 \dots n+m}] <: [l_i : A_i^{i \in 1 \dots n}] \qquad \star <: \star$$

$$\boxed{A \sim B}$$

$$A \sim A \qquad A \sim \star \qquad \star \sim A \qquad \frac{A_1 \sim B_1 \qquad A_2 \sim B_2}{A_1 \to A_2 \sim B_1 \to B_2} \qquad \frac{A_i \sim B_i}{[l_i : A_i] \sim [l_i : B_i]}$$

Fig. 1: Subtyping and type consistency in $\mathbf{FOb}^?_{<:}$

and is type-safe by a type-directed translation to $\lambda\mathsf{B}$, and thus hereditarily preserves parametricity [2].
- We present a complete and sound bidirectional algorithm for implementing the declarative system based on the design principle of Garcia and Cimini [11] and the approach of Dunfield and Krishnaswami [10].
- All of the metatheory of this paper, except some manual proofs for the algorithmic type system, has been mechanically formalized in Coq[2].

2 Background and Motivation

In this section we review a simple gradually typed language with objects [21], to introduce the concept of consistency subtyping. We also briefly talk about the Odersky-Läufer type system for higher-rank types [16], which serves as the original language on which our gradually typed calculus with implicit higher-rank polymorphism is based.

2.1 Gradual Subtyping

Siek and Taha [21] developed a gradual typed system for object-oriented languages that they call $\mathbf{FOb}^?_{<:}$. Central to gradual typing is the concept of *consistency* (written $\sim$) between gradual types, which are types that may involve the unknown type $\star$. The intuition is that consistency relaxes the structure of a type system to tolerate unknown positions in a gradual type. They also defined the subtyping relation in a way that static type safety is preserved. Their key insight is that the unknown type $\star$ is neutral to subtyping, with only $\star <: \star$. Both relations are found in Fig. 1.

[2] All supplementary materials are available at `https://bitbucket.org/xieningning/consistent-subtyping`

A primary contribution of their work is to show that consistency and subtyping are orthogonal. To compose subtyping and consistency, Siek and Taha defined *consistent subtyping* (written $\lesssim$) in two equivalent ways:

Definition 1 (Consistent Subtyping à la Siek and Taha [21]).

- $A \lesssim B$ *if and only if* $A \sim C$ *and* $C <: B$ *for some* C*.*
- $A \lesssim B$ *if and only if* $A <: C$ *and* $C \sim B$ *for some* C*.*

Both definitions are non-deterministic because of the intermediate type C. To remove non-determinism, they came up with a so-called *restriction operator*, written $A|_B$ that masks off the parts of a type A that are unknown in a type B.

$$
\begin{aligned}
A|_B = \;& \textbf{case}\;\; A, B \;\;\textbf{of} \mid (-, \star) \Rightarrow \star \\
& \mid A_1 \to A_2, B_1 \to B_2 = A_1|_{B_1} \to A_2|_{B_2} \\
& \mid [l_1 : A_1, ..., l_n : A_n], [l_1 : B_1, ..., l_m : B_m] \;\;\textbf{if}\;\; n \leq m \Rightarrow [l_1 : A_1|_{B_1}, ..., l_n : A_n|_{B_n}] \\
& \mid [l_1 : A_1, ..., l_n : A_n], [l_1 : B_1, ..., l_m : B_m] \;\;\textbf{if}\;\; n > m \Rightarrow \\
& \qquad [l_1 : A_1|_{B_1}, ..., l_m : A_m|_{B_m}, ..., l_n : A_n] \\
& \mid \textbf{otherwise} \Rightarrow A
\end{aligned}
$$

With the restriction operator, consistent subtyping is simply defined as $A \lesssim B \equiv A|_B <: B|_A$. Then they proved that this definition is equivalent to Definition 1.

2.2 The Odersky-Läufer Type System

The calculus we are combining gradual typing with is the well-established predicative type system for higher-rank types proposed by Odersky and Läufer [16]. One difference is that, for simplicity, we do not account for a let expression, as there is already existing work about gradual type systems with let expressions and let generalization (for example, see Garcia and Cimini [11]). Similar techniques can be applied to our calculus to enable let generalization.

The syntax of the type system, along with the typing and subtyping judgments is given in Fig. 2. An implicit assumption throughout the paper is that variables in contexts are distinct. We save the explanations for the static semantics to Section 4, where we present our gradually typed version of the calculus.

2.3 Motivation: Gradually Typed Higher-Rank Polymorphism

Our work combines implicit (higher-rank) polymorphism with gradual typing. As is well known, a gradually typed language supports both fully static and fully dynamic checking of program properties, as well as the continuum between these two extremes. It also offers programmers fine-grained control over the static-to-dynamic spectrum, i.e., a program can be evolved by introducing more or less precise types as needed [12].

Haskell is a language that supports implicit higher-rank polymorphism, but no gradual typing. Therefore some programs that are safe at run-time may be rejected due to the conservativity of the type system. For example, consider the following Haskell program adapted from Peyton Jones et al. [17]:

Expressions	$e ::= x \mid n \mid \lambda x : A.\ e \mid \lambda x.\ e \mid e\ e$
Types	$A, B ::= \mathsf{Int} \mid a \mid A \to B \mid \forall a.A$
Monotypes	$\tau, \sigma ::= \mathsf{Int} \mid a \mid \tau \to \sigma$
Contexts	$\Psi ::= \varnothing \mid \Psi, x : A \mid \Psi, a$

$$\boxed{\Psi \vdash^{OL} e : A}$$

$$\frac{x : A \in \Psi}{\Psi \vdash^{OL} x : A}\ \textsc{Var} \qquad \frac{}{\Psi \vdash^{OL} n : \mathsf{Int}}\ \textsc{Nat} \qquad \frac{\Psi, x : A \vdash^{OL} e : B}{\Psi \vdash^{OL} \lambda x : A.\ e : A \to B}\ \textsc{LamAnn}$$

$$\frac{\Psi \vdash^{OL} e_1 : A_1 \to A_2 \qquad \Psi \vdash^{OL} e_2 : A_1}{\Psi \vdash^{OL} e_1\ e_2 : A_2}\ \textsc{App} \qquad \frac{\Psi \vdash^{OL} e : A_1 \qquad \Psi \vdash A_1 <: A_2}{\Psi \vdash^{OL} e : A_2}\ \textsc{Sub}$$

$$\frac{\Psi, x : \tau \vdash^{OL} e : B}{\Psi \vdash^{OL} \lambda x.\ e : \tau \to B}\ \textsc{Lam} \qquad \frac{\Psi, a \vdash^{OL} e : A}{\Psi \vdash^{OL} e : \forall a.A}\ \textsc{Gen}$$

$$\boxed{\Psi \vdash A <: B}$$

$$\frac{a \in \Psi}{\Psi \vdash a <: a}\ \text{CS-TVar} \qquad \frac{}{\Psi \vdash \mathsf{Int} <: \mathsf{Int}}\ \text{CS-Int} \qquad \frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] <: B}{\Psi \vdash \forall a.A <: B}\ \textsc{ForallL}$$

$$\frac{\Psi, a \vdash A <: B}{\Psi \vdash A <: \forall a.B}\ \textsc{ForallR} \qquad \frac{\Psi \vdash B_1 <: A_1 \qquad \Psi \vdash A_2 <: B_2}{\Psi \vdash A_1 \to A_2 <: B_1 \to B_2}\ \text{CS-Fun}$$

Fig. 2: Syntax and static semantics of the Odersky-Läufer type system.

foo :: ([**Int**], [**Char**])
foo = **let** *f x* = (*x* [1, 2] , *x* [′*a*′, ′*b*′]) **in** *f* **reverse**

This program is rejected by Haskell's type checker because Haskell implements the Damas-Milner rule that a lambda-bound argument (such as x) can only have a monotype, i.e., the type checker can only assign x the type [**Int**] $\to$ [**Int**], or [**Char**] $\to$ [**Char**], but not $\forall a.a \to a$. Finding such manual polymorphic annotations can be non-trivial. Instead of rejecting the program outright, due to missing type annotations, gradual typing provides a simple alternative by giving x the unknown type (denoted $\star$). With such typing the same program type-checks and produces ([2, 1], [′*b*′, ′*a*′]). By running the program, programmers can gain some additional insight about the run-time behaviour. Then, with such insight, they can also give x a more precise type ($\forall a.a \to a$) à posteriori so that the program continues to type-check via implicit polymorphism and also grants more static safety. In this paper, we envision such a language that combines the benefits of both implicit higher-rank polymorphism and gradual typing.

Types	$A, B ::= \mathsf{Int} \mid a \mid A \to B \mid \forall a. A \mid \star$
Monotypes	$\tau, \sigma ::= \mathsf{Int} \mid a \mid \tau \to \sigma$
Contexts	$\Psi ::= \varnothing \mid \Psi, x : A \mid \Psi, a$

$$\boxed{A \sim B}$$

$$A \sim A \qquad A \sim \star \qquad \star \sim A \qquad \frac{A_1 \sim B_1 \qquad A_2 \sim B_2}{A_1 \to A_2 \sim B_1 \to B_2} \qquad \frac{A \sim B}{\forall a. A \sim \forall a. B}$$

$$\boxed{\Psi \vdash A <: B}$$

$$\frac{\Psi, a \vdash A <: B}{\Psi \vdash A <: \forall a. B}\ \textsc{S-ForallR} \qquad \frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] <: B}{\Psi \vdash \forall a. A <: B}\ \textsc{S-ForallL} \qquad \frac{a \in \Psi}{\Psi \vdash a <: a}\ \textsc{S-TVar}$$

$$\frac{}{\Psi \vdash \mathsf{Int} <: \mathsf{Int}}\ \textsc{S-Int} \qquad \frac{\Psi \vdash B_1 <: A_1 \qquad \Psi \vdash A_2 <: B_2}{\Psi \vdash A_1 \to A_2 <: B_1 \to B_2}\ \textsc{S-Fun} \qquad \frac{}{\Psi \vdash \star <: \star}\ \textsc{S-Unknown}$$

Fig. 3: Syntax of types, consistency, and subtyping in the declarative system.

3 Revisiting Consistent Subtyping

In this section we explore the design space of consistent subtyping. We start with the definitions of consistency and subtyping for polymorphic types, and compare with some relevant work. We then discuss the design decisions involved towards our new definition of consistent subtyping, and justify the new definition by demonstrating its equivalence with that of Siek and Taha [21] and the AGT approach [12] on simple types.

The syntax of types is given at the top of Fig. 3. We write A, B for types. Types are either the integer type Int, type variables a, functions types $A \to B$, universal quantification $\forall a. A$, or the unknown type $\star$. Though we only have one base type Int, we also use Bool for the purpose of illustration. Note that monotypes τ contain all types other than the universal quantifier and the unknown type $\star$. We will discuss this restriction when we present the subtyping rules. Contexts Ψ are *ordered* lists of type variable declarations and term variables.

3.1 Consistency and Subtyping

We start by giving the definitions of consistency and subtyping for polymorphic types, and comparing our definitions with the compatibility relation by Ahmed et al. [1] and type consistency by Igarashi et al. [13].

Consistency The key observation here is that consistency is mostly a structural relation, except that the unknown type $\star$ can be regarded as any type. Following this observation, we naturally extend the definition from Fig. 1 with polymorphic types, as shown at the middle of Fig. 3. In particular a polymorphic type $\forall a. A$ is consistent with another polymorphic type $\forall a. B$ if A is consistent with B.

Subtyping We express the fact that one type is a polymorphic generalization of another by means of the subtyping judgment $\Psi \vdash A <: B$. Compared with the subtyping rules of Odersky and Läufer [16] in Fig. 2, the only addition is the neutral subtyping of $\star$. Notice that in the rule S-ForallL, the universal quantifier is only allowed to be instantiated with a *monotype*. The judgment $\Psi \vdash \tau$ checks all the type variables in τ are bound in the context Ψ. For space reasons, we omit the definition. According to the syntax in Fig. 3, monotypes do not contain the unknown type $\star$. This is because if we were to allow the unknown type to be used for instantiation, we could have $\forall a.a \to a <: \star \to \star$ by instantiating a with $\star$. Since $\star \to \star$ is consistent with any functions $A \to B$, for instance, $\mathsf{Int} \to \mathsf{Bool}$, this means that we could provide an expression of type $\forall a.a \to a$ to a function where the input type is supposed to be $\mathsf{Int} \to \mathsf{Bool}$. However, as we might expect, $\forall a.a \to a$ is definitely not compatible with $\mathsf{Int} \to \mathsf{Bool}$. This does not hold in any polymorphic type systems without gradual typing. So the gradual type system should not accept it either. (This is the so-called *conservative extension* property that will be made precise in Section 4.3.)

Importantly there is a subtle but crucial distinction between a type variable and the unknown type, although they all represent a kind of "arbitrary" type. The unknown type stands for the absence of type information: it could be *any type* at *any instance*. Therefore, the unknown type is consistent with any type, and additional type-checks have to be performed at runtime. On the other hand, a type variable indicates *parametricity*. In other words, a type variable can only be instantiated to a single type. For example, in the type $\forall a.a \to a$, the two occurrences of a represent an arbitrary but single type (e.g., $\mathsf{Int} \to \mathsf{Int}$, $\mathsf{Bool} \to \mathsf{Bool}$), while $\star \to \star$ could be an arbitrary function (e.g., $\mathsf{Int} \to \mathsf{Bool}$) at runtime.

Comparison with Other Relations In other polymorphic gradual calculi, consistency and subtyping are often mixed up to some extent. In the Polymorphic Blame Calculus ($\lambda\mathsf{B}$) [1], the compatibility relation for polymorphic types is defined as follows:

$$\frac{A \prec B}{A \prec \forall X.B}\ \textsc{Comp-AllR} \qquad\qquad \frac{A[X \mapsto \star] \prec B}{\forall X.A \prec B}\ \textsc{Comp-AllL}$$

Notice that, in rule Comp-AllL, the universal quantifier is *always* instantiated to $\star$. However, this way, $\lambda\mathsf{B}$ allows $\forall a.a \to a \prec \mathsf{Int} \to \mathsf{Bool}$, which as we discussed before might not be what we expect. Indeed $\lambda\mathsf{B}$ relies on sophisticated runtime checks to rule out such instances of the compatibility relation à posteriori.

Igarashi et al. [13] introduced the so-called *quasi-polymorphic* types for types that may be used where a $\forall$-type is expected, which is important for their purpose of conservativity over System F. Their type consistency relation, involving polymorphism, is defined as follows[3]:

$$\frac{A \sim B}{\forall a.A \sim \forall a.B} \qquad\qquad \frac{A \sim B \qquad B \neq \forall a.B' \qquad \star \in \mathsf{Types}(B)}{\forall a.A \sim B}$$

[3] This is a simplified version.

$$\begin{tikzcd}
\bot \arrow[r, dash, "\sim"] & (\star \to \mathsf{Int}) \to \mathsf{Int} \\
(\forall a.a \to \mathsf{Int}) \to \mathsf{Int} \arrow[u, "<:"] \arrow[r, dash, "\sim"'] & (\forall a.\star \to \mathsf{Int}) \to \mathsf{Int} \arrow[u, "<:"]
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
\mathsf{Int} \to \mathsf{Int} \arrow[r, dash, "\sim"] & \mathsf{Int} \to \star \\
\forall a.a \arrow[u, "<:"] \arrow[r, dash, "\sim"'] & \bot \arrow[u, "<:"]
\end{tikzcd}$$

(b)

$$\begin{tikzcd}
\bot \arrow[r, dash, "\sim"] & (((\star \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\mathsf{Int} \to \star) \\
(((\forall a.a \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\forall a.a) \arrow[u, "<:"] \arrow[r, dash, "\sim"'] & \bot \arrow[u, "<:"]
\end{tikzcd}$$

(c)

Fig. 4: Examples that break the original definition of consistent subtyping.

Compared with our consistency definition in Fig. 3, their first rule is the same as ours. The second rule says that a non $\forall$-type can be consistent with a $\forall$-type only if it contains $\star$. In this way, their type system is able to reject $\forall a.a \to a \sim \mathsf{Int} \to \mathsf{Bool}$. However, in order to keep conservativity, they also reject $\forall a.a \to a \sim \mathsf{Int} \to \mathsf{Int}$, which is perfectly sensible in their setting (i.e., explicit polymorphism). However with implicit polymorphism, we would expect $\forall a.a \to a$ to be related with $\mathsf{Int} \to \mathsf{Int}$, since a can be instantiated to Int.

Nonetheless, when it comes to interactions between dynamically typed and polymorphically typed terms, both relations allow for example $\forall a.a \to \mathsf{Int}$ to be related with $\star \to \mathsf{Int}$, which in our view, is some sort of (implicit) polymorphic subtyping, and that should be derivable by the more primitive notions in the type system (instead of inventing new relations). One of our design principles is that, subtyping and consistency should be *orthogonal*, and can be naturally superimposed, echoing the same opinion of Siek and Taha [21].

3.2 Towards Consistent Subtyping

With the definitions of consistency and subtyping, the question now is how to compose these two relations so that two types can be compared in a way that takes these two relations into account.

Unfortunately, the original definition of Siek and Taha (Definition 1) does not work well with our definitions of consistency and subtyping for polymorphic types. Consider two types: $(\forall a.a \to \mathsf{Int}) \to \mathsf{Int}$, and $(\star \to \mathsf{Int}) \to \mathsf{Int}$. The first type can only reach the second type in one way (first by applying consistency, then subtyping), but not the other way, as shown in Fig. 4a. We use $\bot$ to mean that we cannot find such a type. Similarly, there are situations where the first type can only reach the second type by the other way (first applying subtyping, and then consistency), as shown in Fig. 4b.

What is worse, if those two examples are composed in a way that those types all appear co-variantly, then the resulting types cannot reach each other in either

$$\begin{tikzcd}
T_1 \arrow[r, dash, "\sim"] & C \\
B \arrow[u, "<:"] \arrow[r, dash, "\sim"] \arrow[ur, dashed, "\lesssim"] & T_2 \arrow[u, "<:"] \\
A \arrow[u, "<:"] \arrow[uur, dashed, bend right=60, "\lesssim"'] &
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
 & A \\
T_1 \arrow[r, dash, "\sim"] & B \arrow[u, "<:"] \\
C \arrow[u, "<:"] \arrow[r, dash, "\sim"'] \arrow[ur, dashed, "\lesssim"] \arrow[uur, dashed, bend left=60, "\lesssim"] & T_2 \arrow[u, "<:"]
\end{tikzcd}$$

(b)

Fig. 5: Observations of consistent subtyping

way. For example, Fig. 4c shows such two types by putting a Bool type in the middle, and neither definition of consistent subtyping works.

Observations on consistent subtyping based on information propagation. In order to develop the correct definition of consistent subtyping for polymorphic types, we need to understand how consistent subtyping works. We first review two important properties of subtyping: (1) subtyping induces the subsumption rule: if $A <: B$, then an expression of type A can be used where B is expected; (2) subtyping is transitive: if $A <: B$, and $B <: C$, then $A <: C$. Though consistent subtyping takes the unknown type into consideration, the subsumption rule should also apply: if $A \lesssim B$, then an expression of type A can also be used where B is expected, given that there might be some information lost by consistency. A crucial difference from subtyping is that consistent subtyping is *not* transitive because information can only be lost once (otherwise, any two types are a consistent subtype of each other). Now consider a situation where we have both $A <: B$, and $B \lesssim C$, this means that A can be used where B is expected, and B can be used where C is expected, with possibly some loss of information. In other words, we should expect that A can be used where C is expected, since there is at most one-time loss of information.

Observation 1 *If $A <: B$, and $B \lesssim C$, then $A \lesssim C$.*

This is reflected in Fig. 5a. A symmetrical observation is given in Fig. 5b:

Observation 2 *If $C \lesssim B$, and $B <: A$, then $C \lesssim A$.*

From the above observations, we see what the problem is with the original definition. In Fig. 5a, if B can reach C by T_1, then by subtyping transitivity, A can reach C by T_1. However, if B can only reach C by T_2, then A cannot reach C through the original definition. A similar problem is shown in Fig. 5b.

However, it turns out that those two problems can be fixed by the same strategy: instead of taking one-step subtyping and one-step consistency, our definition of consistent subtyping allows types to take *one-step subtyping, one-step consistency, and one more step subtyping*. Specifically, $A <: B \sim T_2 <: C$ (in Fig. 5a) and $C <: T_1 \sim B <: A$ (in Fig. 5b) have the same relation chain: subtyping, consistency, and subtyping.

$$\begin{tikzcd}
A_1 \arrow[r, dash, "\sim"] & ((\forall a.\star \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\mathsf{Int} \to \star) \arrow[d, "<:"'] \\
(((\forall a.a \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\forall a.a) \arrow[u, "<:"] \arrow[r, "\lesssim"] & A_2
\end{tikzcd}$$

$$A_1 = ((\forall a.a \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\mathsf{Int} \to \mathsf{Int})$$
$$A_2 = (((\star \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\mathsf{Int} \to \star)$$

Fig. 6: Example that is fixed by the new definition of consistent subtyping.

Definition of consistent subtyping. From the above discussion, we are ready to modify Definition 1, and adapt it to our notation:

Definition 2 (Consistent Subtyping).

$$\frac{\Psi \vdash A <: C \qquad C \sim D \qquad \Psi \vdash D <: B}{\Psi \vdash A \lesssim B}$$

With Definition 2, Figure 6 illustrates the correct relation chain for the broken example shown in Fig. 4c. At first sight, Definition 2 seems worse than the original: we need to guess *two* types! It turns out that Definition 2 is a generalization of Definition 1, and they are equivalent in the system of Siek and Taha [21], whereas Definition 2 in particular is compatible with polymorphic types.

Proposition 1 (Generalization of Consistent Subtyping).

– *Definition 2 subsumes Definition 1. In Definition 2, by choosing $D = B$, we have $A <: C$ and $C \sim B$; by choosing $C = A$, we have $A \sim D$, and $D <: B$.*
– *Definition 1 is equivalent to Definition 2 in the system of Siek and Taha. If $A <: C$, $C \sim D$, and $D <: B$, by Definition 1, $A \sim C'$, $C' <: D$ for some C'. By subtyping transitivity, $C' <: B$. So $A \lesssim B$ by $A \sim C'$ and $C' <: B$.*

3.3 Abstracting Gradual Typing

Garcia et al. [12] presented a new foundation for gradual typing that they call the *Abstracting Gradual Typing* (AGT) approach. In the AGT approach, gradual types are interpreted as sets of static types, where static types refer to types containing no unknown types. In this interpretation, predicates and functions on static types can then be lifted to apply to gradual types. Central to their approach is the so-called *concretization* function. For simple types, a concretization γ from gradual types to a set of static types[4] is defined as follows:

Definition 3 (Concretization).

$$\gamma(\mathsf{Int}) = \{\mathsf{Int}\} \qquad \gamma(A \to B) = \gamma(A) \to \gamma(B) \qquad \gamma(\star) = \{\textit{All static types}\}$$

[4] For simplification, we directly regard type constructor $\to$ as a set-level operator.

Based on the concretization function, subtyping between static types can be lifted to gradual types, resulting in the consistent subtyping relation:

Definition 4 (Consistent Subtyping in AGT). *$A \mathrel{\widetilde{<:}} B$ if and only if $A_1 <: B_1$ for some $A_1 \in \gamma(A)$, $B_1 \in \gamma(B)$.*

Later they proved that this definition of consistent subtyping coincides with that of Siek and Taha [21] (Definition 1). By Proposition 1, we can directly conclude that our definition coincides with AGT:

Proposition 2 (Equivalence to AGT on Simple Types). *$A \lesssim B$ iff $A \mathrel{\widetilde{<:}} B$.*

However, AGT does not show how to deal with polymorphism (e.g. the interpretation of type variables) yet. Still, as noted by Garcia et al. [12] in the conclusion, it is a promising line of future work for AGT, and the question remains whether our definition would coincide with it.

Another note related to AGT is that the definition is later adopted by Castagna and Lanvin [7], where the static types A_1, B_1 in Definition 4 can be algorithmically computed by also accounting for top and bottom types.

3.4 Directed Consistency

Jafery and Dunfield [14] define *directed consistency* based on precision and static subtyping:

$$\frac{A' \sqsubseteq A \qquad A <: B \qquad B' \sqsubseteq B}{A' \lesssim B'}$$

The judgment $A \sqsubseteq B$ is read "A is less precise than B". In their setting, precision is defined for type constructors and subtyping for static types. If we interpret this definition from AGT's point of view, finding a more precise static type[5] has the same effect as concretization. Namely, $A' \sqsubseteq A$ implies $A \in \gamma(A')$ and $B' \sqsubseteq B$ implies $B \in \gamma(B')$. Therefore we consider this definition as AGT-style. From this perspective, this definition naturally coincides with Definition 2.

The value of their definition is that consistent subtyping is derived compositionally from *static subtyping* and *precision*. These are two more atomic relations. At first sight, their definition looks very similar to Definition 2 (replacing $\sqsubseteq$ by $<:$ and $<:$ by $\sim$). Then a question arises as to *which one is more fundamental.* To answer this, we need to discuss the relation between consistency and precision.

Relating Consistency and Precision. Precision is a partial order (anti-symmetric and transitive), while consistency is symmetric but not transitive. Nonetheless, precision and consistency are related by the following proposition:

Proposition 3 (Consistency and Precision).

- *If $A \sim B$, then there exists (static) C, such that $A \sqsubseteq C$, and $B \sqsubseteq C$.*
- *If for some (static) C, we have $A \sqsubseteq C$, and $B \sqsubseteq C$, then we have $A \sim B$.*

[5] The definition of precision of types is given in appendix.

It may seem that precision is a more atomic relation, since consistency can be derived from precision. However, recall that consistency is in fact an equivalence relation lifted from static types to gradual types. Therefore defining consistency independently is straightforward, and it is theoretically viable to validate the definition of consistency directly. On the other hand, precision is usually connected with the gradual criteria [24], and finding a correct partial order that adheres to the criteria is not always an easy task. For example, Igarashi et al. [13] argued that term precision for System F_G is actually nontrivial, leaving the gradual guarantee of the semantics as a conjecture. Thus precision can be difficult to extend to more sophisticated type systems, e.g. dependent types.

Still, it is interesting that those two definitions illustrate the correspondence of different foundations (on simple types): one is defined directly on gradual types, and the other stems from AGT, which is based on static subtyping.

3.5 Consistent Subtyping Without Existentials

Definition 2 serves as a fine specification of how consistent subtyping should behave in general. But it is inherently non-deterministic because of the two intermediate types C and D. As with Definition 1, we need a combined relation to directly compare two types. A natural attempt is to try to extend the restriction operator for polymorphic types. Unfortunately, as we show below, this does not work. However it is possible to devise an equivalent inductive definition instead.

Attempt to extend the restriction operator. Suppose that we try to extend the restriction operator to account for polymorphic types. The original restriction operator is structural, meaning that it works for types of similar structures. But for polymorphic types, two input types could have different structures due to universal quantifiers, e.g, $\forall a.a \to \mathsf{Int}$ and $(\mathsf{Int} \to \star) \to \mathsf{Int}$. If we try to mask the first type using the second, it seems hard to maintain the information that a should be instantiated to a function while ensuring that the return type is masked. There seems to be no satisfactory way to extend the restriction operator in order to support this kind of non-structural masking.

Interpretation of the restriction operator and consistent subtyping. If the restriction operator cannot be extended naturally, it is useful to take a step back and revisit what the restriction operator actually does. For consistent subtyping, two input types could have unknown types in different positions, but we only care about the known parts. What the restriction operator does is (1) erase the type information in one type if the corresponding position in the other type is the unknown type; and (2) compare the resulting types using the normal subtyping relation. The example below shows the masking-off procedure for the types $\mathsf{Int} \to \star \to \mathsf{Bool}$ and $\mathsf{Int} \to \mathsf{Int} \to \star$. Since the known parts have the relation that $\mathsf{Int} \to \star \to \star <: \mathsf{Int} \to \star \to \star$, we conclude that $\mathsf{Int} \to \star \to \mathsf{Bool} \lesssim \mathsf{Int} \to \mathsf{Int} \to \star$.

$$\left.\begin{array}{l} \mathsf{Int} \to \boxed{\star} \to \boxed{\mathsf{Bool}}\ |_{\mathsf{Int} \to \mathsf{Int} \to \star} = \mathsf{Int} \to \star \to \star \\ \mathsf{Int} \to \boxed{\mathsf{Int}} \to \boxed{\star}\ |_{\mathsf{Int} \to \star \to \mathsf{Bool}} = \mathsf{Int} \to \star \to \star \end{array}\right\} <:$$

$$\boxed{\Psi \vdash A \lesssim B}$$

$$\frac{\Psi, a \vdash A \lesssim B}{\Psi \vdash A \lesssim \forall a.B}\ \textsc{CS-ForallR} \qquad \frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] \lesssim B}{\Psi \vdash \forall a.A \lesssim B}\ \textsc{CS-ForallL}$$

$$\frac{\Psi \vdash B_1 \lesssim A_1 \qquad \Psi \vdash A_2 \lesssim B_2}{\Psi \vdash A_1 \to A_2 \lesssim B_1 \to B_2}\ \textsc{CS-Fun} \qquad \frac{a \in \Psi}{\Psi \vdash a \lesssim a}\ \textsc{CS-TVar} \qquad \frac{}{\Psi \vdash \mathsf{Int} \lesssim \mathsf{Int}}\ \textsc{CS-Int}$$

$$\frac{}{\Psi \vdash \star \lesssim A}\ \textsc{CS-UnknownL} \qquad \frac{}{\Psi \vdash A \lesssim \star}\ \textsc{CS-UnknownR}$$

Fig. 7: Consistent Subtyping for implicit polymorphism.

Here differences of the types in boxes are erased because of the restriction operator. Now if we compare the types in boxes directly instead of through the lens of the restriction operator, we can observe that the *consistent subtyping relation always holds between the unknown type and an arbitrary type.* We can interpret this observation directly from Definition 2: the unknown type is neutral to subtyping ($\star <: \star$), the unknown type is consistent with any type ($\star \sim A$), and subtyping is reflexive ($A <: A$). Therefore, *the unknown type is a consistent subtype of any type ($\star \lesssim A$), and vice versa ($A \lesssim \star$).* Note that this interpretation provides a general recipe on how to lift a (static) subtyping relation to a (gradual) consistent subtyping relation, as discussed below.

Defining consistent subtyping directly. From the above discussion, we can define the consistent subtyping relation directly, *without* resorting to subtyping or consistency at all. The key idea is that we replace $<:$ with $\lesssim$ in Fig. 3, get rid of rule S-Unknown and add two extra rules concerning $\star$, resulting in the rules of consistent subtyping in Fig. 7. Of particular interest are the rules CS-UnknownL and CS-UnknownR, both of which correspond to what we just said: the unknown type is a consistent subtype of any type, and vice versa. From now on, we use the symbol $\lesssim$ to refer to the consistent subtyping relation in Fig. 7. What is more, we can prove that those two are equivalent[6]:

$\mathcal{T}$heorem 1 *The following definitions are equivalent:*

– $\Psi \vdash A \lesssim B$.
– $\Psi \vdash A <: C$, $C \sim D$, $\Psi \vdash D <: B$, *for some* C, D.

4 Gradually Typed Implicit Polymorphism

In Section 3 we introduce the consistent subtyping relation that accommodates polymorphic types. In this section we continue with the development by giving a

[6] Theorems with $\mathcal{T}$ are those proved in Coq. The same applies to $\mathcal{L}$emmas.

declarative type system for predicative implicit polymorphism that employs the consistent subtyping relation. The declarative system itself is already quite interesting as it is equipped with both higher-rank polymorphism and the unknown type. The syntax of expressions in the declarative system is given below:

$$\text{Expressions} \quad e ::= x \mid n \mid \lambda x : A.\ e \mid \lambda x.\ e \mid e\ e$$

Meta-variable e ranges over expressions. Expressions are either variables x, integers n, annotated lambda abstractions $\lambda x : A.\ e$, un-annotated lambda abstractions $\lambda x.\ e$ or applications $e_1\ e_2$.

4.1 Typing in Detail

Figure 8 gives the typing rules for our declarative system (the reader is advised to ignore the gray-shaded parts for now). Rule Var extracts the type of the variable from the typing context. Rule Nat always infers integer types. Rule LamAnn puts x with type annotation A into the context, and continues type checking the body e. Rule Lam assigns a monotype τ to x, and continues type checking the body e. Gradual types and polymorphic types are introduced via annotations explicitly. Rule Gen puts a fresh type variable a into the type context and generalizes the typing result A to $\forall a.A$. Rule App first infers the type of e_1, then the matching judgment $\Psi \vdash A \triangleright A_1 \to A_2$ extracts the domain type A_1 and the codomain type A_2 from type A. The type A_3 of the argument e_2 is then compared with A_1 using the consistent subtyping judgment.

Matching. The matching judgment of Siek et al. [24] can be extended to polymorphic types naturally, resulting in $\Psi \vdash A \triangleright A_1 \to A_2$. In M-Forall, a monotype τ is guessed to instantiate the universal quantifier a. This rule is inspired by the *application judgment* $\Phi \vdash A \bullet e \Rightarrow C$ [10], which says that if we apply a term of type A to an argument e, we get something of type C. If A is a polymorphic type, the judgment works by guessing instantiations of polymorphic quantifiers until it reaches an arrow type. Matching further simplifies the application judgment, since it is independent of typing. Rule M-Arr and M-Unknown are the same as Siek et al. [24]. M-Arr returns the domain type A_1 and range type A_2 as expected. If the input is $\star$, then M-Unknown returns $\star$ as both the type for the domain and the range.

Note that matching saves us from having a subsumption rule (Sub in Fig. 2). the subsumption rule is incompatible with consistent subtyping, since the latter is not transitive. Otherwise, we can assign a typed expression any type by applying the subsumption rule twice, once to $\star$, and once to any type we want.

4.2 Type-directed Translation

We give the dynamic semantics of our language by translating it to $\lambda\mathsf{B}$. Below we show a subset of the terms in $\lambda\mathsf{B}$ that are used in the translation:

$$\text{Terms} \quad s ::= x \mid n \mid \lambda x : A.\ s \mid \Lambda a.s \mid s_1\ s_2 \mid \langle A \hookrightarrow B \rangle\ s$$

$$\boxed{\Psi \vdash e : A \rightsquigarrow s}$$

$$\frac{x : A \in \Psi}{\Psi \vdash x : A \rightsquigarrow x}\ \textsc{Var} \qquad \frac{}{\Psi \vdash n : \mathsf{Int} \rightsquigarrow n}\ \textsc{Nat} \qquad \frac{\Psi, a \vdash e : A \rightsquigarrow s}{\Psi \vdash e : \forall a.A \rightsquigarrow \Lambda a.s}\ \textsc{Gen}$$

$$\frac{\Psi, x : A \vdash e : B \rightsquigarrow s}{\Psi \vdash \lambda x : A.\ e : A \to B \rightsquigarrow \lambda x : A.\ s}\ \textsc{LamAnn} \qquad \frac{\Psi, x : \tau \vdash e : B \rightsquigarrow s}{\Psi \vdash \lambda x.\ e : \tau \to B \rightsquigarrow \lambda x : \tau.\ s}\ \textsc{Lam}$$

$$\frac{\Psi \vdash e_1 : A \rightsquigarrow s_1 \qquad \Psi \vdash A \rhd A_1 \to A_2 \qquad \Psi \vdash e_2 : A_3 \rightsquigarrow s_2 \qquad \Psi \vdash A_3 \lesssim A_1}{\Psi \vdash e_1\ e_2 : A_2 \rightsquigarrow (\langle A \hookrightarrow A_1 \to A_2 \rangle\ s_1)\ (\langle A_3 \hookrightarrow A_1 \rangle\ s_2)}\ \textsc{App}$$

$$\boxed{\Psi \vdash A \rhd A_1 \to A_2}$$

$$\frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] \rhd A_1 \to A_2}{\Psi \vdash \forall a.A \rhd A_1 \to A_2}\ \textsc{M-Forall}$$

$$\frac{}{\Psi \vdash (A_1 \to A_2) \rhd (A_1 \to A_2)}\ \textsc{M-Arr} \qquad \frac{}{\Psi \vdash \star \rhd \star \to \star}\ \textsc{M-Unknown}$$

Fig. 8: Declarative typing

A cast $\langle A \hookrightarrow B \rangle\ s$ converts the value of term s from type A to type B. A cast from A to B is permitted only if the types are *compatible*, written $A \prec B$, as briefly mentioned in Section 3.1. The syntax of types in $\lambda\mathsf{B}$ is the same as ours.

The translation is given in the gray-shaded parts in Fig. 8. The only interesting case here is to insert explicit casts in the application rule. Note that there is no need to translate matching or consistent subtyping, instead we insert the source and target types of a cast directly in the translated expressions, thanks to the following two lemmas:

Lemma 1 ($\rhd$ to $\prec$) *If $\Psi \vdash A \rhd A_1 \to A_2$, then $A \prec A_1 \to A_2$.*

Lemma 2 ($\lesssim$ to $\prec$) *If $\Psi \vdash A \lesssim B$, then $A \prec B$.*

In order to show the correctness of the translation, we prove that our translation always produces well-typed expressions in $\lambda\mathsf{B}$. By Lemmas 1 and 2, we have the following theorem:

Theorem 2 (Type Safety) *If $\Psi \vdash e : A \rightsquigarrow s$, then $\Psi \vdash^B s : A$.*

Parametricity An important semantic property of polymorphic types is *relational parametricity* [18]. The parametricity property says that all instances of a polymorphic function should behave *uniformly*. In other words, functions cannot inspect into a type variable, and act differently for different instances of the type variable. A classic example is a function with the type $\forall a.a \to a$. The parametricity property guarantees that a value of this type must be either the identity function (i.e., $\lambda x.x$) or the undefined function (one which never returns a value). However, with the addition of the unknown type $\star$, careful measures are to be taken to ensure parametricity. This is exactly the circumstance that $\lambda\mathsf{B}$ was designed to address. Ahmed et al. [2] proved that $\lambda\mathsf{B}$ satisfies relational parametricity. Based on their result, and by $\mathcal{T}$heorem 2, parametricity is preserved in our system.

Ambiguity from Casts The translation does not always produce a unique target expression. This is because when we guess a monotype τ in rule M-Forall and CS-ForallL, we could have different choices, which inevitably leads to different types. Unlike (non-gradual) polymorphic type systems [17, 10], the choice of monotypes could affect runtime behaviour of the translated programs, since they could appear inside the explicit casts. For example, the following shows two possible translations for the same source expression $\lambda x : \star.\ f\ x$, where the type of f is instantiated to $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Bool} \to \mathsf{Bool}$, respectively:

$$\begin{aligned}
&f : \forall a.a \to a \vdash (\lambda x : \star.\ f\ x) : \star \to \mathsf{Int} \\
&\qquad\qquad \rightsquigarrow (\lambda x : \star.\ (\langle \forall a.a \to a \hookrightarrow \mathsf{Int} \to \mathsf{Int}\rangle\ f)\ (\boxed{\langle \star \hookrightarrow \mathsf{Int}\rangle}\ x)) \\
&f : \forall a.a \to a \vdash (\lambda x : \star.\ f\ x) : \star \to \mathsf{Bool} \\
&\qquad\qquad \rightsquigarrow (\lambda x : \star.\ (\langle \forall a.a \to a \hookrightarrow \mathsf{Bool} \to \mathsf{Bool}\rangle\ f)\ (\boxed{\langle \star \hookrightarrow \mathsf{Bool}\rangle}\ x))
\end{aligned}$$

If we apply $\lambda x : \star.\ f\ x$ to 3, which is fine since the function can take any input, the first translation runs smoothly in $\lambda\mathsf{B}$, while the second one will raise a cast error (Int cannot be cast to Bool). Similarly, if we apply it to true, then the second succeeds while the first fails. The culprit lies in the highlighted parts where any instantiation of a would be put inside the explicit cast. More generally, any choice introduces an explicit cast to that type in the translation, which causes a runtime cast error if the function is applied to a value whose type does not match the guessed type. Note that this does not compromise the type safety of the translated expressions, since cast errors are part of the type safety guarantees.

Coherency The ambiguity of translation seems to imply that the declarative system is *incoherent*. A semantics is coherent if distinct typing derivations of the same typing judgment possess the same meaning [19]. We argue that the declarative system is "coherent up to cast errors" in the sense that a well-typed program produces a unique value, or results in a cast error. In the above example, whatever the translation might be, applying $\lambda x : \star.\ f\ x$ to 3 either results in a cast error, or produces 3, nothing else.

This discrepancy is due to the guessing nature of the *declarative* system. As far as the declarative system is concerned, both $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Bool} \to \mathsf{Bool}$ are equally acceptable. But this is not the case at runtime. The acute reader may have found that the *only* appropriate choice is to instantiate f to $\star \to \star$. However, as specified by rule M-Forall in Fig. 8, we can only instantiate type variables to monotypes, but $\star$ is *not* a monotype! We will get back to this issue in Section 6.2 after we present the corresponding algorithmic system in Section 5.

4.3 Correctness Criteria

Siek et al. [24] present a set of properties that a well-designed gradual typing calculus must have, which they call the refined criteria. Among all the criteria, those related to the static aspects of gradual typing are well summarized by Cimini and Siek [8]. Here we review those criteria and adapt them to our notation. We have proved in Coq that our type system satisfies all these criteria.

Lemma 3 (Correctness Criteria)

- ***Conservative extension:*** *for all static Ψ, e, and A,*
    - *if $\Psi \vdash^{OL} e : A$, then there exists B, such that $\Psi \vdash e : B$, and $\Psi \vdash B <: A$.*
    - *if $\Psi \vdash e : A$, then $\Psi \vdash^{OL} e : A$*
- ***Monotonicity w.r.t. precision:*** *for all Ψ, e, e', A, if $\Psi \vdash e : A$, and $e' \sqsubseteq e$, then $\Psi \vdash e' : B$, and $B \sqsubseteq A$ for some B.*
- ***Type Preservation of cast insertion:*** *for all Ψ, e, A, if $\Psi \vdash e : A$, then $\Psi \vdash e : A \rightsquigarrow s$, and $\Psi \vdash^{B} s : A$ for some s.*
- ***Monotonicity of cast insertion:*** *for all $\Psi, e_1, e_2, e_1', e_2', A$, if $\Psi \vdash e_1 : A \rightsquigarrow e_1'$, and $\Psi \vdash e_2 : A \rightsquigarrow e_2'$, and $e_1 \sqsubseteq e_2$, then $\Psi \mid \Psi \vdash e_1' \sqsubseteq^{B} e_2'$.*

The first criterion states that the gradual type system should be a conservative extension of the original system. In other words, a *static* program that is typeable in the Odersky-Läufer type system if and only if it is typeable in the gradual type system. A static program is one that does not contain any type $\star$[7]. However since our gradual type system does not have the subsumption rule, it produces more general types.

The second criterion states that if a typeable expression loses some type information, it remains typeable. This criterion depends on the definition of the precision relation, written $A \sqsubseteq B$, which is given in the appendix. The relation intuitively captures a notion of types containing more or less unknown types ($\star$). The precision relation over types lifts to programs, i.e., $e_1 \sqsubseteq e_2$ means that e_1 and e_2 are the same program except that e_2 has more unknown types.

The first two criteria are fundamental to gradual typing. They explain for example why these two programs $(\lambda x : \mathsf{Int}.\ x + 1)$ and $(\lambda x : \star.\ x + 1)$ are typeable, as the former is typeable in the Odersky-Läufer type system and the latter is a less-precise version of it.

The last two criteria relate the compilation to the cast calculus. The third criterion is essentially the same as Theorem 2, given that a target expression should always exist, which can be easily seen from Fig. 8. The last criterion ensures that the translation must be monotonic over the precision relation $\sqsubseteq$.

As for the dynamic guarantee, we leave it as an open question, since it is unknown whether it holds in $\lambda\mathsf{B}$. According to Igarashi et al. [13] (where they have System F_C which is similar to $\lambda\mathsf{B}$), the difficulty lies in the definition of term precision that preserves the semantics.

[7] Note that the term *static* has appeared several times with different meanings.

Expressions	$e ::= x \mid n \mid \lambda x : A.\ e \mid \lambda x.\ e \mid e\ e \mid e : A$
Types	$A, B ::= \mathsf{Int} \mid a \mid \widehat{a} \mid A \to B \mid \forall a.A \mid \star$
Monotypes	$\tau, \sigma ::= \mathsf{Int} \mid a \mid \widehat{a} \mid \tau \to \sigma$
Contexts	$\Gamma, \Delta, \Theta ::= \varnothing \mid \Gamma, x : A \mid \Gamma, a \mid \Gamma, \widehat{a} \mid \Gamma, \widehat{a} = \tau$
Complete Contexts	$\Omega ::= \varnothing \mid \Omega, x : A \mid \Omega, a \mid \Omega, \widehat{a} = \tau$

Fig. 9: Syntax of the algorithmic system

5 Algorithmic Type System

In this section we give a bidirectional account of the algorithmic type system that implements the declarative specification. The algorithm is largely inspired by the algorithmic bidirectional system of Dunfield and Krishnaswami [10] (henceforth DK system). However our algorithmic system differs from theirs in three aspects: 1) the addition of the unknown type $\star$; 2) the use of the matching judgment; and 3) the approach of *gradual inference only producing static types* [11]. We then prove that our algorithm is both sound and complete with respect to the declarative type system. Full proofs can be found in the supplementary material.

Algorithmic Contexts The algorithmic context Γ is an *ordered* list containing declarations of type variables a and term variables $x : A$. Unlike declarative contexts, algorithmic contexts also contain declarations of existential type variables $\widehat{a}$, which can be either unsolved (written $\widehat{a}$) or solved to some monotype (written $\widehat{a} = \tau$). Complete contexts Ω are those that contain no unsolved existential type variables. Figure 9 shows the syntax of the algorithmic system. Apart from expressions in the declarative system, we have annotated expressions $e : A$.

5.1 Algorithmic Consistent Subtyping

Figure 10 shows the algorithmic consistent subtyping rules. The first five rules do not manipulate contexts. Rule ACS-Fun is a natural extension of its declarative counterpart. The output context of the first premise is used by the second premise, and the output context of the second premise is the output context of the conclusion. Note that we do not simply check $A_2 \lesssim B_2$, but apply Θ (the input context of the second premise) to both types $\Theta \vdash [\Theta]A_2 \lesssim [\Theta]B_2 \dashv \Delta$. This is to maintain an important invariant: whenever we try to derive $\Gamma \vdash A \lesssim B \dashv \Delta$, the types A and B are already fully applied under Γ (they contain no existential variables already solved in Γ). More generally, every algorithmic judgment form has the property that the input types are fully applied under the input context.

Rule ACS-ForallR looks similar to its declarative counterpart, except that we need to drop the trailing context a, Θ from the concluding output context since they become out of scope. The next rule is essential to eliminating the guessing work, thus appears significantly different from its declarative version. Instead of guessing a monotype τ out of thin air, rule ACS-ForallL generates a fresh existential variable $\widehat{a}$, and replaces a with $\widehat{a}$ in the body A. The new

$$\boxed{\Gamma \vdash A \lesssim B \dashv \Delta}$$

$$\frac{}{\Gamma[a] \vdash a \lesssim a \dashv \Gamma[a]}\ \text{ACS-TVar} \qquad \frac{}{\Gamma[\widehat{a}] \vdash \widehat{a} \lesssim \widehat{a} \dashv \Gamma[\widehat{a}]}\ \text{ACS-ExVar}$$

$$\frac{}{\Gamma \vdash \mathsf{Int} \lesssim \mathsf{Int} \dashv \Gamma}\ \text{ACS-Int} \qquad \frac{}{\Gamma \vdash \star \lesssim A \dashv \Gamma}\ \text{ACS-UnknownL} \qquad \frac{}{\Gamma \vdash A \lesssim \star \dashv \Gamma}\ \text{ACS-UnknownR}$$

$$\frac{\Gamma \vdash B_1 \lesssim A_1 \dashv \Theta \qquad \Theta \vdash [\Theta]A_2 \lesssim [\Theta]B_2 \dashv \Delta}{\Gamma \vdash A_1 \to A_2 \lesssim B_1 \to B_2 \dashv \Delta}\ \text{ACS-Fun}$$

$$\frac{\Gamma, a \vdash A \lesssim B \dashv \Delta, a, \Theta}{\Gamma \vdash A \lesssim \forall a.B \dashv \Delta}\ \text{ACS-ForallR} \qquad \frac{\Gamma, \widehat{a} \vdash A[a \mapsto \widehat{a}] \lesssim B \dashv \Delta}{\Gamma \vdash \forall a.A \lesssim B \dashv \Delta}\ \text{ACS-ForallL}$$

$$\frac{\widehat{a} \notin fv(A) \qquad \Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox A \dashv \Delta}{\Gamma[\widehat{a}] \vdash \widehat{a} \lesssim A \dashv \Delta}\ \text{ACS-InstL} \qquad \frac{\widehat{a} \notin fv(A) \qquad \Gamma[\widehat{a}] \vdash A \lessapprox \widehat{a} \dashv \Delta}{\Gamma[\widehat{a}] \vdash A \lesssim \widehat{a} \dashv \Delta}\ \text{ACS-InstR}$$

Fig. 10: Algorithmic consistent subtyping

existential variable $\widehat{a}$ is then added to the premise's input context. As a side note, when both types are quantifiers, then either ACS-ForallR or ACS-ForallR could be tried. In practice, one can apply ACS-ForallR eagerly.

The last two rules are specific to the algorithm, thus having no counterparts in the declarative version. They together check consistent subtyping with an unsolved existential variable on one side and an arbitrary type on the other side by the help of the instantiation judgment.

5.2 Instantiation

The judgment $\Gamma \vdash \widehat{a} \lessapprox A \dashv \Delta$ defined in Fig. 11 instantiates unsolved existential variables. Judgment $\widehat{a} \lessapprox A$ reads "instantiate $\widehat{a}$ to a consistent subtype of A". For space reasons, we omit its symmetric judgement $\Gamma \vdash A \lessapprox \widehat{a} \dashv \Delta$.

Rule InstLSolve and rule InstLReach set $\widehat{a}$ to τ and $\widehat{b}$ in the output context, respectively. Rule InstLSolveU is similar to ACS-UnknownR in that we put no constraint on $\widehat{a}$ when it meets the unknown type $\star$. This design decision reflects the point that type inference only produces static types [11]. We will get back to this point in Section 6.2. Rule InstLAllR is the instantiation version of rule ACS-ForallR. The last rule InstLArr applies when $\widehat{a}$ meets a function type. It follows that the solution must also be a function type. That is why in the first premise, we generate two fresh existential variables $\widehat{a}_1$ and $\widehat{a}_2$, and insert them just before $\widehat{a}$ in the input context, so that the solution of $\widehat{a}$ can mention them. Note that $A_1 \lessapprox \widehat{a}_1$ switches to the other instantiation judgment.

$$\boxed{\Gamma \vdash \widehat{a} \lessapprox A \dashv \Delta}$$

$$\frac{\Gamma \vdash \tau}{\Gamma, \widehat{a}, \Gamma' \vdash \widehat{a} \lessapprox \tau \dashv \Gamma, \widehat{a} = \tau, \Gamma'}\ \textsc{InstLSolve} \qquad \frac{}{\Gamma[\widehat{a}][\widehat{b}] \vdash \widehat{a} \lessapprox \widehat{b} \dashv \Gamma[\widehat{a}][\widehat{b} = \widehat{a}]}\ \textsc{InstLReach}$$

$$\frac{}{\Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox \star \dashv \Gamma[\widehat{a}]}\ \textsc{InstLSolveU} \qquad \frac{\Gamma[\widehat{a}], b \vdash \widehat{a} \lessapprox B \dashv \Delta, b, \Delta'}{\Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox \forall b.B \dashv \Delta}\ \textsc{InstLAllR}$$

$$\frac{\Gamma[\widehat{a}_2, \widehat{a}_1, \widehat{a} = \widehat{a}_1 \to \widehat{a}_2] \vdash A_1 \lessapprox \widehat{a}_1 \dashv \Theta \qquad \Theta \vdash \widehat{a}_2 \lessapprox [\Theta]A_2 \dashv \Delta}{\Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox A_1 \to A_2 \dashv \Delta}\ \textsc{InstLArr}$$

Fig. 11: Algorithmic instantiation

5.3 Algorithmic Typing

We now turn to the algorithmic typing rules in Fig. 12. The algorithmic system uses bidirectional type checking to accommodate polymorphism. Most of them are quite standard. Perhaps rule AApp (which differs significantly from that in the DK system) deserves attention. It relies on the algorithmic matching judgment $\Gamma \vdash A \rhd A_1 \to A_2 \dashv \Delta$. Rule AM-ForallL replaces a with a fresh existential variable $\widehat{a}$, thus eliminating guessing. Rule AM-Arr and AM-Unknown correspond directly to the declarative rules. Rule AM-Var, which has no corresponding declarative version, is similar to InstRArr/InstLArr: we create $\widehat{a}$ and $\widehat{b}$ and add $\widehat{c} = \widehat{a} \to \widehat{b}$ to the context.

5.4 Completeness and Soundness

We prove that the algorithmic rules are sound and complete with respect to the declarative specifications. We need an auxiliary judgment $\Gamma \longrightarrow \Delta$ that captures a notion of information increase from input contexts Γ to output contexts Δ [10].

Soundness Roughly speaking, soundness of the algorithmic system says that given an expression e that type checks in the algorithmic system, there exists a corresponding expression e' that type checks in the declarative system. However there is one complication: e does not necessarily have more annotations than e'. For example, by ALam we have $\lambda x.\ x \Leftarrow (\forall a.a) \to (\forall a.a)$, but $\lambda x.\ x$ itself cannot have type $(\forall a.a) \to (\forall a.a)$ in the declarative system. To circumvent that, we add an annotation to the lambda abstraction, resulting in $\lambda x : (\forall a.a).\ x$, which is typeable in the declarative system with the same type. To relate $\lambda x.\ x$ and $\lambda x : (\forall a.a).\ x$, we erase all annotations on both expressions. The definition of erasure $\lfloor \cdot \rfloor$ is standard and thus omitted.

Theorem 1 (Soundness of Algorithmic Typing) *Given $\Delta \longrightarrow \Omega$,*

1. *If $\Gamma \vdash e \Rightarrow A \dashv \Delta$ then $\exists e'$ such that $[\Omega]\Delta \vdash e' : [\Omega]A$ and $\lfloor e \rfloor = \lfloor e' \rfloor$.*
2. *If $\Gamma \vdash e \Leftarrow A \dashv \Delta$ then $\exists e'$ such that $[\Omega]\Delta \vdash e' : [\Omega]A$ and $\lfloor e \rfloor = \lfloor e' \rfloor$.*

$$\boxed{\Gamma \vdash e \Rightarrow A \dashv \Delta}$$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A \dashv \Gamma}\ \textsc{AVar} \qquad \frac{}{\Gamma \vdash n \Rightarrow \mathsf{Int} \dashv \Gamma}\ \textsc{ANat}$$

$$\frac{\Gamma, \widehat{a}, \widehat{b}, x : \widehat{a} \vdash e \Leftarrow \widehat{b} \dashv \Delta, x : \widehat{a}, \Theta}{\Gamma \vdash \lambda x.\ e \Rightarrow \widehat{a} \rightarrow \widehat{b} \dashv \Delta}\ \textsc{ALamU} \qquad \frac{\Gamma, x : A \vdash e \Rightarrow B \dashv \Delta, x : A, \Theta}{\Gamma \vdash \lambda x : A.\ e \Rightarrow A \rightarrow B \dashv \Delta}\ \textsc{ALamAnnA}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash e \Leftarrow A \dashv \Delta}{\Gamma \vdash e : A \Rightarrow A \dashv \Delta}\ \textsc{AAnno}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \dashv \Theta_1 \qquad \Theta_1 \vdash [\Theta_1]A \rhd A_1 \rightarrow A_2 \dashv \Theta_2 \qquad \Theta_2 \vdash e_2 \Leftarrow [\Theta_2]A_1 \dashv \Delta}{\Gamma \vdash e_1\ e_2 \Rightarrow A_2 \dashv \Delta}\ \textsc{AApp}$$

$$\boxed{\Gamma \vdash e \Leftarrow A \dashv \Delta}$$

$$\frac{\Gamma, x : A \vdash e \Leftarrow B \dashv \Delta, x : A, \Theta}{\Gamma \vdash \lambda x.\ e \Leftarrow A \rightarrow B \dashv \Delta}\ \textsc{ALam} \qquad \frac{\Gamma, a \vdash e \Leftarrow A \dashv \Delta, a, \Theta}{\Gamma \vdash e \Leftarrow \forall a.A \dashv \Delta}\ \textsc{AGen}$$

$$\frac{\Gamma \vdash e \Rightarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A \lesssim [\Theta]B \dashv \Delta}{\Gamma \vdash e \Leftarrow B \dashv \Delta}\ \textsc{ASub}$$

$$\boxed{\Gamma \vdash A \rhd A_1 \rightarrow A_2 \dashv \Delta}$$

$$\frac{\Gamma, \widehat{a} \vdash A[a \mapsto \widehat{a}] \rhd A_1 \rightarrow A_2 \dashv \Delta}{\Gamma \vdash \forall a.A \rhd A_1 \rightarrow A_2 \dashv \Delta}\ \textsc{AM-Forall} \qquad \frac{}{\Gamma \vdash (A_1 \rightarrow A_2) \rhd (A_1 \rightarrow A_2) \dashv \Gamma}\ \textsc{AM-Arr}$$

$$\frac{}{\Gamma \vdash \star \rhd \star \rightarrow \star \dashv \Gamma}\ \textsc{AM-Unknown} \qquad \frac{}{\Gamma[\widehat{c}] \vdash \widehat{c} \rhd \widehat{a} \rightarrow \widehat{b} \dashv \Gamma[\widehat{a}, \widehat{b}, \widehat{c} = \widehat{a} \rightarrow \widehat{b}]}\ \textsc{AM-Var}$$

Fig. 12: Algorithmic typing

Completeness Completeness of the algorithmic system is the reverse of soundness: given a declarative judgment of the form $[\Omega]\Gamma \vdash [\Omega]\ldots$, we want to get an algorithmic derivation of $\Gamma \vdash \cdots \dashv \Delta$. It turns out that completeness is a bit trickier to state in that the algorithmic rules generate existential variables on the fly, so Δ could contain unsolved existential variables that are not found in Γ, nor in Ω. Therefore the completeness proof must produce another complete context Ω' that extends both the output context Δ, and the given complete context Ω. As with soundness, we need erasure to relate both expressions.

Theorem 2 (Completeness of Algorithmic Typing) *Given $\Gamma \longrightarrow \Omega$ and $\Gamma \vdash A$, if $[\Omega]\Gamma \vdash e : A$ then there exist Δ, Ω', A' and e' such that $\Delta \longrightarrow \Omega'$ and $\Omega \longrightarrow \Omega'$ and $\Gamma \vdash e' \Rightarrow A' \dashv \Delta$ and $A = [\Omega']A'$ and $\lfloor e \rfloor = \lfloor e' \rfloor$.*

6 Discussion

6.1 Top Types

To demonstrate that our definition of consistent subtyping (Definition 2) is applicable to other features, we show how to extend our approach to Top types with all the desired properties preserved.

In order to preserve the orthogonality between subtyping and consistency, we require $\top$ to be a common supertype of all static types, as shown in rule S-Top. This rule might seem strange at first glance, since even if we remove the requirement A *static*, the rule seems reasonable. However, an important point is that because of the orthogonality between subtyping and consistency, subtyping itself should not contain a potential cast in principle! Therefore, subtyping instances such as $\star <: \top$ are not allowed. For consistency, we add the rule that $\top$ is consistent with $\top$, which is actually included in the original reflexive rule $A \sim A$. For consistent subtyping, every type is a consistent subtype of $\top$, for example, $\mathsf{Int} \to \star \lesssim \top$.

$$\frac{A \ \mathit{static}}{\Psi \vdash A <: \top}\ \textsc{S-Top} \qquad\qquad \top \sim \top \qquad\qquad \frac{}{\Psi \vdash A \lesssim \top}\ \textsc{CS-Top}$$

It is easy to verify that Definition 2 is still equivalent to that in Fig. 7 extended with rule CS-Top. That is, Theorem 1 holds:

Proposition 4 (Extension with $\top$). *The following are equivalent:*

- $\Psi \vdash A \lesssim B$.
- $\Psi \vdash A <: C$, $C \sim D$, $\Psi \vdash D <: B$, *for some* C, D.

We extend the definition of concretization (Definition 3) with $\top$ by adding another equation $\gamma(\top) = \{\top\}$. Note that Castagna and Lanvin [7] also have this equation in their calculus. It is easy to verify that Proposition 2 still holds:

Proposition 5 (Equivalent to AGT Extended with $\top$ on Simple Types). $A \lesssim B$ *if only if* $A \mathrel{\widetilde{<:}} B$.

Siek and Taha's definition of consistent subtyping does not work for $\top$ As the analysis in Section 3.2, $\mathsf{Int} \to \star \lesssim \top$ only holds when we first apply consistency, then subtyping. However we cannot find a type A such that $\mathsf{Int} \to \star <: A$ and $A \sim \top$. Also we have a similar problem in extending the restriction operator: *non-structural* masking between $\mathsf{Int} \to \star$ and $\top$ cannot be easily achieved.

6.2 Interpretation of the Dynamic Semantics

In Section 4.2 we have seen an example where a source expression could produce two different target expressions with different runtime behaviour. As we explained, this is due to the guessing nature of the declarative system, and from the typing point of view, no type is particularly better than others. However in practice, this is not desirable. Let us revisit the same example, now from the algorithmic point of view (we omit the translation for space reasons):

$$f : \forall a.a \to a \vdash (\lambda x : \star.\ f\ x) \Rightarrow \star \to \widehat{a} \dashv f : \forall a.a \to a, \widehat{a}$$

Compared with declarative typing, which produces many types ($\star \to \mathsf{Int}$, $\star \to \mathsf{Bool}$, and so on), the algorithm computes the type $\star \to \widehat{a}$ with $\widehat{a}$ unsolved in the output context. What can we know from the output context? The only thing we know is that $\widehat{a}$ is not constrained at all! However, it is possible to make a more refined distinction between different kinds of existential variables. The first kind of existential variables are those that indeed have no constraints at all, as they do not affect the dynamic semantics. The second kind of existential variables (as in this example) are those where the only constraint is that *the variable was once compared with an unknown type* [11].

To emphasize the difference and have better support for dynamic semantics, we could have *gradual variables* in addition to existential variables, with the difference that only unsolved gradual variables are allowed to be unified with the unknown type. An irreversible transition from existential variables to gradual variables occurs when an existential variable is compared with $\star$. After the algorithm terminates, we can set all unsolved existential variables to be any (static) type (or more precisely, as Garcia and Cimini [11], with *static type parameters*), and all unsolved gradual variables to be $\star$ (or *gradual type parameters*). However, this approach requires a more sophisticated declarative/algorithmic type system than the ones presented in this paper, where we only produce static monotypes in type inference. We believe this is a typical trade-off in existing gradual type systems with inference [22, 11]. Here we suppress the complexity of dynamic semantics in favour of the conciseness of static typing.

7 Related Work

Along the way we discussed some of the most relevant work to motivate, compare and promote our gradual typing design. In what follows, we briefly discuss related work on gradual typing and polymorphism.

Gradual Typing The seminal paper by Siek and Taha [20] is the first to propose gradual typing. The original proposal extends the simply typed lambda calculus by introducing the unknown type $\star$ and replacing type equality with type consistency. Casts are introduced to mediate between statically and dynamically typed code. Later Siek and Taha [21] incorporated gradual typing into a simple object oriented language, and showed that subtyping and consistency are orthogonal – an insight that partly inspired our work. We show that subtyping and consistency are orthogonal in a much richer type system with higher-rank polymorphism. In the light of the ever-growing popularity of gradual typing, and its somewhat murky theoretical foundations, Siek et al. [24] felt the urge to have a complete formal characterization of what it means to be gradually typed. They proposed a set of criteria that provides important guidelines for designers of gradually typed languages. Cimini and Siek [8] introduced the *Gradualizer*, a general algorithmic methodology for generating gradual type systems from static type systems. Later they extend it so that the Gradualizer can generate dynamic semantics as well [9]. Garcia et al. [12] introduced the AGT approach based on abstract interpretation.

Gradual Type Systems with Explicit Polymorphism Ahmed et al. [1] proposed the Polymorphic Blame Calculus that extends the blame calculus [28] to incorporate polymorphism. The key novelty of their work is to use dynamic sealing to enforce parametricity. Igarashi et al. [13] also studied integrating gradual typing with parametric polymorphism. They proposed System F_G, a gradually typed extension of System F, and System F_C, a new polymorphic blame calculus. As has been discussed extensively, their definition of type consistency does not apply to our setting (implicit polymorphism). All of these approaches mix consistency with subtyping to some extent, which we argue should be orthogonal.

Gradual Type Inference Siek and Vachharajani [22] studied unification-based type inference for gradual typing, where they show why three straightforward approaches fail to meet their design goals. Their type system infers gradual types, which results in a complicated type system and inference algorithm. Garcia and Cimini [11] presented a new approach that gradual type inference only produces static types, which is adopted in our type system. They also deal with let-polymorphism (rank 1 types). However none of these works deals with higher-ranked implicit polymorphism.

Higher-rank Implicit Polymorphism Odersky and Läufer [16] introduced a type system for higher-rank types. Based on that, Peyton Jones et al. [17] developed an approach for type checking higher-rank predicative polymorphism. Dunfield and Krishnaswami [10] proposed a bidirectional account of higher-rank polymorphism, and an algorithm for implementing the declarative system, which serves as a sole inspiration for our algorithmic system. The key difference, however, is the integration of gradual typing. Vytiniotis et al. [27] defers static type errors to runtime, which is fundamentally different from gradual typing, where programmers can control over static or runtime checks by precision of the annotations.

8 Conclusion

In this paper, we present a generalized definition of consistent subtyping, which is proved to be applicable to both polymorphic and top types. Based on the new definition of consistent subtyping, we have developed a gradually typed calculus with predicative implicit higher-rank polymorphism, and an algorithm to implement it. As future work, we are interested to investigate if our results can scale to real world languages and other programming language features.

Bibliography

[1] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *Proceedings of the 38th Symposium on Principles of Programming Languages*, 2011.

[2] Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for free for free: Parametricity, with and without types. In *Proceedings of the 22nd International Conference on Functional Programming*, 2017.

[3] Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. A theory of gradual effect systems. In *Proceedings of the 19th International Conference on Functional Programming*, 2014.

[4] Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to c#. In *Proceedings of the European Conference on Object-Oriented Programming*, 2010.

[5] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *Proceedings of the 28th European Conference on Object-Oriented Programming*, 2014.

[6] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical optional types for clojure. In *Programming Languages and Systems*. 2016.

[7] Giuseppe Castagna and Victor Lanvin. Gradual typing with union and intersection types. *Proc. ACM Program. Lang.*, 1(ICFP):41:1–41:28, August 2017.

[8] Matteo Cimini and Jeremy G. Siek. The gradualizer: A methodology and algorithm for generating gradual type systems. In *Proceedings of the 43rd Symposium on Principles of Programming Languages*, 2016.

[9] Matteo Cimini and Jeremy G. Siek. Automatically generating the dynamic semantics of gradually typed languages. In *Proceedings of the 44th Symposium on Principles of Programming Languages*, 2017.

[10] Joshua Dunfield and Neelakantan R Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *International Conference on Functional Programming*, 2013.

[11] Ronald Garcia and Matteo Cimini. Principal type schemes for gradual programs. In *Proceedings of the 42nd Symposium on Principles of Programming Languages*, 2015.

[12] Ronald Garcia, Alison M Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd Symposium on Principles of Programming Languages*, 2016.

[13] Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. On polymorphic gradual typing. In *Proceedings of the 22nd International Conference on Functional Programming*, 2017.

[14] Khurram A. Jafery and Joshua Dunfield. Sums of uncertainty: Refinements go gradual. In *Proceedings of the 44th Symposium on Principles of Programming Languages*, 2017.

[15] John C Mitchell. Polymorphic type inference and containment. In *Logical foundations of functional programming*, 1990.

[16] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Proceedings of the 23rd Symposium on Principles of Programming Languages*, 1996.

[17] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1–82, 2007.

[18] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Proceedings of the IFIP 9th World Computer Congress*, 1983.

[19] John C. Reynolds. The coherence of languages with intersection types. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, 1991.

[20] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Proceedings of the 2006 Scheme and Functional Programming Workshop*, 2006.

[21] Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, 2007.

[22] Jeremy G. Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *Proceedings of the 2008 Symposium on Dynamic Languages*, 2008.

[23] Jeremy G. Siek and Philip Wadler. The key to blame: Gradual typing meets cryptography (draft), 2016.

[24] Jeremy G. Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, 2015.

[25] Julien Verlaguet. Facebook: Analyzing php statically. In *Proceedings of Commercial Users of Functional Programming*, 2013.

[26] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for python. In *Proceedings of the 10th Symposium on Dynamic languages*, 2014.

[27] Dimitrios Vytiniotis, Simon Peyton Jones, and José Pedro Magalhães. Equality proofs and deferred type errors: A compiler pearl. In *Proceedings of the 17th International Conference on Functional Programming*, ICFP '12, New York, NY, USA, 2012.

[28] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems*, 2009.