Consistent Subtyping for All

NINGNING XIE, The University of Hong Kong, China
XUAN BI, The University of Hong Kong, China
BRUNO C. D. S. OLIVEIRA, The University of Hong Kong, China
TOM SCHRIJVERS, KU Leuven, Belgium

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. The algorithmic version employs techniques developed by Dunfield and Krishnaswami on higher-rank polymorphism to deal with instantiation. We prove that the new calculus satisfies all static aspects of the refined criteria for gradual typing. We also study an extension of the type system with static and gradual type parameters, in an attempt to support a variant of the dynamic criterion for gradual typing. Assuming a coherence conjecture for the extended calculus, we show that the dynamic gradual guarantee of our source language can be reduced to that of λB, which, at the time of writing, is still an open question. Most of the metatheory of this paper, except some manual proofs for the algorithmic type system and extensions, has been mechanically formalized using the Coq proof assistant.

ACM Reference Format:
Ningning Xie, Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2010. Consistent Subtyping for All. *ACM Trans. Web* 9, 4, Article 39 (March 2010), 85 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Gradual typing [Siek and Taha 2006] 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 [Bonnaire-Sergeant et al. 2016], Python [Vitousek et al. 2014], TypeScript [Bierman et al. 2014], Hack [Verlaguet 2013], and the addition of Dynamic to C# [Bierman et al. 2010], to name a few. On the theoretical side, recent years have seen a large body of research that defines the foundations of gradual typing [Cimini and Siek 2016, 2017; Garcia et al. 2016], explores their use for both functional and object-oriented programming [Siek and Taha 2006, 2007], as well as its applications to many other areas [Bañados Schwerter et al. 2014; Castagna and Lanvin 2017; Jafery and Dunfield 2017; Siek and Wadler 2016].

Authors' addresses: Ningning Xie, The University of Hong Kong, Hong Kong, China, nnxie@cs.hku.hk; Xuan Bi, The University of Hong Kong, Hong Kong, China, xbi@cs.hku.hk; Bruno C. d. S. Oliveira, The University of Hong Kong, Hong Kong, China, bruno@cs.hku.hk; Tom Schrijvers, KU Leuven, Leuven, Belgium, tom.schrijvers@cs.kuleuven.be.

1559-1131/2010/3-ART39 $15.00
https://doi.org/0000001.0000001

A key concept in gradual type systems is *consistency* [Siek and Taha 2006]. Consistency weakens type equality to allow for the presence of *unknown* types $\star$. In some gradual type systems with subtyping, consistency is combined with subtyping to give rise to the notion of *consistent subtyping* [Siek and Taha 2007]. 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 [2007] 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. [2016], 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 [2007]. One important case where such a mismatch happens is in type systems supporting implicit (higher-rank) polymorphism [Dunfield and Krishnaswami 2013; Odersky and Läufer 1996; Peyton Jones et al. 2007]. It is well-known that polymorphic types à la System F induce a subtyping relation that relates polymorphic types to their instantiations [Mitchell 1990; Odersky and Läufer 1996]. However Siek and Taha's definition is not adequate for this kind of subtyping. Moreover the current framework for *Abstracting Gradual Typing* (AGT) [Garcia et al. 2016] does not account for polymorphism either, but the authors acknowledge that it is an interesting avenue for future work.

Existing work on gradual type systems with polymorphism does not use consistent subtyping. The Polymorphic Blame Calculus (λB) [Ahmed et al. 2011, 2017] is an *explicitly* polymorphic calculus with explicit casts, which is often used as a target language for gradual type systems with polymorphism. In λB a notion of *compatibility* is employed to validate conversions allowed by casts. Interestingly λB *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 λB 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 [Dunfield and Krishnaswami 2013; Odersky and Läufer 1996; Peyton Jones et al. 2007] $\forall a.\ a \to a$ and $\mathsf{Int} \to \mathsf{Int}$ are deemed *compatible*, since the latter type is an instantiation of the former. Therefore λB 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. [2017]. Like λB their calculus is explicitly polymorphic. However, 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 λB. Nonetheless, there is still some flavor 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 $\star \to \mathsf{Int}$, where some sort of (implicit) polymorphic subtyping is involved.

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. Our 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, and does not require the *restriction* operator of Siek and Taha [2007]. 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 the design of GPC, which stands for **G**radually **P**olymorphic **C**alculus: 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 the one hand, 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 are several existing works on integrating *explicit* polymorphism into gradual typing [Ahmed et al. 2011; Igarashi et al. 2017]. Yet no work investigates how to move its expressive power into a source language with 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 polymorphism into a gradually typed source language. We prove that our calculus satisfies all of the *static* aspects of the refined criteria for gradual typing [Siek et al. 2015].

As a step towards the *dynamic gradual guarantee* property [Siek et al. 2015], we propose an extension of our calculus. This extension employs *static type parameters*, which are placeholders for monotypes, and *gradual type parameters*, which are placeholders for monotypes that are consistent with the unknown type. The concept of static type parameters and gradual type parameters in the context of gradual typing was first introduced by Garcia and Cimini [2015], and later played a central role in the work of Igarashi et al. [2017][2]. With this extension it becomes possible to talk about *representative translations*: those translations that generalize a number of other translations using specific monotypes. Our work recasts the dynamic gradual guarantee in terms of representative translations. Assuming a coherence conjecture regarding representative translations, the dynamic gradual guarantee of our extended source language now can be reduced to that of λB, which, at the time of writing, is still an open question. Nonetheless, we believe our discussion of representative translations is helpful in shedding some light on this issue.

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, and can deal with polymorphism and top types, and
  - a syntax-directed version of consistent subtyping that is sound and complete with respect to our definition of consistent subtyping, but still guesses instantiations.

[1]Note here that we borrow Siek and Taha's motto mostly to talk about the static semantics. As Ahmed et al. [2011] show there are several non-trivial interactions between polymorphism and casts at the level of the dynamic semantics.
[2]The static and gradual type variables in their work.

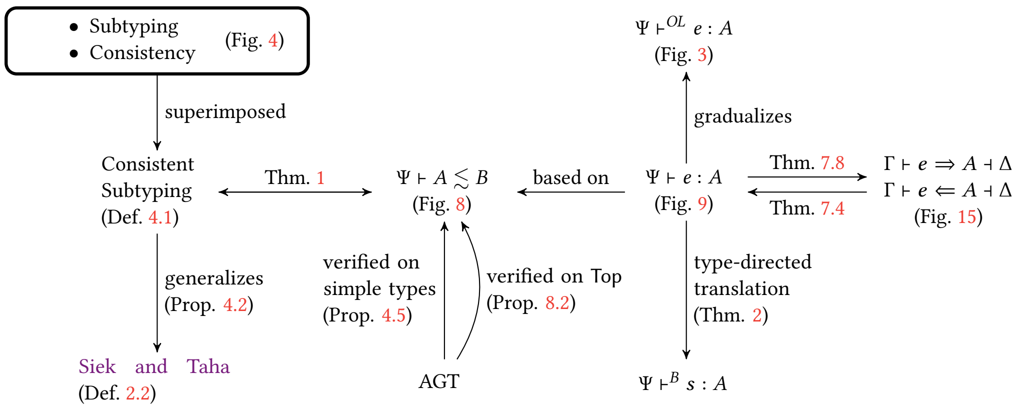


Fig. 1. Some key results in the paper.

- Based on consistent subtyping, we present GPC: 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 [Siek et al. 2015], and is type-safe by a type-directed translation to λB [Ahmed et al. 2011].
- We present a sound and complete bidirectional algorithm for implementing the declarative system based on the design principle of Garcia and Cimini [2015] and the approach of Dunfield and Krishnaswami [2013]. A Haskell implementation of the type checker is also available.
- We propose an extension of the type system with type parameters [Garcia and Cimini 2015] as a step towards restoring the *dynamic gradual guarantee* [Siek et al. 2015]. The extension significantly changes the algorithmic system. The new algorithm features a novel use of existential variables with a different solution space, which is a natural extension of the approach by Dunfield and Krishnaswami [2013].
- All of the metatheory of this paper, except some manual proofs for the algorithmic type system and extensions, has been mechanically formalized in Coq.

Figure 1 summarizes some key results in the paper. The reader is advised to refer back to this figure while reading the rest of the paper, as what it depicts will gradually come to make sense.

This article is a significantly expanded version of a conference paper [Xie et al. 2018]. Besides many improvements and expansions of the conference paper materials, there are several novel extensions. Firstly, we have added **let** expressions to our gradually typed calculus. Therefore it can now be seen as a complete gradual version of the implicitly polymorphic lambda calculus by Odersky and Läufer [1996]. Secondly, we have significantly expanded the discussion of applications of the calculus. In particular we now show how we can model algebraic datatypes, and how the combination of gradual typing and polymorphism enables simple (gradual) formulations of heterogeneous data structures [Kiselyov et al. 2004; McBride 2002]. Thirdly, we provide an extensive discussion of a variant of the calculus with a subsumption rule and prove its soundness and completeness. Fourthly, we study an extension of the calculus with type parameters and discuss the *dynamic gradual guarantee* [Siek et al. 2015] in detail. Furthermore we formalize the notion of *coherence up to cast errors* in Section 5. We also include detailed proofs on decidability of the algorithmic system. Finally, we provide an implementation of our type checking algorithm.

All supplementary materials, including Coq mechanization, manual proofs, and the Haskell implementation of the algorithm, are available at https://github.com/xnning/Consistent-Subtyping-for-All.

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

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

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

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

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

Fig. 2. Subtyping and type consistency in $\mathbf{FOb}^{?}_{<:}$

2 BACKGROUND

In this section we review a simple gradually typed language with objects [Siek and Taha 2007], to introduce the concept of consistent subtyping. We also briefly talk about the Odersky and Läufer type system for higher-rank types [Odersky and Läufer 1996], 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 [2007] developed a gradual type 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 defined in Fig. 2.

A primary contribution of their work is to show that consistency and subtyping are orthogonal. However, the orthogonality of consistency and subtyping does not lead to a deterministic relation. Thus Siek and Taha defined *consistent subtyping* (written $\lesssim$) based on a *restriction operator*, written $A|_B$ that "masks off" the parts of type A that are unknown in type B. For example, $\mathsf{Int} \to \mathsf{Int}|_{\mathsf{Bool} \to \star} = \mathsf{Int} \to \star$, and $\mathsf{Bool} \to \star|_{\mathsf{Int} \to \mathsf{Int}} = \mathsf{Bool} \to \star$. The definition of the restriction operator is given below:

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

With the restriction operator, consistent subtyping is simply defined as:

Definition 2.1 (Algorithmic Consistent Subtyping of Siek and Taha [2007]). $A \lesssim B \equiv A|_B <: B|_A$.

Later they show a proposition that consistent subtyping is equivalent to two declarative definitions, which we refer to as *declarative* consistent subtyping because it servers as a good guideline

Types	A, B	::=	$\mathsf{Int} \mid a \mid A \to B \mid \forall a.\, A$
Monotypes	τ, σ	::=	$\mathsf{Int} \mid a \mid \tau \to \sigma$
Terms	e	::=	$x \mid n \mid \lambda x : A.\, e \mid \lambda x.\, e \mid e_1\, e_2 \mid \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2$
Contexts	Ψ	::=	$\bullet \mid \Psi, x : A \mid \Psi, a$

$\boxed{\Psi \vdash^{OL} e : A}$ *(Typing)*

$$\frac{(x : A) \in \Psi}{\Psi \vdash^{OL} x : A}\ \textsc{u-var} \qquad \frac{}{\Psi \vdash^{OL} n : \mathsf{Int}}\ \textsc{u-int} \qquad \frac{\Psi, x : A \vdash^{OL} e : B}{\Psi \vdash^{OL} \lambda x : A.\, e : A \to B}\ \textsc{u-lamann}$$

$$\frac{\Psi, x : \tau \vdash^{OL} e : B}{\Psi \vdash^{OL} \lambda x.\, e : \tau \to B}\ \textsc{u-lam} \qquad \frac{\Psi \vdash^{OL} e_1 : A_1 \to A_2 \qquad \Psi \vdash^{OL} e_2 : A_1}{\Psi \vdash^{OL} e_1\, e_2 : A_2}\ \textsc{u-app}$$

$$\frac{\Psi \vdash^{OL} e : A_1 \qquad \Psi \vdash A_1 <: A_2}{\Psi \vdash^{OL} e : A_2}\ \textsc{u-sub} \qquad \frac{\Psi, a \vdash^{OL} e : A}{\Psi \vdash^{OL} e : \forall a.\, A}\ \textsc{u-gen}$$

$$\frac{\Psi \vdash^{OL} e_1 : A \qquad \Psi, x : A \vdash^{OL} e_2 : B}{\Psi \vdash^{OL} \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 : B}\ \textsc{u-let}$$

$\boxed{\Psi \vdash A <: B}$ *(Subtyping)*

$$\frac{a \in \Psi}{\Psi \vdash a <: a}\ \textsc{s-tvar} \qquad \frac{}{\Psi \vdash \mathsf{Int} <: \mathsf{Int}}\ \textsc{s-int} \qquad \frac{\Psi \vdash B_1 <: A_1 \qquad \Psi \vdash A_2 <: B_2}{\Psi \vdash A_1 \to A_2 <: B_1 \to B_2}\ \textsc{s-arrow}$$

$$\frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] <: B}{\Psi \vdash \forall a.\, A <: B}\ \textsc{s-forallL} \qquad \frac{\Psi, a \vdash A <: B}{\Psi \vdash A <: \forall a.\, B}\ \textsc{s-forallR}$$

Fig. 3. Syntax and static semantics of the Odersky-Läufer type system.

on superimposing consistency and subtyping. Both definitions are non-deterministic because of the intermediate type C.

Definition 2.2 (Declarative Consistent Subtyping of Siek and Taha [2007]). The following two are equivalent:

(1) $A \lesssim B$ if and only if $A \sim C$ and $C <: B$ for some C.
(2) $A \lesssim B$ if and only if $A <: C$ and $C \sim B$ for some C.

In our later discussion, it will always be clear which definition we are referring to. For example, we focus more on Definition 2.2 in Section 4.2, and more on Definition 2.1 in Section 4.5.

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 [1996].

The syntax of the type system, along with the typing and subtyping judgments is given in Fig. 3. An implicit assumption throughout the paper is that variables in contexts are distinct. Most typing rules are standard. The rule u-sub (the subsumption rule) allows us to convert the type A_1 of an expression e to some supertype A_2. The rule u-gen generalizes type variables to polymorphic types. These two rules can be applied anywhere. Most subtyping rules are standard as well. Rule s-forallL guesses a monotype τ to instantiate the type variable a, and the subtyping relation holds if the the instantiated type $A[a \mapsto \tau]$ is a subtype of B. In rule S-FORALLR, the type variable a is put into the typing context and subtyping continues with A and B. We save additional explanation about the static semantics for Section 5, where we present our gradually typed version of the calculus.

3 MOTIVATION AND APPLICATIONS

In this section we motivate why the combination of gradual typing and implicit polymorphism is useful. We then illustrate two concrete applications related to algebraic datatypes. The first application shows how gradual typing helps in defining Scott encodings of algebraic datatypes [Curry et al. 1958; Parigot 1992], which are impossible to encode in plain System F. The second application shows how gradual typing makes it easy to model and use heterogeneous containers.

3.1 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 [Garcia et al. 2016].

Haskell is a language renowned for its advanced type system, but it does not feature gradual typing. Of particular interest to us is its support for implicit higher-rank polymorphism, which is supported via explicit type annotations. In Haskell 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. [2007]:

```
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 [Damas and Milner 1982; Hindley 1969] 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] → [Int]`, or `[Char] → [Char]`, but not `∀a. [a] → [a]`. Finding such manual polymorphic annotations can be non-trivial, especially when the program scales up and the annotation is long and complicated.

Instead of rejecting the program outright, due to missing type annotations, gradual typing provides a simple alternative by giving `x` the unknown type $\star$. With this type the same program type-checks and produces `([2, 1], ['b', 'a'])`. By running the program, programmers can gain more insight about its run-time behaviour. Then, with this insight, they can also give `x` a more precise type (`∀a. [a] → [a]`) a 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.

3.2 Application: Efficient (Partly) Typed Encodings of ADTs

Our calculus does not provide built-in support for algebraic datatypes (ADTs). Nevertheless, the calculus is expressive enough to support efficient function-based encodings of (optionally polymorphic) ADTs[3]. This offers an immediate way to model algebraic datatypes in our calculus without requiring extensions to our calculus or, more importantly, to its target—the polymorphic blame calculus. While we believe that such extensions are possible, they would likely require non-trivial extensions to the polymorphic blame calculus. Thus the alternative of being able to model algebraic datatypes without extending λB is appealing. The encoding also paves the way to provide built-in

[3]In a type system with impure features, such as non-termination or exceptions, the encoded types can have valid inhabitants with side-effects. However, that is beyond the scope of this paper.

support for algebraic datatypes in the source language, while elaborating them via the encoding into λB.

Church and Scott Encodings. It is well-known that polymorphic calculi such as System F can encode datatypes via Church encodings. However these encodings have well-known drawbacks. In particular, some operations are hard to define, and they can have a time complexity that is greater than that of the corresponding functions for built-in algebraic datatypes. A good example is the definition of the predecessor function for Church numerals [Church 1941]. Its definition requires significant ingenuity (while it is trivial with built-in algebraic datatypes), and it has *linear* time complexity (versus the *constant* time complexity for a definition using built-in algebraic datatypes).

An alternative to Church encodings are the so-called Scott encodings [Curry et al. 1958]. They address the two drawbacks of Church encodings: they allow simple definitions that directly correspond to programs implemented with built-in algebraic datatypes, and those definitions have the same time complexity to programs using algebraic datatypes.

Unfortunately, Scott encodings, or more precisely, their typed variant [Parigot 1992], cannot be expressed in System F: in the general case they require recursive types, which System F does not support. However, with gradual typing, we can remove the need for recursive types, thus enabling Scott encodings in our calculus.

A Scott Encoding of Parametric Lists. Consider for instance the typed Scott encoding of parametric lists in a system with implicit polymorphism:

$$\begin{aligned} \mathsf{List}\ a &\triangleq \mu L.\, \forall b.\, b \to (a \to L \to b) \to b \\ \mathsf{nil} &\triangleq \mathbf{fold}_{\mathsf{List}\ a}\, (\lambda n.\, \lambda c.\, n) : \forall a.\, \mathsf{List}\ a \\ \mathsf{cons} &\triangleq \lambda x.\, \lambda xs.\, \mathbf{fold}_{\mathsf{List}\ a}\, (\lambda n.\, \lambda c.\, c\, x\, xs) : \forall a.\, a \to \mathsf{List}\ a \to \mathsf{List}\ a \end{aligned}$$

This encoding requires both polymorphic and recursive types[4]. Like System F, our calculus only supports the former, but not the latter. Nevertheless, gradual types still allow us to use the Scott encoding in a partially typed fashion. The trick is to omit the recursive type binder μL and replace the recursive occurrence of L by the unknown type $\star$:

$$\mathsf{List}_\star\ a \triangleq \forall b.\, b \to (a \to \star \to b) \to b$$

As a consequence, we need to replace the term-level witnesses of the iso-recursion by explicit type annotations to respectively forget or recover the type structure of the recursive occurrences:

$$\begin{aligned} \mathbf{fold}_{\mathsf{List}_\star\ a} &\triangleq \lambda x.\, x : (\forall b.\, b \to (a \to \mathsf{List}_\star\ a \to b) \to b) \to \mathsf{List}_\star\ a \\ \mathbf{unfold}_{\mathsf{List}_\star\ a} &\triangleq \lambda x.\, x : \mathsf{List}_\star\ a \to (\forall b.\, b \to (a \to \mathsf{List}_\star\ a \to b) \to b) \end{aligned}$$

With the reinterpretation of **fold** and **unfold** as functions instead of built-in primitives, we have exactly the same definitions of $\mathsf{nil}_\star$ and $\mathsf{cons}_\star$.

Note that when we elaborate our calculus into the polymorphic blame calculus, the above type annotations give rise to explicit casts. For instance, after elaboration $\mathbf{fold}_{\mathsf{List}_\star\ a}\ e$ results in the cast $\langle(\forall b.\, b \to (a \to \mathsf{List}_\star\ a \to b) \to b) \hookrightarrow \mathsf{List}_\star\ a\rangle s$ where s is the elaboration of e.

In order to perform recursive traversals on lists, e.g., to compute their length, we need a fixpoint combinator like the Y combinator. Unfortunately, this combinator cannot be assigned a type in the simply typed lambda calculus or System F. Yet, we can still provide a gradual typing for it in our system.

$$\mathsf{fix} \triangleq \lambda f.\, (\lambda x : \star.\, f\, (x\, x))\, (\lambda x : \star.\, f\, (x\, x)) : \forall a.\, (a \to a) \to a$$

[4]Here we use iso-recursive types, but equi-recursive types can be used too.

This allows us for instance to compute the length of a list.

$$\mathsf{length} \triangleq \mathsf{fix}\,(\lambda len.\,\lambda l.\,\mathsf{zero}_\star\,(\lambda xs.\,\mathsf{succ}_\star\,(len\,xs)))$$

Here $\mathsf{zero}_\star : \mathsf{Nat}_\star$ and $\mathsf{succ}_\star : \mathsf{Nat}_\star \to \mathsf{Nat}_\star$ are the encodings of the constructors for natural numbers $\mathsf{Nat}_\star$. In practice, for performance reasons, we could extend our language with a **letrec** construct in a standard way to support general recursion, instead of defining a fixpoint combinator.

Observe that the gradual typing of lists still enforces that all elements in the list are of the same type. For instance, a heterogeneous list like $\mathsf{cons}_\star\,\mathsf{zero}_\star\,(\mathsf{cons}_\star\,\mathsf{true}_\star\,\mathsf{nil}_\star)$, is rejected because $\mathsf{zero}_\star : \mathsf{Nat}_\star$ and $\mathsf{true}_\star : \mathsf{Bool}_\star$ have different types.

Heterogeneous Containers. Heterogeneous containers are datatypes that can store data of different types, which is very useful in various scenarios. One typical application is that an XML element is heterogeneously typed. Moreover, the result of a SQL query contains heterogeneous rows.

In statically typed languages, there are several ways to obtain heterogeneous lists. For example, in Haskell, one option is to use *dynamic types*. Haskell's library **Data.Dynamic** provides the special type **Dynamic** along with its injection **toDyn** and projection **fromDyn**. The drawback is that the code is littered with **toDyn** and **fromDyn**, which obscures the program logic. One can also use the HList library [Kiselyov et al. 2004], which provides strongly typed data structures for heterogeneous collections. The library requires several Haskell extensions, such as multi-parameter classes [Jones et al. 1997] and functional dependencies [Jones 2000]. With fake dependent types [McBride 2002], heterogeneous vectors are also possible with type-level constructors.

In our type system, with explicit type annotations that set the element types to the unknown type we can disable the homogeneous typing discipline for the elements and get gradually typed heterogeneous lists[5]. Such gradually typed heterogeneous lists are akin to Haskell's approach with Dynamic types, but much more convenient to use since no injections and projections are needed, and the $\star$ type is built-in and natural to use.

An example of such gradually typed heterogeneous collections is:

$$l \triangleq \mathsf{cons}_\star\,(\mathsf{zero}_\star : \star)\,(\mathsf{cons}_\star\,(\mathsf{true}_\star : \star)\,\mathsf{nil}_\star)$$

Here we annotate each element with type annotation $\star$ and the type system is happy to type-check that $l : \mathsf{List}_\star\,\star$. Note that we are being meticulous about the syntax, but with proper implementation of the source language, we could write more succinct programs akin to Haskell's syntax, such as `[0, True]`.

4 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 in our new definition of consistent subtyping, and justify the new definition by demonstrating its equivalence with that of Siek and Taha [2007] and the AGT approach [Garcia et al. 2016] on simple types.

The syntax of types is given at the top of Fig. 4. 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 in examples. 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.

[5]This argument is based on the extended type system in Section 9.

$$
\begin{array}{llcl}
\text{Types} & A, B & ::= & \mathsf{Int} \mid a \mid A \to B \mid \forall a.\, A \mid \star \\
\text{Monotypes} & \tau, \sigma & ::= & \mathsf{Int} \mid a \mid \tau \to \sigma \\
\text{Contexts} & \Psi & ::= & \bullet \mid \Psi, x : A \mid \Psi, a
\end{array}
$$

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

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

$\boxed{\Psi \vdash A <: B}$ *(Subtyping)*

$$
\frac{a \in \Psi}{\Psi \vdash a <: a}\ \textsc{s-tvar} \qquad \frac{}{\Psi \vdash \mathsf{Int} <: \mathsf{Int}}\ \textsc{s-int} \qquad \frac{\Psi \vdash B_1 <: A_1 \qquad \Psi \vdash A_2 <: B_2}{\Psi \vdash A_1 \to A_2 <: B_1 \to B_2}\ \textsc{s-arrow}
$$

$$
\frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] <: B}{\Psi \vdash \forall a.\, A <: B}\ \textsc{s-forallL} \qquad \frac{\Psi, a \vdash A <: B}{\Psi \vdash A <: \forall a.\, B}\ \textsc{s-forallR} \qquad \frac{}{\Psi \vdash \star <: \star}\ \textsc{s-unknown}
$$

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

$$
\frac{}{\Psi \vdash \mathsf{Int}} \qquad \frac{}{\Psi \vdash \star} \qquad \frac{a \in \Psi}{\Psi \vdash a} \qquad \frac{\Psi \vdash A \qquad \Psi \vdash B}{\Psi \vdash A \to B} \qquad \frac{\Psi, a \vdash A}{\Psi \vdash \forall a.\, A}
$$

Fig. 4. Syntax of types, consistency, subtyping, well-formedness of types in the declarative system.

4.1 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. [2011] and type consistency by Igarashi et al. [2017].

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. In other words, consistency is an equivalence relation lifted from static types to gradual types [Garcia et al. 2016]. Following this observation, we naturally extend the definition from Fig. 2 with polymorphic types, as shown in the middle of Fig. 4. 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 [1996] in Fig. 3, the only addition is the neutral subtyping of $\star$. Notice that, in rule $\textsc{s-forallL}$, the universal quantifier is only allowed to be instantiated with a *monotype*. The judgment $\Psi \vdash A$ checks whether all the type variables in A are bound in the context Ψ. According to the syntax in Fig. 4, monotypes do not include 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 know, $\forall a.\, a \to a$ is definitely not compatible with $\mathsf{Int} \to \mathsf{Bool}$. Indeed, 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 *conservative extension* property that will be made precise in Section 5.3.)

Importantly there is a subtle distinction between a type variable and the unknown type, although they both 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 λB [Ahmed et al. 2011], 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, λB allows $\forall a.\, a \to a \prec \mathsf{Int} \to \mathsf{Bool}$, which as we discussed before might not be what we expect. Indeed λB relies on sophisticated runtime checks to rule out such instances of the compatibility relation a posteriori.

Igarashi et al. [2017] 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[6]:

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

Compared with our consistency definition in Fig. 4, 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 of 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 $\forall a.\, a \to \mathsf{Int}$ to be related with $\star \to \mathsf{Int}$ for example, which in our view, is a kind of (implicit) polymorphic subtyping combined with type consistency, 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 and *orthogonal*, and can be naturally superimposed, echoing the opinion of Siek and Taha [2007].

4.2 Towards Consistent Subtyping

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

Unfortunately, the original declarative definition of Siek and Taha (Definition 2.2) 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. 5a. We use $\emptyset$ to mean that we cannot find such a type. Similarly, there are situations where the first type

[6]This is a simplified version. These two rules are presented in Section 3.1 in their paper as one of the key ideas of the design of type consistency, which are later amended with *labels*.

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

(a)

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

(b)

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

(c)

Fig. 5. Examples that break the original definition of consistent subtyping.

can only reach the second type by the other way (first applying subtyping, and then consistency), as shown in Fig. 5b.

What is worse, if those two examples are composed in a way that those types all appear covariantly, then the resulting types cannot reach each other in either way. For example, Fig. 5c shows two such 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 a 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.

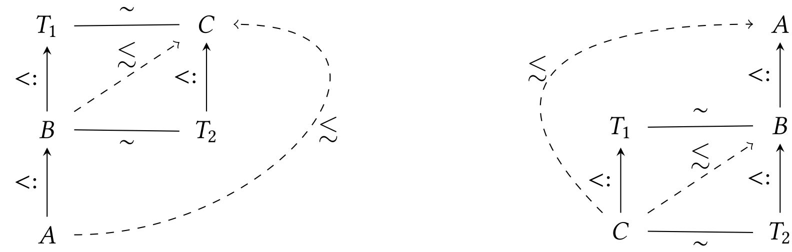
Observation 1. *If $A <: B$, and $B \lesssim C$, then $A \lesssim C$.*

This is reflected in Fig. 6a. A symmetrical observation is given in Fig. 6b:

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

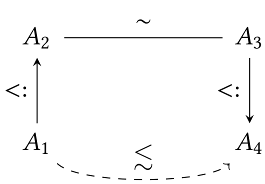
From the above observations, we see what the problem is with the original definition. In Fig. 6a, if B can reach C by T_1, then by subtyping transitivity, A can reach C by T_1. However, if B can only reach C by T_2, then A cannot reach C through the original definition. A similar problem is shown in Fig. 6b.

It turns out that these two problems can be fixed using 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, $A <: B \sim T_2 <: C$



(a) (b)

Fig. 6. Observations of consistent subtyping



$$
\begin{aligned}
A_1 &= (((\forall a.a \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\forall a.a)\\
A_2 &= ((\forall a.a \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\mathsf{Int} \to \mathsf{Int})\\
A_3 &= ((\forall a.\star \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\mathsf{Int} \to \star)\\
A_4 &= (((\star \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\mathsf{Int} \to \star)
\end{aligned}
$$

Fig. 7. Example that is fixed by the new definition of consistent subtyping.

(in Fig. 6a) and $C <: T_1 \sim B <: A$ (in Fig. 6b) have the same relation chain: subtyping, consistency, and subtyping.

Definition of Consistent subtyping. From the above discussion, we are ready to modify Definition 2.2, and adapt it to our notation[7]:

Definition 4.1 (Consistent Subtyping). $\Psi \vdash A \lesssim B$ if and only if $\Psi \vdash A <: A'$, $A' \sim B'$ and $\Psi \vdash B' <: B$ for some A' and B'.

With Definition 4.1, Figure 7 illustrates the correct relation chain for the broken example shown in Fig. 5c.

At first sight, Definition 4.1 seems worse than the original: we need to guess *two* types! It turns out that Definition 4.1 is a generalization of Definition 2.2, and they are equivalent in the system of Siek and Taha [2007]. However, more generally, Definition 4.1 is compatible with polymorphic types.

Proposition 4.2 (Generalization of Declarative Consistent Subtyping).

- *Definition 4.1 subsumes Definition 2.2.*
  In Definition 4.1, 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 2.2 is equivalent to Definition 4.1 in the system of Siek and Taha.*
  If $A <: C$, $C \sim D$, and $D <: B$, by Definition 2.2, $A \sim C'$, $C' <: D$ for some C'. By subtyping transitivity, $C' <: B$. So $A \lesssim B$ by $A \sim C'$ and $C' <: B$.

[7] For readers who are familiar with category theory, this defines consistent subtyping as the least subtyping bimodule extending consistency.

4.3 Abstracting Gradual Typing

Garcia et al. [2016] 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 is defined as follows:

Definition 4.3 (Concretization).

$$\gamma(\mathsf{Int}) = \{\mathsf{Int}\} \qquad \gamma(A \to B) = \{A' \to B' \mid A' \in \gamma(A), B' \in \gamma(B)\} \qquad \gamma(\star) = \{\text{All static types}\}$$

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

Definition 4.4 (Consistent Subtyping in AGT). $A \mathrel{\widetilde{<:}} B$ if and only if $A_1 <: B_1$ for some *static types* A_1 and B_1 such that $A_1 \in \gamma(A)$ and $B_1 \in \gamma(B)$.

Later they proved that this definition of consistent subtyping coincides with that of Siek and Taha [2007] (Definition 2.2). By Proposition 4.2, we can directly conclude that our definition coincides with AGT:

Corollary 4.5 (Equivalence to AGT on Simple Types). *$A \lesssim B$ if and only if $A \mathrel{\widetilde{<:}} B$.*

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. [2016], 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 [2017] in a gradual type system with union and intersection types, where the static types A_1, B_1 in Definition 4.4 can be algorithmically computed by also accounting for top and bottom types.

4.4 Directed Consistency

Directed consistency [Jafery and Dunfield 2017] is defined in terms of precision and 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".[8] In their setting, precision is first defined for type constructors and then lifted to gradual types, and subtyping is defined for gradual types. If we interpret this definition from the AGT point of view, finding a more precise static type has the same effect as concretization. Namely, $A' \sqsubseteq A$ implies $A \in \gamma(A')$ and $B' \sqsubseteq B$ implies $B \in \gamma(B')$ if A and B are static types. Therefore we consider this definition as AGT-style. From this perspective, this definition naturally coincides with Definition 4.4, and by Corollary 4.5, it coincides with Definition 4.1.

The value of their definition is that consistent subtyping is derived compositionally from *gradual subtyping* and *precision*. Arguably, gradual types play a role in both definitions, which is different from Definition 4.1 where subtyping is neutral to unknown types. Still, the definition is interesting as it takes precision into consideration, rather than consistency. Then a question arises as to *how are consistency and precision related.*

[8] Jafery and Dunfield actually read $A \sqsubseteq$ B as "A is *more precise* than B". We, however, use the "less precise" interpretation (which is also adopted by Cimini and Siek [2016]) throughout the paper. The full rules can be found in Fig. 10.

Consistency and Precision. Precision is a partial order (anti-symmetric and transitive), while consistency is symmetric but not transitive. Recall that consistency is in fact an equivalence relation lifted from static types to gradual types [Garcia et al. 2016], which embodies the key role of gradual types in typing. 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 [Siek et al. 2015], and finding a correct partial order that adheres to the criteria is not always an easy task. For example, Igarashi et al. [2017] argued that term precision for gradual System F 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.

Nonetheless, in our system, precision and consistency can be related by the following lemma [9]:

$\mathcal{L}$EMMA 1 (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 $A \sim B$.*

4.5 Consistent Subtyping Without Existentials

Definition 4.1 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 Definition 2.1, 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 Definition 2.1 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 $\mathsf{Int} \to \star \to \mathsf{Bool} \lesssim \mathsf{Int} \to \mathsf{Int} \to \star$.

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

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*

[9] Lemmas with $\mathcal{L}$ are those proved in Coq. The same applies to $\mathcal{T}$heorems.

$\boxed{\Psi \vdash A \lesssim B}$ *(Consistent Subtyping)*

$$\frac{a \in \Psi}{\Psi \vdash a \lesssim a}\ \textsc{cs-tvar} \qquad \frac{}{\Psi \vdash \mathsf{Int} \lesssim \mathsf{Int}}\ \textsc{cs-int} \qquad \frac{\Psi \vdash B_1 \lesssim A_1 \qquad \Psi \vdash A_2 \lesssim B_2}{\Psi \vdash A_1 \to A_2 \lesssim B_1 \to B_2}\ \textsc{cs-arrow}$$

$$\frac{\Psi, a \vdash A \lesssim B}{\Psi \vdash A \lesssim \forall a.\, B}\ \textsc{cs-forallR} \qquad \frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] \lesssim B}{\Psi \vdash \forall a.\, A \lesssim B}\ \textsc{cs-forallL} \qquad \frac{}{\Psi \vdash \star \lesssim A}\ \textsc{cs-unknownL}$$

$$\frac{}{\Psi \vdash A \lesssim \star}\ \textsc{cs-unknownR}$$

Fig. 8. Consistent Subtyping for implicit polymorphism.

arbitrary type. We can interpret this observation directly from Definition 4.1: 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 for lifting 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. 4, get rid of rule S-UNKNOWN and add two extra rules concerning $\star$, resulting in the rules of consistent subtyping in Fig. 8. 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. 8. What is more, we can prove that the two definitions are equivalent.

Theorem 1. *$\Psi \vdash A \lesssim B \Leftrightarrow \Psi \vdash A <: A', A' \sim B', \Psi \vdash B' <: B$ for some A', B'.*

5 GRADUALLY TYPED IMPLICIT POLYMORPHISM

In Section 4 we introduced our consistent subtyping relation that accommodates polymorphic types. In this section we continue with the development by giving a 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:

Expressions	e	$::=$	$x \mid n \mid \lambda x : A.\, e \mid \lambda x.\, e \mid e_1\, e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$

Meta-variable e ranges over expressions. Expressions include variables x, integers n, annotated lambda abstractions $\lambda x : A.\, e$, un-annotated lambda abstractions $\lambda x.\, e$, applications $e_1\, e_2$, and let expressions $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$.

5.1 Typing in Detail

Figure 9 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 INT 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 explicit annotations. Rule GEN puts a fresh type variable a into the type context and generalizes the typing result A to

$\boxed{\Psi \vdash e : A \rightsquigarrow s}$ *(Typing)*

$$\frac{(x : A) \in \Psi}{\Psi \vdash x : A \rightsquigarrow x}\ \text{VAR} \qquad \frac{}{\Psi \vdash n : \mathsf{Int} \rightsquigarrow n}\ \text{INT} \qquad \frac{\Psi, a \vdash e : A \rightsquigarrow s}{\Psi \vdash e : \forall a. A \rightsquigarrow \Lambda a. s}\ \text{GEN}$$

$$\frac{\Psi, x : A \vdash e : B \rightsquigarrow s}{\Psi \vdash \lambda x : A.\, e : A \to B \rightsquigarrow \lambda x : A.\, s}\ \text{LAMANN} \qquad \frac{\Psi, x : \tau \vdash e : B \rightsquigarrow s}{\Psi \vdash \lambda x.\, e : \tau \to B \rightsquigarrow \lambda x : \tau.\, s}\ \text{LAM}$$

$$\frac{\Psi \vdash e_1 : A \rightsquigarrow s_1 \qquad \Psi, x : A \vdash e_2 : B \rightsquigarrow s_2}{\Psi \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : B \rightsquigarrow (\lambda x : A.\, s_2)\, s_1}\ \text{LET}$$

$$\frac{\Psi \vdash e_1 : A \rightsquigarrow s_1 \qquad \Psi \vdash A \triangleright A_1 \to A_2 \qquad \Psi \vdash e_2 : A_3 \rightsquigarrow s_2 \qquad \Psi \vdash A_3 \lesssim A_1}{\Psi \vdash e_1\, e_2 : A_2 \rightsquigarrow (\langle A \hookrightarrow A_1 \to A_2 \rangle s_1)(\langle A_3 \hookrightarrow A_1 \rangle s_2)}\ \text{APP}$$

$\boxed{\Psi \vdash A \triangleright A_1 \to A_2}$ *(Matching)*

$$\frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] \triangleright A_1 \to A_2}{\Psi \vdash \forall a. A \triangleright A_1 \to A_2}\ \text{M-FORALL} \qquad \frac{}{\Psi \vdash A_1 \to A_2 \triangleright A_1 \to A_2}\ \text{M-ARR}$$

$$\frac{}{\Psi \vdash \star \triangleright \star \to \star}\ \text{M-UNKNOWN}$$

Fig. 9. Declarative typing

$\forall a. A$. Rule LET infers the type A of e_1, then puts $x : A$ in the context to infer the type of e_2. 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. [2015] is extended to polymorphic types naturally, resulting in $\Psi \vdash A \triangleright A_1 \to A_2$. In rule M-FORALL, a monotype τ is guessed to instantiate the universal quantifier a. This rule is inspired by the *application judgment* $\Phi \vdash A \bullet e \implies C$ [Dunfield and Krishnaswami 2013], which says that if we apply a term of type A to an argument e, then we get a term of type C. If A is a polymorphic type, the judgment works by guessing instantiations until it reaches an arrow type. Matching further simplifies the application judgment, since it is independent of typing. Rules M-ARR and M-UNKNOWN are the same as Siek et al. [2015]. Rule M-ARR returns the domain type A_1 and range type A_2 as expected. If the input is $\star$, then rule M-UNKNOWN returns $\star$ as both the type for the domain and the range.

Note that matching saves us from having a subsumption rule (rule U-SUB in Fig. 3). The subsumption rule is incompatible with consistent subtyping, since the latter is not transitive. A discussion of a subsumption rule based on normal subtyping can be found in Section 8.2.

5.2 Type-directed Translation

We give the dynamic semantics of our language by translating it to λB [Ahmed et al. 2011]. Below we show a subset of the terms in λB that are used in the translation:

λB Terms	s	$::=$	$x \mid n \mid \lambda x : A.\, s \mid \Lambda a.\, s \mid s_1\, s_2 \mid \langle A \hookrightarrow B \rangle s$

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 4.1. The syntax of types in λB is the same as ours.

The translation is given in the gray-shaded parts in Fig. 9. 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 2 ($\triangleright$ to $\prec$). *If $\Psi \vdash A \triangleright A_1 \to A_2$, then $A \prec A_1 \to A_2$.*

Lemma 3 ($\lesssim$ 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 λB. By Lemmas 2 and 3, 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* [Reynolds 1983]. The parametricity property says that all instances of a polymorphic function should behave *uniformly*. 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. Our translation target λB is taken from Ahmed et al. [2011], where relational parametricity is enforced by dynamic sealing [Matthews and Ahmed 2008; Neis et al. 2009], but there is no rigorous proof. Later, Ahmed et al. [2017] imposed a syntactic restriction on terms of λB, where all type abstractions must have *values* as their body. With this invariant, they proved that the restricted λB satisfies relational parametricity. It remains to see if our translation process can be adjusted to target restricted λB. One possibility is to impose similar restriction to the rule GEN:

$$\frac{\Psi, a \vdash e : A \rightsquigarrow v}{\Psi \vdash e : \forall a.\, A \rightsquigarrow \Lambda a.\, v}\ \textsc{Gen2}$$

where we only generate type abstractions if the inner body is a value. However, the type system with this rule is a weaker calculus, which is not a conservative extension of the Odersky-Läufer type system.

Ambiguity from Casts. The translation does not always produce a unique target expression. This is because when guessing some monotype τ in rules M-FORALL and CS-FORALLL, we could have many choices, which inevitably leads to different types. This is usually not a problem for (non-gradual) System F-like systems [Dunfield and Krishnaswami 2013; Peyton Jones et al. 2007] because they adopt a type-erasure semantics [Pierce 2002]. However, in our case, the choice of monotypes may affect the runtime behaviour of translated programs, since they could appear inside the explicit casts. For instance, the following example shows two possible translations for the same source expression $(\lambda x : \star.\, f\ x) : \star \to \mathsf{Int}$, where the type of f is instantiated to $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Bool} \to \mathsf{Int}$, respectively:

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

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 λB, 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 the instantiation of a appears in 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.

The semantic discrepancy is due to the guessing nature of the *declarative* system. As far as the static semantics is concerned, both $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Bool} \to \mathsf{Int}$ are equally acceptable. But this is not the case at runtime. The astute reader may have found that the *only* appropriate choice is to instantiate the type of f to $\star \to \mathsf{Int}$ in the matching judgment. However, as specified by rule M-FORALL in Fig. 9, we can only instantiate type variables to monotypes, but $\star$ is *not* a monotype! We will get back to this issue in Section 9.

Coherence. 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 [Reynolds 1991]. 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, suppose f is defined as $(\lambda x.\ 1)$, then whatever the translation might be, applying $(\lambda x : \star.\, f\ x)$ to 3 either results in a cast error, or produces 1, nothing else.

We defined contextual equivalence [Morris Jr 1969] to formally characterize that two open expressions have the same behavior. The definition of contextual equivalence requires a notion of well-typed expression contexts C, written $C : (\Psi \vdash^B A) \rightsquigarrow (\Psi' \vdash^B A')$. The definitions of contexts and context typing are standard and thus omitted. As is common, we first define contextual approximation. In our setting, we need to relax the notion of contextual approximation of λB [Ahmed et al. 2017] to also take into consideration of cast errors. We write $\Psi \vdash s_1 \preceq_{ctx} s_2 : A$ to say that s_2 mimics the behaviour of s_1 at type A in the sense that whenever a program containing s_1 reduces to an integer, replacing it with s_2 either reduces to the same integer, or emits a cast error. We restrict the program results to integers to eliminate the role of types in values. If it is not an integer, it is always possible to embed it into another context that reduces to an integer. Then we write $\Psi \vdash s_1 \simeq_{ctx} s_2 : A$ to say s_1 and s_2 are contextually equivalent, that is, they approximate each other.

Definition 5.1 (Contextual Approximation and Equivalence up to Cast Errors).

$$\begin{array}{rcl}
\Psi \vdash s_1 \preceq_{ctx} s_2 : A & \triangleq & \Psi \vdash^B s_1 : A \wedge \Psi \vdash^B s_2 : A \wedge \\
& & \text{for all } C.\, C : (\Psi \vdash^B A) \rightsquigarrow (\bullet \vdash^B \mathsf{Int}) \Longrightarrow \\
& & \quad C\{s_1\} \Downarrow n \Longrightarrow (C\{s_2\} \Downarrow n \vee C\{s_2\} \Downarrow \mathsf{blame}) \\
\Psi \vdash s_1 \simeq_{ctx} s_2 : A & \triangleq & \Psi \vdash s_1 \preceq_{ctx} s_2 : A \wedge \Psi \vdash s_2 \preceq_{ctx} s_1 : A
\end{array}$$

Before presenting the formal definition of coherence, first we observe that after erasing types and casts, all translations of the same expression are exactly the same. This is easy to see by examining each elaboration rule. We use $\lfloor s \rfloor$ to denote an expression in λB after erasure.

LEMMA 5.2. *If $\Psi \vdash e : A \rightsquigarrow s_1$, and $\Psi \vdash e : A \rightsquigarrow s_2$, then $\lfloor s_1 \rfloor \equiv_\alpha \lfloor s_2 \rfloor$.*

Second, at runtime, the only role of types and casts is to emit cast errors caused by type mismatch. Therefore, By Lemma 5.2 coherence follows as a corollary:

LEMMA 5.3 (COHERENCE UP TO CAST ERRORS). *For any expression e such that $\Psi \vdash e : A \rightsquigarrow s_1$ and $\Psi \vdash e : A \rightsquigarrow s_2$, we have $\Psi \vdash s_1 \simeq_{ctx} s_2 : A$.*

$\boxed{A \sqsubseteq B}$ Type precision

$$\frac{}{\star \sqsubseteq A}\ \textsc{L-Unknown} \qquad \frac{}{\mathsf{Int} \sqsubseteq \mathsf{Int}}\ \textsc{L-Nat} \qquad \frac{A_1 \sqsubseteq B_1 \qquad A_2 \sqsubseteq B_2}{A_1 \rightarrow A_2 \sqsubseteq B_1 \rightarrow B_2}\ \textsc{L-Arrow} \qquad \frac{}{a \sqsubseteq a}\ \textsc{L-TVar}$$

$$\frac{A \sqsubseteq B}{\forall a.A \sqsubseteq \forall a.B}\ \textsc{L-Forall}$$

$\boxed{e_1 \sqsubseteq e_2}$ Term precision

$$\frac{}{e \sqsubseteq e}\ \textsc{L-Refl} \qquad \frac{A_1 \sqsubseteq A_2 \qquad e_1 \sqsubseteq e_2}{\lambda x : A_1.\ e_1 \sqsubseteq \lambda x : A_2.\ e_2}\ \textsc{L-LamAnn} \qquad \frac{e_1 \sqsubseteq e_3 \qquad e_2 \sqsubseteq e_4}{e_1\ e_2 \sqsubseteq e_3\ e_4}\ \textsc{L-App}$$

$\boxed{\Psi_1 \mid \Psi_2 \vdash e_1 \sqsubseteq^B e_2}$ Term less precision in λB

$$\frac{}{x \sqsubseteq^B x}\ \textsc{L-Var} \qquad \frac{}{n \sqsubseteq^B n}\ \textsc{L-Nat} \qquad \frac{A_1 \sqsubseteq A_2 \qquad e_1 \sqsubseteq^B e_2}{\lambda x : A_1.\ e_1 \sqsubseteq^B \lambda x : A_2.\ e_2}\ \textsc{L-LamAnn} \qquad \frac{e_1 \sqsubseteq^B e_2}{\Lambda a.e_1 \sqsubseteq^B \Lambda a.e_2}\ \textsc{L-LamAnn}$$

$$\frac{e_1 \sqsubseteq^B e_3 \qquad e_2 \sqsubseteq^B e_4}{e_1\ e_2 \sqsubseteq^B e_3\ e_4}\ \textsc{L-App} \qquad \frac{A_1 \sqsubseteq B_1 \qquad A_2 \sqsubseteq B_2 \qquad e_1 \sqsubseteq^B e_2}{\langle A_1 \hookrightarrow A_2 \rangle\ e_1 \sqsubseteq^B \langle B_1 \hookrightarrow B_2 \rangle\ e_2}\ \textsc{L-Cast}$$

Fig. 10. Less precision

5.3 Correctness Criteria

Siek et al. [2015] present a set of properties, the *refined criteria*, that a well-designed gradual typing calculus must have. Among all the criteria, those related to the static aspects of gradual typing are well summarized by Cimini and Siek [2016]. 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 4 (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 $\Psi, e_1, e_2, s_1, s_2, A$, if $\Psi \vdash e_1 : A \rightsquigarrow s_1$, and $\Psi \vdash e_2 : A \rightsquigarrow s_2$, and $e_1 \sqsubseteq e_2$, then $\Psi \mid \Psi \vdash s_1 \sqsubseteq^B s_2$.*

The first criterion states that the gradual type system should be a conservative extension of the original system. In other words, a *static* program 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 $\star$[10]. However since our gradual type system does not have the subsumption rule, it produces more general types.

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

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 Fig. 10. 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_1 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. 9. The last criterion ensures that the translation must be monotonic over the precision relation $\sqsubseteq$. Ahmed et al. [2011] does not include a formal definition of precision, but an *approximation* definition and a *simulation relation*. Here we adapt the simulation relation as the precision, and a subset of it that is used in our system is given at the bottom of Fig. 10.

The Dynamic Gradual Guarantee. Besides the static criteria, there is also a criterion concerning the dynamic semantics, known as *the dynamic gradual guarantee* [Siek et al. 2015].

Definition 5.4 (Dynamic Gradual Guarantee). Suppose $e' \sqsubseteq e$, and $\bullet \vdash e : A \rightsquigarrow s$ and $\bullet \vdash e' : A' \rightsquigarrow s'$,

- if $s \Downarrow v$, then $s' \Downarrow v'$ and $v' \sqsubseteq v$. If $s \Uparrow$ then $s' \Uparrow$.
- if $s' \Downarrow v'$, then $s \Downarrow v$ where $v' \sqsubseteq v$, or $s \Downarrow \mathsf{blame}$. If $s' \Uparrow$ then $s \Uparrow$ or $s \Downarrow \mathsf{blame}$.

The first part of the dynamic gradual guarantee says that if a gradually typed program evaluates to a value, then making type annotations less precise always produces a program that evaluates to an less precise value. Unfortunately, coherence up to cast errors in the declarative system breaks the dynamic gradual guarantee. For instance:

$$(\lambda f : \forall a.a \to \mathsf{Int}.\ \lambda x : \mathsf{Int}.\ f\ x)\ (\lambda x.\ 1)\ 3 \qquad (\lambda f : \forall a.a \to \mathsf{Int}.\ \lambda x : \star.\ f\ x)\ (\lambda x.\ 1)\ 3$$

The left one evaluates to 1, whereas its less precise version (right) will give a cast error if a is instantiated to Bool for example. In Section 9, we will present an extension of the declarative system that will alleviate the issue.

6 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 [2013] (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* [Garcia and Cimini 2015]. 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 appendix. We also provide an implementation, which can be found in the supplementary materials.[11]

Algorithmic Contexts. Figure 11 shows the syntax of the algorithmic system. A noticeable difference are the algorithmic contexts Γ, which are represented as 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 $\widehat{a} = \tau$). Finally, algorithmic contexts include a *marker* $\blacktriangleright_{\widehat{a}}$ (read

[11]Note that the proofs in the appendix and the implementation are for the extended system in Section 9, which subsumes the algorithmic system presented in this section.

Expressions	e	$::=$	$x \mid n \mid \lambda x : A.\, e \mid \lambda x.\, e \mid e_1\, e_2 \mid e : A \mid \textbf{let}\, x = e_1 \,\textbf{in}\, e_2$
Types	A, B	$::=$	$\mathsf{Int} \mid a \mid \widehat{a} \mid A \to B \mid \forall a.\, A \mid \star$
Monotypes	τ, σ	$::=$	$\mathsf{Int} \mid a \mid \widehat{a} \mid \tau \to \sigma$
Algorithmic Contexts	Γ, Δ, Θ	$::=$	$\bullet \mid \Gamma, x : A \mid \Gamma, a \mid \Gamma, \widehat{a} \mid \Gamma, \widehat{a} = \tau \mid \Gamma, \blacktriangleright_{\widehat{a}}$
Complete Contexts	Ω	$::=$	$\bullet \mid \Omega, x : A \mid \Omega, a \mid \Omega, \widehat{a} = \tau \mid \Omega, \blacktriangleright_{\widehat{a}}$

Fig. 11. Syntax of the algorithmic system

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

$$\frac{}{\Gamma \vdash \mathsf{Int}}\ \textsc{ad-int} \qquad \frac{}{\Gamma \vdash \star}\ \textsc{ad-unknown} \qquad \frac{}{\Gamma[a] \vdash a}\ \textsc{ad-tvar} \qquad \frac{}{\Gamma[\widehat{a}] \vdash \widehat{a}}\ \textsc{ad-evar}$$

$$\frac{}{\Gamma[\widehat{a} = \tau] \vdash \widehat{a}}\ \textsc{ad-solved} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B}\ \textsc{ad-arrow} \qquad \frac{\Gamma, a \vdash A}{\Gamma \vdash \forall a.\, A}\ \textsc{ad-forall}$$

$\boxed{\vdash \Gamma}$ *(Well-formedness of algorithmic contexts)*

$$\frac{}{\vdash \bullet}\ \textsc{wf-empty} \qquad \frac{\vdash \Gamma \qquad x \notin \textsc{fv}(\Gamma) \qquad \Gamma \vdash A}{\vdash \Gamma, x : A}\ \textsc{wf-var} \qquad \frac{\vdash \Gamma \qquad a \notin \textsc{fv}(\Gamma)}{\vdash \Gamma, a}\ \textsc{wf-tvar}$$

$$\frac{\vdash \Gamma \qquad \widehat{a} \notin \textsc{fv}(\Gamma)}{\vdash \Gamma, \widehat{a}}\ \textsc{wf-evar} \qquad \frac{\vdash \Gamma \qquad \widehat{a} \notin \textsc{fv}(\Gamma) \qquad \Gamma \vdash \tau}{\vdash \Gamma, \widehat{a} = \tau}\ \textsc{wf-solved} \qquad \frac{\vdash \Gamma \qquad \blacktriangleright_{\widehat{a}} \notin \textsc{fv}(\Gamma)}{\vdash \Gamma, \blacktriangleright_{\widehat{a}}}\ \textsc{wf-marker}$$

Fig. 12. Well-formedness of types and contexts in the algorithmic system

"marker $\widehat{a}$"), which is used to delineate existential variables created by the algorithm. We will have more to say about markers when we examine the rules. Complete contexts Ω are the same as contexts, except that they contain no unsolved variables.

Apart from expressions in the declarative system, we add annotated expressions $e : A$. The well-formedness judgments for types and contexts are shown in Fig. 12.

Notational convenience. Following DK system, we use contexts as substitutions on types. We write $[\Gamma]A$ to mean Γ applied as a substitution to type A. We also use a hole notation, which is useful when manipulating contexts by inserting and replacing declarations in the middle. The hole notation is used extensively in proving soundness and completeness. For example, $\Gamma[\Theta]$ means Γ has the form $\Gamma_L, \Theta, \Gamma_R$; if we have $\Gamma[\widehat{a}] = (\Gamma_L, \widehat{a}, \Gamma_R)$, then $\Gamma[\widehat{a} = \tau] = (\Gamma_L, \widehat{a} = \tau, \Gamma_R)$. Occasionally, we will see a context with two *ordered* holes, e.g., $\Gamma = \Gamma_0[\Theta_1][\Theta_2]$ means Γ has the form $\Gamma_L, \Theta_1, \Gamma_M, \Theta_2, \Gamma_R$.

Input and output contexts. The algorithmic system, compared with the declarative system, includes similar judgment forms, except that we replace the declarative context Ψ with an algorithmic context Γ (the *input context*), and add an *output context* Δ after a backward turnstile, e.g., $\Gamma \vdash A \lesssim B \dashv \Delta$ is the judgment form for the algorithmic consistent subtyping. All algorithmic rules manipulate input and output contexts in a way that is consistent with the notion of *context extension*, which will be described in Section 7.1.

We start with the explanation of the algorithmic consistent subtyping as it involves manipulating existential type variables explicitly (and solving them if possible).

$\boxed{\Gamma \vdash A \lesssim B \dashv \Delta}$ *(Under input context Γ, A is a consistent subtype of B, with output context Δ)*

$$\frac{}{\Gamma[a] \vdash a \lesssim a \dashv \Gamma[a]}\ \textsc{as-tvar} \qquad \frac{}{\Gamma \vdash \mathsf{Int} \lesssim \mathsf{Int} \dashv \Gamma}\ \textsc{as-int} \qquad \frac{}{\Gamma[\widehat{a}] \vdash \widehat{a} \lesssim \widehat{a} \dashv \Gamma[\widehat{a}]}\ \textsc{as-evar}$$

$$\frac{}{\Gamma \vdash \star \lesssim A \dashv \Gamma}\ \textsc{as-unknownL} \qquad \frac{}{\Gamma \vdash A \lesssim \star \dashv \Gamma}\ \textsc{as-unknownR}$$

$$\frac{\Gamma \vdash B_1 \lesssim A_1 \dashv \Theta \qquad \Theta \vdash [\Theta]A_2 \lesssim [\Theta]B_2 \dashv \Delta}{\Gamma \vdash A_1 \to A_2 \lesssim B_1 \to B_2 \dashv \Delta}\ \textsc{as-arrow} \qquad \frac{\Gamma, a \vdash A \lesssim B \dashv \Delta, a, \Theta}{\Gamma \vdash A \lesssim \forall a.\, B \dashv \Delta}\ \textsc{as-forallR}$$

$$\frac{\Gamma, \blacktriangleright_{\widehat{a}}, \widehat{a} \vdash A[a \mapsto \widehat{a}] \lesssim B \dashv \Delta, \blacktriangleright_{\widehat{a}}, \Theta}{\Gamma \vdash \forall a.\, A \lesssim B \dashv \Delta}\ \textsc{as-forallL} \qquad \frac{\widehat{a} \notin \textsc{fv}(A) \qquad \Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox A \dashv \Delta}{\Gamma[\widehat{a}] \vdash \widehat{a} \lesssim A \dashv \Delta}\ \textsc{as-instL}$$

$$\frac{\widehat{a} \notin \textsc{fv}(A) \qquad \Gamma[\widehat{a}] \vdash A \lessapprox \widehat{a} \dashv \Delta}{\Gamma[\widehat{a}] \vdash A \lesssim \widehat{a} \dashv \Delta}\ \textsc{as-instR}$$

Fig. 13. Algorithmic consistent subtyping

6.1 Algorithmic Consistent Subtyping

Figure 13 presents the rules of algorithmic consistent subtyping $\Gamma \vdash A \lesssim B \dashv \Delta$, which says that under input context Γ, A is a consistent subtype of B, with output context Δ. The first five rules do not manipulate contexts, but illustrate how contexts are propagated.

Rules AS-TVAR and AS-INT do not involve existential variables, so the output contexts remain unchanged. Rule AS-EVAR says that any unsolved existential variable is a consistent subtype of itself. The output is still the same as the input context as the rule gives no clue as to what is the solution of that existential variable. Rules AS-UNKNOWNL and AS-UNKNOWNR are the counterparts of rules CS-UNKNOWNL and CS-UNKNOWNR.

Rule AS-ARROW 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 (e.g., $[\Theta]A_2$). This is to maintain an important invariant: whenever $\Gamma \vdash A \lesssim B \dashv \Delta$ holds, the types A and B are fully applied under input context Γ (they contain no existential variables already solved in Γ). The same invariant applies to every algorithmic judgment.

Rule AS-FORALLR, similar to the declarative rule CS-FORALLR, adds a to the input context. Note that the output context of the premise allows additional existential variables to appear after the type variable a, in a trailing context Θ. These existential variables could depend on a; since a goes out of scope in the conclusion, we need to drop them from the concluding output, resulting in Δ. The next rule is essential to eliminating the guessing work. Instead of guessing a monotype τ out of thin air, rule AS-FORALLL generates a fresh existential variable $\widehat{a}$, and replaces a with $\widehat{a}$ in the body A. The new existential variable $\widehat{a}$ is then added to the input context, just before the marker $\blacktriangleright_{\widehat{a}}$. The output context $(\Delta, \blacktriangleright_{\widehat{a}}, \Theta)$ allows additional existential variables to appear after $\blacktriangleright_{\widehat{a}}$ in Θ. For the same reasons as in rule AS-FORALLR, we drop them from the output context. A central idea behind these two rules is that we defer the decision of picking a monotype for a type variable, and hope that it could be solved later when we have more information at hand. As a side note, when both types are universal quantifiers, then either rule AS-FORALLR or AS-FORALLL could be tried. In practice, one can apply rule AS-FORALLR eagerly as it is invertible.

$\boxed{\Gamma \vdash \widehat{a} \lessapprox A \dashv \Delta}$ *(Under input context Γ, instantiate $\widehat{a}$ such that $\widehat{a} \lesssim A$, with output context Δ)*

$$\frac{\Gamma \vdash \tau}{\Gamma, \widehat{a}, \Gamma' \vdash \widehat{a} \lessapprox \tau \dashv \Gamma, \widehat{a} = \tau, \Gamma'}\ \textsc{instl-solve} \qquad \frac{}{\Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox \star \dashv \Gamma[\widehat{a}]}\ \textsc{instl-solveU}$$

$$\frac{}{\Gamma[\widehat{a}][\widehat{b}] \vdash \widehat{a} \lessapprox \widehat{b} \dashv \Gamma[\widehat{a}][\widehat{b} = \widehat{a}]}\ \textsc{instl-reach} \qquad \frac{\Gamma[\widehat{a}], b \vdash \widehat{a} \lessapprox B \dashv \Delta, b, \Theta}{\Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox \forall b.\, B \dashv \Delta}\ \textsc{instl-forallR}$$

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

$\boxed{\Gamma \vdash A \lessapprox \widehat{a} \dashv \Delta}$ *(Under input context Γ, instantiate $\widehat{a}$ such that $A \lesssim \widehat{a}$, with output context Δ)*

$$\frac{\Gamma \vdash \tau}{\Gamma, \widehat{a}, \Gamma' \vdash \tau \lessapprox \widehat{a} \dashv \Gamma, \widehat{a} = \tau, \Gamma'}\ \textsc{instr-solve} \qquad \frac{}{\Gamma[\widehat{a}] \vdash \star \lessapprox \widehat{a} \dashv \Gamma[\widehat{a}]}\ \textsc{instr-solveU}$$

$$\frac{}{\Gamma[\widehat{a}][\widehat{b}] \vdash \widehat{b} \lessapprox \widehat{a} \dashv \Gamma[\widehat{a}][\widehat{b} = \widehat{a}]}\ \textsc{instr-reach} \qquad \frac{\Gamma[\widehat{a}], \blacktriangleright_{\widehat{b}}, \widehat{b} \vdash B[b \mapsto \widehat{b}] \lessapprox \widehat{a} \dashv \Delta, \blacktriangleright_{\widehat{b}}, \Theta}{\Gamma[\widehat{a}] \vdash \forall b.\, B \lessapprox \widehat{a} \dashv \Delta}\ \textsc{instr-forallL}$$

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

Fig. 14. Algorithmic instantiation

The last two rules (AS-INSTL and AS-INSTR) are specific to the algorithm, thus having no counterparts in the declarative version. They both check consistent subtyping with an unsolved existential variable on one side and an arbitrary type on the other side. Apart from checking that the existential variable does not occur in the type A, both rules do not directly solve the existential variables, but leave the real work to the instantiation judgment.

6.2 Instantiation

Two symmetric judgments $\Gamma \vdash \widehat{a} \lessapprox A \dashv \Delta$ and $\Gamma \vdash A \lessapprox \widehat{a} \dashv \Delta$ defined in Fig. 14 instantiate unsolved existential variables. They read "under input context Γ, instantiate $\widehat{a}$ to a consistent subtype (or supertype) of A, with output context Δ". The judgments are extended naturally from DK system, whose original inspiration comes from Cardelli [1993]. Since these two judgments are mutually defined, we discuss them together.

Rule INSTL-SOLVE is the simplest one – when an existential variable meets a monotype – where we simply set the solution of $\widehat{a}$ to the monotype τ in the output context. We also need to check that the monotype τ is well-formed under the prefix context Γ.

Rule INSTL-SOLVEU is similar to rule AS-UNKNOWNR in that we put no constraint[12] on $\widehat{a}$ when it meets the unknown type $\star$. This design decision reflects the point that type inference only produces static types [Garcia and Cimini 2015].

Rule INSTL-REACH deals with the situation where two existential variables meet. Recall that $\Gamma[\widehat{a}][\widehat{b}]$ denotes a context where some unsolved existential variable $\widehat{a}$ is declared before $\widehat{b}$. In this situation, the only logical thing we can do is to set the solution of one existential variable to

[12] As we will see in Section 9 where we present a more refined system, the "no constraint" statement is not entirely true.

$\boxed{\Gamma \vdash e \Rightarrow A \dashv \Delta}$ *(Under input context Γ, e infers output type A, with output context Δ)*

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A \dashv \Gamma}\ \textsc{inf-var} \qquad \frac{}{\Gamma \vdash n \Rightarrow \mathsf{Int} \dashv \Gamma}\ \textsc{inf-int} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash e \Leftarrow A \dashv \Delta}{\Gamma \vdash e : A \Rightarrow A \dashv \Delta}\ \textsc{inf-anno}$$

$$\frac{\Gamma \vdash A \qquad \Gamma, \widehat{b}, x : A \vdash e \Leftarrow \widehat{b} \dashv \Delta, x : A, \Theta}{\Gamma \vdash \lambda x : A.\, e \Rightarrow A \to \widehat{b} \dashv \Delta}\ \textsc{inf-lamann} \qquad \frac{\Gamma, \widehat{a}, \widehat{b}, x : \widehat{a} \vdash e \Leftarrow \widehat{b} \dashv \Delta, x : \widehat{a}, \Theta}{\Gamma \vdash \lambda x.\, e \Rightarrow \widehat{a} \to \widehat{b} \dashv \Delta}\ \textsc{inf-lam}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \dashv \Theta_1 \qquad \Theta_1, \widehat{a}, x : A \vdash e_2 \Leftarrow \widehat{a} \dashv \Delta, x : A, \Theta_2}{\Gamma \vdash \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 \Rightarrow \widehat{a} \dashv \Delta}\ \textsc{inf-let}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \dashv \Theta_1 \qquad \Theta_1 \vdash [\Theta_1]A \triangleright A_1 \to A_2 \dashv \Theta_2 \qquad \Theta_2 \vdash e_2 \Leftarrow [\Theta_2]A_1 \dashv \Delta}{\Gamma \vdash e_1\, e_2 \Rightarrow A_2 \dashv \Delta}\ \textsc{inf-app}$$

$\boxed{\Gamma \vdash e \Leftarrow A \dashv \Delta}$ *(Under input context Γ, e checks against input type A, with output context Δ)*

$$\frac{\Gamma, x : A \vdash e \Leftarrow B \dashv \Delta, x : A, \Theta}{\Gamma \vdash \lambda x.\, e \Leftarrow A \to B \dashv \Delta}\ \textsc{chk-lam} \qquad \frac{\Gamma, a \vdash e \Leftarrow A \dashv \Delta, a, \Theta}{\Gamma \vdash e \Leftarrow \forall a.\, A \dashv \Delta}\ \textsc{chk-gen}$$

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

$\boxed{\Gamma \vdash A \triangleright A_1 \to A_2 \dashv \Delta}$ *(Under input context Γ, A matches output type $A_1 \to A_2$, with output context Δ)*

$$\frac{\Gamma, \widehat{a} \vdash A[a \mapsto \widehat{a}] \triangleright A_1 \to A_2 \dashv \Delta}{\Gamma \vdash \forall a.\, A \triangleright A_1 \to A_2 \dashv \Delta}\ \textsc{am-forall} \qquad \frac{}{\Gamma \vdash A_1 \to A_2 \triangleright A_1 \to A_2 \dashv \Gamma}\ \textsc{am-arr}$$

$$\frac{}{\Gamma \vdash \star \triangleright \star \to \star \dashv \Gamma}\ \textsc{am-unknown} \qquad \frac{}{\Gamma[\widehat{a}] \vdash \widehat{a} \triangleright \widehat{a}_1 \to \widehat{a}_2 \dashv \Gamma[\widehat{a}_1, \widehat{a}_2, \widehat{a} = \widehat{a}_1 \to \widehat{a}_2]}\ \textsc{am-var}$$

Fig. 15. Algorithmic typing

the other one, depending on which one is declared before. For example, in the output context of rule INSTL-REACH, we have $\widehat{b} = \widehat{a}$ because in the input context, $\widehat{a}$ is declared before $\widehat{b}$.

Rule INSTL-FORALLR is the instantiation version of rule AS-FORALLR. Since our system is predicative, $\widehat{a}$ cannot be instantiated to $\forall b.\, B$, but we can decompose $\forall b.\, B$ in the same way as in rule AS-FORALLR. Rule INSTR-FORALLL is the instantiation version of rule AS-FORALLL.

Rule INSTL-ARR applies when $\widehat{a}$ meets an arrow type. It follows that the solution must also be an arrow type. This 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 we can solve $\widehat{a}$ to $\widehat{a}_1 \to \widehat{a}_2$. Note that the first premise $A_1 \lessapprox \widehat{a}_1$ switches to the other instantiation judgment.

6.3 Algorithmic Typing

We now turn to the algorithmic typing rules in Fig. 15. Because general type inference for System F is undecidable [Wells 1999], our algorithmic system uses bidirectional type checking to accommodate (first-class) polymorphism. Traditionally, two modes are employed in bidirectional systems: the checking mode $\Gamma \vdash e \Leftarrow A \dashv \Theta$, which takes a term e and a type A as input, and ensures that the term e checks against A; the inference mode $\Gamma \vdash e \Rightarrow A \dashv \Theta$, which takes a term e and produces a type A. We first discuss rules in the inference mode.

Rules INF-VAR and INF-INT do not generate any new information and simply propagate the input context. Rule INF-ANNO is standard, switching to the checking mode in the premise.

In rule INF-LAMANN, we generate a fresh existential variable $\widehat{b}$ for the function codomain, and check the function body against $\widehat{b}$. Note that it is tempting to write $\Gamma, x : A \vdash e \Rightarrow B \dashv \Delta, x : A, \Theta$ as the premise (in the hope of better matching its declarative counterpart rule LAMANN), which has a subtle consequence. Consider the expression $\lambda x : \mathsf{Int}.\, \lambda y.\, y$. Under the new premise, this is untypable because of $\bullet \vdash \lambda x : \mathsf{Int}.\, \lambda y.\, y \Rightarrow \mathsf{Int} \rightarrow \widehat{a} \rightarrow \widehat{a} \dashv \bullet$ where $\widehat{a}$ is not found in the output context. This explains why we put $\widehat{b}$ *before* $x : A$ so that it remains in the output context Δ. Rule INF-LAM, which corresponds to rule LAM, one of the guessing rules, is similar to rule INF-LAMANN. As with the other algorithmic rules that eliminate guessing, we create new existential variables $\widehat{a}$ (for function domain) and $\widehat{b}$ (for function codomain) and check the function body against $\widehat{b}$. Rule INF-LET is similar to rule INF-LAMANN.

Algorithmic Matching. Rule INF-APP (which differs significantly from that of [Dunfield and Krishnaswami 2013]) deserves attention. It relies on the algorithmic matching judgment $\Gamma \vdash A \triangleright A_1 \rightarrow A_2 \dashv \Delta$. The matching judgment algorithmically synthesizes an arrow type from an arbitrary type. Rule AM-FORALL replaces a with a fresh existential variable $\widehat{a}$, thus eliminating guessing. Rules AM-ARR and AM-UNKNOWN correspond directly to the declarative rules. Rule AM-VAR, which has no corresponding declarative version, is similar to rule INSTL-ARR/INSTR-ARR: we create $\widehat{a}_1$ and $\widehat{a}_2$ and solve $\widehat{a}$ to $\widehat{a}_1 \rightarrow \widehat{a}_2$ in the output context.

Back to the rule INF-APP. This rule first infers the type of e_1, producing an output context Θ_1. Then it applies Θ_1 to A and goes into the matching judgment, which delivers an arrow type $A_1 \rightarrow A_2$ and another output context Θ_2. Θ_2 is used as the input context when checking e_2 against $[\Theta_2]A_1$, where we go into the checking mode.

Rules in the checking mode are quite standard. Rule CHK-LAM checks against $A \rightarrow B$. Rule CHK-GEN, like the declarative rule GEN, adds a type variable a to the input context. Rule CHK-SUB uses the algorithmic consistent subtyping judgment.

6.4 Decidability

Our algorithmic system is decidable. It is not at all obvious to see why this is the case, as many rules are not strictly structural (e.g., many rules have $[\Gamma]A$ in the premises). This implies that we need a more sophisticated measure metric to support the argument. Since the typing rules (Fig. 15) depend on the consistent subtyping rules (Fig. 13), which in turn depends on the instantiation rules (Fig. 14), to show the decidability of the typing judgment, we need to show that the instantiation and consistent subtyping judgments are decidable. The proof strategy mostly follows that of the DK system. Here only highlights of the proofs are given; the full proofs can be found in Appendix C.

Decidability of Instantiation. The basic idea is that we need to show A in the instantiation judgments $\Gamma \vdash \widehat{a} \lessapprox A \dashv \Delta$ and $\Gamma \vdash A \lessapprox \widehat{a} \dashv \Delta$ always gets smaller. Most of the rules are structural and thus easy to verify (e.g., rule INSTL-FORALLR); the non-trivial cases are rules INSTL-ARR and INSTR-ARR where context applications appear in the premises. The key observation there is that the instantiation rules preserve the size of (substituted) types. The formal statement of decidability of instantiation needs a few pre-conditions: assuming $\widehat{a}$ is unsolved in the input context Γ, that A is well-formed under the context Γ, that A is fully applied under the input context Γ ($[\Gamma]A = A$), and that $\widehat{a}$ does not occur in A. Those conditions are actually met when instantiation is invoked: rule CHK-SUB applies the input context, and the subtyping rules apply input context when needed.

THEOREM 6.1 (DECIDABILITY OF INSTANTIATION). *If $\Gamma = \Gamma_0[\widehat{a}]$ and $\Gamma \vdash A$ such that $[\Gamma]A = A$ and $\widehat{a} \notin \mathit{FV}(A)$ then:*

(1) Either there exists Δ *such that* $\Gamma \vdash \widehat{a} \lessapprox A \dashv \Delta$*, or not.*
(2) Either there exists Δ *such that* $\Gamma \vdash A \lessapprox \widehat{a} \dashv \Delta$*, or not.*

Decidability of Algorithmic Consistent Subtyping. Proving decidability of the algorithmic consistent subtyping is a bit more involved, as the induction measure consists of several parts. We measure the judgment $\Gamma \vdash A \lesssim B \dashv \Delta$ lexicographically by

(M1) the number of $\forall$-quantifiers in A and B;
(M2) the number of unknown types in A and B;
(M3) $|\textsc{unsolved}(\Gamma)|$: the number of unsolved existential variables in Γ;
(M4) $|\Gamma \vdash A| + |\Gamma \vdash B|$.

Notice that because of our gradual setting, we also need to measure the number of unknown types (M2). This is a key difference from the DK system. We refer the reader to Appendix C for more details. For (M4), we use *contextual size*—the size of well-formed types under certain contexts, which penalizes solved variables (*).

Definition 6.2 (Contextual Size).

$$
\begin{array}{lcll}
|\Gamma \vdash \mathsf{Int}| & = & 1 & \\
|\Gamma \vdash \star| & = & 1 & \\
|\Gamma \vdash a| & = & 1 & \\
|\Gamma \vdash \widehat{a}| & = & 1 & \\
|\Gamma[\widehat{a} = \tau] \vdash \widehat{a}| & = & 1 + |\Gamma[\widehat{a} = \tau] \vdash \tau| & (*) \\
|\Gamma \vdash \forall a.\, A| & = & 1 + |\Gamma, a \vdash A| & \\
|\Gamma \vdash A \rightarrow B| & = & 1 + |\Gamma \vdash A| + |\Gamma \vdash B| &
\end{array}
$$

Theorem 6.3 (Decidability of Algorithmic Consistent Subtyping). *Given a context* Γ *and types* A*,* B *such that* $\Gamma \vdash A$ *and* $\Gamma \vdash B$ *and* $[\Gamma]A = A$ *and* $[\Gamma]B = B$*, it is decidable whether there exists* Δ *such that* $\Gamma \vdash A \lesssim B \dashv \Delta$*.*

Decidability of Algorithmic Typing. Similar to proving decidability of algorithmic consistent subtyping, the key is to come up with a correct measure. Since the typing rules depend on the matching judgment, we first show decidability of the algorithmic matching.

Lemma 6.4 (Decidability of Algorithmic Matching). *Given a context* Γ *and a type* A *it is decidable whether there exist types* A_1*,* A_2 *and a context* Δ *such that* $\Gamma \vdash A \triangleright A_1 \rightarrow A_2 \dashv \Delta$*.*

Now we are ready to show decidability of typing. The following induction measure suffices:

$$
\left\langle \quad e, \quad \begin{array}{c} \Rightarrow \\ \Leftarrow, \end{array} \quad |\Gamma \vdash A| \quad \right\rangle
$$

where $\langle \ldots \rangle$ denotes lexicographic order, and where (when comparing two judgments of typing terms of the same size) the inference judgment (first line) is considered *smaller* than the checking judgment (second line). The above measure is much simpler than the corresponding one in the DK system, where they also need to consider the application judgment together with the inference and checking judgments. This shows another benefit (besides the independence from typing) of adopting the matching judgment.

Theorem 6.5 (Decidability of Algorithmic Typing).

(1) Inference: Given a context Γ *and a term* e*, it is decidable whether there exist a type* A *and a context* Δ *such that* $\Gamma \vdash e \Rightarrow A \dashv \Delta$*.*
(2) Checking: Given a context Γ*, a term* e *and a type* B *such that* $\Gamma \vdash B$*, it is decidable whether there exists a context* Δ *such that* $\Gamma \vdash e \Leftarrow B \dashv \Delta$*.*

$\boxed{\Gamma \longrightarrow \Delta}$ *(Context extension)*

$$\frac{}{\bullet \longrightarrow \bullet}\ \textsc{ext-id} \qquad \frac{\Gamma \longrightarrow \Delta \qquad [\Delta]A = [\Delta]A'}{\Gamma, x : A \longrightarrow \Delta, x : A'}\ \textsc{ext-var} \qquad \frac{\Gamma \longrightarrow \Delta}{\Gamma, a \longrightarrow \Delta, a}\ \textsc{ext-tvar}$$

$$\frac{\Gamma \longrightarrow \Delta}{\Gamma, \widehat{a} \longrightarrow \Delta, \widehat{a}}\ \textsc{ext-evar} \qquad \frac{\Gamma \longrightarrow \Delta \qquad [\Delta]\tau = [\Delta]\tau'}{\Gamma, \widehat{a} = \tau \longrightarrow \Delta, \widehat{a} = \tau'}\ \textsc{ext-solved} \qquad \frac{\Gamma \longrightarrow \Delta}{\Gamma, \widehat{a} \longrightarrow \Delta, \widehat{a} = \tau}\ \textsc{ext-solve}$$

$$\frac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, \widehat{a}}\ \textsc{ext-add} \qquad \frac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, \widehat{a} = \tau}\ \textsc{ext-addSolve} \qquad \frac{\Gamma \longrightarrow \Delta}{\Gamma, \blacktriangleright_{\widehat{a}} \longrightarrow \Delta, \blacktriangleright_{\widehat{a}}}\ \textsc{ext-marker}$$

Fig. 16. Context extension

7 SOUNDNESS AND COMPLETENESS

To be confident that our algorithmic type system and the declarative type system agree with each other, we need to prove that the algorithmic rules are sound and complete with respect to the declarative specification. Before we give the formal statements of the soundness and completeness theorems, we need a meta-theoretical device, called *context extension* [Dunfield and Krishnaswami 2013], to capture a notion of information increase from input contexts to output contexts.

7.1 Context Extension

A context extension judgment $\Gamma \longrightarrow \Delta$ reads "Γ is extended by Δ". Intuitively, this judgment says that Δ has at least as much information as Γ: some unsolved existential variables in Γ may be solved in Δ. The full inductive definition can be found Fig. 16. We refer the reader to Dunfield and Krishnaswami [2013, Section 4] for further explanation of context extension.

7.2 Soundness

Roughly speaking, soundness of the algorithmic system says that given a derivation of an algorithmic judgment with input context Γ, output context Δ, and a complete context Ω that extends Δ, applying Ω throughout the given algorithmic judgment should yield a derivable declarative judgment. For example, let us consider an algorithmic typing judgment $\bullet \vdash \lambda x.\, x \Rightarrow \widehat{a} \to \widehat{a} \dashv \widehat{a}$, and any complete context, say, $\Omega = (\widehat{a} = \mathsf{Int})$, then applying Ω to the above judgment yields $\bullet \vdash \lambda x.\, x : \mathsf{Int} \to \mathsf{Int}$, which is derivable in the declarative system.

However there is one complication: applying Ω to the algorithmic expression does not necessarily yield a typable declarative expression. For example, by rule chk-lam we have $\lambda x.\, x \Leftarrow (\forall a.\, a \to a) \to (\forall a.\, a \to a)$, but $\lambda x.\, x$ itself cannot have type $(\forall a.\, a \to a) \to (\forall a.\, a \to a)$ in the declarative system. To circumvent that, we add an annotation to the lambda abstraction, resulting in $\lambda x : (\forall a.\, a \to a).\, x$, which is typeable in the declarative system with the same type. To relate $\lambda x.\, x$ and $\lambda x : (\forall a.\, a \to a).\, x$, we erase all annotations on both expressions.

Definition 7.1 (Type annotation erasure). The erasure function is denoted as $\lfloor \cdot \rfloor$, and defined as follows:

$$\begin{aligned} \lfloor x \rfloor &= x & \lfloor n \rfloor &= n \\ \lfloor \lambda x : A.\, e \rfloor &= \lambda x.\, \lfloor e \rfloor & \lfloor \lambda x.\, e \rfloor &= \lambda x.\, \lfloor e \rfloor \\ \lfloor e_1\, e_2 \rfloor &= \lfloor e_1 \rfloor\, \lfloor e_2 \rfloor & \lfloor e : A \rfloor &= \lfloor e \rfloor \end{aligned}$$

Theorem 7.2 (Instantiation Soundness). *Given $\Delta \longrightarrow \Omega$ and $[\Gamma]A = A$ and $\widehat{a} \notin \mathit{FV}(A)$:*
(1) If $\Gamma \vdash \widehat{a} \lessapprox A \dashv \Delta$ then $[\Omega]\Delta \vdash [\Omega]\widehat{a} \lesssim [\Omega]A$.

(2) If $\Gamma \vdash A \lessapprox \widehat{a} \dashv \Delta$ *then* $[\Omega]\Delta \vdash [\Omega]A \lesssim [\Omega]\widehat{a}$.

Notice that the declarative judgment uses $[\Omega]\Delta$, an operation that applies a complete context Ω to the algorithmic context Δ, essentially plugging in all known solutions and removing all declarations of existential variables (both solved and unsolved), resulting in a declarative context.

With instantiation soundness, next we show that the algorithmic consistent subtyping is sound:

Theorem 7.3 (Soundness of Algorithmic Consistent Subtyping). *If* $\Gamma \vdash A \lesssim B \dashv \Delta$ *where* $[\Gamma]A = A$ *and* $[\Gamma]B = B$ *and* $\Delta \longrightarrow \Omega$ *then* $[\Omega]\Delta \vdash [\Omega]A \lesssim [\Omega]B$.

Finally the soundness theorem of algorithmic typing is:

Theorem 7.4 (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* $\lfloor e \rfloor = \lfloor e' \rfloor$.

7.3 Completeness

Completeness of the algorithmic system is the reverse of soundness: given a declarative judgment of the form $[\Omega]\Gamma \vdash [\Omega] \dots$, 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 7.5 (Instantiation Completeness). *Given* $\Gamma \longrightarrow \Omega$ *and* $A = [\Gamma]A$ *and* $\widehat{a} \notin$ *unsolved*(Γ) *and* $\widehat{a} \notin$ *fv*(A)*:*

(1) If $[\Omega]\Gamma \vdash [\Omega]\widehat{a} \lesssim [\Omega]A$ *then there are* Δ, Ω' *such that* $\Omega \longrightarrow \Omega'$ *and* $\Delta \longrightarrow \Omega'$ *and* $\Gamma \vdash \widehat{a} \lessapprox A \dashv \Delta$.
(2) If $[\Omega]\Gamma \vdash [\Omega]A \lesssim [\Omega]\widehat{a}$ *then there are* Δ, Ω' *such that* $\Omega \longrightarrow \Omega'$ *and* $\Delta \longrightarrow \Omega'$ *and* $\Gamma \vdash A \lessapprox \widehat{a} \dashv \Delta$.

Next is the completeness of consistent subtyping:

Theorem 7.6 (Generalized Completeness of Consistent Subtyping). *If* $\Gamma \longrightarrow \Omega$ *and* $\Gamma \vdash A$ *and* $\Gamma \vdash B$ *and* $[\Omega]\Gamma \vdash [\Omega]A \lesssim [\Omega]B$ *then there exist* Δ *and* Ω' *such that* $\Delta \longrightarrow \Omega'$ *and* $\Omega \longrightarrow \Omega'$ *and* $\Gamma \vdash [\Gamma]A \lesssim [\Gamma]B \dashv \Delta$.

We prove that the algorithmic matching is complete with respect to the declarative matching:

Theorem 7.7 (Matching Completeness). *Given* $\Gamma \longrightarrow \Omega$ *and* $\Gamma \vdash A$, *if* $[\Omega]\Gamma \vdash [\Omega]A \triangleright A_1 \rightarrow A_2$ *then there exist* Δ, Ω', A_1' *and* A_2' *such that* $\Gamma \vdash [\Gamma]A \triangleright A_1' \rightarrow A_2' \dashv \Delta$ *and* $\Delta \longrightarrow \Omega'$ *and* $\Omega \longrightarrow \Omega'$ *and* $A_1 = [\Omega']A_1'$ *and* $A_2 = [\Omega']A_2'$.

Finally here is the completeness theorem of the algorithmic typing:

Theorem 7.8 (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* $\lfloor e \rfloor = \lfloor e' \rfloor$.

8 SIMPLE EXTENSIONS AND VARIANTS

This section considers two simple variations on the presented system. The first variation extends the system with a top type, while the second variation considers a more declarative formulation using a subsumption rule.

8.1 Top Types

We argued that our definition of consistent subtyping (Definition 4.1) generalizes the original definition by Siek and Taha [2007]. We have shown its applicability to polymorphic types, for which Siek and Taha [2007] approach cannot be extended naturally. To strengthen our argument, we show how to extend our approach to Top types, which is also not supported by Siek and Taha [2007] approach.

Consistent Subtyping with $\top$. 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 still seems reasonable. However, an important point is that, because of the orthogonality between subtyping and consistency, subtyping itself should not contain a potential information loss! 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, $\mathsf{Int} \to \star \lesssim \top$.

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

It is easy to verify that Definition 4.1 is still equivalent to that in Fig. 8 extended with rule CS-Top. That is, $\mathcal{T}$heorem 1 holds:

Proposition 8.1 (Extension with $\top$). *$\Psi \vdash A \lesssim B \Leftrightarrow \Psi \vdash A <: C, C \sim D, \Psi \vdash D <: B$, for some C, D.*

We extend the definition of concretization (Definition 4.3) with $\top$ by adding another equation $\gamma(\top) = \{\top\}$. Note that Castagna and Lanvin [2017] also have this equation in their calculus. It is easy to verify that Corollary 4.5 still holds:

Proposition 8.2 (Equivalent to AGT on $\top$). *$A \lesssim B$ if only if $A \;\widetilde{<:}\; B$.*

Siek and Taha's Definition of Consistent Subtyping Does Not Work for $\top$. As with the analysis in Section 4.2, $\mathsf{Int} \to \star \lesssim \top$ only holds when we first apply consistency, then subtyping. However we cannot find a type A such that $\mathsf{Int} \to \star <: A$ and $A \sim \top$. The following diagram depicts the situation:

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

Additionally we have a similar problem in extending the restriction operator: *non-structural* masking between $\mathsf{Int} \to \star$ and $\top$ cannot be easily achieved.

Note that both the top and universally quantified types can be seen as special cases of intersection types. Indeed, top is the intersection of the empty set, while a universally quantified type is the intersection of the infinite set of its instantiations [Davies and Pfenning 2000]. Recall from Section 4.3 that Castagna and Lanvin [2017] shows that consistent subtyping from AGT works well for intersection types, and our definition coincides with AGT (Corollary 4.5 and Proposition 8.2). We thus believe that our definition is compatible with conventional binary intersection types as well. Yet, a rigorous formalization would be needed to substantiate this belief.

8.2 A More Declarative Type System

In Section 5 we present our declarative system in terms of the matching and consistent subtyping judgments. The rationale behind this design choice is that the resulting declarative system combines subtyping and type consistency in the application rule, thus making it easier to design an algorithmic system accordingly. Still, one may wonder if it is possible to design a more declarative specification. For example, even though we mentioned that the subsumption rule is incompatible with consistent subtyping, it might be possible to accommodate a subsumption rule for normal subtyping (instead of consistent subtyping). In this section, we discuss an alternative for the design of the declarative system.

Wrong Design. A naive design that does not work is to replace rule App in Fig. 9 with the following two rules:

$$\frac{\Psi \vdash e : A \qquad A <: B}{\Psi \vdash e : B}\ \text{V-Sub} \qquad \frac{\Psi \vdash e_1 : A \qquad \Psi \vdash e_2 : A_1 \qquad A \sim A_1 \rightarrow A_2}{\Psi \vdash e_1\ e_2 : A_2}\ \text{V-App1}$$

Rule V-Sub is the standard subsumption rule: if an expression e has type A, then it can be assigned some type B that is a supertype of A. Rule V-App1 first infers that e_1 has type A, and e_2 has type A_1, then it finds some A_2 so that A is consistent with $A_1 \rightarrow A_2$.

There would be two obvious benefits of this variant if it did work: firstly this approach closely resembles the traditional declarative type systems for calculi with subtyping; secondly it saves us from discussing various forms of A in rule V-App1, leaving the job to the consistency judgment.

The design is wrong because of the information loss caused by the choice of A_2 in rule V-App1. Suppose we have $\Psi \vdash \mathsf{plus} : \mathsf{Int} \rightarrow \mathsf{Int} \rightarrow \mathsf{Int}$, then we can apply it to 1 to get

$$\frac{\Psi \vdash \mathsf{plus} : \mathsf{Int} \rightarrow \mathsf{Int} \rightarrow \mathsf{Int} \qquad \Psi \vdash 1 : \mathsf{Int} \qquad \mathsf{Int} \rightarrow \mathsf{Int} \rightarrow \mathsf{Int} \sim \mathsf{Int} \rightarrow \star \rightarrow \mathsf{Int}}{\Psi \vdash \mathsf{plus}\ 1 : \star \rightarrow \mathsf{Int}}\ \text{V-App1}$$

Further applying it to true we get

$$\frac{\Psi \vdash \mathsf{plus}\ 1 \Rightarrow \star \rightarrow \mathsf{Int} \qquad \Psi \vdash \mathsf{true} : \mathsf{Bool} \qquad \star \rightarrow \mathsf{Int} \sim \mathsf{Bool} \rightarrow \mathsf{Int}}{\Psi \vdash \mathsf{plus}\ 1\ \mathsf{true} : \mathsf{Int}}\ \text{V-App1}$$

which is obviously wrong! The type consistency in rule V-App1 causes information loss for both the argument type A_1 and the return type A_2. The problem is that information of A_2 can get lost again if it appears in further applications. The moral of the story is that we should be very careful when using type consistency. We hypothesize that it is inevitable to do case analysis for the type of the function in an application (i.e., A in rule V-App1).

Proper Declarative Design. The proper design refines the first variant by using a matching judgment to carefully distinguish two cases for the typing result of e_1 in rule V-App1: (1) when it is an arrow type, and (2) when it is an unknown type. This variant replaces rule App in Fig. 9 with the following rules:

$$\frac{\Gamma \vdash e : A \qquad A <: B}{\Gamma \vdash e : B}\ \text{V-Sub} \qquad \frac{\Psi \vdash e : A \qquad \Psi \vdash A \triangleright A_1 \rightarrow A_2 \qquad \Psi \vdash e_2 : A_3 \qquad A_1 \sim A_3}{\Psi \vdash e_1\ e_2 : A_2}\ \text{V-App2}$$

$$\frac{}{\Psi \vdash A_1 \rightarrow A_2 \triangleright A_1 \rightarrow A_2} \qquad \frac{}{\Psi \vdash \star \triangleright \star \rightarrow \star}$$

Rule V-Sub is the same as in the first variant. In rule V-App2, we infer that e_1 has type A, and use the matching judgment to get an arrow type $A_1 \rightarrow A_2$. Then we need to ensure that the argument type A_3 is *consistent with* (rather than a consistent subtype of) A_1, and use A_2 as the result type of the application. The matching judgment only deals with two cases, as polymorphic types are handled by rule V-SUB. These rules are closely related to the ones in Siek and Taha [2006] and Siek and Taha [2007].

The more declarative nature of this system also implies that it is highly non-syntax-directed, and it does not offer any insight into combining subtyping and consistency. We have proved in Coq the following lemmas to establish soundness and completeness of this system with respect to our original system (to avoid ambiguity, we use the notation $\vdash_m$ to indicate the more declarative version):

LEMMA 5 (COMPLETENESS OF $\vdash_m$). *If* $\Gamma \vdash e : A$, *then* $\Gamma \vdash_m e : A$.

LEMMA 6 (SOUNDNESS OF $\vdash_m$). *If* $\Gamma \vdash_m e : A$, *then there exists some* B, *such that* $\Gamma \vdash e : B$ *and* $\Gamma \vdash B <: A$.

9 RESTORING THE DYNAMIC GRADUAL GUARANTEE WITH TYPE PARAMETERS

In Section 5.2 we have seen an example where a single source expression could produce two different target expressions with different runtime behaviors. As we explained, this is due to the guessing nature of the declarative system, and, from the (source) typing point of view, no guessed type is particularly better than any other. As a consequence, this breaks the dynamic gradual guarantee as discussed in Section 5.3.

To alleviate this situation, we introduce *static type parameters*, which are placeholders for monotypes, and *gradual type parameters*, which are placeholders for monotypes that are consistent with the unknown type. The concept of static type parameters and gradual type parameters in the context of gradual typing was first introduced by Garcia and Cimini [2015], and later played a central role in the work of Igarashi et al. [2017]. In our type system, type parameters mainly help capture the notion of *representative translations*, and should not appear in a source program. With them we are able to recast the dynamic gradual guarantee in terms of representative translations, and to prove that every well-typed source expression possesses at least one representative translation. With a coherence conjecture regarding representative translations, the dynamic gradual guarantee of our extended source language now can be reduced to that of λB, which, at the time of writing, is still an open question.

9.1 Declarative Type System

The new syntax of types is given at the top of Fig. 17, with the differences highlighted. In addition to the types of Fig. 4, we add *static type parameters* $\mathcal{S}$, and *gradual type parameters* $\mathcal{G}$. Both kinds of type parameters are monotypes. The addition of type parameters, however, leads to two new syntactic categories of types. *Castable types* $\mathbb{C}$ represent types that can be cast from or to $\star$. It includes all types, except those that contain static type parameters. *Castable monotypes* t are those castable types that are also monotypes.

Consistent Subtyping. The new definition of consistent subtyping is given at the bottom of Fig. 17, again with the differences highlighted. Now the unknown type is only a consistent subtype of all castable types, rather than of all types (rule CS-UNKNOWNLL), and vice versa (rule CS-UNKNOWNRR). Moreover, the static type parameter $\mathcal{S}$ is a consistent subtype of itself (rule CS-SPAR), and similarly for the gradual type parameter (rule CS-GPAR). From this definition it follows immediately that $\star$ is incomparable with types that contain static type parameters $\mathcal{S}$, such as $\mathcal{S} \to \mathsf{Int}$.

Typing and Translation. Given these extensions to types and consistent subtyping, the typing process remains the same as in Fig. 9. To account for the changes in the translation, if we extend λB with type parameters as in Garcia and Cimini [2015], then the translation remains the same as well.

Types	A, B	$::=$	$\mathsf{Int} \mid a \mid A \to B \mid \forall a.\, A \mid \star \mid \mathcal{S} \mid \mathcal{G}$
Monotypes	τ, σ	$::=$	$\mathsf{Int} \mid a \mid \tau \to \sigma \mid \mathcal{S} \mid \mathcal{G}$
Castable Types	$\mathbb{C}$	$::=$	$\mathsf{Int} \mid a \mid \mathbb{C}_1 \to \mathbb{C}_2 \mid \forall a.\, \mathbb{C} \mid \star \mid \mathcal{G}$
Castable Monotypes	t	$::=$	$\mathsf{Int} \mid a \mid t_1 \to t_2 \mid \mathcal{G}$

$\boxed{\Psi \vdash A \lesssim B}$ *(Consistent Subtyping)*

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

$$\frac{\Psi, a \vdash A \lesssim B}{\Psi \vdash A \lesssim \forall a.\, B}\ \textsc{cs-ForallR} \qquad \frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] \lesssim B}{\Psi \vdash \forall a.\, A \lesssim B}\ \textsc{cs-ForallL} \qquad \frac{}{\Psi \vdash \star \lesssim \mathbb{C}}\ \textsc{cs-UnknownLL}$$

$$\frac{}{\Psi \vdash \mathbb{C} \lesssim \star}\ \textsc{cs-UnknownRR} \qquad \frac{}{\Psi \vdash \mathcal{S} \lesssim \mathcal{S}}\ \textsc{cs-SPar} \qquad \frac{}{\Psi \vdash \mathcal{G} \lesssim \mathcal{G}}\ \textsc{cs-GPar}$$

Fig. 17. Syntax of types, and consistent subtyping in the extended declarative system.

9.2 Substitutions and Representative Translations

As we mentioned, type parameters serve as placeholders for monotypes. As a consequence, wherever a type parameter is used, any *suitable* monotype could appear just as well. To formalize this observation, we define substitutions for type parameters as follows:

Definition 9.1 (Substitution). Substitutions for type parameters are defined as:

(1) Let $S^{\mathcal{S}} : \mathcal{S} \to \tau$ be a total function mapping static type parameters to monotypes.
(2) Let $S^{\mathcal{G}} : \mathcal{G} \to t$ be a total function mapping gradual type parameters to castable monotypes.
(3) Let $S^{\mathcal{P}} = S^{\mathcal{G}} \cup S^{\mathcal{S}}$ be a union of $S^{\mathcal{S}}$ and $S^{\mathcal{G}}$ mapping static and gradual type parameters accordingly.

Note that since $\mathcal{G}$ might be compared with $\star$, only castable monotypes are suitable substitutes, whereas $\mathcal{S}$ can be replaced by any monotypes. Therefore, we can substitute $\mathcal{G}$ for $\mathcal{S}$, but not the other way around.

Let us go back to our example and its two translations in Section 5.2. The problem with those translations is that neither $\mathsf{Int} \to \mathsf{Int}$ nor $\mathsf{Bool} \to \mathsf{Int}$ is general enough. With type parameters, however, we can state a more *general* translation that covers both through substitution:

$$f : \forall a.a \to \mathsf{Int} \vdash (\lambda x : \star.\ f\ x) : \star \to \mathsf{Int}$$
$$\rightsquigarrow (\lambda x : \star.\ (\langle \forall a.a \to \mathsf{Int} \hookrightarrow \mathcal{G} \to \mathsf{Int}\rangle\ f)\ (\langle \star \hookrightarrow \mathcal{G}\rangle\ x))$$

The advantage of type parameters is that they help reasoning about the dynamic semantics. Now we are not limited to a particular choice, such as $\mathsf{Int} \to \mathsf{Int}$ or $\mathsf{Bool} \to \mathsf{Int}$, which might or might not emit a cast error at runtime. Instead we have a general choice $\mathcal{G} \to \mathsf{Int}$.

What does the more general choice with type parameters tell us? First, we know that in this case, there is no concrete constraint on a, so we can instantiate it with a type parameter. Second, the fact that the general choice uses $\mathcal{G}$ rather than $\mathcal{S}$ indicates that any chosen instantiation needs to be a castable type. It follows that any concrete instantiation will have an impact on the runtime behavior; therefore it is best to instantiate a with $\star$. However, type inference cannot instantiate a with $\star$, and substitution cannot replace $\mathcal{G}$ with $\star$ either. This means that we need a syntactic refinement process of the translated programs in order to replace type parameters with allowed gradual types.

Syntactic Refinement. We define syntactic refinement of the translated expressions as follows. As $\mathcal{S}$ denotes no constraints at all, substituting it with any monotype would work. Here we arbitrarily use Int. We interpret $\mathcal{G}$ as $\star$ since any monotype could possibly lead to a cast error.

Definition 9.2 (Syntactic Refinement). The syntactic refinement of a translated expression s is denoted by $\lceil s \rceil$, and defined as follows:

$$
\begin{array}{lcl@{\qquad}lcl}
\lceil \mathsf{Int} \rceil & = & \mathsf{Int} & \lceil a \rceil & = & a \\
\lceil A \to B \rceil & = & \lceil A \rceil \to \lceil B \rceil & \lceil \forall a.A \rceil & = & \forall a.\lceil A \rceil \\
\lceil \star \rceil & = & \star & \lceil \mathcal{S} \rceil & = & \mathsf{Int} \\
\lceil \mathcal{G} \rceil & = & \star & & &
\end{array}
$$

Applying the syntactic refinement to the translated expression, we get

$$(\lambda x : \star.\ (\langle \forall a.a \to \mathsf{Int} \hookrightarrow \boxed{\star} \to \mathsf{Int} \rangle\ f)\ (\langle \star \hookrightarrow \boxed{\star} \rangle\ x))$$

where two $\mathcal{G}$ are refined by $\star$ as highlighted. It is easy to verify that both applying this expression to 3 and to *true* now results in a translation that evaluates to a value.

Representative Translations. To decide whether one translation is more general than the other, we define a preorder between translations.

Definition 9.3 (Translation Pre-order). Suppose $\Psi \vdash e : A \rightsquigarrow s_1$ and $\Psi \vdash e : A \rightsquigarrow s_2$, we define $s_1 \le s_2$ to mean $s_2 \equiv_\alpha S^{\mathcal{P}}\ (s_1)$ for some $S^{\mathcal{P}}$.

Proposition 9.4. *If $s_1 \le s_2$ and $s_2 \le s_1$, then s_1 and s_2 are α-equivalent (i.e., equivalent up to renaming of type parameters).*

The preorder between translations gives rise to a notion of what we call *representative translations*:

Definition 9.5 (Representative Translation). A translation s is said to be a representative translation of a typing derivation $\Psi \vdash e : A \rightsquigarrow s$ if and only if for any other translation $\Psi \vdash e : A \rightsquigarrow s'$ such that $s' \le s$, we have $s \le s'$. From now on we use r to denote a representative translation.

An important property of representative translations, which we conjecture for the lack of rigorous proof, is that if there exists any translation of an expression that (after syntactic refinement) can reduce to a value, so can a representative translation of that expression. Conversely, if a representative translation runs into a blame, then no translation of that expression can reduce to a value.

Conjecture 9.6 (Property of Representative Translations). *For any expression e such that $\Psi \vdash e : A \rightsquigarrow s$ and $\Psi \vdash e : A \rightsquigarrow r$ and $\forall C.\, C : (\Psi \vdash^B A) \rightsquigarrow (\bullet \vdash^B \mathsf{Int})$, we have*

- *If $C\{\lceil s \rceil\} \Downarrow n$, then $C\{\lceil r \rceil\} \Downarrow n$.*
- *If $C\{\lceil r \rceil\} \Downarrow \mathsf{blame}$, then $C\{\lceil s \rceil\} \Downarrow \mathsf{blame}$.*

Given this conjecture, we can state a stricter coherence property (without the "up to casts" part) between any two representative translations. We first strengthen Definition 5.1 following Ahmed et al. [2017]:

Definition 9.7 (Contextual Approximation à la Ahmed et al. [2017]).

$$\begin{array}{rcl} \Psi \vdash s_1 \preceq_{ctx} s_2 : A & \triangleq & \Psi \vdash^B s_1 : A \wedge \Psi \vdash^B s_2 : A \wedge \\ & & \text{for all } C.\, C : (\Psi \vdash^B A) \rightsquigarrow (\bullet \vdash^B \mathsf{Int}) \Longrightarrow \\ & & \quad (C\{\lceil s_1 \rceil\} \Downarrow n \Longrightarrow C\{\lceil s_2 \rceil\} \Downarrow n) \wedge \\ & & \quad (C\{\lceil s_1 \rceil\} \Downarrow \mathsf{blame} \Longrightarrow C\{\lceil s_2 \rceil\} \Downarrow \mathsf{blame}) \end{array}$$

The only difference is that now when a program containing s_1 reduces to a value, so does one containing s_2.

From Conjecture 9.6, it follows that coherence holds between two representative translations of the same expression.

Corollary 9.8 (Coherence for Representative Translations). *For any expression e such that $\Psi \vdash e : A \rightsquigarrow r_1$ and $\Psi \vdash e : A \rightsquigarrow r_2$, we have $\Psi \vdash r_1 \simeq_{ctx} r_2 : A$.*

We have proved that for every typing derivation, at least one representative translation exists.

Lemma 9.9 (Representative Translation for Typing). *For any typing derivation $\Psi \vdash e : A$ there exists at least one representative translation r such that $\Psi \vdash e : A \rightsquigarrow r$.*

For our example, $(\lambda x : \star.\, ((\langle \forall a.a \to \mathsf{Int} \hookrightarrow \mathcal{G} \to \mathsf{Int} \rangle\ f)\ (\langle \star \hookrightarrow \mathcal{G} \rangle\ x))$ is a representative translation, while the other two are not.

9.3 Dynamic Gradual Guarantee, Reloaded

Given the above propositions, we are ready to revisit the dynamic gradual guarantee. The nice thing about representative translations is that the dynamic gradual guarantee of our source language is essentially that of λB, our target language. However, the dynamic gradual guarantee for λB is still an open question. According to Igarashi et al. [2017], the difficulty lies in the definition of term precision that preserves the semantics. We leave it here as a conjecture as well. From a declarative point of view, we cannot prevent the system from picking undesirable instantiations, but we know that some choices are better than the others, so we can restrict the discussion of dynamic gradual guarantee to representative translations.

Conjecture 9.10 (Dynamic Gradual Guarantee in terms of Representative Translations). *Suppose $e' \sqsubseteq e$,*

(1) If $\bullet \vdash e : A \rightsquigarrow r$, $\lceil r \rceil \Downarrow v$, then for some B and r', we have $\bullet \vdash e' : B \rightsquigarrow r'$, and $B \sqsubseteq A$, and $\lceil r' \rceil \Downarrow v'$, and $v' \sqsubseteq v$.

(2) If $\bullet \vdash e' : B \rightsquigarrow r'$, $\lceil r' \rceil \Downarrow v'$, then for some A and r, we have $\bullet \vdash e : A \rightsquigarrow r$, and $B \sqsubseteq A$. Moreover, $\lceil r \rceil \Downarrow v$ and $v' \sqsubseteq v$, or $\lceil r \rceil \Downarrow \mathsf{blame}$.

For the example in Section 5.3, now we know that the representative translation of the right one will evaluate to 1 as well.

$$(\lambda f : \forall a.a \to \mathsf{Int}.\ \lambda x : \mathsf{Int}.\ f\ x)\ (\lambda x.\, 1)\ 3 \qquad (\lambda f : \forall a.a \to \mathsf{Int}.\ \lambda x : \star.\ f\ x)\ (\lambda x.\, 1)\ 3$$

More importantly, in what follows, we show that our extended algorithm is able to find those representative translations.

9.4 Extended Algorithmic Type System

To understand the design choices involved in the new algorithmic system, we consider the following algorithmic typing example:

$$f : \forall a.\, a \to \mathsf{Int}, x : \star \vdash f\ x \Rightarrow \mathsf{Int} \dashv f : \forall a.\, a \to \mathsf{Int}, x : \star, \widehat{a}$$

Compared with the declarative typing, where we have many choices (e.g., $\mathsf{Int} \to \mathsf{Int}$, $\mathsf{Bool} \to \mathsf{Int}$, and so on) to instantiate $\forall a.\, a \to \mathsf{Int}$, the algorithm computes the instantiation $\widehat{a} \to \mathsf{Int}$ with $\widehat{a}$

Types	A, B	$::=$	$\mathsf{Int} \mid a \mid \widehat{a} \mid A \to B \mid \forall a. A \mid \star \mid \mathcal{S} \mid \mathcal{G}$
Monotypes	τ, σ	$::=$	$\mathsf{Int} \mid a \mid \widehat{a} \mid \tau \to \sigma \mid \mathcal{S} \mid \mathcal{G}$
Existential variables	$\widehat{a}$	$::=$	$\widehat{a}_S \mid \widehat{a}_G$
Castable Types	$\mathbb{C}$	$::=$	$\mathsf{Int} \mid a \mid \widehat{a} \mid \mathbb{C}_1 \to \mathbb{C}_2 \mid \forall a. \mathbb{C} \mid \star \mid \mathcal{G}$
Castable Monotypes	t	$::=$	$\mathsf{Int} \mid a \mid \widehat{a} \mid t_1 \to t_2 \mid \mathcal{G}$
Algorithmic Contexts	Γ, Δ, Θ	$::=$	$\bullet \mid \Gamma, x : A \mid \Gamma, a \mid \Gamma, \widehat{a} \mid \Gamma, \widehat{a}_S = \tau \mid \Gamma, \widehat{a}_G = t \mid \Gamma, \blacktriangleright_{\widehat{a}}$
Complete Contexts	Ω	$::=$	$\bullet \mid \Omega, x : A \mid \Omega, a \mid \Omega, \widehat{a}_S = \tau \mid \Omega, \widehat{a}_G = t \mid \Omega, \blacktriangleright_{\widehat{a}}$

$\boxed{\Gamma \vdash A \lesssim B \dashv \Delta}$ *(Algorithmic Consistent Subtyping)*

$$\frac{}{\Gamma[a] \vdash a \lesssim a \dashv \Gamma[a]}\ \text{AS-TVAR} \qquad \frac{}{\Gamma \vdash \mathsf{Int} \lesssim \mathsf{Int} \dashv \Gamma}\ \text{AS-INT} \qquad \frac{}{\Gamma[\widehat{a}] \vdash \widehat{a} \lesssim \widehat{a} \dashv \Gamma[\widehat{a}]}\ \text{AS-EVAR}$$

$$\frac{}{\Gamma \vdash \mathcal{S} \lesssim \mathcal{S} \dashv \Gamma}\ \text{AS-SPAR} \qquad \frac{}{\Gamma \vdash \mathcal{G} \lesssim \mathcal{G} \dashv \Gamma}\ \text{AS-GPAR}$$

$$\frac{}{\Gamma \vdash \star \lesssim \mathbb{C} \dashv \mathsf{contaminate}(\Gamma, \mathbb{C})}\ \text{AS-UNKNOWNLL}$$

$$\frac{}{\Gamma \vdash \mathbb{C} \lesssim \star \dashv \mathsf{contaminate}(\Gamma, \mathbb{C})}\ \text{AS-UNKNOWNRR}$$

$$\frac{\Gamma \vdash B_1 \lesssim A_1 \dashv \Theta \qquad \Theta \vdash [\Theta]A_2 \lesssim [\Theta]B_2 \dashv \Delta}{\Gamma \vdash A_1 \to A_2 \lesssim B_1 \to B_2 \dashv \Delta}\ \text{AS-ARROW} \qquad \frac{\Gamma, a \vdash A \lesssim B \dashv \Delta, a, \Theta}{\Gamma \vdash A \lesssim \forall a. B \dashv \Delta}\ \text{AS-FORALLR}$$

$$\frac{\Gamma, \blacktriangleright_{\widehat{a}_S}, \widehat{a}_S \vdash A[a \mapsto \widehat{a}_S] \lesssim B \dashv \Delta, \blacktriangleright_{\widehat{a}_S}, \Theta}{\Gamma \vdash \forall a. A \lesssim B \dashv \Delta}\ \text{AS-FORALLLL}$$

$$\frac{\widehat{a} \notin \mathrm{FV}(A) \qquad \Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox A \dashv \Delta}{\Gamma[\widehat{a}] \vdash \widehat{a} \lesssim A \dashv \Delta}\ \text{AS-INSTL} \qquad \frac{\widehat{a} \notin \mathrm{FV}(A) \qquad \Gamma[\widehat{a}] \vdash A \lessapprox \widehat{a} \dashv \Delta}{\Gamma[\widehat{a}] \vdash A \lesssim \widehat{a} \dashv \Delta}\ \text{AS-INSTR}$$

$\mathsf{contaminate}(\bullet, A)$	$=$	$\bullet$	
$\mathsf{contaminate}((\Gamma, x : A), A)$	$=$	$\mathsf{contaminate}(\Gamma, A), x : A$	
$\mathsf{contaminate}((\Gamma, a), A)$	$=$	$\mathsf{contaminate}(\Gamma, A), a$	
$\mathsf{contaminate}((\Gamma, \widehat{a}_S), A)$	$=$	$\mathsf{contaminate}(\Gamma, A), \widehat{a}_G, \widehat{a}_S = \widehat{a}_G$	if $\widehat{a}_S$ occurs in A
$\mathsf{contaminate}((\Gamma, \widehat{a}_S), A)$	$=$	$\mathsf{contaminate}(\Gamma, A), \widehat{a}_S$	if $\widehat{a}_S$ does not occur in A
$\mathsf{contaminate}((\Gamma, \widehat{a}_G), A)$	$=$	$\mathsf{contaminate}(\Gamma, A), \widehat{a}_G$	
$\mathsf{contaminate}((\Gamma, \widehat{a} = \tau), A)$	$=$	$\mathsf{contaminate}(\Gamma, A), \widehat{a} = \tau$	
$\mathsf{contaminate}((\Gamma, \blacktriangleright_{\widehat{a}}), A)$	$=$	$\mathsf{contaminate}(\Gamma, A), \blacktriangleright_{\widehat{a}}$	

Fig. 18. Syntax of types, contexts and consistent subtyping in the extended algorithmic system.

unsolved in the output context. What can we know from the algorithmic typing? First we know that, here $\widehat{a}$ is *not constrained* by the typing problem. Second, and more importantly, $\widehat{a}$ has been compared with an unknown type (when typing $(f\ x)$). Therefore, 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;

while the second kind (as in this example) are those where the only constraint is that *the variable was once compared with an unknown type* [Garcia and Cimini 2015].

The syntax of types is shown at the top of Fig. 18. A notable difference, apart from the addition of static and gradual parameters, is that we further split existential variables $\widehat{a}$ into static existential variables $\widehat{a}_S$ and gradual existential variables $\widehat{a}_G$. Depending on whether an existential variable has been compared with $\star$ or not, its solution space changes. More specifically, static existential variables can be solved to a monotype τ, whereas gradual existential variables can only be solved to a castable monotype t, as can be seen in the changes of algorithmic contexts and complete contexts. As a result, the typing result for the above example now becomes

$$f : \forall a.\, a \to \mathsf{Int}, x : \star \vdash f\ x \Rightarrow \mathsf{Int} \dashv f : \forall a.\, a \to \mathsf{Int}, x : \star, \boxed{\widehat{a}_G}$$

since we can solve any unconstrained $\widehat{a}_G$ to $\mathcal{G}$, it is easy to verify that the resulting translation is indeed a representative translation.

Our extended algorithm is novel in the following aspects. We naturally extend the concept of existential variables [Dunfield and Krishnaswami 2013] to deal with comparisons between existential variables and unknown types. Unlike Garcia and Cimini [2015], where they use an extra set to store types that have been compared with unknown types, our two kinds of existential variables emphasize the type distinction better, and correspond more closely to the two kinds of type parameters, as we can solve $\widehat{a}_S$ to $\mathcal{S}$ and $\widehat{a}_G$ to $\mathcal{G}$.

The implementation of the algorithm can be found in the supplementary materials.

Extended Algorithmic Consistent Subtyping. While the changes in the syntax seem negligible, the addition of static and gradual type parameters changes the algorithmic judgments in a significant way. We first discuss the algorithmic consistent subtyping, which is shown at the bottom of Fig. 18. For notational convenience, when static and gradual existential variables have the same rule form, we compress them into one rule. For example, rule AS-EVAR is really two rules $\Gamma[\widehat{a}_S] \vdash \widehat{a}_S \lesssim \widehat{a}_S \dashv \Gamma[\widehat{a}_S]$ and $\Gamma[\widehat{a}_G] \vdash \widehat{a}_G \lesssim \widehat{a}_G \dashv \Gamma[\widehat{a}_G]$; same for rules AS-INSTL and AS-INSTR.

Rules AS-SPAR and AS-GPAR are direct analogies of rules CS-SPAR and CS-GPAR. Though looking simple, rules AS-UNKNOWNLL and AS-UNKNOWNRR deserve much explanation. To understand what the output context contaminate$(\Gamma, \mathbb{C})$ is for, let us first see why this seemingly intuitive rule $\Gamma \vdash \star \lesssim \mathbb{C} \dashv \Gamma$ (like rule AS-UNKNOWNL in the original algorithmic system) is wrong. Consider the judgment $\widehat{a}_S \vdash \star \lesssim \widehat{a}_S \to \widehat{a}_S \dashv \widehat{a}_S$, which seems fine. If this holds, then – since $\widehat{a}_S$ is unsolved in the output context – we can solve it to $\mathcal{S}$ for example (recall that $\widehat{a}_S$ can be solved to some monotype), resulting in $\star \lesssim \mathcal{S} \to \mathcal{S}$. However, this is in direct conflict with rule CS-UNKNOWNLL in the declarative system precisely because $\mathcal{S} \to \mathcal{S}$ is not a castable type! A possible solution would be to transform all static existential variables to gradual existential variables within $\mathbb{C}$ whenever it is being compared to $\star$: while $\widehat{a}_S \vdash \star \lesssim \widehat{a}_S \to \widehat{a}_S \dashv \widehat{a}_S$ does not hold, $\widehat{a}_G \vdash \star \lesssim \widehat{a}_G \to \widehat{a}_G \dashv \widehat{a}_G$ does. While substituting static existential variables with gradual existential variables seems to be intuitively correct, it is rather hard to formulate—not only do we need to perform substitution in $\mathbb{C}$, we also need to substitute accordingly in both the input and output contexts in order to ensure that no existential variables become unbound. However, making such changes is at odds with the interpretation of input contexts: they are "input", which evolve into output contexts with more variables solved. Therefore, in line with the use of input contexts, a simple solution is to generate a new gradual existential variable and solve the static existential variable to it in the output context, without touching $\mathbb{C}$ at all. So we have $\widehat{a}_S \vdash \star \lesssim \widehat{a}_S \to \widehat{a}_S \dashv \widehat{a}_G, \widehat{a}_S = \widehat{a}_G$.

Based on the above discussion, we show the definition of contaminate(Γ, A) in the bottom of Fig. 18, which solves all static existential variables found within A to fresh gradual existential variables in Γ. Notice the case for contaminate$((\Gamma, \widehat{a}_S), A)$ is exactly what we have just described.

$\boxed{\Gamma \vdash \widehat{a} \lessapprox A \dashv \Delta}$ *(Instantiation I)*

$$\frac{\Gamma \vdash \tau}{\Gamma, \widehat{a}_S, \Gamma' \vdash \widehat{a}_S \lessapprox \tau \dashv \Gamma, \widehat{a}_S = \tau, \Gamma'}\ \textsc{instl-solveS} \qquad \frac{\Gamma \vdash t}{\Gamma, \widehat{a}_G, \Gamma' \vdash \widehat{a}_G \lessapprox t \dashv \Gamma, \widehat{a}_G = t, \Gamma'}\ \textsc{instl-solveG}$$

$$\frac{}{\Gamma[\widehat{a}_S] \vdash \widehat{a}_S \lessapprox \star \dashv \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G]}\ \textsc{instl-solveUS} \qquad \frac{}{\Gamma[\widehat{a}_G] \vdash \widehat{a}_G \lessapprox \star \dashv \Gamma[\widehat{a}_G]}\ \textsc{instl-solveUG}$$

$$\frac{}{\Gamma[\widehat{a}_S][\widehat{b}_G] \vdash \widehat{a}_S \lessapprox \widehat{b}_G \dashv \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G][\widehat{b}_G = \widehat{a}_G]}\ \textsc{instl-reachSG1}$$

$$\frac{}{\Gamma[\widehat{b}_S][\widehat{a}_G] \vdash \widehat{a}_G \lessapprox \widehat{b}_S \dashv \Gamma[\widehat{b}_G, \widehat{b}_S = \widehat{b}_G][\widehat{a}_G = \widehat{b}_G]}\ \textsc{instl-reachSG2}$$

$$\frac{}{\Gamma[\widehat{a}][\widehat{b}] \vdash \widehat{a} \lessapprox \widehat{b} \dashv \Gamma[\widehat{a}][\widehat{b} = \widehat{a}]}\ \textsc{instl-reachOther} \qquad \frac{\Gamma[\widehat{a}], b \vdash \widehat{a} \lessapprox B \dashv \Delta, b, \Theta}{\Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox \forall b.\, B \dashv \Delta}\ \textsc{instl-forallR}$$

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

$\boxed{\Gamma \vdash A \lessapprox \widehat{a} \dashv \Delta}$ *(Instantiation II, excerpt)*

$$\frac{\Gamma[\widehat{a}], \blacktriangleright_{\widehat{b}_S}, \widehat{b}_S \vdash B[b \mapsto \widehat{b}_S] \lessapprox \widehat{a} \dashv \Delta, \blacktriangleright_{\widehat{b}_S}, \Theta}{\Gamma[\widehat{a}] \vdash \forall b.\, B \lessapprox \widehat{a} \dashv \Delta}\ \textsc{instr-forallLL}$$

Fig. 19. Instantiation in the extended algorithmic system

Rule AS-FORALLLL is slightly different from rule AS-FORALLL in the original algorithmic system in that we replace a with a new static existential variable $\widehat{a}_S$. Note that $\widehat{a}_S$ might be solved to a gradual existential variable later. The rest of the rules are the same as those in the original system.

Extended Instantiation. The instantiation judgments shown in Fig. 19 also change significantly. The complication comes from the fact that now we have two different kinds of existential variables, and the relative order they appear in the context affects their solutions.

Rules INSTL-SOLVES and INSTL-SOLVEG are the refinement to rule INSTL-SOLVE in the original system. The next two rules deal with situations where one side is an existential variable and the other side is an unknown type. Rule INSTL-SOLVEUS is a special case of rule AS-UNKNOWNRR where we create a new gradual existential variable $\widehat{a}_G$ and set the solution of $\widehat{a}_S$ to be $\widehat{a}_G$ in the output context. Rule INSTL-SOLVEUG is the same as rule INSTL-SOLVEU in the original system and simply propagates the input context. The next two rules INSTL-REACHSG1 and INSTL-REACHSG2 are a bit involved, but they both answer to the same question: how to solve a gradual existential variable when it is declared after some static existential variable. More concretely, in rule INSTL-REACHSG1, we feel that we need to solve $\widehat{b}_G$ to another existential variable. However, simply setting $\widehat{b}_G = \widehat{a}_S$ and leaving $\widehat{a}_S$ untouched in the output context is wrong. The reason is that $\widehat{b}_G$ could be a gradual existential variable created by rule AS-UNKNOWNLL/AS-UNKNOWNRR and solving $\widehat{b}_G$ to a static existential variable would result in the same problem as we have discussed. Instead, we create another new gradual existential variable $\widehat{a}_G$ and set the solutions of both $\widehat{a}_S$ and $\widehat{b}_G$ to it; similarly in rule INSTL-REACHSG2. Rule INSTL-REACHOTHER deals with the other cases (e.g., $\widehat{a}_S \lessapprox \widehat{b}_S$, $\widehat{a}_G \lessapprox \widehat{b}_G$ and so on). In those cases, we employ the same strategy as in the original system.

As for the other instantiation judgment, most of the rules are symmetric and thus omitted. The only interesting rule is INSTR-FORALLLL, which is similar to what we did for rule AS-FORALLLL.

Algorithmic Typing and Metatheory. Fortunately, the changes in the algorithmic bidirectional system are minimal: we replace every existential variable with a static existential variable. Furthermore, we proved that the extended algorithmic system is sound and complete with respect to the extended declarative system. The proofs can be found in the appendix.

Do We Really Need Type Parameters in the Algorithmic System? As we mentioned earlier, type parameters in the declarative system are merely an analysis tool, and in practice, type parameters are inaccessible to programmers. For the sake of proving soundness and completeness, we have to endow the algorithmic system with type parameters. However, the algorithmic system already has static and gradual existential variables, which can serve the same purpose. In that regard, we could directly solve every *unsolved* static and gradual existential variable in the output context to Int and $\star$, respectively.

9.5 Restricted Generalization

In Section 5.2, we discussed the issue that the translation produces multiple target expressions due to the different choices for instantiations, and those translations have different dynamic semantics. Besides that, there is another cause for multiple translations: redundant generalization during translation by rule GEN. Consider the simple expression $(\lambda x : \mathsf{Int}.\, x)\, 1$, the following shows two possible translations:

$$\bullet \vdash (\lambda x : \mathsf{Int}.\, x)\, 1 : \mathsf{Int} \rightsquigarrow (\lambda x : \mathsf{Int}.\, x)\, (\langle \mathsf{Int} \hookrightarrow \mathsf{Int} \rangle 1)$$
$$\bullet \vdash (\lambda x : \mathsf{Int}.\, x)\, 1 : \mathsf{Int} \rightsquigarrow (\lambda x : \mathsf{Int}.\, x)\, (\langle \forall a.\, \mathsf{Int} \hookrightarrow \mathsf{Int} \rangle (\Lambda a.\, 1))$$

The difference comes from the fact that in the second translation, we apply rule GEN while typing 1 to get $\bullet \vdash 1 : \forall a.\, \mathsf{Int}$. As a consequence, the translation of 1 is accompanied by a cast from $\forall a.\, \mathsf{Int}$ to Int since the former is a consistent subtype of the latter. This difference is harmless, because obviously these two expressions will reduce to the same value in λB, thus preserving coherence (up to cast error). While it is not going to break coherence, it does result in multiple representative translations for one expression (e.g., the above two translations are both the representative translations).

There are several ways to make the translation process more deterministic. For example, we can restrict generalization to happen only in let expressions and require let expressions to include annotations, as $\mathbf{let}\, x : A = e_1 \,\mathbf{in}\, e_2$. Another feasible option would be to give a declarative, bidirectional system as the specification (instead of the type assignment one), in the same spirit of Dunfield and Krishnaswami [2013]. Then we can restrict generalization to be performed through annotations in checking mode.

With restricted generalization, we hypothesize that now each expression has exactly one representative translation (up to renaming of fresh type parameters). Instead of calling it a *representative* translation, we can say it is a *principal* translation. Of course the above is only a sketch; we have not defined the corresponding rules, nor studied metatheory.

10 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 [2006] is the first to propose gradual typing, which enables programmers to mix static and dynamic typing in a program by providing a mechanism to control which parts of a program are statically checked. 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 [2007] 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. Siek et al. [2009] explores the design space of different dynamic semantics for simply typed lambda calculus with casts and unknown types. In the light of the ever-growing popularity of gradual typing, and its somewhat murky theoretical foundations, Siek et al. [2015] 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 [2016] introduced the *Gradualizer*, a general methodology for generating gradual type systems from static type systems. Later they also develop an algorithm to generate dynamic semantics [Cimini and Siek 2017]. Garcia et al. [2016] introduced the AGT approach based on abstract interpretation. As we discussed, none of these approaches instructed us how to define consistent subtyping for polymorphic types.

There is some work on integrating gradual typing with rich type disciplines. Bañados Schwerter et al. [2014] establish a framework to combine gradual typing and effects, with which a static effect system can be transformed to a dynamic effect system or any intermediate blend. Jafery and Dunfield [2017] present a type system with *gradual sums*, which combines refinement and imprecision. We have discussed the interesting definition of *directed consistency* in Section 4. Castagna and Lanvin [2017] develop a gradual type system with intersection and union types, with consistent subtyping defined by following the idea of Garcia et al. [2016]. TypeScript [Bierman et al. 2014] has a distinguished dynamic type, written any, whose fundamental feature is that any type can be implicitly converted to and from any. Our treatment of the unknown type in Fig. 8 is similar to their treatment of any. However, their type system does not have polymorphic types. Also, Unlike our consistent subtyping which inserts runtime casts, in TypeScript, type information is erased after compilation so there are no runtime casts, which makes runtime type errors possible.

Gradual Type Systems with Explicit Polymorphism. Morris [1973] dynamically enforces parametric polymorphism and uses *sealing* functions as the dynamic type mechanism. More recent works on integrating gradual typing with parametric polymorphism include the dynamic type of Abadi et al. [1995] and the *Sage* language of Gronski et al. [2006]. None of these has carefully studied the interaction between statically and dynamically typed code. Ahmed et al. [2011] proposed λB that extends the blame calculus [Wadler and Findler 2009] to incorporate polymorphism. The key novelty of their work is to use dynamic sealing to enforce parametricity. As such, they end up with a sophisticated dynamic semantics. Later, Ahmed et al. [2017] prove that with more restrictions, λB satisfies parametricity. Compared to their work, our type system can catch more errors earlier since, as we argued, their notion of *compatibility* is too permissive. For example, the following is rejected (more precisely, the corresponding source program never gets elaborated) by our type system:

$$(\lambda x : \star.\ x + 1) : \forall a.a \to a \rightsquigarrow \langle \star \to \mathsf{Int} \hookrightarrow \forall a.a \to a \rangle\ (\lambda x : \star.\ x + 1)$$

while the type system of λB would accept the translation, though at runtime, the program would result in a cast error as it violates parametricity. We emphasize that it is the combination of our powerful type system together with the powerful dynamic semantics of λB that makes it possible to have implicit higher-rank polymorphism in a gradually typed setting. Devriese et al. [2017] proved that embedding of System F terms into λB is not fully abstract. Igarashi et al. [2017] 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. On a side note, it seems that our calculus can also be safely translated to System F_C. However we do not understand all the tradeoffs involved in the choice between λB and System F_C as a target.

Gradual Type Inference. Siek and Vachharajani [2008] studied unification-based type inference for gradual typing, where they show why three straightforward approaches fail to meet their design goals. One of their main observations is that simply ignoring dynamic types during unification does not work. Therefore, their type system assigns unknown types to type variables and infers gradual types, which results in a complicated type system and inference algorithm. In our algorithm presented in Section 9, comparisons between existential variables and unknown types are emphasized by the distinction between static existential variables and gradual existential variables. By syntactically refining unsolved gradual existential variables with unknown types, we gain a similar effect as assigning unknown types, while keeping the algorithm relatively simple. Garcia and Cimini [2015] presented a new approach where gradual type inference only produces static types, which is adopted in our type system. They also deal with let-polymorphism (rank 1 types). They proposed the distinction between static and gradual type parameters, which inspired our extension to restore the dynamic gradual guarantee. Although those existing works all involve gradual types and inference, none of these works deal with higher-rank implicit polymorphism.

Higher-rank Implicit Polymorphism. Odersky and Läufer [1996] introduced a type system for higher-rank implicit polymorphic types. Based on that, Peyton Jones et al. [2007] developed an approach for type checking higher-rank predicative polymorphism. Dunfield and Krishnaswami [2013] proposed a bidirectional account of higher-rank polymorphism, and an algorithm for implementing the declarative system, which serves as the main inspiration for our algorithmic system. The key difference, however, is the integration of gradual typing. As our work, those works are in a *predicative* setting, since complete type inference for higher-rank types in an impredicative setting is undecidable. Still, there are many type systems trying to infer some impredicative types, such as ML^F [Le Botlan and Rémy 2003, 2009; Rémy and Yakobowski 2008], the HML system [Leijen 2009], the FPH system [Vytiniotis et al. 2008] and so on. Those type systems usually end up with non-standard System F types, and sophisticated forms of type inference.

11 CONCLUSION

In this paper, we have presented a generalized definition of consistent subtyping that works for polymorphic types. Based on this new definition, we have developed GPC: a gradually typed calculus with predicative implicit higher-rank polymorphism, and corresponding algorithms that can be used to implement the calculus.

As far as we know, our work is the first to integrate gradual typing with implicit (higher-rank) polymorphism, which we believe is a major step towards gradualizing modern functional languages, such as Haskell. Moreover, our extension with type parameters and the extensive discussion of related properties (e.g., representative translations) provides insight into the dynamic semantics for gradual languages with implicit polymorphism. With respect to the dynamic gradual guarantee, we discuss an extension of the calculus with static and gradual type parameters. We propose a variant of the dynamic gradual guarantee with representative translations. Then we show that our calculus supports this property if: 1) λB does indeed have the dynamic gradual guarantee (which is unknown at the time of writing); and 2) our coherence conjecture can be proved.

As future work, we want to investigate whether our notion of consistent subtyping has a more fundamental conceptual explanation, for example, whether it coincides with AGT on polymorphic types. It is also interesting to see whether our results can scale to real-world languages (e.g. Haskell) and other programming language features, such as recursive types, union types and intersection types. Recent work by Castagna and Lanvin [2017] on gradual typing with union and intersection types in a simply typed setting may shed some light on this direction.

ACKNOWLEDGEMENTS

We thank Ronald Garcia, Dustin Jamner, and the anonymous reviewers for their helpful comments. This work has been sponsored by the Hong Kong Research Grant Council projects number 17210617 and 17258816, and by the Research Foundation - Flanders.

REFERENCES

Martin Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. 1995. Dynamic Typing in Polymorphic Languages. *Journal of Functional Programming* 5, 1 (1995), 111–130.

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

Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for Free for Free: Parametricity, With and Without Types. In *Proceedings of the 22nd International Conference on Functional Programming*.

Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2014. A Theory of Gradual Effect Systems. In *Proceedings of the 19th International Conference on Functional Programming*.

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

Gavin Bierman, Erik Meijer, and Mads Torgersen. 2010. Adding Dynamic Types to C#. In *Proceedings of the European Conference on Object-Oriented Programming*.

Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. 2016. Practical Optional Types for Clojure. In *Programming Languages and Systems*.

Luca Cardelli. 1993. *An implementation of FSub*. Technical Report. Research Report 97, Digital Equipment Corporation Systems Research Center.

Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 41 (Aug. 2017), 28 pages.

Alonzo Church. 1941. *The calculi of lambda-conversion*. Number 6. Princeton University Press.

Matteo Cimini and Jeremy G. Siek. 2016. The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems. In *Proceedings of the 43rd Symposium on Principles of Programming Languages*.

Matteo Cimini and Jeremy G. Siek. 2017. Automatically Generating the Dynamic Semantics of Gradually Typed Languages. In *Proceedings of the 44th Symposium on Principles of Programming Languages*.

Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin. 1958. *Combinatory logic*. Vol. 1. North-Holland Amsterdam.

Luis Damas and Robin Milner. 1982. Principal Type-schemes for Functional Programs *(POPL '82)*. 6.

Rowan Davies and Frank Pfenning. 2000. Intersection Types and Computational Effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, New York, NY, USA, 198–208. https://doi.org/10.1145/351240.351259

Dominique Devriese, Marco Patrignani, and Frank Piessens. 2017. Parametricity versus the universal type. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 38.

Joshua Dunfield and Neelakantan R Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism. In *International Conference on Functional Programming*. https://arxiv.org/abs/1306.6032.

Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *Proceedings of the 42nd Symposium on Principles of Programming Languages*.

Ronald Garcia, Alison M Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd Symposium on Principles of Programming Languages*.

Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N Freund, and Cormac Flanagan. 2006. Sage: Hybrid Checking for Flexible Specifications. In *Scheme and Functional Programming Workshop*.

J. Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* 146 (1969), 29–60.

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On Polymorphic Gradual Typing. In *Proceedings of the 22nd International Conference on Functional Programming*.

Khurram A. Jafery and Joshua Dunfield. 2017. Sums of Uncertainty: Refinements Go Gradual. In *Proceedings of the 44th Symposium on Principles of Programming Languages*. 14.

Mark P Jones. 2000. Type classes with functional dependencies. In *European Symposium on Programming*. Springer, 230–244.

Simon Peyton Jones, Mark Jones, and Erik Meijer. 1997. Type classes: exploring the design space. In *Haskell workshop*, Vol. 1997.

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*. ACM, 96–107.

Didier Le Botlan and Didier Rémy. 2003. MLF: Raising ML to the Power of System F *(ICFP '03)*. 12.

Didier Le Botlan and Didier Rémy. 2009. Recasting MLF. *Information and Computation* 207, 6 (2009), 726–785.

Daan Leijen. 2009. Flexible Types: Robust Type Inference for First-class Polymorphism *(POPL '09)*. 12.

Jacob Matthews and Amal Ahmed. 2008. Parametric polymorphism through run-time sealing or, theorems for low, low prices!. In *European Symposium on Programming*. Springer, 16–31.

Conor McBride. 2002. Faking it Simulating dependent types in Haskell. *Journal of functional programming* 12, 4-5 (2002), 375–392.

John C Mitchell. 1990. Polymorphic Type Inference and Containment. In *Logical foundations of functional programming*.

James H. Morris, Jr. 1973. Types Are Not Sets. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '73)*. ACM, New York, NY, USA, 120–124. https://doi.org/10.1145/512927.512938

James Hiram Morris Jr. 1969. *Lambda-calculus models of programming languages.* Ph.D. Dissertation. Massachusetts Institute of Technology.

Georg Neis, Derek Dreyer, and Andreas Rossberg. 2009. Non-parametric Parametricity. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 135–148. https://doi.org/10.1145/1596550.1596572

Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *Proceedings of the 23rd Symposium on Principles of Programming Languages*.

Michel Parigot. 1992. Recursive programming with proofs. *Theoretical Computer Science* 94, 2 (1992), 335–356.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical Type Inference for Arbitrary-Rank Types. *Journal of Functional Programming* 17, 1 (2007), 1–82.

Benjamin C Pierce. 2002. *Types and programming languages.*

Didier Rémy and Boris Yakobowski. 2008. From ML to MLF: Graphic Type Constraints with Efficient Type Inference *(ICFP '08)*. 12.

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Proceedings of the IFIP 9th World Computer Congress*.

John C. Reynolds. 1991. The Coherence of Languages with Intersection Types. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software*.

Jeremy Siek, Ronald Garcia, and Walid Taha. 2009. Exploring the design space of higher-order casts. In *European Symposium on Programming*. 17–31.

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

Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *European Conference on Object-Oriented Programming*.

Jeremy G. Siek and Manish Vachharajani. 2008. Gradual Typing with Unification-based Inference. In *Proceedings of the 2008 Symposium on Dynamic Languages*.

Jeremy G. Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *LIPIcs-Leibniz International Proceedings in Informatics*.

Jeremy G. Siek and Philip Wadler. 2016. The Key to Blame: Gradual Typing Meets Cryptography (draft). (2016).

Julien Verlaguet. 2013. Facebook: Analyzing PHP statically. In *Proceedings of Commercial Users of Functional Programming*.

Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. In *Proceedings of the 10th Symposium on Dynamic languages*.

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2008. FPH: First-class Polymorphism for Haskell *(ICFP '08)*. 12.

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems*.

J.B. Wells. 1999. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic* 98, 1 (1999), 111 – 156. https://doi.org/10.1016/S0168-0072(98)00047-5

Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira. 2018. Consistent Subtyping for All. In *European Symposium on Programming*. 3–30.

A SOME PROOFS ABOUT THE DECLARATIVE SYSTEM

LEMMA 5.2. *If* $\Psi \vdash e : A \rightsquigarrow s_1$, *and* $\Psi \vdash e : A \rightsquigarrow s_2$, *then* $\lfloor s_1 \rfloor \equiv_\alpha \lfloor s_2 \rfloor$.

PROOF. By straightforward induction on the typing derivation. □

LEMMA 5.3 (COHERENCE UP TO CAST ERRORS). *For any expression* e *such that* $\Psi \vdash e : A \rightsquigarrow s_1$ *and* $\Psi \vdash e : A \rightsquigarrow s_2$, *we have* $\Psi \vdash s_1 \simeq_{ctx} s_2 : A$.

PROOF. According to Lemma 5.2, after erasure of types and casts, $\mathcal{C}\{s_1\}$ and $\mathcal{C}\{s_2\}$ are equivalent. So if $\mathcal{C}\{s_1\} \Downarrow n$, it is impossible for $\mathcal{C}\{s_2\}$ to reduce to a different integer according to the dynamic semantics. □

PROPOSITION 8.1 (EXTENSION WITH $\top$). $\Psi \vdash A \lesssim B \Leftrightarrow \Psi \vdash A <: C, C \sim D, \Psi \vdash D <: B$, *for some* C, D.

PROOF.

- From first to second: By induction on the derivation of consistent subtyping. We have extra case rule CS-Top now, where $B = \top$. We can choose $C = A$, and D by replacing the unknown types in C by Int. Namely, D is a static type, so by rule S-Top we are done.
- From second to first: By induction on the derivation of second subtyping. We have extra case rule S-Top now, where $B = \top$, so $A \lesssim B$ holds by rule CS-Top.

□

PROPOSITION 8.2 (EQUIVALENT TO AGT ON $\top$). $A \lesssim B$ *if only if* $A \mathrel{\widetilde{<:}} B$.

PROOF.

- From left to right: By induction on the derivation of consistent subtyping. We have case rule CS-Top now. It follows that for every static type $A_1 \in \gamma(A)$, we can derive $A_1 <: \top$ by rule S-Top. We have $B_1 = B = \top$ and we are done.
- From right to left: By induction on the derivation of subtyping and inversion on the concretization. We have extra case rule S-Top now, where B is $\top$. So consistent subtyping directly holds.

□

PROPOSITION 9.4. *If* $s_1 \leq s_2$ *and* $s_2 \leq s_1$, *then* s_1 *and* s_2 *are* α*-equivalent (i.e., equivalent up to renaming of type parameters).*

PROOF. Follows directly from the definition of Translation Pre-order. □

Definition A.1 (Measurements of Translation). There are three measurements of a translation s,

(1) $[\![s]\!]_{\mathcal{E}}$, the size of the expression
(2) $[\![s]\!]_{\mathcal{S}}$, the number of distinct static type parameters in s
(3) $[\![s]\!]_{\mathcal{G}}$, the number of distinct gradual type parameters in s

We use $[\![s]\!]$ to denote the lexicographical order of the triple $([\![s]\!]_{\mathcal{E}}, -[\![s]\!]_{\mathcal{S}}, -[\![s]\!]_{\mathcal{G}})$.

Definition A.2 (Size of types).

$$\begin{aligned}
[\![\mathsf{Int}]\!] &= 1 \\
[\![a]\!] &= 1 \\
[\![A \to B]\!] &= [\![A]\!] + [\![B]\!] + 1 \\
[\![\forall a.\, A]\!] &= [\![A]\!] + 1 \\
[\![\star]\!] &= 1 \\
[\![\mathcal{S}]\!] &= 1 \\
[\![\mathcal{G}]\!] &= 1
\end{aligned}$$

Definition A.3 (Size of expressions).

$$\begin{aligned}
[\![x]\!]_{\mathcal{E}} &= 1 \\
[\![n]\!]_{\mathcal{E}} &= 1 \\
[\![\lambda x : A.\, s]\!]_{\mathcal{E}} &= [\![A]\!] + [\![s]\!]_{\mathcal{E}} + 1 \\
[\![\Lambda a.\, s]\!]_{\mathcal{E}} &= [\![s]\!]_{\mathcal{E}} + 1 \\
[\![s_1\, s_2]\!]_{\mathcal{E}} &= [\![s_1]\!]_{\mathcal{E}} + [\![s_2]\!]_{\mathcal{E}} + 1 \\
[\![\langle A \hookrightarrow B \rangle s]\!]_{\mathcal{E}} &= [\![s]\!]_{\mathcal{E}} + [\![A]\!] + [\![B]\!] + 1
\end{aligned}$$

Lemma A.4. *If* $\Psi \vdash e : A \rightsquigarrow s$ *then* $[\![s]\!]_{\mathcal{E}} \geq [\![e]\!]_{\mathcal{E}}$.

Proof. Immediate by inspecting each typing rule. □

Corollary A.5. *If* $\Psi \vdash e : A \rightsquigarrow s$ *then* $[\![s]\!] > ([\![e]\!]_{\mathcal{E}}, -[\![e]\!]_{\mathcal{E}}, -[\![e]\!]_{\mathcal{E}})$.

Proof. By Lemma A.4 and note that $[\![s]\!]_{\mathcal{E}} > [\![s]\!]_{\mathcal{S}}$ and $[\![s]\!]_{\mathcal{E}} > [\![s]\!]_{\mathcal{G}}$ □

Lemma A.6. $[\![A]\!] \leq [\![S^{\mathcal{P}}(A)]\!]$.

Proof. By induction on the structure of A. The interesting cases are $A = \mathcal{S}$ and $A = \mathcal{G}$. When $A = \mathcal{S}$, $S^{\mathcal{P}}(A) = \tau$ for some monotype τ and it is immediate that $[\![\mathcal{S}]\!] \leq [\![\tau]\!]$ (note that $[\![\mathcal{S}]\!] < [\![\mathcal{G}]\!]$ by definition). □

Lemma A.7 (Substitution Decreases Measurement). *If* $s_1 \leq s_2$*, then* $[\![s_1]\!] \leq [\![s_2]\!]$*; unless* $s_2 \leq s_1$ *also holds, otherwise we have* $[\![s_1]\!] < [\![s_2]\!]$.

Proof. Since $s_1 \leq s_2$, we know $s_2 = S^{\mathcal{P}}(s_1)$ for some $S^{\mathcal{P}}$. By induction on the structure of s_1.

- Case $s_1 = \lambda x : A.\, s$. We have $s_2 = \lambda x : S^{\mathcal{P}}(A).\, S^{\mathcal{P}}(s)$. By Lemma A.6 we have $[\![A]\!] \leq [\![S^{\mathcal{P}}(A)]\!]$. By i.h., we have $[\![s]\!] \leq [\![S^{\mathcal{P}}(s)]\!]$. Therefore $[\![\lambda x : A.\, s]\!] \leq [\![\lambda x : S^{\mathcal{P}}(A).\, S^{\mathcal{P}}(s)]\!]$.
- Case $s_1 = \langle A \hookrightarrow B \rangle s$. We have $s_2 = \langle S^{\mathcal{P}}(A) \hookrightarrow S^{\mathcal{P}}(B) \rangle S^{\mathcal{P}}(s)$. By Lemma A.6 we have $[\![A]\!] \leq [\![S^{\mathcal{P}}(A)]\!]$ and $[\![B]\!] \leq [\![S^{\mathcal{P}}(B)]\!]$. By i.h., we have $[\![s]\!] \leq [\![S^{\mathcal{P}}(s)]\!]$. Therefore $[\![\langle A \hookrightarrow B \rangle s]\!] \leq [\![\langle S^{\mathcal{P}}(A) \hookrightarrow S^{\mathcal{P}}(B) \rangle S^{\mathcal{P}}(s)]\!]$.
- The rest of cases are immediate.

□

Lemma 9.9 (Representative Translation for Typing). *For any typing derivation* $\Psi \vdash e : A$ *there exists at least one representative translation* r *such that* $\Psi \vdash e : A \rightsquigarrow r$.

Proof. We already know that at least one translation $s = s_1$ exists for every typing derivation. If s_1 is a representative translation then we are done. Otherwise there exists another translation s_2 such that $s_2 \leq s_1$ and $s_1 \not\leq s_2$. By Lemma A.7, we have $[\![s_2]\!] < [\![s_1]\!]$. We continue with $s = s_2$, and get a strictly decreasing sequence $[\![s_1]\!], [\![s_2]\!], \ldots$. By Corollary A.5, we know this sequence cannot be infinite long. Suppose it ends at $[\![s_j]\!]$, by the construction of the sequence, we know that s_j is a representative translation of e. ◻

B THE EXTENDED ALGORITHMIC SYSTEM

B.1 Syntax

Expressions	e	$::=$	$x \mid n \mid \lambda x : A.\, e \mid \lambda x.\, e \mid e_1\, e_2 \mid e : A \mid \textbf{let}\, x = e_1 \,\textbf{in}\, e_2$
Types	A, B	$::=$	$\mathsf{Int} \mid a \mid \widehat{a} \mid A \rightarrow B \mid \forall a.\, A \mid \star \mid \mathcal{S} \mid \mathcal{G}$
Monotypes	τ, σ	$::=$	$\mathsf{Int} \mid a \mid \widehat{a} \mid \tau \rightarrow \sigma \mid \mathcal{S} \mid \mathcal{G}$
Existential variables	$\widehat{a}$	$::=$	$\widehat{a}_S \mid \widehat{a}_G$
Castable Types	$\mathbb{C}$	$::=$	$\mathsf{Int} \mid a \mid \widehat{a} \mid \mathbb{C}_1 \rightarrow \mathbb{C}_2 \mid \forall a.\, \mathbb{C} \mid \star \mid \mathcal{G}$
Castable Monotypes	t	$::=$	$\mathsf{Int} \mid a \mid \widehat{a} \mid t_1 \rightarrow t_2 \mid \mathcal{G}$
Algorithmic Contexts	Γ, Δ, Θ	$::=$	$\bullet \mid \Gamma, x : A \mid \Gamma, a \mid \Gamma, \widehat{a} \mid \Gamma, \widehat{a}_S = \tau \mid \Gamma, \widehat{a}_G = t \mid \Gamma, \blacktriangleright_{\widehat{a}}$
Complete Contexts	Ω	$::=$	$\bullet \mid \Omega, x : A \mid \Omega, a \mid \Omega, \widehat{a}_S = \tau \mid \Omega, \widehat{a}_G = t \mid \Omega, \blacktriangleright_{\widehat{a}}$

B.2 Type System

$\boxed{\Gamma \vdash A \lesssim B \dashv \Delta}$ *(Algorithmic Consistent Subtyping)*

$$\frac{}{\Gamma[a] \vdash a \lesssim a \dashv \Gamma[a]}\ \textsc{as-tvar} \qquad \frac{}{\Gamma[\widehat{a}] \vdash \widehat{a} \lesssim \widehat{a} \dashv \Gamma[\widehat{a}]}\ \textsc{as-evar} \qquad \frac{}{\Gamma \vdash \mathsf{Int} \lesssim \mathsf{Int} \dashv \Gamma}\ \textsc{as-int}$$

$$\frac{\Gamma \vdash B_1 \lesssim A_1 \dashv \Theta \qquad \Theta \vdash [\Theta]A_2 \lesssim [\Theta]B_2 \dashv \Delta}{\Gamma \vdash A_1 \rightarrow A_2 \lesssim B_1 \rightarrow B_2 \dashv \Delta}\ \textsc{as-arrow} \qquad \frac{\Gamma, a \vdash A \lesssim B \dashv \Delta, a, \Theta}{\Gamma \vdash A \lesssim \forall a.\, B \dashv \Delta}\ \textsc{as-forallR}$$

$$\frac{\Gamma, \blacktriangleright_{\widehat{a}_S}, \widehat{a}_S \vdash A[a \mapsto \widehat{a}_S] \lesssim B \dashv \Delta, \blacktriangleright_{\widehat{a}_S}, \Theta}{\Gamma \vdash \forall a.\, A \lesssim B \dashv \Delta}\ \textsc{as-forallLL} \qquad \frac{}{\Gamma \vdash \mathcal{S} \lesssim \mathcal{S} \dashv \Gamma}\ \textsc{as-spar}$$

$$\frac{}{\Gamma \vdash \mathcal{G} \lesssim \mathcal{G} \dashv \Gamma}\ \textsc{as-gpar} \qquad \frac{}{\Gamma \vdash \star \lesssim \mathbb{C} \dashv \mathsf{contaminate}(\Gamma, \mathbb{C})}\ \textsc{as-unknownLL}$$

$$\frac{}{\Gamma \vdash \mathbb{C} \lesssim \star \dashv \mathsf{contaminate}(\Gamma, \mathbb{C})}\ \textsc{as-unknownRR} \qquad \frac{\widehat{a} \notin \textsc{fv}(A) \qquad \Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox A \dashv \Delta}{\Gamma[\widehat{a}] \vdash \widehat{a} \lesssim A \dashv \Delta}\ \textsc{as-instL}$$

$$\frac{\widehat{a} \notin \textsc{fv}(A) \qquad \Gamma[\widehat{a}] \vdash A \lessapprox \widehat{a} \dashv \Delta}{\Gamma[\widehat{a}] \vdash A \lesssim \widehat{a} \dashv \Delta}\ \textsc{as-instR}$$

$\boxed{\Gamma \vdash \widehat{a} \lessapprox A \dashv \Delta}$ *(Instantiation I)*

$$\frac{\Gamma \vdash \tau}{\Gamma, \widehat{a}_S, \Gamma' \vdash \widehat{a}_S \lessapprox \tau \dashv \Gamma, \widehat{a}_S = \tau, \Gamma'}\ \textsc{instl-solveS} \qquad \frac{\Gamma \vdash t}{\Gamma, \widehat{a}_G, \Gamma' \vdash \widehat{a}_G \lessapprox t \dashv \Gamma, \widehat{a}_G = t, \Gamma'}\ \textsc{instl-solveG}$$

$$\frac{}{\Gamma[\widehat{a}_S] \vdash \widehat{a}_S \lessapprox \star \dashv \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G]}\ \textsc{instl-solveUS} \qquad \frac{}{\Gamma[\widehat{a}_G] \vdash \widehat{a}_G \lessapprox \star \dashv \Gamma[\widehat{a}_G]}\ \textsc{instl-solveUG}$$

$$\frac{}{\Gamma[\widehat{a}_S][\widehat{b}_G] \vdash \widehat{a}_S \lessapprox \widehat{b}_G \dashv \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G][\widehat{b}_G = \widehat{a}_G]}\ \textsc{instl-reachSG1}$$

$$\frac{}{\Gamma[\widehat{b}_S][\widehat{a}_G] \vdash \widehat{a}_G \lessapprox \widehat{b}_S \dashv \Gamma[\widehat{b}_G, \widehat{b}_S = \widehat{b}_G][\widehat{a}_G = \widehat{b}_G]}\ \textsc{instl-reachSG2}$$

$$\frac{}{\Gamma[\widehat{a}][\widehat{b}] \vdash \widehat{a} \lessapprox \widehat{b} \dashv \Gamma[\widehat{a}][\widehat{b} = \widehat{a}]}\ \textsc{instl-reachOther}$$

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

$$\frac{\Gamma[\widehat{a}], b \vdash \widehat{a} \lessapprox B \dashv \Delta, b, \Theta}{\Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox \forall b.\, B \dashv \Delta} \textsc{instl-forallR}$$

$\boxed{\Gamma \vdash A \lessapprox \widehat{a} \dashv \Delta}$ *(Instantiation II)*

$$\frac{\Gamma \vdash \tau}{\Gamma, \widehat{a}_S, \Gamma' \vdash \tau \lessapprox \widehat{a}_S \dashv \Gamma, \widehat{a}_S = \tau, \Gamma'} \textsc{instr-solveS} \qquad \frac{\Gamma \vdash t}{\Gamma, \widehat{a}_G, \Gamma' \vdash t \lessapprox \widehat{a}_G \dashv \Gamma, \widehat{a}_G = t, \Gamma'} \textsc{instr-solveG}$$

$$\frac{}{\Gamma[\widehat{a}_S] \vdash \star \lessapprox \widehat{a}_S \dashv \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G]} \textsc{instr-solveUS} \qquad \frac{}{\Gamma[\widehat{a}_G] \vdash \star \lessapprox \widehat{a}_G \dashv \Gamma[\widehat{a}_G]} \textsc{instr-solveUG}$$

$$\frac{}{\Gamma[\widehat{a}_S][\widehat{b}_G] \vdash \widehat{b}_G \lessapprox \widehat{a}_S \dashv \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G][\widehat{b}_G = \widehat{a}_G]} \textsc{instr-reachSG1}$$

$$\frac{}{\Gamma[\widehat{b}_S][\widehat{a}_G] \vdash \widehat{b}_S \lessapprox \widehat{a}_G \dashv \Gamma[\widehat{b}_G, \widehat{b}_S = \widehat{b}_G][\widehat{a}_G = \widehat{b}_G]} \textsc{instr-reachSG2}$$

$$\frac{}{\Gamma[\widehat{a}][\widehat{b}] \vdash \widehat{b} \lessapprox \widehat{a} \dashv \Gamma[\widehat{a}][\widehat{b} = \widehat{a}]} \textsc{instr-reachOther}$$

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

$$\frac{\Gamma[\widehat{a}], \blacktriangleright_{\widehat{b}_S}, \widehat{b}_S \vdash B[b \mapsto \widehat{b}_S] \lessapprox \widehat{a} \dashv \Delta, \blacktriangleright_{\widehat{b}_S}, \Theta}{\Gamma[\widehat{a}] \vdash \forall b.\, B \lessapprox \widehat{a} \dashv \Delta} \textsc{instr-forallLL}$$

$\boxed{\Gamma \vdash e \Rightarrow A \dashv \Delta}$ *(Inference)*

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A \dashv \Gamma} \textsc{inf-var} \qquad \frac{}{\Gamma \vdash n \Rightarrow \mathsf{Int} \dashv \Gamma} \textsc{inf-int}$$

$$\frac{\Gamma \vdash A \qquad \Gamma, \widehat{b}_S, x : A \vdash e \Leftarrow \widehat{b}_S \dashv \Delta, x : A, \Theta}{\Gamma \vdash \lambda x : A.\, e \Rightarrow A \to \widehat{b}_S \dashv \Delta} \textsc{inf-lamann2}$$

$$\frac{\Gamma, \widehat{a}_S, \widehat{b}_S, x : \widehat{a}_S \vdash e \Leftarrow \widehat{b}_S \dashv \Delta, x : \widehat{a}_S, \Theta}{\Gamma \vdash \lambda x.\, e \Rightarrow \widehat{a}_S \to \widehat{b}_S \dashv \Delta} \textsc{inf-lam2} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash e \Leftarrow A \dashv \Delta}{\Gamma \vdash e : A \Rightarrow A \dashv \Delta} \textsc{inf-anno}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \dashv \Theta_1 \qquad \Theta_1 \vdash [\Theta_1]A \triangleright A_1 \to A_2 \dashv \Theta_2 \qquad \Theta_2 \vdash e_2 \Leftarrow [\Theta_2]A_1 \dashv \Delta}{\Gamma \vdash e_1\, e_2 \Rightarrow A_2 \dashv \Delta} \textsc{inf-app}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \dashv \Theta_1 \qquad \Theta_1, \widehat{a}_S, x : A \vdash e_2 \Leftarrow \widehat{a}_S \dashv \Delta, x : A, \Theta_2}{\Gamma \vdash \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 \Rightarrow \widehat{a}_S \dashv \Delta} \textsc{inf-let2}$$

$\boxed{\Gamma \vdash e \Leftarrow A \dashv \Delta}$ *(Checking)*

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

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

$\boxed{\Gamma \vdash A \triangleright A_1 \rightarrow A_2 \dashv \Delta}$ *(Algorithmic Matching)*

$$\frac{\Gamma, \widehat{a}_S \vdash A[a \mapsto \widehat{a}_S] \triangleright A_1 \rightarrow A_2 \dashv \Delta}{\Gamma \vdash \forall a.\, A \triangleright A_1 \rightarrow A_2 \dashv \Delta}\ \textsc{am-forallL} \qquad \frac{}{\Gamma \vdash A_1 \rightarrow A_2 \triangleright A_1 \rightarrow A_2 \dashv \Gamma}\ \textsc{am-arr}$$

$$\frac{}{\Gamma \vdash \star \triangleright \star \rightarrow \star \dashv \Gamma}\ \textsc{am-unknown} \qquad \frac{}{\Gamma[\widehat{a}] \vdash \widehat{a} \triangleright \widehat{a}_1 \rightarrow \widehat{a}_2 \dashv \Gamma[\widehat{a}_1, \widehat{a}_2, \widehat{a} = \widehat{a}_1 \rightarrow \widehat{a}_2]}\ \textsc{am-var}$$

C DECIDABILITY

The decidability proofs mostly follow that of DK system. Whenever possible, we only show the new cases; otherwise we provide full detailed proofs.

C.1 Decidability of Instantiation

Lemma C.1 (Left Unsolvedness Preservation). *Let* $\Gamma = \Gamma_0, \widehat{a}, \Gamma_1$. *If* $\Gamma \vdash \widehat{a} \lesssim A \dashv \Delta$ *or* $\Gamma \vdash A \lesssim \widehat{a} \dashv \Delta$, *and* $\widehat{b} \in \textsc{unsolved}(\Gamma_0)$, *then* $\Delta = (\Delta_0, \widehat{b}, \Delta_1)$ *or* $\Delta = (\Delta_0, \widehat{b}', \widehat{b} = \widehat{b}', \Delta_1)$ *where* $\widehat{b}'$ *is a fresh unsolved existential.*

Proof. By induction on the given derivation. We show the new cases.

- Case

$$\frac{}{\Gamma_0, \widehat{a}_S, \Gamma_1 \vdash \widehat{a}_S \lesssim \star \dashv \Gamma_0, \widehat{a}_G, \widehat{a}_S = \widehat{a}_G, \Gamma_1}\ \textsc{instl-solve}US$$

  First notice that $\widehat{b}$ cannot be $\widehat{a}_G$. Then to the left of $\widehat{a}_S$, the contexts Δ and Γ are the same Γ_0.
- Case

$$\frac{}{\Gamma[\widehat{a}_G] \vdash \widehat{a}_G \lesssim \star \dashv \Gamma[\widehat{a}_G]}\ \textsc{instl-solve}UG$$

  Immediate, since to the left of $\widehat{a}_G$, the contexts Δ and Γ are the same.
- Case

$$\frac{}{\Gamma[\widehat{a}_S][\widehat{b}_G] \vdash \widehat{a}_S \lesssim \widehat{b}_G \dashv \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G][\widehat{b}_G = \widehat{a}_G]}\ \textsc{instl-reach}SG1$$

  First notice that $\widehat{b}$ cannot be $\widehat{a}_G$. Then to the left of $\widehat{a}_S$, the contexts Δ and Γ are the same.
- Case

$$\frac{}{\Gamma[\widehat{b}_S][\widehat{a}_G] \vdash \widehat{a}_G \lesssim \widehat{b}_S \dashv \Gamma[\widehat{b}_G, \widehat{b}_S = \widehat{b}_G][\widehat{a}_G = \widehat{b}_G]}\ \textsc{instl-reach}SG2$$

  If $\widehat{b} \neq \widehat{b}_S$, immediate, since to the left of $\widehat{a}_G$ ($\widehat{b}$ cannot be $\widehat{b}_G$), the contexts Δ and Γ are the same. Otherwise, $\widehat{b}_S$'s solution (i.e., $\widehat{b}_G$) is a fresh unsolved existential that lies just before $\widehat{b}_S$.
- Case instr-solveUS is similar to case instl-solveUS.
- Case instr-solveUG is similar to case instl-solveUG.
- Case instr-reachSG1 is similar to case instl-reachSG1.
- Case instr-reachSG2 is similar to case instl-reachSG2.

□

Lemma C.2 (Left Free Variable Preservation). *Let* $\Gamma = \Gamma_0, \widehat{a}, \Gamma_1$. *If* $\Gamma \vdash \widehat{a} \lesssim A \dashv \Delta$ *or* $\Gamma \vdash A \lesssim \widehat{a} \dashv \Delta$, *and* $\Gamma \vdash B$ *and* $\widehat{a} \notin \textsc{fv}([\Gamma]B)$ *and* $\widehat{b} \in \textsc{unsolved}(\Gamma_0)$ *and* $\widehat{b} \notin \textsc{fv}([\Gamma]B)$, *then* $\widehat{b} \notin \textsc{fv}([\Delta]B)$.

Proof. By induction on the given derivation. We show the new cases.

- Case

$$\frac{}{\Gamma[\widehat{a}_S] \vdash \widehat{a}_S \lesssim \star \dashv \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G]}\ \textsc{instl-solve}US$$

  Since Δ differs from Γ only in solving $\widehat{a}_S$ to $\widehat{a}_G$, and $\widehat{a}_G$ is fresh, applying Δ to a type will not introduce $\widehat{b}$. We have $\widehat{b} \notin \textsc{fv}([\Gamma]B)$, so $\widehat{b} \notin \textsc{fv}([\Delta]B)$.
- Case

$$\frac{}{\Gamma[\widehat{a}_G] \vdash \widehat{a}_G \lesssim \star \dashv \Gamma[\widehat{a}_G]}\ \textsc{instl-solve}UG$$

  Immediate, since Δ and Γ are the same.

- Case
$$\frac{}{\Gamma[\widehat{a}_S][\widehat{b}_G] \vdash \widehat{a}_S \lessapprox \widehat{b}_G \dashv \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G][\widehat{b}_G = \widehat{a}_G]} \text{ INSTL-REACHSG1}$$
Since Δ differs from Γ only in solving $\widehat{a}_S$ and $\widehat{b}_G$ to $\widehat{a}_G$, and $\widehat{a}_G$ is fresh, applying Δ to a type will not introduce $\widehat{b}$. We have $\widehat{b} \notin \text{FV}([\Gamma]B)$, so $\widehat{b} \notin \text{FV}([\Delta]B)$.
- Case
$$\frac{}{\Gamma[\widehat{b}_S][\widehat{a}_G] \vdash \widehat{a}_G \lessapprox \widehat{b}_S \dashv \Gamma[\widehat{b}_G, \widehat{b}_S = \widehat{b}_G][\widehat{a}_G = \widehat{b}_G]} \text{ INSTL-REACHSG2}$$
Since Δ differs from Γ only in solving $\widehat{b}_S$ and $\widehat{a}_G$ to $\widehat{b}_G$, and $\widehat{b}_G$ is fresh, applying Δ to a type will not introduce $\widehat{b}$. We have $\widehat{b} \notin \text{FV}([\Gamma]B)$, so $\widehat{b} \notin \text{FV}([\Delta]B)$.
- Case INSTR-SOLVEUS is similar to case INSTL-SOLVEUS.
- Case INSTR-SOLVEUG is similar to case INSTL-SOLVEUG.
- Case INSTR-REACHSG1 is similar to case INSTL-REACHSG1.
- Case INSTR-REACHSG2 is similar to case INSTL-REACHSG2.

□

Lemma C.3 (Instantiation Size Preservation). *If $\Gamma = \Gamma_0, \widehat{a}, \Gamma_1$ and $\Gamma \vdash \widehat{a} \lessapprox A \dashv \Delta$ or $\Gamma \vdash A \lessapprox \widehat{a} \dashv \Delta$, and $\Gamma \vdash B$ and $\widehat{a} \notin \text{FV}([\Gamma]B)$, then $|[\Gamma]B| = |[\Delta]B|$, where $|C|$ is the plain size of C.*

Proof. By induction on the given derivation. We show the new cases.

- Case
$$\frac{}{\Gamma[\widehat{a}_S] \vdash \widehat{a}_S \lessapprox \star \dashv \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G]} \text{ INSTL-SOLVEUS}$$
Since Δ differs Γ only in solving $\widehat{a}_S$, and we know $\widehat{a}_S \notin \text{FV}([\Gamma]B)$, we have $[\Delta]B = [\Gamma]B$, so $|[\Gamma]B| = |[\Delta]B|$.
- Case
$$\frac{}{\Gamma[\widehat{a}_G] \vdash \widehat{a}_G \lessapprox \star \dashv \Gamma[\widehat{a}_G]} \text{ INSTL-SOLVEUG}$$
Immediate, since Δ and Γ are the same.
- Case
$$\frac{}{\Gamma[\widehat{a}_S][\widehat{b}_G] \vdash \widehat{a}_S \lessapprox \widehat{b}_G \dashv \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G][\widehat{b}_G = \widehat{a}_G]} \text{ INSTL-REACHSG1}$$
Since Δ differs Γ only in solving $\widehat{a}_S$ and $\widehat{b}_G$, and we know $\widehat{a}_S \notin \text{FV}([\Gamma]B)$, even if $\widehat{b}_G$ occurs in $[\Gamma]B$, its solution is again an existential variable, so the size does not change, so $|[\Gamma]B| = |[\Delta]B|$.
- Case
$$\frac{}{\Gamma[\widehat{b}_S][\widehat{a}_G] \vdash \widehat{a}_G \lessapprox \widehat{b}_S \dashv \Gamma[\widehat{b}_G, \widehat{b}_S = \widehat{b}_G][\widehat{a}_G = \widehat{b}_G]} \text{ INSTL-REACHSG2}$$
Since Δ differs Γ only in solving $\widehat{a}_G$ and $\widehat{b}_S$, and we know $\widehat{a}_G \notin \text{FV}([\Gamma]B)$, even if $\widehat{b}_S$ occurs in $[\Gamma]B$, its solution is again an existential variable, so the size does not change, so $|[\Gamma]B| = |[\Delta]B|$.
- Case INSTR-SOLVEUS is similar to case INSTL-SOLVEUS.
- Case INSTR-SOLVEUG is similar to case INSTL-SOLVEUG.
- Case INSTR-REACHSG1 is similar to case INSTL-REACHSG1.
- Case INSTR-REACHSG2 is similar to case INSTL-REACHSG2.

□

Theorem 6.1 (Decidability of Instantiation). *If $\Gamma = \Gamma_0[\widehat{a}]$ and $\Gamma \vdash A$ such that $[\Gamma]A = A$ and $\widehat{a} \notin \text{FV}(A)$ then:*

(1) Either there exists Δ such that $\Gamma \vdash \widehat{a} \lessapprox A \dashv \Delta$, or not.
(2) Either there exists Δ such that $\Gamma \vdash A \lessapprox \widehat{a} \dashv \Delta$, or not.

Proof. By induction on the derivation of $\Gamma \vdash A$. We show the new cases.

- Case
$$\frac{}{\Gamma \vdash \star}\ \textsc{ad-unknown}$$
By rule instl-solveUS or rule instl-solveUG.
- Case
$$\frac{}{\Gamma \vdash \mathcal{S}}\ \textsc{ad-static}$$
By rule instl-solveS.
- Case
$$\frac{}{\Gamma \vdash \mathcal{G}}\ \textsc{ad-gradual}$$
By rule instl-solveS or rule instl-solveG.
- Case
$$\frac{}{\Gamma_0, \widehat{a}_S, \Gamma_1 \vdash \widehat{a}_G}\ \textsc{ad-evar}$$
If $\widehat{a}_G \in \Gamma_0$, then we have a derivation by rule instl-reachOther. If $\widehat{a}_G \in \Gamma_1$, then we have a derivation by rule instl-reachSG1.
- Case
$$\frac{}{\Gamma_0, \widehat{a}_G, \Gamma_1 \vdash \widehat{a}_S}\ \textsc{ad-evar}$$
If $\widehat{a}_S \in \Gamma_0$, then we have a derivation by rule instl-reachSG2. If $\widehat{a}_S \in \Gamma_1$, then we have a derivation by rule instl-reachOther.

□

C.2 Decidability of Algorithmic Consistent Subtyping

Lemma C.4 (Monotypes Solve Variables). *If $\Gamma \vdash \widehat{a} \lessapprox \tau \dashv \Delta$ or $\Gamma \vdash \tau \lessapprox \widehat{a} \dashv \Delta$, then if $[\Gamma]\tau = \tau$ and $\widehat{a} \notin$ fv$([\Gamma]\tau)$, then $|$unsolved$(\Gamma)| = |$unsolved$(\Delta)| + 1$.*

Proof. By induction on the given derivation. Since our syntax of monotypes differ from DK only in having static and gradual parameters, we show only two affected cases.

- Case
$$\frac{\Gamma \vdash \tau}{\Gamma, \widehat{a}_S, \Gamma' \vdash \widehat{a}_S \lessapprox \tau \dashv \Gamma, \widehat{a}_S = \tau, \Gamma'}\ \textsc{instl-solveS}$$
It is immediate that $|\textsc{unsolved}(\Gamma, \widehat{a}_S, \Gamma')| = |\textsc{unsolved}(\Gamma, \widehat{a}_S = \tau, \Gamma')| + 1$.
- Case
$$\frac{\Gamma \vdash t}{\Gamma, \widehat{a}_G, \Gamma' \vdash \widehat{a}_G \lessapprox t \dashv \Gamma, \widehat{a}_G = t, \Gamma'}\ \textsc{instl-solveG}$$
It is immediate that $|\textsc{unsolved}(\Gamma, \widehat{a}_G, \Gamma')| = |\textsc{unsolved}(\Gamma, \widehat{a}_G = t, \Gamma')| + 1$.

□

Lemma C.5 (Monotype Monotonicity). *If $\Gamma \vdash \tau_1 \lessapprox \tau_2 \dashv \Delta$ then $|$unsolved$(\Delta)| \leq |$unsolved$(\Gamma)|$.*

Proof. By induction on the derivation. We show the new cases.

- Case as-spar and as-gpar: In these rules, $\Delta = \Gamma$, so $|\textsc{unsolved}(\Delta)| = |\textsc{unsolved}(\Gamma)|$.

□

Lemma C.6 (Substitution Decreases Size). *If $\Gamma \vdash A$, then $|\Gamma \vdash [\Gamma]A| \leq |\Gamma \vdash A|$.*

Proof. By induction on $|\Gamma \vdash A|$. We show the new cases.

- $A = \star$, or $A = \mathcal{S}$, or $A = \mathcal{G}$ then $[\Gamma]A = A$. Therefore $|\Gamma \vdash [\Gamma]A| = |\Gamma \vdash A|$.

□

Lemma C.7 (Monotype Context Invariance). *If* $\Gamma \vdash \tau \lesssim \tau' \dashv \Delta$ *where* $[\Gamma]\tau = \tau$ *and* $[\Gamma]\tau' = \tau'$ *and* $|\textit{unsolved}(\Gamma)| = |\textit{unsolved}(\Delta)|$, *then* $\Delta = \Gamma$.

Proof. By induction on the derivation. We show the new cases.

- Cases as-spar and as-gpar: In these rules, the output context is the same as the input context, so the result is immediate.
- Case

$$\frac{\widehat{a} \notin \text{FV}(A) \qquad \Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox A \dashv \Delta}{\Gamma[\widehat{a}] \vdash \widehat{a} \lesssim A \dashv \Delta}\ \text{as-instL}$$

  By Lemma C.4, $|\text{unsolved}(\Delta)| < |\text{unsolved}(\Gamma[\widehat{a}])|$, which is contrary to what is given, so this case is impossible.
- Case as-instR is similar to as-instL.

□

Theorem 6.3 (Decidability of Algorithmic Consistent Subtyping). *Given a context* Γ *and types* A, B *such that* $\Gamma \vdash A$ *and* $\Gamma \vdash B$ *and* $[\Gamma]A = A$ *and* $[\Gamma]B = B$, *it is decidable whether there exists* Δ *such that* $\Gamma \vdash A \lesssim B \dashv \Delta$.

Proof. Let the judgment $\Gamma \vdash A \lesssim B \dashv \Delta$ be measured lexicographically by

(M1) the number of $\forall$-quantifiers in A and B;
(M2) the number of unknown types in A and B;
(M3) $|\text{unsolved}(\Gamma)|$: the number of unsolved existential variables in Γ;
(M4) $|\Gamma \vdash A| + |\Gamma \vdash B|$.

We focus on the interesting (and new) cases.

- Cases as-spar, as-gpar, as-unknownLL, and as-unknownRR have no premises.
- Case

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

We discuss each premise separately:

First premise: If A_2 or B_2 has a quantifier, then the first premise is smaller by (M1). Otherwise, if A_2 or B_2 has a unknown type, then first premise is smaller by (M2). Otherwise, the first premise shares the same input context as the conclusion, so it has the same (M3), but the types B_1 and A_1 are subterms of the conclusion's types, so the first premise is smaller by (M4).

Second premise: If B_1 or A_1 has a quantifier, then the second premise is smaller by (M1) because applying contexts will not introduce quantifiers. Otherwise, if B_1 or A_1 has a unknown type, then the second premise is smaller by (M2) because applying contexts will not introduce unknown types. Otherwise, at this point, we know B_1 and A_1 are monotypes, so by Lemma C.5 on the first premise, we have $|\text{unsolved}(\Theta)| \le |\text{unsolved}(\Gamma)|$.

- If $|\text{unsolved}(\Theta)| < |\text{unsolved}(\Gamma)|$, then the second premise is smaller by (M3).
- If $|\text{unsolved}(\Theta)| = |\text{unsolved}(\Gamma)|$, then we have the same (M3). By Lemma C.7 on the first premise, we know $\Theta = \Gamma$, so $|\Theta \vdash [\Theta]A_2| = |\Gamma \vdash [\Gamma]A_2|$. By Lemma C.6 we know $|\Gamma \vdash [\Gamma]A_2| \le |\Gamma \vdash A_2|$. Therefore we have

$$|\Theta \vdash [\Theta]A_2| \le |\Gamma \vdash A_2|$$

  Same for B_2:

$$|\Theta \vdash [\Theta]B_2| \le |\Gamma \vdash B_2|$$

Therefore,

$$|\Theta \vdash [\Theta]A_2| + |\Theta \vdash [\Theta]B_2| \leq |\Gamma \vdash A_2| + |\Gamma \vdash B_2| < |\Gamma \vdash A_1 \rightarrow A_2| + |\Gamma \vdash B_1 \rightarrow B_2|$$

and the second premise is smaller by (M4).

□

C.3 Decidability of Algorithmic Typing

Lemma 6.4 (Decidability of Algorithmic Matching). *Given a context Γ and a type A it is decidable whether there exist types A_1, A_2 and a context Δ such that $\Gamma \vdash A \triangleright A_1 \rightarrow A_2 \dashv \Delta$.*

Proof. Rules AM-ARR, AM-UNKNOWN, and AM-VAR do not have premises. For rule AM-FORALL, the size of A is decreasing in the premise. □

Theorem 6.5 (Decidability of Algorithmic Typing).

(1) Inference: Given a context Γ and a term e, it is decidable whether there exist a type A and a context Δ such that $\Gamma \vdash e \Rightarrow A \dashv \Delta$.

(2) Checking: Given a context Γ, a term e and a type B such that $\Gamma \vdash B$, it is decidable whether there exists a context Δ such that $\Gamma \vdash e \Leftarrow B \dashv \Delta$.

Proof. We consider the following measure:

$$\left\langle \quad e, \quad \begin{matrix} \Rightarrow \\ \Leftarrow, \end{matrix} \quad |\Gamma \vdash A| \quad \right\rangle$$

and show every inference/checking premise is smaller than the conclusion.

- Rules INF-VAR and INF-INT do not have premises.
- Rules INF-ANNO, INF-LAMANN, INF-LAM, INF-LET, and CHK-LAM all have strictly smaller e in the premises.
- Rule INF-APP: The first and third premises have strictly smaller e. The second (matching) judgment is decidable by Lemma 6.4.
- Rule CHK-GEN: Both the premise and conclusion type the same term, and both are the checking judgments. However $|\Gamma, a \vdash A| < |\Gamma \vdash \forall a.\, A|$, so the premise is smaller.
- Rule CHK-SUB: The first premise uses inference mode, so it is smaller. The second premise is decidable by Theorem 6.3.

□

D PROPERTIES OF CONSISTENT SUBTYPING

Lemma 7 (Consistent Subtyping is Reflexive). *If $\Psi \vdash A$ then $\Psi \vdash A \lesssim A$.*

Lemma 8 (Monotype Equality). *If $\Psi \vdash \tau \lesssim \sigma$ then $\tau = \sigma$.*

Lemma D.1 (Invertibility). *If $\Psi \vdash A \lesssim \forall b. B$ then $\Psi, b \vdash A \lesssim B$.*

Proof. By induction on the given derivation.

- Rules cs-arrow, cs-tvar, cs-int, cs-unknownRR, cs-spar, and cs-gpar are impossible since the supertype is not a forall type.
- Case

$$\frac{\Psi, a \vdash A \lesssim B}{\Psi \vdash A \lesssim \forall a. B}\ \textsc{cs-forallR}$$

The premise is exactly what we need.
- Case

$$\frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] \lesssim B}{\Psi \vdash \forall a. A \lesssim B}\ \textsc{cs-forallL}$$

where $B = \forall b. B_0$. By i.h., we have $\Psi, b \vdash A[a \mapsto \tau] \lesssim B_0$. By rule cs-forallL we have $\Psi, b \vdash \forall a. A \lesssim B_0$.
- Case

$$\frac{}{\Psi \vdash \star \lesssim \mathbb{C}}\ \textsc{cs-unknownLL}$$

where $\mathbb{C} = \forall b. \mathbb{C}_0$. By rule cs-unknownLL we have $\Psi, b \vdash \star \lesssim \mathbb{C}_0$.

□

E PROPERTIES OF CONTEXT EXTENSION

E.1 Syntactic Properties

Since the definition of the context extension judgment ($\Gamma \longrightarrow \Delta$, Fig. 16) is exactly the same as that of the DK system, we refer the reader to their technical report [Dunfield and Krishnaswami 2013] for the proofs of the following syntactic properties of context extension.

Lemma E.1 (Reverse Declaration Order Preservation). *If $\Gamma \longrightarrow \Delta$ and a and b are both declared in Γ and a is declared to the left of b in Δ, then a is declared to the left of b in Γ.*

Lemma E.2 (Reflexivity). *If Γ is well-formed then $\Gamma \longrightarrow \Gamma$.*

Lemma E.3 (Transitivity). *If $\Gamma \longrightarrow \Delta$ and $\Delta \longrightarrow \Theta$ then $\Gamma \longrightarrow \Theta$.*

Definition E.4 (Softness). A context Θ is soft iff it consists only of $\widehat{a}$ and $\widehat{a} = \tau$ declarations.

Lemma E.5 (Substitution Extension Invariance). *If $\Theta \vdash A$ and $\Theta \longrightarrow \Gamma$ then $[\Gamma]A = [\Gamma]([\Theta]A)$ and $[\Gamma]A = [\Theta]([\Gamma]A)$.*

Lemma E.6 (Extension Order). *We have the following:*

(1) If $\Gamma_L, a, \Gamma_R \longrightarrow \Delta$ then $\Delta = (\Delta_L, a, \Delta_R)$ where $\Gamma_L \longrightarrow \Delta_L$. Moreover, if Γ_R is soft then Δ_R is soft.
(2) If $\Gamma_L, \blacktriangleright_{\widehat{a}}, \Gamma_R \longrightarrow \Delta$ then $\Delta = (\Delta_L, \blacktriangleright_{\widehat{a}}, \Delta_R)$ where $\Gamma_L \longrightarrow \Delta_L$. Moreover, if Γ_R is soft then Δ_R is soft.
(3) If $\Gamma_L, \widehat{a}, \Gamma_R \longrightarrow \Delta$ then $\Delta = (\Delta_L, \Theta, \Delta_R)$ where $\Gamma_L \longrightarrow \Delta_L$ and Θ is either $\widehat{a}$ or $\widehat{a} = \tau$ for some τ.
(4) If $\Gamma_L, \widehat{a} = \tau, \Gamma_R \longrightarrow \Delta$ then $\Delta = (\Delta_L, \widehat{a} = \tau', \Delta_R)$ where $\Gamma_L \longrightarrow \Delta_L$ and and $[\Delta_L]\tau = [\Delta_L]\tau'$.
(5) If $\Gamma_L, x : A, \Gamma_R \longrightarrow \Delta$ then $\Delta = (\Delta_L, x : A', \Delta_R)$ where $\Gamma_L \longrightarrow \Delta_L$ and $[\Delta_L]A = [\Delta_L]A'$. Moreover, Γ_R is soft if and only if Δ_R is soft.

Lemma E.7 (Solution Admissibility for Extension). *If $\Gamma_L \vdash \tau$ then $\Gamma_L, \widehat{a}, \Gamma_R \longrightarrow \Gamma_L, \widehat{a} = \tau, \Gamma_R$.*

Lemma E.8 (Unsolved Variable Addition for Extension). *We have that $\Gamma_L, \Gamma_R \longrightarrow \Gamma_L, \widehat{a}, \Gamma_R$*

Lemma E.9 (Parallel Admissibility). *If $\Gamma_L \longrightarrow \Delta_L$ and $\Gamma_L, \Gamma_R \longrightarrow \Delta_L, \Delta_R$ then:*

(1) $\Gamma_L, \widehat{a}, \Gamma_R \longrightarrow \Delta_L, \widehat{a}, \Delta_R$
(2) If $\Delta_L \vdash \tau'$ then $\Gamma_L, \widehat{a}, \Gamma_R \longrightarrow \Delta_L, \widehat{a} = \tau', \Delta_R$.
(3) If $\Gamma_L \vdash \tau$ and $\Delta_L \vdash \tau'$ and $[\Delta_L]\tau = [\Delta_L]\tau'$, then $\Gamma_L, \widehat{a} = \tau, \Gamma_R \longrightarrow \Delta_L, \widehat{a} = \tau', \Delta_R$.

Lemma E.10 (Parallel Extension Solution). *If $\Gamma_L, \widehat{a}, \Gamma_R \longrightarrow \Delta_L, \widehat{a} = \tau', \Delta_R$ and $\Gamma_L \vdash \tau$ and $[\Delta_L]\tau = [\Delta_L]\tau'$, then $\Gamma_L, \widehat{a} = \tau, \Gamma_R \longrightarrow \Delta_L, \widehat{a} = \tau', \Delta_R$.*

Lemma E.11 (Drop Variable for Extension). *If $\Gamma, \widehat{a} \longrightarrow \Delta$ then $\Gamma \longrightarrow \Delta$.*

Lemma E.12 (Finishing Types). *If $\Omega \vdash A$ and $\Omega \longrightarrow \Omega'$ then $[\Omega]A = [\Omega']A$.*

Lemma E.13 (Finishing completions). *If $\Omega \longrightarrow \Omega'$ then $[\Omega]\Omega = [\Omega']\Omega'$.*

Lemma E.14 (Confluence of Completeness). *If $\Delta_1 \longrightarrow \Omega$ and $\Delta_2 \longrightarrow \Omega$ then $[\Omega]\Delta_1 = [\Omega]\Delta_2$.*

Lemma E.15 (Variable Preservation). *If $(x : A) \in \Delta$ or $(x : A) \in \Omega$ and $\Delta \longrightarrow \Omega$ then $(x : [\Omega]A) \in [\Omega]\Delta$.*

Lemma E.16 (Softness Goes Away). *If $\Delta, \Theta \longrightarrow \Omega, \Omega_Z$ where $\Delta \longrightarrow \Omega$ and Θ is soft, then $[\Omega, \Omega_Z](\Delta, \Theta) = [\Omega]\Delta$.*

Lemma E.17 (Stability of Complete Contexts). *If $\Gamma \longrightarrow \Omega$ then $[\Omega]\Gamma = [\Omega]\Omega$.*

E.2 Instantiation Extends

Lemma E.18 (Instantiation Extension). *If* $\Gamma \vdash \widehat{a} \lessapprox A \dashv \Delta$ *or* $\Gamma \vdash A \lessapprox \widehat{a} \dashv \Delta$ *then* $\Gamma \longrightarrow \Delta$.

Proof. By induction on the given instantiation derivation.

- Rules instl-solveS, instl-solveG, instl-reachOther, instr-solveS, instr-solveG, and instr-reachOther are immediate from Lemma E.7.
- Case
$$\frac{}{\Gamma[\widehat{a}_S] \vdash \widehat{a}_S \lessapprox \star \dashv \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G]}\ \textsc{instl-solveUS}$$
By Lemma E.8 we have $\Gamma[\widehat{a}_S] \longrightarrow \Gamma[\widehat{a}_G, \widehat{a}_S]$. By Lemma E.7 we have $\Gamma[\widehat{a}_G, \widehat{a}_S] \longrightarrow \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G]$. By Lemma E.3 we have $\Gamma[\widehat{a}_S] \longrightarrow \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G]$.
- Case
$$\frac{}{\Gamma[\widehat{a}_G] \vdash \widehat{a}_G \lessapprox \star \dashv \Gamma[\widehat{a}_G]}\ \textsc{instl-solveUG}$$
Immediate by Lemma E.2.
- Case
$$\frac{}{\Gamma[\widehat{a}_S][\widehat{b}_G] \vdash \widehat{a}_S \lessapprox \widehat{b}_G \dashv \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G][\widehat{b}_G = \widehat{a}_G]}\ \textsc{instl-reachSG1}$$
By Lemma E.8 we have $\Gamma[\widehat{a}_S][\widehat{b}_G] \longrightarrow \Gamma[\widehat{a}_G, \widehat{a}_S][\widehat{b}_G]$. By applying Lemma E.7 twice, we have $\Gamma[\widehat{a}_G, \widehat{a}_S][\widehat{b}_G] \longrightarrow \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G][\widehat{b}_G = \widehat{a}_G]$. By Lemma E.3 we have $\Gamma[\widehat{a}_S][\widehat{b}_G] \longrightarrow \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G][\widehat{b}_G = \widehat{a}_G]$.
- Case
$$\frac{}{\Gamma[\widehat{b}_S][\widehat{a}_G] \vdash \widehat{a}_G \lessapprox \widehat{b}_S \dashv \Gamma[\widehat{b}_G, \widehat{b}_S = \widehat{b}_G][\widehat{a}_G = \widehat{b}_G]}\ \textsc{instl-reachSG2}$$
Same as the case for rule instl-reachSG1.
- Case
$$\frac{\Gamma[\widehat{a}_2, \widehat{a}_1, \widehat{a} = \widehat{a}_1 \to \widehat{a}_2] \vdash A_1 \lessapprox \widehat{a}_1 \dashv \Theta \qquad \Theta \vdash \widehat{a}_2 \lessapprox [\Theta]A_2 \dashv \Delta}{\Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox A_1 \to A_2 \dashv \Delta}\ \textsc{instl-arr}$$
By applying Lemma E.8 twice, we have $\Gamma[\widehat{a}] \longrightarrow \Gamma[\widehat{a}_2, \widehat{a}_1, \widehat{a}]$. By Lemma E.7 we have $\Gamma[\widehat{a}_2, \widehat{a}_1, \widehat{a}] \longrightarrow \Gamma[\widehat{a}_2, \widehat{a}_1, \widehat{a} = \widehat{a}_1 \to \widehat{a}_2]$. By i.h., we have $\Gamma[\widehat{a}_2, \widehat{a}_1, \widehat{a} = \widehat{a}_1 \to \widehat{a}_2] \longrightarrow \Theta$ and $\Theta \longrightarrow \Delta$. By Lemma E.3 we have $\Gamma[\widehat{a}] \longrightarrow \Delta$.
- Case
$$\frac{\Gamma[\widehat{a}], b \vdash \widehat{a} \lessapprox B \dashv \Delta, b, \Theta}{\Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox \forall b.\, B \dashv \Delta}\ \textsc{instl-forallR}$$
By i.h., we have $\Gamma[\widehat{a}], b \longrightarrow \Delta, b, \Theta$. By Lemma E.6 (1), we have $\Gamma[\widehat{a}] \longrightarrow \Delta$.
- Case
$$\frac{}{\Gamma[\widehat{a}_S] \vdash \star \lessapprox \widehat{a}_S \dashv \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G]}\ \textsc{instr-solveUS}$$
Same as the case for rule instl-solveUS.
- Case
$$\frac{}{\Gamma[\widehat{a}_G] \vdash \star \lessapprox \widehat{a}_G \dashv \Gamma[\widehat{a}_G]}\ \textsc{instr-solveUG}$$
Same as the case for rule instl-solveUG.
- Case
$$\frac{}{\Gamma[\widehat{a}_S][\widehat{b}_G] \vdash \widehat{b}_G \lessapprox \widehat{a}_S \dashv \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G][\widehat{b}_G = \widehat{a}_G]}\ \textsc{instr-reachSG1}$$
Same as the case for rule instl-reachSG1.

- Case
$$\frac{}{\Gamma[\widehat{b}_S][\widehat{a}_G] \vdash \widehat{b}_S \lessapprox \widehat{a}_G \dashv \Gamma[\widehat{b}_G, \widehat{b}_S = \widehat{b}_G][\widehat{a}_G = \widehat{b}_G]}\ \textsc{instr-reachSG2}$$
Same as the case for rule instl-reachSG1.
- Case
$$\frac{\Gamma[\widehat{a}_2, \widehat{a}_1, \widehat{a} = \widehat{a}_1 \to \widehat{a}_2] \vdash \widehat{a}_1 \lessapprox A_1 \dashv \Theta \qquad \Theta \vdash [\Theta]A_2 \lessapprox \widehat{a}_2 \dashv \Delta}{\Gamma[\widehat{a}] \vdash A_1 \to A_2 \lessapprox \widehat{a} \dashv \Delta}\ \textsc{instr-arr}$$
Same as the case for rule instl-arr.
- Case
$$\frac{\Gamma[\widehat{a}], \blacktriangleright_{\widehat{b}_S}, \widehat{b}_S \vdash B[b \mapsto \widehat{b}_S] \lessapprox \widehat{a} \dashv \Delta, \blacktriangleright_{\widehat{b}_S}, \Theta}{\Gamma[\widehat{a}] \vdash \forall b.\, B \lessapprox \widehat{a} \dashv \Delta}\ \textsc{instr-forallLL}$$
By i.h., we have $\Gamma[\widehat{a}], \blacktriangleright_{\widehat{b}}, \widehat{b}_S \longrightarrow \Delta, \blacktriangleright_{\widehat{b}}, \Theta$. By Lemma E.6(2) we have $\Gamma[\widehat{a}] \longrightarrow \Delta$.

□

E.3 Consistent Subtyping Extends

Lemma E.19. *If* $\Gamma \vdash A$ *then* $\Gamma \longrightarrow \mathsf{contaminate}(\Gamma, A)$.

Proof. By induction on the structure of Γ. The only interesting case is when $\Gamma = \Gamma', \widehat{a}_S$. By ??, we have $\mathsf{contaminate}((\Gamma', \widehat{a}_S), A) = \mathsf{contaminate}(\Gamma', A), \widehat{a}_G, \widehat{a}_S = \widehat{a}_G$. By i.h., we have $\Gamma' \longrightarrow \mathsf{contaminate}(\Gamma', A)$. By definition of context extension we have $\Gamma', \widehat{a}_S \longrightarrow \mathsf{contaminate}(\Gamma', A), \widehat{a}_S$. By Lemma E.8 we have $\mathsf{contaminate}(\Gamma', A), \widehat{a}_S \longrightarrow \mathsf{contaminate}(\Gamma', A), \widehat{a}_G, \widehat{a}_S$. By Lemma E.7 we have $\mathsf{contaminate}(\Gamma', A), \widehat{a}_G, \widehat{a}_S \longrightarrow \mathsf{contaminate}(\Gamma', A), \widehat{a}_G, \widehat{a}_S = \widehat{a}_G$. By Lemma E.3 we have $\Gamma', \widehat{a}_S \longrightarrow \mathsf{contaminate}(\Gamma', A), \widehat{a}_G, \widehat{a}_S = \widehat{a}_G$. □

Lemma E.20 (Consistent Subtyping Extension). *If* $\Gamma \vdash A \lesssim B \dashv \Delta$ *then* $\Gamma \longrightarrow \Delta$.

Proof. By induction on the derivation of consistent subtyping.

- Rules as-tvar, as-evar, as-int, as-spar, and as-gpar are immediate from Lemma E.2.
- Case
$$\frac{\Gamma \vdash B_1 \lesssim A_1 \dashv \Theta \qquad \Theta \vdash [\Theta]A_2 \lesssim [\Theta]B_2 \dashv \Delta}{\Gamma \vdash A_1 \to A_2 \lesssim B_1 \to B_2 \dashv \Delta}\ \textsc{as-arrow}$$
By i.h., we have $\Gamma \longrightarrow \Theta$ and $\Theta \longrightarrow \Delta$. By Lemma E.3, we have $\Gamma \longrightarrow \Delta$.
- Case
$$\frac{\Gamma, a \vdash A \lesssim B \dashv \Delta, a, \Theta}{\Gamma \vdash A \lesssim \forall a.\, B \dashv \Delta}\ \textsc{as-forallR}$$
By i.h., we have $\Gamma, a \longrightarrow \Delta, a, \Theta$. By Lemma E.6 (1), we have $\Gamma \longrightarrow \Delta$.
- Case
$$\frac{\Gamma, \blacktriangleright_{\widehat{a}_S}, \widehat{a}_S \vdash A[a \mapsto \widehat{a}_S] \lesssim B \dashv \Delta, \blacktriangleright_{\widehat{a}_S}, \Theta}{\Gamma \vdash \forall a.\, A \lesssim B \dashv \Delta}\ \textsc{as-forallLL}$$
By i.h., we have $\Gamma, \blacktriangleright_{\widehat{a}}, \widehat{a}_S \longrightarrow \Delta, \blacktriangleright_{\widehat{a}}, \Theta$. By Lemma E.6 (2), we have $\Gamma \longrightarrow \Delta$.
- Case
$$\frac{}{\Gamma \vdash \star \lesssim \mathbb{C} \dashv \mathsf{contaminate}(\Gamma, \mathbb{C})}\ \textsc{as-unknownLL}$$
Immediate by Lemma E.19.
- Case
$$\frac{}{\Gamma \vdash \mathbb{C} \lesssim \star \dashv \mathsf{contaminate}(\Gamma, \mathbb{C})}\ \textsc{as-unknownRR}$$
Immediate by Lemma E.19.

- Rules AS-INSTL and AS-INSTR are immediate.

$\square$

F SOUNDNESS OF CONSISTENT SUBTYPING

Definition F.1 (Filling). The filling of a context $|\Gamma|$ solves all unsolved variables:

$$\begin{aligned} |\bullet| &= \bullet \\ |\Gamma, x : A| &= |\Gamma|, x : A \\ |\Gamma, a| &= |\Gamma|, a \\ |\Gamma, \widehat{a} = \tau| &= |\Gamma|, \widehat{a} = \tau \\ |\Gamma, \widehat{a}| &= |\Gamma|, \widehat{a} = \mathsf{Int} \\ |\Gamma, \blacktriangleright_{\widehat{a}}| &= |\Gamma|, \blacktriangleright_{\widehat{a}} \end{aligned}$$

Lemma F.2 (Substitution Stability). *For any well-formed complete context (Ω, Ω_Z), if $\Omega \vdash A$ then $[\Omega]A = [\Omega, \Omega_Z]A$.*

Proof. By induction on Ω_Z. If $\Omega_Z = \bullet$, the result is immediate. Otherwise use the i.h. and the fact that $\Omega \vdash A$ implies $\mathrm{FV}(A) \cap \mathrm{DOM}(\Omega_Z) = \emptyset$. □

Lemma F.3 (Filling Completes). *If $\Gamma \longrightarrow \Omega$ and (Γ, Θ) is well-formed, then $\Gamma, \Theta \longrightarrow \Omega, |\Theta|$.*

Proof. By induction on Θ, following Definition F.1 and applying the rules for context extension. □

Theorem 7.2 (Instantiation Soundness). *Given $\Delta \longrightarrow \Omega$ and $[\Gamma]A = A$ and $\widehat{a} \notin \mathrm{FV}(A)$:*
(1) If $\Gamma \vdash \widehat{a} \lessapprox A \dashv \Delta$ then $[\Omega]\Delta \vdash [\Omega]\widehat{a} \lesssim [\Omega]A$.
(2) If $\Gamma \vdash A \lessapprox \widehat{a} \dashv \Delta$ then $[\Omega]\Delta \vdash [\Omega]A \lesssim [\Omega]\widehat{a}$.

Proof. By induction on the given instantiation derivation.

- Case

$$\frac{\Gamma \vdash \tau}{\Gamma, \widehat{a}_S, \Gamma' \vdash \widehat{a}_S \lessapprox \tau \dashv \Gamma, \widehat{a}_S = \tau, \Gamma'}\ \textsc{instl-solveS}$$

  Immediate from $\mathcal{L}$emma 7.
- Case

$$\frac{\Gamma \vdash t}{\Gamma, \widehat{a}_G, \Gamma' \vdash \widehat{a}_G \lessapprox t \dashv \Gamma, \widehat{a}_G = t, \Gamma'}\ \textsc{instl-solveG}$$

  Immediate from $\mathcal{L}$emma 7.
- Case

$$\frac{}{\Gamma[\widehat{a}_S] \vdash \widehat{a}_S \lessapprox \star \dashv \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G]}\ \textsc{instl-solveUS}$$

  We know $[\Omega]\widehat{a}_S = t$ for some castable monotype t, and $t \in \mathbb{C}$. By rule cs-unknownRR, we have $[\Omega](\Gamma[\widehat{a}_S]) \vdash t \lesssim \star$
- Case

$$\frac{}{\Gamma[\widehat{a}_G] \vdash \widehat{a}_G \lessapprox \star \dashv \Gamma[\widehat{a}_G]}\ \textsc{instl-solveUG}$$

  Similar to the case for rule instl-solveUS.
- Case

$$\frac{}{\Gamma[\widehat{a}_S][\widehat{b}_G] \vdash \widehat{a}_S \lessapprox \widehat{b}_G \dashv \Gamma[\widehat{a}_G, \widehat{a}_S = \widehat{a}_G][\widehat{b}_G = \widehat{a}_G]}\ \textsc{instl-reachSG1}$$

  We know $[\Omega]\widehat{a}_S = [\Omega]\widehat{a}_G = t$ and $[\Omega]\widehat{b}_G = [\Omega]\widehat{a}_G = t$ for some castable monotype t. By $\mathcal{L}$emma 7 we have $[\Omega](\Gamma[\widehat{a}_S][\widehat{b}_G]) \vdash t \lesssim t$.

- Case

$$\frac{}{\Gamma[\widehat{b}_S][\widehat{a}_G] \vdash \widehat{a}_G \lessapprox \widehat{b}_S \dashv \Gamma[\widehat{b}_G, \widehat{b}_S = \widehat{b}_G][\widehat{a}_G = \widehat{b}_G]} \textsc{ instl-reachSG2}$$

Similar to the case for rule INSTL-REACHSG1.

- Case

$$\frac{}{\Gamma[\widehat{a}][\widehat{b}] \vdash \widehat{a} \lessapprox \widehat{b} \dashv \Gamma[\widehat{a}][\widehat{b} = \widehat{a}]} \textsc{ instl-reachOther}$$

Let $\Delta = \Gamma[\widehat{a}][\widehat{b}]$, we have $[\Omega]\widehat{a} = \tau$ and $[\Omega]\widehat{b} = [\Omega]\widehat{a} = \tau$ for some monotype τ. By $\mathcal{L}$emma 7 we have $[\Omega]\Delta \vdash \tau \lesssim \tau$.

- Case

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

Let $\Gamma_1 = \Gamma[\widehat{a}_2, \widehat{a}_1, \widehat{a} = \widehat{a}_1 \to \widehat{a}_2]$:

$\Theta \vdash \widehat{a}_2 \lessapprox [\Theta]A_2 \dashv \Delta$	Premise
$\Theta \longrightarrow \Delta$	By Lemma E.18
$\Delta \longrightarrow \Omega$	Given
$\Theta \longrightarrow \Omega$	By Lemma E.3
$\Gamma_1 \vdash A_1 \lessapprox \widehat{a}_1 \dashv \Theta$	Given
$[\Omega]\Delta \vdash [\Omega]A_1 \lesssim [\Omega]\widehat{a}_1$	By i.h. and Lemma E.14
$\Theta \vdash \widehat{a}_2 \lessapprox [\Theta]A_2 \dashv \Delta$	Premise
$[\Omega]\Delta \vdash [\Omega]\widehat{a}_2 \lesssim [\Omega]([\Theta]A_2)$	By i.h.
$\Theta \longrightarrow \Delta$	Above
$[\Omega]\Delta \vdash [\Omega]\widehat{a}_2 \lesssim [\Omega]A_2$	By Lemma E.5
$[\Omega]\Delta \vdash [\Omega]\widehat{a}_1 \to [\Omega]\widehat{a}_2 \lesssim [\Omega]A_1 \to [\Omega]A_2$	By rule CS-ARROW
$[\Omega]\Delta \vdash [\Omega]\widehat{a} \lesssim [\Omega](A_1 \to A_2)$	By def. of substitution

- Case

$$\frac{\Gamma[\widehat{a}], b \vdash \widehat{a} \lessapprox B \dashv \Delta, b, \Theta}{\Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox \forall b.\, B \dashv \Delta} \textsc{ instl-forallR}$$

$\Delta, b, \Theta \longrightarrow \Omega, b, \lvert\Theta\rvert$	By Lemma F.3
$\Gamma[\widehat{a}], b \vdash \widehat{a} \lessapprox B \dashv \Delta, b, \Theta$	Given
$[\Omega, b, \lvert\Theta\rvert](\Delta, b, \Theta) \vdash [\Omega, b, \lvert\Theta\rvert]\widehat{a} \lesssim [\Omega, b, \lvert\Theta\rvert]B$	By i.h.
$[\Omega, b, \lvert\Theta\rvert](\Delta, b, \Theta) \vdash [\Omega, b]\widehat{a} \lesssim [\Omega, b]B$	Free variables in $\widehat{a}$ and B are declared in (Ω, b)
$[\Omega, b](\Delta, b) \vdash [\Omega, b]\widehat{a} \lesssim [\Omega, b]B$	By context partitioning and thinning
$[\Omega]\Delta, b \vdash [\Omega]\widehat{a} \lesssim [\Omega]B$	By context substitution
$[\Omega]\Delta \vdash [\Omega]\widehat{a} \lesssim \forall b.\, [\Omega]B$	By rule CS-FORALLR
$[\Omega]\Delta \vdash [\Omega]\widehat{a} \lesssim [\Omega](\forall b.\, B)$	By def. of substitution

- Case

$$\frac{\Gamma[\widehat{a}], \blacktriangleright_{\widehat{b}_S}, \widehat{b}_S \vdash B[b \mapsto \widehat{b}_S] \lessapprox \widehat{a} \dashv \Delta, \blacktriangleright_{\widehat{b}_S}, \Theta}{\Gamma[\widehat{a}] \vdash \forall b.\, B \lessapprox \widehat{a} \dashv \Delta} \textsc{ instr-forallLL}$$

$\Delta, \blacktriangleright_{\widehat{b}}, \Theta \longrightarrow \Omega, \blacktriangleright_{\widehat{b}}, \lvert\Theta\rvert$ By Lemma F.3

$\Gamma[\widehat{a}], \blacktriangleright_{\widehat{b}}, \widehat{b}_S \vdash B[b \mapsto \widehat{b}_S] \lessapprox \widehat{a} \dashv \Delta, \blacktriangleright_{\widehat{b}}, \Theta$ Premise

$[\Omega, \blacktriangleright_{\widehat{b}}, \lvert\Theta\rvert](\Delta, \blacktriangleright_{\widehat{b}}, \Theta) \vdash [\Omega, \blacktriangleright_{\widehat{b}}, \lvert\Theta\rvert](B[b \mapsto \widehat{b}_S]) \lesssim [\Omega, \blacktriangleright_{\widehat{b}}, \lvert\Theta\rvert]\widehat{a}$ By i.h.

$[\Omega]\Delta \vdash ([\Omega]B)[b \mapsto [\Omega, \blacktriangleright_{\widehat{b}}, |\Theta|]\widehat{b}_S] \lesssim [\Omega]\widehat{a}$ By distributivity of substitution
$[\Omega]\Delta \vdash [\Omega, \blacktriangleright_{\widehat{b}}, |\Theta|]\widehat{b}_S$ Follows from def. of context application
$[\Omega]\Delta \vdash \forall b.\, [\Omega]B \lesssim [\Omega]\widehat{a}$ By rule CS-FORALLL and $[\Omega, \blacktriangleright_{\widehat{b}}, |\Theta|]\widehat{b}_S$ is a monotype
$[\Omega]\Delta \vdash [\Omega](\forall b.\, B) \lesssim [\Omega]\widehat{a}$ By def. of substitution

- The rest of the cases are similar to the above cases.

□

Theorem 7.3 (Soundness of Algorithmic Consistent Subtyping). *If* $\Gamma \vdash A \lesssim B \dashv \Delta$ *where* $[\Gamma]A = A$ *and* $[\Gamma]B = B$ *and* $\Delta \longrightarrow \Omega$ *then* $[\Omega]\Delta \vdash [\Omega]A \lesssim [\Omega]B$.

Proof. By induction on the derivation of consistent subtyping.

- Case

$$\frac{}{\Gamma[a] \vdash a \lesssim a \dashv \Gamma[a]}\ \textsc{as-tvar}$$

$a \in \Gamma[a]$	Given
$a \in [\Omega](\Gamma[a])$	Follows from def. of context application
$[\Omega](\Gamma[a]) \vdash a \lesssim a$	By rule CS-TVAR
$[\Omega](\Gamma[a]) \vdash [\Omega]a \lesssim [\Omega]a$	By def. of substitution

- Case

$$\frac{}{\Gamma[\widehat{a}] \vdash \widehat{a} \lesssim \widehat{a} \dashv \Gamma[\widehat{a}]}\ \textsc{as-evar}$$

$[\Omega]\widehat{a}$ defined	Follows from def. of context application
$[\Omega]\Delta \vdash [\Omega]\widehat{a}$	Follows from $\Delta = [\Gamma]\widehat{a}$
$[\Omega]\Delta \vdash [\Omega]\widehat{a} \lesssim [\Omega]\widehat{a}$	By $\mathcal{L}$emma 7

- Case

$$\frac{}{\Gamma \vdash \mathsf{Int} \lesssim \mathsf{Int} \dashv \Gamma}\ \textsc{as-int}$$

  Immediate.
- Case

$$\frac{\Gamma \vdash B_1 \lesssim A_1 \dashv \Theta \qquad \Theta \vdash [\Theta]A_2 \lesssim [\Theta]B_2 \dashv \Delta}{\Gamma \vdash A_1 \to A_2 \lesssim B_1 \to B_2 \dashv \Delta}\ \textsc{as-arrow}$$

$\Gamma \vdash B_1 \lesssim A_1 \dashv \Theta$	Premise
$\Delta \longrightarrow \Omega$	Given
$\Theta \longrightarrow \Omega$	By Lemma E.3
$[\Omega]\Theta \vdash [\Omega]B_1 \lesssim [\Omega]A_1$	By i.h.
$[\Omega]\Delta \vdash [\Omega]B_1 \lesssim [\Omega]A_1$	By Lemma E.14
$\Theta \vdash [\Theta]A_2 \lesssim [\Theta]B_2 \dashv \Delta$	Premise
$[\Omega]\Delta \vdash [\Omega]([\Theta]A_2) \lesssim [\Omega]([\Theta]B_2)$	By i.h.
$[\Omega]([\Theta]A_2) = [\Omega]A_2$	By Lemma E.5
$[\Omega]([\Theta]B_2) = [\Omega]B_2$	By Lemma E.5
$[\Omega]\Delta \vdash [\Omega]A_2 \lesssim [\Omega]B_2$	By above equalities
$[\Omega]\Delta \vdash [\Omega]A_1 \to [\Omega]A_2 \lesssim [\Omega]B_1 \to [\Omega]B_2$	By rule CS-ARROW
$[\Omega]\Delta \vdash [\Omega](A_1 \to A_2) \lesssim [\Omega](B_1 \to B_2)$	By def. of substitution

- Case

$$\frac{\Gamma, a \vdash A \lesssim B \dashv \Delta, a, \Theta}{\Gamma \vdash A \lesssim \forall a.\, B \dashv \Delta}\ \textsc{as-forallR}$$

$\Gamma, a \longrightarrow \Delta, a, \Theta$	By Lemma E.20
Θ is soft	By Lemma E.6 (1) where $\Gamma_R = \bullet$
$\Delta \longrightarrow \Omega$	Given
$\underbrace{\Delta, a, \Theta}_{\Delta'} \longrightarrow \underbrace{\Omega, a, \lvert\Theta\rvert}_{\Omega'}$	By Lemma F.3
$\Gamma, a \vdash A \lesssim B \dashv \Delta, a, \Theta$	Given
$[\Omega']\Delta' \vdash [\Omega']A \lesssim [\Omega']B$	By i.h.
$[\Omega']A = [\Omega, a]A$	By Lemma F.2
$[\Omega']B = [\Omega, a]B$	By Lemma F.2
$[\Omega']\Delta' = [\Omega, a](\Delta, a)$	By Lemma E.16
$[\Omega, a](\Delta, a) \vdash [\Omega, a]A \lesssim [\Omega, a]B$	By above equalities
$[\Omega]\Delta, a \vdash [\Omega]A \lesssim [\Omega]B$	By def. of substitution
$[\Omega]\Delta \vdash [\Omega]A \lesssim \forall a.\, [\Omega]B$	By rule CS-FORALLR
$[\Omega]\Delta \vdash [\Omega]A \lesssim [\Omega](\forall a.\, B)$	By def. of substitution

- Case

$$\frac{\Gamma, \blacktriangleright_{\widehat{a}_S}, \widehat{a}_S \vdash A[a \mapsto \widehat{a}_S] \lesssim B \dashv \Delta, \blacktriangleright_{\widehat{a}_S}, \Theta}{\Gamma \vdash \forall a.\, A \lesssim B \dashv \Delta}\ \textsc{as-forallLL}$$

Let $\Omega' = \Omega, \blacktriangleright_{\widehat{a}_S}, \lvert\Theta\rvert$	
$\Delta \longrightarrow \Omega$	Given
$\Delta, \blacktriangleright_{\widehat{a}_S}, \Theta \longrightarrow \Omega'$	By Lemma F.3
$\Gamma, \blacktriangleright_{\widehat{a}_S}, \widehat{a}_S \vdash A[a \mapsto \widehat{a}_S] \lesssim B \dashv \Delta, \blacktriangleright_{\widehat{a}_S}, \Theta$	Premise
$[\Omega'](\Delta, \blacktriangleright_{\widehat{a}_S}, \Theta) \vdash [\Omega'](A[a \mapsto \widehat{a}_S]) \lesssim [\Omega']B$	By i.h.
$[\Omega']B = [\Omega, a]B$	By Lemma F.2
$[\Omega](\Delta, \blacktriangleright_{\widehat{a}_S}, \Theta) \vdash [\Omega']A[a \mapsto [\Omega']\widehat{a}_S] \lesssim [\Omega]B$	By distributivity of substitution
$[\Omega](\Delta, \blacktriangleright_{\widehat{a}_S}, \Theta) \vdash [\Omega']\widehat{a}_S$	Follows from def. of context application
$[\Omega](\Delta, \blacktriangleright_{\widehat{a}_S}, \Theta) \vdash \forall a.\, [\Omega']A \lesssim [\Omega]B$	By rule CS-FORALLL
$[\Omega]\Delta \vdash \forall a.\, [\Omega]A \lesssim [\Omega]B$	By context partitioning
$[\Omega]\Delta \vdash [\Omega](\forall a.\, A) \lesssim [\Omega]B$	By def. of substitution

- Case

$$\frac{}{\Gamma \vdash \mathcal{S} \lesssim \mathcal{S} \dashv \Gamma}\ \textsc{as-spar}$$

Immediate from rule CS-SPAR.

- Case

$$\frac{}{\Gamma \vdash \mathcal{G} \lesssim \mathcal{G} \dashv \Gamma}\ \textsc{as-gpar}$$

Immediate from rule CS-GPAR.

- Case

$$\frac{}{\Gamma \vdash \star \lesssim \mathbb{C} \dashv \mathsf{contaminate}(\Gamma, \mathbb{C})}\ \textsc{as-unknownLL}$$

Immediate from rule CS-UNKNOWNLL.

- Case

$$\frac{}{\Gamma \vdash \mathbb{C} \lesssim \star \dashv \mathsf{contaminate}(\Gamma, \mathbb{C})}\ \textsc{as-unknownRR}$$

Immediate from rule CS-UNKNOWNRR.

- Case

$$\frac{\widehat{a} \notin \mathrm{FV}(A) \qquad \Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox A \dashv \Delta}{\Gamma[\widehat{a}] \vdash \widehat{a} \lesssim A \dashv \Delta}\ \textsc{as-instL}$$

$\Gamma[\widehat{a}] \vdash \widehat{a} \lessapprox A \dashv \Delta$	Premise
$[\Omega]\Delta \vdash [\Omega]\widehat{a} \lesssim [\Omega]A$	By Theorem 7.2

- Case

$$\frac{\widehat{a} \notin \mathrm{FV}(A) \qquad \Gamma[\widehat{a}] \vdash A \lessapprox \widehat{a} \dashv \Delta}{\Gamma[\widehat{a}] \vdash A \lesssim \widehat{a} \dashv \Delta}\ \textsc{as-instR}$$

Similar to the case for rule AS-INSTL.

□

G SOUNDNESS OF TYPING

Note: We use ♦ to improve readability when the conclusion has several parts.

Lemma G.1 (Matching Extension). *If* $\Gamma \vdash A \triangleright A_1 \to A_2 \dashv \Delta$ *then* $\Gamma \longrightarrow \Delta$.

Proof. By induction on the given derivation.

- Case
$$\frac{\Gamma, \widehat{a}_S \vdash A[a \mapsto \widehat{a}_S] \triangleright A_1 \to A_2 \dashv \Delta}{\Gamma \vdash \forall a.\, A \triangleright A_1 \to A_2 \dashv \Delta}\ \textsc{am-forallL}$$
By i.h., we have $\Gamma, \widehat{a}_S \longrightarrow \Delta$. By Lemma E.11, we have $\Gamma \longrightarrow \Delta$.
- Case
$$\frac{}{\Gamma \vdash A_1 \to A_2 \triangleright A_1 \to A_2 \dashv \Gamma}\ \textsc{am-arr}$$
Immediate by Lemma E.2.
- Case
$$\frac{}{\Gamma \vdash \star \triangleright \star \to \star \dashv \Gamma}\ \textsc{am-unknown}$$
Immediate by Lemma E.2.
- Case
$$\frac{}{\Gamma[\widehat{a}] \vdash \widehat{a} \triangleright \widehat{a}_1 \to \widehat{a}_2 \dashv \Gamma[\widehat{a}_1, \widehat{a}_2, \widehat{a} = \widehat{a}_1 \to \widehat{a}_2]}\ \textsc{am-var}$$
By applying Lemma E.8 twice, we have $\Gamma[\widehat{a}] \longrightarrow \Gamma[\widehat{a}_1, \widehat{a}_2, \widehat{a}]$. By Lemma E.7, we have $\Gamma[\widehat{a}_1, \widehat{a}_2, \widehat{a}] \longrightarrow \Gamma[\widehat{a}_1, \widehat{a}_2, \widehat{a} = \widehat{a}_1 \to \widehat{a}_2]$. By Lemma E.3, we have $\Gamma[\widehat{a}] \longrightarrow \Gamma[\widehat{a}_1, \widehat{a}_2, \widehat{a} = \widehat{a}_1 \to \widehat{a}_2]$.

□

Lemma G.2 (Typing Extension). *If* $\Gamma \vdash e \Rightarrow A \dashv \Delta$ *or* $\Gamma \vdash e \Leftarrow A \dashv \Delta$ *then* $\Gamma \longrightarrow \Delta$.

Proof. By induction on the given derivation.

- Case
$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A \dashv \Gamma}\ \textsc{inf-var}$$
Immediate by Lemma E.2.
- Case
$$\frac{}{\Gamma \vdash n \Rightarrow \mathsf{Int} \dashv \Gamma}\ \textsc{inf-int}$$
Immediate by Lemma E.2.
- Case
$$\frac{\Gamma, \widehat{a}_S, \widehat{b}_S, x : \widehat{a}_S \vdash e \Leftarrow \widehat{b}_S \dashv \Delta, x : \widehat{a}_S, \Theta}{\Gamma \vdash \lambda x.\, e \Rightarrow \widehat{a}_S \to \widehat{b}_S \dashv \Delta}\ \textsc{inf-lam2}$$
By i.h., we have $\Gamma, \widehat{a}_S, \widehat{b}_S, x : \widehat{a}_S \longrightarrow \Delta, x : \widehat{a}_S, \Theta$. By Lemma E.6, we have $\Gamma, \widehat{a}_S, \widehat{b}_S \longrightarrow \Delta$. By definition, we have $\Gamma \longrightarrow \Gamma, \widehat{a}_S, \widehat{b}_S$. By Lemma E.3 we have $\Gamma \longrightarrow \Delta$.
- Case
$$\frac{\Gamma \vdash A \qquad \Gamma, \widehat{b}_S, x : A \vdash e \Leftarrow \widehat{b}_S \dashv \Delta, x : A, \Theta}{\Gamma \vdash \lambda x : A.\, e \Rightarrow A \to \widehat{b}_S \dashv \Delta}\ \textsc{inf-lamann2}$$
By i.h., we have $\Gamma, \widehat{b}_S, x : A \longrightarrow \Delta, x : A, \Theta$. By Lemma E.6, we have $\Gamma \longrightarrow \Delta$.

- Case

$$\frac{\Gamma \vdash e_1 \Rightarrow A \dashv \Theta_1 \qquad \Theta_1 \vdash [\Theta_1]A \triangleright A_1 \rightarrow A_2 \dashv \Theta_2 \qquad \Theta_2 \vdash e_2 \Leftarrow [\Theta_2]A_1 \dashv \Delta}{\Gamma \vdash e_1\, e_2 \Rightarrow A_2 \dashv \Delta}\ \textsc{inf-app}$$

By i.h., we have $\Gamma \longrightarrow \Theta_1, \Theta_2 \longrightarrow \Delta$. By Lemma G.1, we have $\Theta_1 \longrightarrow \Theta_2$. By Lemma E.3 , we have $\Gamma \longrightarrow \Delta$.

- Case

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

By i.h., we have $\Gamma \longrightarrow \Delta$.

- Case

$$\frac{\Gamma, a \vdash e \Leftarrow A \dashv \Delta, a, \Theta}{\Gamma \vdash e \Leftarrow \forall a.\, A \dashv \Delta}\ \textsc{chk-gen}$$

By i.h., we have $\Gamma, a \longrightarrow \Delta, a, \Theta$. By Lemma E.6 we have $\Gamma \longrightarrow \Delta$.

- Case

$$\frac{\Gamma, x : A \vdash e \Leftarrow B \dashv \Delta, x : A, \Theta}{\Gamma \vdash \lambda x.\, e \Leftarrow A \rightarrow B \dashv \Delta}\ \textsc{chk-lam}$$

By i.h., we have $\Gamma, x : A \longrightarrow \Delta, x : A, \Theta$. By Lemma E.6 we have $\Gamma \longrightarrow \Delta$.

- Case

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

By i.h., we have $\Gamma \longrightarrow \Theta$. By Lemma E.20 we have $\Theta \longrightarrow \Delta$. By Lemma E.3 we have $\Gamma \longrightarrow \Delta$.

□

Theorem G.3 (Matching Soundness). *If* $\Gamma \vdash A \triangleright A_1 \rightarrow A_2 \dashv \Delta$ *where* $[\Gamma]A = A$ *and* $\Delta \longrightarrow \Omega$ *then* $[\Omega]\Delta \vdash [\Omega]A \triangleright [\Omega]A_1 \rightarrow [\Omega]A_2$.

Proof. By induction on the given derivation.

- Case

$$\frac{\Gamma, \widehat{a}_S \vdash A[a \mapsto \widehat{a}_S] \triangleright A_1 \rightarrow A_2 \dashv \Delta}{\Gamma \vdash \forall a.\, A \triangleright A_1 \rightarrow A_2 \dashv \Delta}\ \textsc{am-forallL}$$

$\Gamma, \widehat{a}_S \vdash A[a \mapsto \widehat{a}_S] \triangleright A_1 \rightarrow A_2 \dashv \Delta$	Premise
$\Delta \longrightarrow \Omega$	Given
$[\Omega]\Delta \vdash [\Omega](A[a \mapsto \widehat{a}_S]) \triangleright [\Omega]A_1 \rightarrow [\Omega]A_2$	By i.h.
$[\Omega]\Delta \vdash [\Omega]A[a \mapsto [\Omega]\widehat{a}_S] \triangleright [\Omega]A_1 \rightarrow [\Omega]A_2$	By distributivity of substitution
$[\Omega]\Delta \vdash [\Omega]\widehat{a}_S$	Follows from def. of context application
$[\Omega]\Delta \vdash \forall a.\, [\Omega]A \triangleright [\Omega]A_1 \rightarrow [\Omega]A_2$	By rule M-FORALL
$[\Omega]\Delta \vdash [\Omega](\forall a.\, A) \triangleright [\Omega]A_1 \rightarrow [\Omega]A_2$	By def. of substitution

- Case

$$\frac{}{\Gamma \vdash A_1 \rightarrow A_2 \triangleright A_1 \rightarrow A_2 \dashv \Gamma}\ \textsc{am-arr}$$

Immediate from rule M-ARR.

- Case

$$\frac{}{\Gamma \vdash \star \triangleright \star \rightarrow \star \dashv \Gamma}\ \textsc{am-unknown}$$

Immediate from rule M-UNKNOWN.

- Case

$$\frac{}{\Gamma[\widehat{a}] \vdash \widehat{a} \rhd \widehat{a}_1 \rightarrow \widehat{a}_2 \dashv \Gamma[\widehat{a}_1, \widehat{a}_2, \widehat{a} = \widehat{a}_1 \rightarrow \widehat{a}_2]}\ \textsc{am-var}$$

$\Delta \longrightarrow \Omega$	Given
$[\Omega]\widehat{a} = [\Omega]\widehat{a}_1 \rightarrow [\Omega]\widehat{a}_2$	By def. of context application
$[\Omega]\Delta \vdash [\Omega]\widehat{a}_1 \rightarrow [\Omega]\widehat{a}_2 \rhd [\Omega]\widehat{a}_1 \rightarrow [\Omega]\widehat{a}_2$	By rule M-ARR

□

Theorem 7.4 (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* $\lfloor e \rfloor = \lfloor e' \rfloor$*.*

Proof. By induction on the given derivation.

- Case

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A \dashv \Gamma}\ \textsc{inf-var}$$

	$(x : A) \in \Gamma$	Premise
	$(x : A) \in \Delta$	$\Delta = \Omega$
	$\Delta \longrightarrow \Omega$	Given
	$(x : [\Omega]A) \in [\Omega]\Gamma$	By Lemma E.15
♦	$[\Omega]\Gamma \vdash x : [\Omega]A$	By rule VAR
♦	$\lfloor x \rfloor = \lfloor x \rfloor$	By def. of erasure

- Case

$$\frac{}{\Gamma \vdash n \Rightarrow \mathsf{Int} \dashv \Gamma}\ \textsc{inf-int}$$

♦	$[\Omega]\Gamma \vdash n : \mathsf{Int}$	By rule INT
♦	$\lfloor n \rfloor = \lfloor n \rfloor$	By def. of erasure

- Case

$$\frac{\Gamma \vdash A \qquad \Gamma, \widehat{b}_S, x : A \vdash e \Leftarrow \widehat{b}_S \dashv \Delta, x : A, \Theta}{\Gamma \vdash \lambda x : A.\, e \Rightarrow A \rightarrow \widehat{b}_S \dashv \Delta}\ \textsc{inf-lamann2}$$

$\Gamma, \widehat{a}_S, x : A \longrightarrow \Delta, x : A, \Theta$	By Lemma G.2
Θ is soft	By Lemma E.6 where $\Gamma_R = \bullet$
$\Delta \longrightarrow \Omega$	Given
$\underbrace{\Delta, x : A, \Theta}_{\Delta'} \longrightarrow \underbrace{\Omega, x : A, \lvert\Theta\rvert}_{\Omega'}$	By Lemma F.3
$\Gamma, x : A \vdash e \Rightarrow B \dashv \Delta, x : A, \Theta$	Premise
$[\Omega']\Delta' \vdash e' : [\Omega']\widehat{b}_S$	By i.h.
$\lfloor e \rfloor = \lfloor e' \rfloor$	above
$[\Omega']\widehat{b}_S = [\Omega, x : A]\widehat{b}_S = [\Omega]\widehat{b}_S$	By Lemma F.2 and def. of substitution
$[\Omega']\Delta' = [\Omega]\Delta, x : [\Omega]A$	By Lemma E.16 and def. of context substitution
$[\Omega]\Delta, x : [\Omega]A \vdash e' : [\Omega]\widehat{b}_S$	By above equalities
$[\Omega]\Delta \vdash \lambda x : [\Omega]A.\, e' : [\Omega]A \rightarrow [\Omega]\widehat{b}_S$	By rule LAMANN
$[\Omega]A = A$	Type annotations cannot contain evars

$$
\begin{array}{ll|l}
 & [\Omega]\Delta \vdash \lambda x : A.\ e' : [\Omega]A \rightarrow [\Omega]\widehat{b}_S & \text{By above equality} \\
\blacklozenge & [\Omega]\Delta \vdash \lambda x : A.\ e' : [\Omega](A \rightarrow \widehat{b}_S) & \text{By def. of substitution} \\
\blacklozenge & \lfloor \lambda x : A.\ e' \rfloor = \lambda x.\ \lfloor e' \rfloor = \lambda x.\ \lfloor e \rfloor = \lfloor \lambda x : A.\ e \rfloor & \text{By def. of erasure}
\end{array}
$$

- Case

$$
\frac{\Gamma, \widehat{a}, \widehat{b}, x : \widehat{a} \vdash e \Leftarrow \widehat{b} \dashv \Delta, x : \widehat{a}, \Theta}{\Gamma \vdash \lambda x.\ e \Rightarrow \widehat{a} \rightarrow \widehat{b} \dashv \Delta}\ \textsc{inf-lam}
$$

$$
\begin{array}{l|l}
\Gamma, \widehat{a}_S, \widehat{b}_S, x : \widehat{a}_S \longrightarrow \Delta, x : \widehat{a}_S, \Theta & \text{By Lemma G.2} \\
\Gamma, \widehat{a}_S, \widehat{b}_S \longrightarrow \Delta & \text{By Lemma E.6} \\
\Theta \text{ is soft} & \text{Above}
\end{array}
$$

$$
\begin{array}{l|l}
\Delta \longrightarrow \Omega & \text{Given} \\
\Delta, x : \widehat{a}_S \longrightarrow \Omega, x : [\Omega]\widehat{a}_S & \text{By def} \\
\underbrace{\Delta, x : \widehat{a}_S, \Theta}_{\Delta'} \longrightarrow \underbrace{\Omega, x : [\Omega]\widehat{a}_S, |\Theta|}_{\Omega'} & \text{By Lemma F.3}
\end{array}
$$

$$
\begin{array}{ll|l}
 & \Gamma, \widehat{a}_S, \widehat{b}_S, x : \widehat{a}_S \vdash e \Leftarrow \widehat{b}_S \dashv \Delta, x : \widehat{a}_S, \Theta & \text{Premise} \\
 & [\Omega']\Delta' \vdash e' : [\Omega']\widehat{b}_S & \text{By i.h.} \\
 & \lfloor e \rfloor = \lfloor e' \rfloor & \text{Above} \\
 & [\Omega']\widehat{b}_S = [\Omega]\widehat{b}_S & \text{By def. of context substitution} \\
 & [\Omega']\Delta' = [\Omega]\Delta, x : [\Omega]\widehat{a}_S & \text{By def. of context substitution} \\
 & [\Omega]\Delta, x : [\Omega]\widehat{a}_S \vdash e' : [\Omega]\widehat{b}_S & \text{By above equalities} \\
 & [\Omega]\widehat{a}_S \text{ is a monotype} & \Omega \text{ is predicative} \\
 & [\Omega]\Delta \vdash \lambda x.\ e' : [\Omega]\widehat{a}_S \rightarrow [\Omega]\widehat{b}_S & \text{By rule } \textsc{lam} \\
\blacklozenge & [\Omega]\Delta \vdash \lambda x.\ e' : [\Omega](\widehat{a}_S \rightarrow \widehat{b}_S) & \text{By def. of substitution} \\
\blacklozenge & \lfloor \lambda x.\ e \rfloor = \lambda x.\ \lfloor e \rfloor = \lambda x.\ \lfloor e' \rfloor = \lfloor \lambda x.\ e' \rfloor & \text{By def. of erasure}
\end{array}
$$

- Case

$$
\frac{\Gamma \vdash e_1 \Rightarrow A \dashv \Theta_1 \qquad \Theta_1 \vdash [\Theta_1]A \triangleright A_1 \rightarrow A_2 \dashv \Theta_2 \qquad \Theta_2 \vdash e_2 \Leftarrow [\Theta_2]A_1 \dashv \Delta}{\Gamma \vdash e_1\ e_2 \Rightarrow A_2 \dashv \Delta}\ \textsc{inf-app}
$$

$$
\begin{array}{l|l}
\Delta \longrightarrow \Omega & \text{Given} \\
\Theta_1 \longrightarrow \Omega & \text{By Lemmas G.1 to E.3} \\
\Gamma \vdash e_1 \Rightarrow A \dashv \Theta_1 & \text{Premise} \\
[\Omega]\Theta_1 \vdash e_1' : [\Omega]A & \text{By i.h.} \\
\lfloor e_1' \rfloor = \lfloor e_1 \rfloor & \text{above} \\
[\Omega]\Theta_1 = [\Omega]\Delta & \text{By Lemma E.14} \\
[\Omega]\Delta \vdash e_1' : [\Omega]A & \text{By above equality} \\
\Theta_2 \vdash e_2 \Leftarrow [\Theta_2]A_1 \dashv \Delta & \text{Premise} \\
[\Omega]\Delta \vdash e_2' : [\Omega]A_1 & \text{By i.h.} \\
\lfloor e_2' \rfloor = \lfloor e_2 \rfloor & \text{Above} \\
\Theta_1 \vdash [\Theta_1]A \triangleright A_1 \rightarrow A_2 \dashv \Theta_2 & \text{Premise} \\
[\Omega]\Theta_2 \vdash [\Omega]([\Theta_1]A) \triangleright [\Omega]A_1 \rightarrow [\Omega]A_2 & \text{By Theorem G.3} \\
[\Omega]\Theta_2 = [\Omega]\Delta & \text{By Lemma E.14} \\
[\Omega]([\Theta_1]A) = [\Omega]A & \text{By Lemma E.5}
\end{array}
$$

	$[\Omega]\Delta \vdash [\Omega]A \triangleright [\Omega]A_1 \to [\Omega]A_2$	By above equalities
	$[\Omega]\Delta \vdash [\Omega]A_1 \lesssim [\Omega]A_1$	By $\mathcal{L}$emma 7
♦	$[\Omega]\Delta \vdash e_1' \, e_2' : [\Omega]A_2$	By rule APP
♦	$\lfloor e_1' \, e_2' \rfloor = \lfloor e_1' \rfloor \, \lfloor e_2' \rfloor = \lfloor e_1 \rfloor \, \lfloor e_2 \rfloor = \lfloor e_1 \, e_2 \rfloor$	By def. of erasure

- Case

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

	$\Gamma \vdash e \Leftarrow A \dashv \Delta$	Premise
♦	$[\Omega]\Delta \vdash e' : [\Omega]A$	By i.h.,
	$\lfloor e \rfloor = \lfloor e' \rfloor$	Above
♦	$\lfloor e : A \rfloor = \lfloor e \rfloor = \lfloor e' \rfloor$	By above equality and the def. of erasure

- Case

$$\frac{\Gamma, a \vdash e \Leftarrow A \dashv \Delta, a, \Theta}{\Gamma \vdash e \Leftarrow \forall a.\, A \dashv \Delta}\ \textsc{chk-gen}$$

	$\Delta \longrightarrow \Omega$	Given
	$\Delta, a \longrightarrow \Omega, a$	By def
	$\Gamma, a \longrightarrow \Delta, a, \Theta$	By Lemma G.2
	Θ is soft	By Lemma E.6
	$\underbrace{\Delta, a, \Theta}_{\Delta'} \longrightarrow \underbrace{\Omega, a, \lvert\Theta\rvert}_{\Omega'}$	By Lemma F.3
	$\Gamma, a \vdash e \Leftarrow A \dashv \Delta, a, \Theta$	Premise
	$[\Omega']\Delta' \vdash e' : [\Omega']A$	By i.h.,
♦	$\lfloor e \rfloor = \lfloor e' \rfloor$	Above
	$[\Omega']A = [\Omega]A$	By Lemma F.2
	$[\Omega']\Delta' = [\Omega]\Delta, a$	By Lemma E.16 and def. of context substitution
	$[\Omega]\Delta, a \vdash e' : [\Omega]A$	By above equalities
	$[\Omega]\Delta \vdash e' : \forall a.\, [\Omega]A$	By rule GEN
♦	$[\Omega]\Delta \vdash e' : [\Omega](\forall a.\, A)$	By def. of substitution

- Case

$$\frac{\Gamma, x : A \vdash e \Leftarrow B \dashv \Delta, x : A, \Theta}{\Gamma \vdash \lambda x.\, e \Leftarrow A \to B \dashv \Delta}\ \textsc{chk-lam}$$

$\Delta \longrightarrow \Omega$	Given
$\Delta, x : A \longrightarrow \Omega, x : [\Omega]A$	By def
$\Gamma, x : A \longrightarrow \Delta, x : A, \Theta$	By Lemma G.2
Θ is soft	By Lemma E.6
$\underbrace{\Delta, x : A, \Theta}_{\Delta'} \longrightarrow \underbrace{\Omega, x : [\Omega]A, \lvert\Theta\rvert}_{\Omega'}$	By Lemma F.3
$\Gamma, x : A \vdash e \Leftarrow B \dashv \Delta, x : A, \Theta$	Premise
$[\Omega']\Delta' \vdash e' : [\Omega']B$	By i.h.,
$\lfloor e \rfloor = \lfloor e' \rfloor$	Above
$[\Omega']B = [\Omega]B$	By Lemma F.2
$[\Omega']\Delta' = [\Omega]\Delta, x : [\Omega]A$	By Lemma E.16 and def. of context substitution

	$[\Omega]\Delta, x : [\Omega]A \vdash e' : [\Omega]B$	By above equalities
	$[\Omega]\Delta \vdash \lambda x : [\Omega]A.\, e' : [\Omega]A \rightarrow [\Omega]B$	By rule LAMANN
♦	$[\Omega]\Delta \vdash \lambda x : [\Omega]A.\, e' : [\Omega](A \rightarrow B)$	By def. of substitution
♦	$\lfloor \lambda x.\, e \rfloor = \lambda x.\, \lfloor e \rfloor = \lambda x.\, \lfloor e' \rfloor = \lfloor \lambda x : [\Omega]A.\, e' \rfloor$	By the def. of erasure

- Case

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

	$\Theta \vdash [\Theta]A \lesssim [\Theta]B \dashv \Delta$	Premise
	$\Theta \longrightarrow \Delta$	By Lemma E.20
	$\Delta \longrightarrow \Omega$	Given
	$\Theta \longrightarrow \Omega$	By Lemma E.3
	$\Gamma \vdash e \Rightarrow A \dashv \Theta$	Premise
	$[\Omega]\Theta \vdash e' : [\Omega]A$	By i.h.,
	$\lfloor e \rfloor = \lfloor e' \rfloor$	Above
	$[\Omega]\Theta = [\Omega]\Delta$	By Lemma E.14
	$[\Omega]\Delta \vdash e' : [\Omega]A$	By above equality
	$[\Omega]\Delta \vdash [\Omega]([\Theta]A) \lesssim [\Omega]([\Theta]B)$	By Theorem 7.3
	$[\Omega]([\Theta]A) = [\Omega]A$	By Lemma E.5
	$[\Omega]([\Theta]B) = [\Omega]B$	By Lemma E.5
	$[\Omega]\Delta \vdash [\Omega]A \lesssim [\Omega]B$	By above equalities
♦	$[\Omega]\Delta \vdash (e' : [\Omega]B) : [\Omega]B$	By def. annotation
♦	$\lfloor (e' : [\Omega]B) \rfloor = \lfloor e' \rfloor = \lfloor e \rfloor$	By def. erasure

□

H COMPLETENESS OF CONSISTENT SUBTYPING

Theorem 7.5 (Instantiation Completeness). *Given* $\Gamma \longrightarrow \Omega$ *and* $A = [\Gamma]A$ *and* $\widehat{a} \notin \textsc{unsolved}(\Gamma)$ *and* $\widehat{a} \notin \textsc{fv}(A)$*:*

(1) If $[\Omega]\Gamma \vdash [\Omega]\widehat{a} \lesssim [\Omega]A$ *then there are* Δ, Ω' *such that* $\Omega \longrightarrow \Omega'$ *and* $\Delta \longrightarrow \Omega'$ *and* $\Gamma \vdash \widehat{a} \lessapprox A \dashv \Delta$.

(2) If $[\Omega]\Gamma \vdash [\Omega]A \lesssim [\Omega]\widehat{a}$ *then there are* Δ, Ω' *such that* $\Omega \longrightarrow \Omega'$ *and* $\Delta \longrightarrow \Omega'$ *and* $\Gamma \vdash A \lessapprox \widehat{a} \dashv \Delta$.

Proof. By mutual induction on the given derivation.

(1) We have $[\Omega]\Gamma \vdash [\Omega]\widehat{a} \lesssim [\Omega]A$. We case analyze the shape of A.

- Case $A = \star, \widehat{a} = \widehat{a}_S$:

	$[\Omega]\Gamma \vdash [\Omega]\widehat{a}_S \lesssim [\Omega]\star$	Given
	$[\Omega]\star = \star$	
	$[\Omega]\Gamma \vdash [\Omega]\widehat{a}_S \lesssim \star$	By above equality
	$\widehat{a}_S \notin \textsc{unsolved}(\Gamma)$	Given
	$\Gamma = \Gamma_L, \widehat{a}_S, \Gamma_R$	Above
	$\Gamma_L, \widehat{a}_S, \Gamma_R \longrightarrow \Omega$	Given
	$\Omega = \Omega_L, \widehat{a}_S = t, \Omega_R$	By Lemma E.6 and Ω is complete and $[\Omega]\widehat{a}_S \in \mathbb{C}$
	$\Gamma_L \longrightarrow \Omega_L$	Above
	Let $\Delta = \Gamma_L, \widehat{a}_G, \widehat{a}_S = \widehat{a}_G, \Gamma_R$	
	and $\Omega' = \Omega_L, \widehat{a}_G = t, \widehat{a}_S = t, \Omega_R$	
♦	$\Gamma \vdash \widehat{a}_S \lessapprox \star \dashv \Delta$	By rule instl-solveUS
♦	$\Delta \longrightarrow \Omega'$	By Lemmas E.9 and E.10
♦	$\Omega \longrightarrow \Omega'$	By Lemmas E.7 and E.8

- Case $A = \star, \widehat{a} = \widehat{a}_G$:

	$[\Omega]\Gamma \vdash [\Omega]\widehat{a}_G \lesssim [\Omega]\star$	Given
	$[\Omega]\star = \star$	
	$[\Omega]\Gamma \vdash [\Omega]\widehat{a}_G \lesssim \star$	By above equality
	$\widehat{a}_G \notin \textsc{unsolved}(\Gamma)$	Given
	$\Gamma = \Gamma_0[\widehat{a}_G]$	Above
	Let $\Delta = \Gamma_0[\widehat{a}_G]$ and $\Omega' = \Omega$	
♦	$\Gamma \vdash \widehat{a}_G \lessapprox \star \dashv \Delta$	By rule instl-solveUG
♦	$\Delta \longrightarrow \Omega'$	Given
♦	$\Omega \longrightarrow \Omega'$	By Lemma E.2

- Case $A = \widehat{b}_G, \widehat{a} = \widehat{a}_S$:

$[\Omega]\Gamma \vdash [\Omega]\widehat{a}_S \lesssim [\Omega]\widehat{b}_G$	Given
$[\Omega]\Gamma \vdash \tau \lesssim t$	Let $[\Omega]\widehat{a}_S = \tau$ and $[\Omega]\widehat{b}_G = t$ and Ω is predicative
$\tau = t$	By $\mathcal{L}$emma 8

$[\Gamma]\widehat{b}_G = \widehat{b}_G$	Given
$\widehat{b}_G \in \textsc{unsolved}(\Gamma)$	Above

Now consider whether $\widehat{a}_S$ is declared to the left of $\widehat{b}_G$.

- Case $\Gamma = \Gamma_0, \widehat{a}_S, \Gamma_1, \widehat{b}_G, \Gamma_2$

Let $\Delta = \Gamma_0, \widehat{a}_G, \widehat{a}_S = \widehat{a}_G, \Gamma_1, \widehat{b}_G = \widehat{a}_G, \Gamma_2$	
♦ $\Gamma \vdash \widehat{a}_S \lessapprox \widehat{b}_G \dashv \Delta$	By rule INSTL-REACHSG1
$\Gamma \longrightarrow \Omega$	Given
$\Omega = \Omega_0, \widehat{a}_S = t, \Omega_1, \widehat{b}_G = t, \Omega_2$	By Lemma E.6
Let $\Omega' = \Omega_0, \widehat{a}_G = t, \widehat{a}_S = t, \Omega_1, \widehat{b}_G = t, \Omega_2$	
♦ $\Omega \longrightarrow \Omega'$	By Lemmas E.7 and E.8
♦ $\Delta \longrightarrow \Omega'$	By Lemmas E.9 and E.10

– Case $\Gamma = \Gamma_0, \widehat{b}_G, \Gamma_1, \widehat{a}_S, \Gamma_2$

Let $\Delta = \Gamma_0, \widehat{b}_G, \Gamma_1, \widehat{a}_S = \widehat{b}_G, \Gamma_2$	
♦ $\Gamma \vdash \widehat{a}_S \lessapprox \widehat{b}_G \dashv \Delta$	By rule INSTL-REACHOTHER
♦ $\Delta \longrightarrow \Omega$	By Lemma E.10
♦ $\Omega \longrightarrow \Omega$	By Lemma E.2

- Case $A = \widehat{b}_S$ is similar to the above case.
- Case $A = a$:

$[\Omega]\Gamma \vdash [\Omega]\widehat{a} \lesssim [\Omega]a$	Given
$[\Omega]\Gamma \vdash [\Omega]\widehat{a} \lesssim a$	From $[\Omega]a = a$
$[\Omega]\widehat{a} = a$	By inversion of rule CS-TVAR
a is declared to the left of $\widehat{a}$ in Ω	Ω is well-formed
$\Gamma \longrightarrow \Omega$	Given
a is declared to the left of $\widehat{a}$ in Γ	By Lemma E.1
Let $\Gamma = \Gamma_0[a][\widehat{a}]$	
Let $\Delta = \Gamma_0[a][\widehat{a} = a]$	
♦ $\Gamma \vdash \widehat{a} \lessapprox a \dashv \Delta$	By rule INSTL-SOLVES or rule INSTL-SOLVEG
♦ $\Delta \longrightarrow \Omega$	By Lemma E.10
♦ $\Omega \longrightarrow \Omega$	By Lemma E.2

- Case $A = A_1 \rightarrow A_2$:

$[\Omega]\Gamma \vdash [\Omega]\widehat{a} \lesssim [\Omega]A_1 \rightarrow [\Omega]A_2$	Given
$[\Omega]\widehat{a} = \tau_1 \rightarrow \tau_2$	Ω is predicative
$[\Omega]\Gamma \vdash [\Omega]A_1 \lesssim \tau_1$	By inversion of rule CS-ARROW
$[\Omega]\Gamma \vdash \tau_2 \lesssim [\Omega]A_2$	Above
$\Gamma = \Gamma_0[\widehat{a}]$	From $\widehat{a} \in \text{UNSOLVED}(\Gamma)$
$\Gamma_0[\widehat{a}] \longrightarrow \underbrace{\Gamma_0[\widehat{a}_2, \widehat{a}_1, \widehat{a} = \widehat{a}_1 \rightarrow \widehat{a}_2]}_{\Gamma_1}$	
$\Gamma \longrightarrow \Omega$	Given
$\Omega = \Omega_0[\widehat{a} = \tau_0]$	From $\widehat{a} \in \text{UNSOLVED}(\Gamma)$
$\Omega_0[\widehat{a} = \tau_0] \longrightarrow \underbrace{\Omega_0[\widehat{a}_2 = \tau_2, \widehat{a}_1 = \tau_1, \widehat{a} = \widehat{a}_1 \rightarrow \widehat{a}_2]}_{\Omega_1}$	

$[\Omega]\Gamma = [\Omega_1]\Gamma_1$	By Lemma E.13
$[\Omega]A_1 = [\Omega_1]A_1$	By Lemma E.12
$\tau_1 = [\Omega_1]\widehat{a}_1$	From def. of Ω_1
$[\Omega_1]\Gamma_1 \vdash [\Omega_1]A_1 \lesssim [\Omega_1]\widehat{a}_1$	By above equalities

	$\Gamma_1 \vdash A_1 \lessapprox \widehat{a}_1 \dashv \Delta_2$	By i.h.
	$\Delta_2 \longrightarrow \Omega_2$ and $\Omega_1 \longrightarrow \Omega_2$	Above
	$[\Omega]\Gamma = [\Omega_2]\Gamma_2$	By Lemma E.13
	$[\Omega]A_2 = [\Omega_2]A_2 = [\Omega_2]([\Delta_2]A_2)$	By Lemma E.12
	$\tau_2 = [\Omega_2]\widehat{a}_2$	By $\Omega_1 \longrightarrow \Omega_2$
	$[\Omega_2]\Delta_2 \vdash [\Omega_2]\widehat{a}_2 \lesssim [\Omega_2]([\Delta_2]A_2)$	By above equalities
	$\Delta_2 \vdash \widehat{a}_2 \lessapprox [\Delta_2]A_2 \dashv \Delta$	By i.h.
	$\Omega_2 \longrightarrow \Omega'$	Above
♦	$\Delta \longrightarrow \Omega'$	Above
♦	$\Gamma_0[\widehat{a}] \vdash \widehat{a} \lessapprox A_1 \to A_2 \dashv \Delta$	By rule INSTL-ARR
♦	$\Omega \longrightarrow \Omega'$	By Lemma E.3

- Case $A = \mathsf{Int}$:

$[\Omega]\Gamma \vdash [\Omega]\widehat{a} \lesssim [\Omega]\mathsf{Int}$	Given
$[\Omega]\mathsf{Int} = \mathsf{Int}$	
$[\Omega]\Gamma \vdash [\Omega]\widehat{a} \lesssim \mathsf{Int}$	By above equality
$[\Omega]\widehat{a} = \mathsf{Int}$	Ω is predicative
$\widehat{a} \in \textsc{unsolved}(\Gamma)$	Given
$\Gamma = \Gamma_0[\widehat{a}]$	Above
Let $\Delta = \Gamma_0[\widehat{a} = \mathsf{Int}]$ and $\Omega' = \Omega$	
$\Gamma_0[\widehat{a}] \vdash \widehat{a} \lessapprox \mathsf{Int} \dashv \Delta$	By rule INSTL-SOLVES or rule INSTL-SOLVEG
$\Gamma \longrightarrow \Omega$	Given
$\Gamma_0[\widehat{a} = \mathsf{Int}] \longrightarrow \Omega$	By Lemma E.10

- Case $A = \forall b.\, B$:

	$[\Omega]\Gamma \vdash [\Omega]\widehat{a} \lesssim \forall b.\, [\Omega]B$	Given
	$[\Omega]\widehat{a}$ cannot be a quantifier	Ω is predicative
	$[\Omega]\Gamma, b \vdash [\Omega]\widehat{a} \lesssim [\Omega]B$	By inversion of rule CS-FORALLR
	$[\Omega]\Gamma, b = [\Omega, b](\Gamma, b)$	By def. of context substitution
	$[\Omega]\widehat{a} = [\Omega, b]\widehat{a}$	By def. of substitution
	$[\Omega]B = [\Omega, b]B$	By def. of substitution
	$[\Omega, b](\Gamma, b) \vdash [\Omega, b]\widehat{a} \lesssim [\Omega, b]B$	By above equalities
	$\Gamma, b \vdash \widehat{a} \lessapprox B \dashv \Delta_0$	By i.h.
	$\Delta_0 \longrightarrow \Omega'$	Above
	$\Omega, b \longrightarrow \Omega'$	Above
♦	$\Omega \longrightarrow \Omega'$	By Lemma E.11
	$\Gamma, b \longrightarrow \Delta_0$	By Lemma E.18
	$\Delta_0 = \Delta, b, \Delta'$	By Lemma E.6
	$\Gamma \longrightarrow \Delta$	Above
♦	$\Delta \longrightarrow \Omega'$	
♦	$\Gamma \vdash \widehat{a} \lessapprox \forall b.\, B \dashv \Delta$	By rule INSTL-FORALLR

(2) Now we have $[\Omega]\Gamma \vdash [\Omega]A \lesssim [\Omega]\widehat{a}$. These cases are mostly symmetric. The one exception is when $A = \forall b.\, B$.

- Case $A = \forall b.\, B$:

$[\Omega]\Gamma \vdash \forall b.\, [\Omega]B \lesssim [\Omega]\widehat{a}$ Given
$[\Omega]\widehat{a}$ cannot be a quantifier Ω is predicative
$[\Omega]\Gamma \vdash \tau$ By inversion of rule CS-FORALLL
$[\Omega]\Gamma \vdash ([\Omega]B)[b \mapsto \tau] \lesssim [\Omega]\widehat{a}$ Above

$[\Omega]\Gamma = [\Omega, \blacktriangleright_{\widehat{b_S}}, \widehat{b_S} = \tau](\Gamma, \blacktriangleright_{\widehat{b_S}}, \widehat{b_S})$ By def. of context application
$([\Omega]B)[b \mapsto \tau] = [\Omega, \blacktriangleright_{\widehat{b_S}}, \widehat{b_S} = \tau](B[b \mapsto \widehat{b_S}])$ by def. of substitution
$[\Omega]\widehat{a} = [\Omega, \blacktriangleright_{\widehat{b_S}}, \widehat{b_S} = \tau]\widehat{a}$ By def. of substitution
$[\Omega, \blacktriangleright_{\widehat{b_S}}, \widehat{b_S} = \tau](\Gamma, \blacktriangleright_{\widehat{b_S}}, \widehat{b_S}) \vdash [\Omega, \blacktriangleright_{\widehat{b_S}}, \widehat{b_S} = \tau](B[b \mapsto \widehat{b_S}]) \lesssim [\Omega, \widehat{b_S} = \tau]\widehat{a}$ By above equalities
$\Gamma, \blacktriangleright_{\widehat{b_S}}, \widehat{b_S} \vdash B[b \mapsto \widehat{b_S}] \lessapprox \widehat{a} \dashv \Delta$ By i.h.
$\Gamma, \blacktriangleright_{\widehat{b_S}}, \widehat{b_S} \longrightarrow \Delta$ By Lemma E.18
$\Delta = \Delta_L, \blacktriangleright_{\widehat{b_S}}, \Delta_R$ By Lemma E.6
$\Gamma \longrightarrow \Delta_L$ Above
$\Omega, \blacktriangleright_{\widehat{b_S}}, \widehat{b_S} = \tau \longrightarrow \Omega'$ Above
$\Omega' = \Omega_L, \blacktriangleright_{\widehat{b_S}}, \Omega_R$ By Lemma E.6
♦ $\Omega \longrightarrow \Omega_L$ Above
♦ $\Delta_L \longrightarrow \Omega_L$ Lemma E.3
♦ $\Gamma \vdash \forall b.\, B \lessapprox \widehat{a} \dashv \Delta_L$ By rule INSTR-FORALLLL

□

$[\Gamma]A$ \ $[\Gamma]B$	$\forall b.B'$	$\int$	a	$\widehat{b}$	$\star$	$B_1 \to B_2$	$\mathcal{S}$	$\mathcal{G}$
$\forall a.A'$	1 (B poly)	2.Poly	2.Poly	2.Poly	1 (B unknown)	2.Poly	2.Poly	2.Poly
$\int$	1 (B poly)	2.Ints	*Impossible*	2.BEx.Int	1 (B unknown)	*Impossible*	*Impossible*	*Impossible*
a	1 (B poly)	*Impossible*	2.UVars	2.BEx.UVar	1 (B unknown)	*Impossible*	*Impossible*	*Impossible*
$\widehat{a}$	1 (B poly)	2.AEx.Int	2.AEx.UVar	2.AEx.SameEx 2.AEx.OtherEx	1 (B unknown)	2.AEx.Arrow	2.AEx.S	2.AEx.G
$\star$	1 (B poly)	2.Unknown	2.Unknown	2.Unknown	1 (B unknown)	2.Unknown	*Impossible*	2.Unknown
$A_1 \to A_2$	1 (B poly)	*Impossible*	*Impossible*	2.BEx.Arrow	1 (B unknown)	2.Arrows	*Impossible*	*Impossible*
$\mathcal{S}$	1 (B poly)	*Impossible*	*Impossible*	2.BEx.S	*Impossible*	*Impossible*	2.S	*Impossible*
$\mathcal{G}$	1 (B poly)	*Impossible*	*Impossible*	2.BEx.G	1 (B unknown)	*Impossible*	*Impossible*	2.G

Table 31. List of cases

THEOREM 7.6 (GENERALIZED COMPLETENESS OF CONSISTENT SUBTYPING). *If* $\Gamma \longrightarrow \Omega$ *and* $\Gamma \vdash A$ *and* $\Gamma \vdash B$ *and* $[\Omega]\Gamma \vdash [\Omega]A \lesssim [\Omega]B$ *then there exist* Δ *and* Ω' *such that* $\Delta \longrightarrow \Omega'$ *and* $\Omega \longrightarrow \Omega'$ *and* $\Gamma \vdash [\Gamma]A \lesssim [\Gamma]B \dashv \Delta$.

PROOF. By induction on the given declarative derivation. We list all the possible cases in Table 31. We first split on$[\Gamma]B$.

- Case 1 (B poly) : $[\Gamma]B$ is polymorphic: $[\Gamma]B = \forall b.\, B'$:

$B = \forall b.\, B_0$	Γ is predicative
$B' = [\Gamma]B_0$	Γ is predicative
$[\Omega]B = \forall b.\, [\Omega]B_0$	By def. of substitution
$[\Omega]\Gamma \vdash [\Omega]A \lesssim [\Omega]B$	Premise
$[\Omega]\Gamma \vdash [\Omega]A \lesssim \forall b.\, [\Omega]B_0$	By above equality
$[\Omega]\Gamma, b \vdash [\Omega]A \lesssim [\Omega]B_0$	By Lemma D.1
$[\Omega]\Gamma, b = [\Omega, b](\Gamma, b)$	By def. of substitution
$[\Omega]A = [\Omega, b]A$	By def. of substitution
$[\Omega]B = [\Omega, b]B$	By def. of substitution
$[\Omega, b](\Gamma, b) \vdash [\Omega, b]A \lesssim [\Omega, b]B_0$	By above equalities
$\Gamma, b \vdash [\Gamma, b]A \lesssim [\Gamma, b]B_0 \dashv \Delta'$	By i.h.
$\Delta' \longrightarrow \Omega'_0$	Above
$\Omega, b \longrightarrow \Omega'_0$	Above
$\Gamma, b \vdash [\Gamma]A \lesssim [\Gamma]B_0 \dashv \Delta'$	By def. of substitution

	$\Gamma, b \longrightarrow \Delta'$	By Lemma E.18
	$\Delta' = \Delta, b, \Theta$	By Lemma E.6
	$\Gamma \longrightarrow \Delta$	Above
	$\Delta, b, \Theta \longrightarrow \Omega'_0$	By $\Delta' \longrightarrow \Omega'_0$ and above equality
	$\Omega'_0 = \Omega', b, \Omega_R$	By Lemma E.6
♦	$\Delta \longrightarrow \Omega'$	Above
	$\Omega, b \longrightarrow \Omega', b, \Omega_R$	By above equality
♦	$\Omega \longrightarrow \Omega'$	By Lemma E.6

	$\Gamma, b \vdash [\Gamma]A \lesssim [\Gamma]B_0 \dashv \Delta, b, \Theta$	By above equality
	$\Gamma \vdash [\Gamma]A \lesssim \forall b.\, [\Gamma]B_0 \dashv \Delta$	By rule AS-FORALLR
♦	$\Gamma \vdash [\Gamma]A \lesssim \forall b.\, B' \dashv \Delta$	By above equality

- Case 1 (B unknown) : $[\Gamma]B = \star$:

$[\Omega]B = \star$	
$[\Omega]\Gamma \vdash [\Omega]A \lesssim \star$	Given
$[\Omega]A \in \mathbb{C}$	Above
$\Gamma \longrightarrow \Omega$	Given
$[\Gamma]A \in \mathbb{C}$	Above
$\Gamma \vdash [\Gamma]A \lesssim \star \dashv \mathsf{contaminate}(\Gamma, [\Gamma]A)$	By rule AS-UNKNOWNRR
There exists Ω' such that $\mathsf{contaminate}(\Gamma, [\Gamma]A) \longrightarrow \Omega'$ and $\Omega \longrightarrow \Omega'$	

- Case 2.*: $[\Gamma]B$ is not polymorphic. We split on the form of $[\Omega]A$.
  - Case 2.Poly : $[\Omega]A$ is polymorphic: $[\Gamma]A = \forall a.\, A'$:

	$A = \forall a.\, A_0$	Γ is predicative
	$A' = [\Gamma]A_0$	Γ is predicative
	$[\Omega]A = \forall a.\, [\Omega]A_0$	By def. of substitution
	$[\Omega]\Gamma \vdash [\Omega]A \lesssim [\Omega]B$	Premise
	$[\Omega]\Gamma \vdash \forall a.\, [\Omega]A_0 \lesssim [\Omega]B$	By above equality
	$[\Omega]\Gamma \vdash ([\Omega]A_0)[a \mapsto \tau] \lesssim [\Omega]B$	By inversion on rule CS-FORALLL
	$[\Omega]\Gamma \vdash \tau$	Above

	$[\Omega]\Gamma = [\Omega, \widehat{a} = \tau](\Gamma, \widehat{a})$	By def. of substitution
	$([\Omega]A_0)[a \mapsto \tau] = [\Omega, \widehat{a} = \tau](A_0[a \mapsto \widehat{a}])$	By def. of substitution
	$[\Omega]B = [\Omega, \widehat{a} = \tau]B$	By def. of substitution
	$[\Omega, \widehat{a} = \tau](\Gamma, \widehat{a}) \vdash [\Omega, \widehat{a} = \tau](A_0[a \mapsto \widehat{a}]) \lesssim [\Omega, \widehat{a} = \tau]B$	By above equalities
	$\Gamma, \widehat{a} \vdash [\Gamma, \widehat{a}](A_0[a \mapsto \widehat{a}]) \lesssim [\Gamma, \widehat{a}]B \dashv \Delta$	By i.h.
♦	$\Delta \longrightarrow \Omega'$	Above
	$\Omega, \widehat{a} = \tau \longrightarrow \Omega'$	Above
♦	$\Omega \longrightarrow \Omega'$	By Lemma E.11
	$[\Gamma, \widehat{a}](A_0[a \mapsto \widehat{a}]) = ([\Gamma]A_0)[a \mapsto \widehat{a}]$	By def. of substitution
	$[\Gamma, \widehat{a}]B = [\Gamma]B$	By def. of substitution
	$\Gamma, \widehat{a} \vdash ([\Gamma]A_0)[a \mapsto \widehat{a}] \lesssim [\Gamma]B \dashv \Delta$	By above equality
	$\Gamma \vdash \forall a.\, ([\Gamma]A_0) \lesssim [\Gamma]B \dashv \Delta$	By rule AS-FORALLLL
♦	$\Gamma \vdash \forall a.\, A' \lesssim [\Gamma]B \dashv \Delta$	By above equality

– Case 2.Unknown : $[\Gamma]A = \star$:

$[\Omega]A = \star$	Obviously, what else?
$[\Omega]\Gamma \vdash \star \lesssim [\Omega]B$	Given
$[\Omega]B \in \mathbb{C}$	Above
$\Gamma \longrightarrow \Omega$	Given
$[\Gamma]B \in \mathbb{C}$	Above
$\Gamma \vdash \star \lesssim [\Gamma]B \dashv \mathsf{contaminate}(\Gamma, [\Gamma]B)$	By rule AS-UNKNOWNLL
There exists Ω' such that $\mathsf{contaminate}(\Gamma, [\Gamma]B) \longrightarrow \Omega'$ and $\Omega \longrightarrow \Omega'$	

– Case 2.AEx.* : $[\Gamma]A$ is an existential variable: $[\Gamma]A = \widehat{a}$. We split on the form of $[\Gamma]B$.

* Case 2.AEx.SameEx . $[\Gamma]B$ is the same existential variable $[\Gamma]B = \widehat{a}$:

	$\Gamma \vdash \widehat{a} \lesssim \widehat{a} \dashv \Gamma$	By rule AS-EVAR
♦	$\Gamma \vdash [\Gamma]A \lesssim [\Gamma]B \dashv \Gamma$	By above equality
♦	$\Delta \longrightarrow \Omega$	$\Delta = \Gamma$
♦	$\Omega \longrightarrow \Omega'$	By Lemma E.2 and $\Omega' = \Omega$

* Case 2.AEx.OtherEx . $[\Gamma]B$ is a different existential variable $[\Gamma]B = \widehat{b}$ where $\widehat{b} \neq \widehat{a}$:

	$[\Omega]A = [\Omega]([\Gamma]A) = [\Omega]\widehat{a}$	By Lemma E.5
	$[\Omega]B = [\Omega]([\Gamma]B) = [\Omega]\widehat{b}$	By Lemma E.5
	$[\Omega]\Gamma \vdash [\Omega]A \lesssim [\Omega]B$	Given
	$[\Omega]\Gamma \vdash [\Omega]\widehat{a} \lesssim [\Omega]\widehat{b}$	By above equalities
	$\Gamma \vdash \widehat{a} \lessapprox \widehat{b} \dashv \Delta$	By Theorem 7.5
♦	$\Delta \longrightarrow \Omega'$	Above

♦	$\Omega \longrightarrow \Omega'$	Above
	$\Gamma \vdash \widehat{a} \lesssim \widehat{b} \dashv \Delta$	By rule AS-INSTL
♦	$\Gamma \vdash [\Gamma]A \lesssim [\Gamma]B \dashv \Delta$	By above equalities

* Case 2.AEx.Int . We have $[\Gamma]B = \mathsf{Int}$:

	$\Gamma \longrightarrow \Omega$	Given
	$[\Omega]B = \mathsf{Int} = [\Omega]\mathsf{Int}$	By def. of substitution
	$[\Omega]A = [\Omega]([\Gamma]A) = [\Omega]\widehat{a}$	By Lemma E.5
	$[\Omega]\Gamma \vdash [\Omega]A \lesssim [\Omega]B$	Given
	$[\Omega]\Gamma \vdash [\Omega]\widehat{a} \lesssim [\Omega]\mathsf{Int}$	By above equalities
	$\Gamma \vdash \widehat{a} \lessapprox \mathsf{Int} \dashv \Delta$	By Theorem 7.5
♦	$\Delta \longrightarrow \Omega'$	Above
♦	$\Omega \longrightarrow \Omega'$	Above
	$\Gamma \vdash \widehat{a} \lesssim \mathsf{Int} \dashv \Delta$	By rule AS-INSTL
♦	$\Gamma \vdash [\Gamma]A \lesssim [\Gamma]B \dashv \Delta$	By above equalities

* Case 2.AEx.UVar . We have $[\Gamma]B = b$. Similar to Case 2.AEx.Int .
* Case 2.AEx.Arrow . $[\Gamma]B = B_1 \rightarrow B_2$. We prove $\widehat{a} \notin \textsc{fv}([\Gamma]B)$. Suppose for a contradiction, that $\widehat{a} \in \textsc{fv}([\Gamma]B)$, then $\widehat{a}$ must be a subterm of $[\Gamma]B$, so is $[\Omega]\widehat{a}$ a subterm of $[\Omega]([\Gamma]B)$. The latter is equal to $[\Omega]B$, so $[\Omega]\widehat{a}$ is a subterm of $[\Omega]B$. Since $[\Gamma]B = B_1 \rightarrow B_2$, then $[\Omega]B$ must have the form $B_1' \rightarrow B_2'$. Therefore $[\Omega]\widehat{a}$ must occur in either B_1' or B_2'. But we have $[\Omega]\Gamma \vdash [\Omega]\widehat{a} \lesssim [\Omega]B$. That is , $[\Omega]\widehat{a}$ cannot be a subterm of $[\Omega]B$. This is a contradiction.

	$\widehat{a} \notin \textsc{fv}([\Gamma]B)$	Proved above
	$\Gamma \longrightarrow \Omega$	Given
	$[\Omega]B = [\Omega]([\Gamma]B)$	By Lemma E.5
	$[\Omega]\Gamma \vdash [\Omega]\widehat{a} \lesssim [\Omega]B$	Given
	$[\Omega]\Gamma \vdash [\Omega]\widehat{a} \lesssim [\Omega]([\Gamma]B)$	By above equality
	$\Gamma \vdash \widehat{a} \lessapprox [\Gamma]B \dashv \Delta$	By Theorem 7.5
♦	$\Delta \longrightarrow \Omega'$	Above
♦	$\Omega \longrightarrow \Omega'$	Above
	$\Gamma \vdash \widehat{a} \lesssim [\Gamma]B \dashv \Delta$	By rule CS-INSTL
♦	$\Gamma \vdash [\Gamma]A \lesssim [\Gamma]B \dashv \Delta$	By above equalities

* Case 2.AEx.S and 2.AEx.S . Similar to Case 2.AEx.Int .

– Case 2.BEx.* . $[\Gamma]A$ is not polymorphic and $[\Gamma]B$ is an existential variable: $[\Gamma]B = \widehat{b}$. We split on the form of $[\Gamma]A$.
  * Case 2.BEx.Int . Similar to Case 2.AEx.Unit .
  * Case 2.BEx.UVar . Similar to Case 2.AEx.UVar .
  * Case 2.BEx.Arrow . Similar to Case 2.AEx.Arrow .
  * Case 2.BEx.S . Similar to Case 2.AEx.S .
  * Case 2.BEx.G . Similar to Case 2.AEx.G .

  We use the second part of Theorem 7.5 and apply rule AS-INSTR.

– Case 2.Ints . $[\Gamma]A = [\Gamma]B = \mathsf{Int}$:

♦	$\Gamma \vdash \mathsf{Int} \lesssim \mathsf{Int} \dashv \Gamma$	By rule AS-INT

$$\begin{array}{ll|l} & \Gamma \longrightarrow \Omega & \text{Given} \\ \blacklozenge & \Delta \longrightarrow \Omega' & \Delta = \Gamma \\ \blacklozenge & \Omega \longrightarrow \Omega' & \text{By Lemma E.2 and } \Omega' = \Omega \end{array}$$

– Case $\boxed{\text{2.UVars}}$. $[\Gamma]A = [\Gamma]B = a$:

$$\begin{array}{ll|l} \blacklozenge & \Gamma \vdash a \lesssim a \dashv \Gamma & \text{By rule AS-TVAR} \\ & \Gamma \longrightarrow \Omega & \text{Given} \\ \blacklozenge & \Delta \longrightarrow \Omega' & \Delta = \Gamma \\ \blacklozenge & \Omega \longrightarrow \Omega' & \text{By Lemma E.2 and } \Omega' = \Omega \end{array}$$

– Case $\boxed{\text{2.Arrows}}$. Let $[\Gamma]A = A_1 \to A_2$ and $[\Gamma]B = B_1 \to B_2$:

$$\begin{array}{ll|l} & \Gamma \longrightarrow \Omega & \text{Given} \\ & [\Omega]A = [\Omega]([\Gamma]A) = [\Omega]A_1 \to [\Omega]A_2 & \text{By Lemma E.5} \\ & [\Omega]B = [\Omega]([\Gamma]B) = [\Omega]B_1 \to [\Omega]B_2 & \text{By Lemma E.5} \\ & [\Omega]\Gamma \vdash [\Omega]A \lesssim [\Omega]B & \text{Given} \\ & [\Omega]\Gamma \vdash [\Omega]B_1 \lesssim [\Omega]A_1 & \text{Premise} \\ & \Gamma \vdash [\Gamma]B_1 \lesssim [\Gamma]A_1 \dashv \Theta & \text{By i.h.} \\ & \Theta \longrightarrow \Omega_0 & \text{Above} \\ & \Omega \longrightarrow \Omega_0 & \text{Above} \\ & \Gamma \longrightarrow \Omega_0 & \text{By Lemma E.3} \end{array}$$

$$\begin{array}{ll|l} & [\Omega]\Gamma = [\Omega_0]\Theta & \text{By Lemma E.14} \\ & [\Omega]A_2 = [\Omega_0]([\Gamma]A_2) & \text{By Lemma E.5} \\ & [\Omega]B_2 = [\Omega_0]([\Gamma]B_2) & \text{By Lemma E.5} \\ & [\Omega]\Gamma \vdash [\Omega]A_2 \lesssim [\Omega]B_2 & \text{Premise} \\ & [\Omega_0]\Theta \vdash [\Omega_0]([\Gamma]A_2) \lesssim [\Omega_0]([\Gamma]B_2) & \text{By above equalities} \\ & \Theta \vdash [\Theta]([\Gamma]A_2) \lesssim [\Theta]([\Gamma]B_2) \dashv \Delta & \text{By i.h.} \\ \blacklozenge & \Delta \longrightarrow \Omega' & \text{Above} \\ & \Omega_0 \longrightarrow \Omega' & \text{Above} \end{array}$$

$$\begin{array}{ll|l} \blacklozenge & \Gamma \vdash [\Gamma](A_1 \to A_2) \lesssim [\Gamma](B_1 \to B_2) \dashv \Delta & \text{By rule AS-ARROW} \\ \blacklozenge & \Omega \longrightarrow \Omega' & \text{By Lemma E.3} \end{array}$$

– Case $\boxed{\text{2.S}}$: $[\Gamma]A = [\Gamma]B = \mathcal{S}$.

$$\begin{array}{ll|l} \blacklozenge & \Gamma \vdash \mathcal{S} \lesssim \mathcal{S} \dashv \Gamma & \text{By rule AS-SPAR} \\ & \Gamma \longrightarrow \Omega & \text{Given} \\ \blacklozenge & \Delta \longrightarrow \Omega' & \Delta = \Gamma \\ \blacklozenge & \Omega \longrightarrow \Omega' & \text{By Lemma E.2 and } \Omega' = \Omega \end{array}$$

– Case $\boxed{\text{2.G}}$. Similar to Case $\boxed{\text{2.S}}$.

□

I COMPLETENESS OF TYPING

Theorem 7.7 (Matching Completeness). *Given* $\Gamma \longrightarrow \Omega$ *and* $\Gamma \vdash A$, *if* $[\Omega]\Gamma \vdash [\Omega]A \triangleright A_1 \to A_2$ *then there exist* Δ, Ω', A_1' *and* A_2' *such that* $\Gamma \vdash [\Gamma]A \triangleright A_1' \to A_2' \dashv \Delta$ *and* $\Delta \longrightarrow \Omega'$ *and* $\Omega \longrightarrow \Omega'$ *and* $A_1 = [\Omega']A_1'$ *and* $A_2 = [\Omega']A_2'$.

Proof. By induction on the given derivation. We split on $[\Gamma]A$.

- $[\Gamma]A = \forall a. A'$:

$A = \forall a. A_0$	Γ is predicative
$A' = [\Gamma]A_0$	Γ is predicative
$[\Omega]A = \forall a. [\Omega]A_0$	By def. of substitution
$[\Omega]\Gamma \vdash [\Omega]A \triangleright A_1 \to A_2$	Given
$[\Omega]\Gamma \vdash \forall a. [\Omega]A_0 \triangleright A_1 \to A_2$	By above equality
$[\Omega]\Gamma \vdash ([\Omega]A_0)[a \mapsto \tau] \triangleright A_1 \to A_2$	By inversion
$[\Omega]\Gamma \vdash \tau$	Above
$\Gamma \longrightarrow \Omega$	Given
$\Gamma, \widehat{a}_S \longrightarrow \Omega, \widehat{a}_S$	By def. of context extension

	$[\Omega]\Gamma = [\Omega, \widehat{a}_S = \tau](\Gamma, \widehat{a}_S)$	By def. of context application
	$([\Omega]A_0)[a \mapsto \tau] = [\Omega, \widehat{a}_S = \tau](A_0[a \mapsto \widehat{a}_S])$	By def. of substitution
	$[\Omega, \widehat{a}_S = \tau](\Gamma, \widehat{a}_S) \vdash [\Omega, \widehat{a}_S = \tau](A_0[a \mapsto \widehat{a}_S]) \triangleright A_1 \to A_2$	By above equalities
	$\Gamma, \widehat{a}_S \vdash [\Gamma, \widehat{a}_S](A_0[a \mapsto \widehat{a}_S]) \triangleright A_1' \to A_2' \dashv \Delta$	By i.h.
♦	$\Delta \longrightarrow \Omega'$ and $\Omega, \widehat{a}_S = \tau \longrightarrow \Omega'$	Above
♦	$A_1 = [\Omega']A_1'$ and $A_2 = [\Omega']A_2'$	Above
	$[\Gamma, \widehat{a}_S](A_0[a \mapsto \widehat{a}_S]) = ([\Gamma]A_0)[a \mapsto \widehat{a}_S]$	By def. of substitution
	$\Gamma, \widehat{a}_S \vdash ([\Gamma]A_0)[a \mapsto \widehat{a}_S] \triangleright A_1' \to A_2' \dashv \Delta$	By above equality
	$\Gamma \vdash \forall a. [\Gamma]A_0 \triangleright A_1' \to A_2' \dashv \Delta$	By rule AM-FORALLL
	$[\Gamma]A = \forall a. A' = \forall a. [\Gamma]A_0$	By above equalities
♦	$\Gamma \vdash [\Gamma]A \triangleright A_1' \to A_2' \dashv \Delta$	Above

- $[\Gamma]A = A_1' \to A_2'$:

	$[\Omega]A = [\Omega]([\Gamma]A) = [\Omega]A_1' \to [\Omega]A_2'$	By Lemma E.5
	$[\Omega]\Gamma \vdash [\Omega]A_1' \to [\Omega]A_2' \triangleright A_1 \to A_2$	Given
	Let $\Delta = \Gamma$ and $\Omega' = \Omega$	
♦	$[\Omega]A_1' = A_1$ and $[\Omega]A_2' = A_2$	
♦	$\Gamma \vdash A_1' \to A_2' \triangleright A_1' \to A_2' \dashv \Gamma$	By rule AM-ARR
♦	$\Delta \longrightarrow \Omega'$	Given $\Gamma \longrightarrow \Omega$
♦	$\Omega \longrightarrow \Omega'$	By Lemma E.2

- $[\Gamma]A = \star$:

	$[\Omega]A = [\Omega]([\Gamma]A) = \star$	By Lemma E.5
	$[\Omega]\Gamma \vdash \star \triangleright A_1 \to A_2$	Given
	Let $\Delta = \Gamma$ and $\Omega' = \Omega$	
♦	$A_1 = \star$ and $A_2 = \star$	
♦	$\Gamma \vdash \star \triangleright \star \to \star \dashv \Gamma$	By rule AM-UNKNOWN
♦	$\Delta \longrightarrow \Omega'$	Given $\Gamma \longrightarrow \Omega$
♦	$\Omega \longrightarrow \Omega'$	By Lemma E.2

- $[\Gamma]A = \widehat{a}$:

	$\Gamma = \Gamma_0[\widehat{a}]$	Since $\widehat{a} \in \textsc{unsolved}(\Gamma)$
	$[\Omega]A = [\Omega]([\Gamma]A) = [\Omega]\widehat{a}$	By Lemma E.5
	$[\Omega]\Gamma \vdash [\Omega]\widehat{a} \triangleright A_1 \to A_2$	Given
	$[\Omega]\widehat{a} = \tau_1 \to \tau_2$ and $A_1 = \tau_1$ and $A_2 = \tau_2$	Ω is predicate
	$\Omega = \Omega_0[\widehat{a} = \tau']$ and $[\Omega]\tau' = \tau_1 \to \tau_2$	Above
	Let $\Delta = \Gamma_0[\widehat{a}_1, \widehat{a}_2, \widehat{a} = \widehat{a}_1 \to \widehat{a}_2]$	
	Let $\Omega' = \Omega_0[\widehat{a}_1 = \tau_1, \widehat{a}_2 = \tau_2, \widehat{a} = \widehat{a}_1 \to \widehat{a}_2]$	
♦	$\Delta \longrightarrow \Omega'$	By Lemma E.9 twice
♦	$\Omega \longrightarrow \Omega'$	By Lemma E.10 and Lemma E.9
♦	$\Gamma_0[\widehat{a}] \vdash \widehat{a} \triangleright \widehat{a}_1 \to \widehat{a}_2 \dashv \Delta$	By rule AM-VAR
♦	$A_1 = \tau_1 = [\Omega']\widehat{a}_1$ and $A_2 = \tau_2 = [\Omega']\widehat{a}_2$	Above

□

Theorem 7.8 (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 $\lfloor e \rfloor = \lfloor e' \rfloor$.*

Proof. By induction on the given derivation.

- Case

$$\frac{(x : A) \in [\Omega]\Gamma}{[\Omega]\Gamma \vdash x : A}\ \textsc{var}$$

	$(x : A) \in [\Omega]\Gamma$	Premise
	$\Gamma \longrightarrow \Omega$	Given
	$(x : A') \in \Gamma$ where $[\Omega]A' = [\Omega]A$	From def. of context application
	Let $\Delta = \Gamma$ and $\Omega' = \Omega$.	
♦	$\Gamma \longrightarrow \Omega$	Given
♦	$\Omega \longrightarrow \Omega$	By Lemma E.2
♦	$\Gamma \vdash x \Rightarrow A' \dashv \Gamma$	By rule INF-VAR
♦	$[\Omega]A' = [\Omega]A = A$	A is well-formed in $[\Omega]\Gamma$
♦	$\lfloor x \rfloor = \lfloor x \rfloor$	By def. of erasure

- Case

$$\frac{}{[\Omega]\Gamma \vdash n : \mathsf{Int}}\ \textsc{int}$$

	Let $A' = \mathsf{Int}$ and $\Delta = \Gamma$ and $\Omega' = \Omega$.	
♦	$\Gamma \longrightarrow \Omega$	Given
♦	$\Omega \longrightarrow \Omega$	By Lemma E.2
♦	$\Gamma \vdash n \Rightarrow \mathsf{Int} \dashv \Gamma$	By rule INF-INT
♦	$[\Omega]\mathsf{Int} = \mathsf{Int}$	
♦	$\lfloor n \rfloor = \lfloor n \rfloor$	By def. of erasure

- Case

$$\frac{[\Omega]\Gamma, x : A \vdash e : B}{[\Omega]\Gamma \vdash \lambda x : A.\, e : A \to B}\ \textsc{lamann}$$

Let $\Omega_0 = \Omega, x : A$.	
$[\Omega_0](\Gamma, x : A) = [\Omega]\Gamma, x : A$	From def. of context application
$[\Omega_0](\Gamma, x : A) \vdash e : B$	By above equality and premise
$\Gamma, x : A \vdash e' \Rightarrow B_0 \dashv \Delta_0$	By i.h.
$\Delta_0 \longrightarrow \Omega'$	Above
$\Omega_0 \longrightarrow \Omega'$	Above
$B = [\Omega']B_0$	Above
$\lfloor e \rfloor = \lfloor e' \rfloor$	Above
$\Gamma, x : A \longrightarrow \Delta_0$	By Lemma G.2
$\Delta_0 = \Delta_1, x : A', \Delta_2$	By Lemma E.6
$[\Delta_1]A = [\Delta_1]A'$	Above
$\Gamma \longrightarrow \Delta_1$	Above
$A = [\Delta_1]A'$	A has no evar
$\Gamma, x : A \vdash e' \Rightarrow B_0 \dashv \Delta_1, x : A, \Delta_2$	By above equalities
$\Gamma, x : A \vdash e' \Leftarrow B_0 \dashv \Delta_1, x : A, \Delta_2$	By rule CHK-SUB

$\Delta_1, x : A', \Delta_2 \longrightarrow \Omega'$	By above equalities
$\Omega' = \Omega_1, x : A'', \Omega_2$	By Lemma E.6
$[\Omega_1]A' = [\Omega_1]A''$	Above
♦ $\Delta_1 \longrightarrow \Omega_1$	Above
$\Omega, x : A \longrightarrow \Omega_1, x : A'', \Omega_2$	By above equalities
♦ $\Omega \longrightarrow \Omega_1$	By Lemma E.6

$\Gamma \vdash \lambda x.\, e' \Leftarrow A \to B_0 \dashv \Delta_1$	rule CHK-LAM
♦ $\Gamma \vdash (\lambda x.\, e') : A \to B_0 \Rightarrow A \to B_0 \dashv \Delta_1$	rule INF-ANNO
♦ $[\Omega_1](A \to B_0) = A \to [\Omega']B_0 = A \to B$	From above equality
♦ $\lfloor \lambda x : A.\, e \rfloor = \lambda x.\, \lfloor e \rfloor = \lambda x.\, \lfloor e' \rfloor = \lfloor (\lambda x.\, e') : A \to B_0 \rfloor$	By def. of erasure

- Case

$$\frac{[\Omega]\Gamma \vdash e_1 : A \qquad [\Omega]\Gamma \vdash A \triangleright A_1 \to A_2 \qquad [\Omega]\Gamma \vdash e_2 : A_3 \qquad [\Omega]\Gamma \vdash A_3 \lesssim A_1}{[\Omega]\Gamma \vdash e_1\, e_2 : A_2}\ \textsc{app}$$

$[\Omega]\Gamma \vdash e_1 : A$	Premise
$\Gamma \longrightarrow \Omega$	Given
$\Gamma \vdash e_1' \Rightarrow A' \dashv \Theta_1$	By i.h.
$\Theta_1 \longrightarrow \Omega_0'$	Above
$\Omega \longrightarrow \Omega_0'$	Above
$A = [\Omega_0']A'$	Above
$\lfloor e_1 \rfloor = \lfloor e_1' \rfloor$	Above

$[\Omega]\Gamma \vdash A \triangleright A_1 \to A_2$	Premise
$[\Omega]\Gamma = [\Omega]\Omega$	By Lemma E.17
$= [\Omega_0']\Omega_0'$	By Lemma E.13
$= [\Omega_0']\Gamma$	By Lemma E.17
$= [\Omega_0']\Theta_1$	By Lemma E.14
$[\Omega_0']\Theta_1 \vdash [\Omega_0']A' \triangleright A_1 \to A_2$	By above equalities
$\Theta_1 \vdash [\Theta_1]A' \triangleright A_1' \to A_2' \dashv \Theta_2$	By Theorem 7.7

$\Theta_2 \longrightarrow \Omega'$	Above
$\Omega'_0 \longrightarrow \Omega'$	Above
$A_1 = [\Omega']A'_1$	Above
$A_2 = [\Omega']A'_2$	Above

$[\Omega]\Gamma \vdash e_2 : A_3$	Premise
$[\Omega]\Gamma = [\Omega]\Omega$	By Lemma E.17
$= [\Omega']\Omega'$	By Lemma E.13
$= [\Omega']\Gamma$	By Lemma E.17
$= [\Omega']\Theta_2$	By Lemma E.14
$[\Omega']\Theta_2 \vdash e_2 : A_3$	By above equality
$\Theta_2 \vdash e'_2 \Rightarrow A'_3 \dashv \Theta_3$	By i.h.
$\Theta_3 \longrightarrow \Omega'_1$	Above
$\Omega' \longrightarrow \Omega'_1$	Above
$A_3 = [\Omega'_1]A'_3$	Above
$\lfloor e_2 \rfloor = \lfloor e'_2 \rfloor$	Above

	$[\Omega]\Gamma \vdash A_3 \lesssim A_1$	Premise
	$[\Omega]\Gamma = [\Omega]\Omega$	By Lemma E.17
	$= [\Omega'_1]\Omega'_1$	By Lemma E.13
	$= [\Omega'_1]\Gamma$	By Lemma E.17
	$= [\Omega'_1]\Theta_3$	By Lemma E.14
	$A_3 = [\Omega'_1]A'_3$	Above
	$A_1 = [\Omega']A'_1 = [\Omega'_1]A'_1$	By Lemma E.12
	$[\Omega'_1]\Theta_3 \vdash [\Omega'_1]A'_3 \lesssim [\Omega'_1]A'_1$	By above equalities
	$\Theta_3 \vdash [\Theta_3]A'_3 \lesssim [\Theta_3]A'_1 \dashv \Delta$	By Theorem 7.6
	$[\Theta_3]A'_1 = [\Theta_3]([\Theta_2]A_1)$	By Lemma E.5
	$\Theta_2 \vdash e'_2 \Leftarrow [\Theta_2]A'_1 \dashv \Delta$	By rule CHK-SUB
♦	$\Delta \longrightarrow \Omega'_2$	Above
	$\Omega'_1 \longrightarrow \Omega'_2$	Above
♦	$\Gamma \vdash e'_1\, e'_2 \Rightarrow A'_2 \dashv \Delta$	By rule INF-APP
♦	$A_2 = [\Omega']A'_2 = [\Omega'_2]A'_2$	Lemma E.12
♦	$\Gamma \longrightarrow \Omega'_2$	By Lemma E.3
♦	$\lfloor e_1\, e_2 \rfloor = \lfloor e_1 \rfloor\, \lfloor e_2 \rfloor = \lfloor e'_1 \rfloor\, \lfloor e'_2 \rfloor = \lfloor e'_1\, e'_2 \rfloor$	By def. of erasure

- Case

$$\frac{[\Omega]\Gamma, x : \tau \vdash e : B}{[\Omega]\Gamma \vdash \lambda x.\, e : \tau \to B}\ \text{LAM}$$

$[\Omega]\Gamma, x : \tau \vdash e : B$	Given
$[\Omega]\Gamma, x : \tau = [\Omega, x : \tau](\Gamma, x : \tau)$	By def. of context substitution
$[\Omega, x : \tau](\Gamma, x : \tau) \vdash e : B$	By above equality
$\Gamma, x : \tau \vdash e' \Rightarrow B' \dashv \Delta'$	By i.h.,
$\Delta' \longrightarrow \Omega'$	Above
$\Omega, x : \tau \longrightarrow \Omega'$	Above
$B = [\Omega']B'$	Above
$\lfloor e \rfloor = \lfloor e' \rfloor$	Above
$\Gamma, x : \tau \longrightarrow \Delta'$	By Lemma G.2

$$
\begin{array}{ll|l}
 & \Delta' = \Delta, x : \tau, \Theta & \text{By Lemma E.6} \\
 & \Gamma, x : \tau \vdash e' \Rightarrow B' \dashv \Delta, x : \tau, \Theta & \text{By above equality} \\
\blacklozenge & \Gamma \vdash \lambda x : \tau .\, e' \Rightarrow \tau \rightarrow B' \dashv \Delta & \text{By rule INF-LAMANN} \\
\blacklozenge & \Delta \longrightarrow \Omega' & \text{By context extension} \\
\blacklozenge & \Omega \longrightarrow \Omega' & \text{By context extension} \\
\blacklozenge & \tau \rightarrow B = \tau \rightarrow [\Omega']B' = [\Omega'](\tau \rightarrow B') & \text{By def. of substitution} \\
\blacklozenge & \lfloor \lambda x.\, e \rfloor = \lambda x.\, \lfloor e \rfloor = \lambda x.\, \lfloor e' \rfloor = \lfloor \lambda x : \tau .\, e' \rfloor & \text{By def. of erasure}
\end{array}
$$

- Case

$$
\frac{[\Omega]\Gamma, a \vdash e : A}{[\Omega]\Gamma \vdash e : \forall a.\, A}\ \textsc{gen}
$$

$$
\begin{array}{ll|l}
 & [\Omega]\Gamma, a \vdash e : A & \text{Given} \\
 & [\Omega]\Gamma, a = [\Omega, a](\Gamma, a) & \text{By def. of context substitution} \\
 & [\Omega, a](\Gamma, a) \vdash e : A & \text{By above equality} \\
 & \Gamma, a \vdash e' \Rightarrow A' \dashv \Delta' & \text{By i.h.,} \\
 & \Delta' \longrightarrow \Omega' & \text{Above} \\
 & \Omega, a \longrightarrow \Omega' & \text{Above} \\
 & A = [\Omega']A' & \text{Above} \\
\blacklozenge & \lfloor e \rfloor = \lfloor e' \rfloor & \text{Above} \\
 & \Gamma, a \longrightarrow \Delta' & \text{By Lemma G.2} \\
 & \Delta' = \Delta, a, \Theta & \text{By Lemma E.6} \\
\blacklozenge & \Delta \longrightarrow \Omega' & \text{By context extension} \\
\blacklozenge & \Omega \longrightarrow \Omega' & \text{By context extension} \\
 & \Gamma, a \vdash e' \Rightarrow A' \dashv \Delta, a, \Theta & \text{By above equality} \\
 & \Delta, a, \Theta \vdash [\Delta, a, \Theta]A' \lesssim [\Delta, a, \Theta]A' \dashv \Delta, a, \Theta & \text{By reflexivity of consistent subtyping} \\
 & \Gamma, a \vdash e' \Leftarrow A' \dashv \Delta, a, \Theta & \text{By rule CHK-SUB} \\
 & \Gamma \vdash e' \Leftarrow \forall a.\, A' \dashv \Delta & \text{By rule CHK-GEN} \\
\blacklozenge & \Gamma \vdash e' : \forall a.\, A' \Rightarrow \forall a.\, A' \dashv \Delta & \text{By rule INF-ANNO} \\
\blacklozenge & \forall a.\, A = \forall a.[\Omega']A' = [\Omega'](\forall a.\, A') & \text{By def. of substitution}
\end{array}
$$

$\square$