Elementary Type Inference

Jinxu Zhao
Department of Computer Science, The University of Hong Kong, China

Bruno C. d. S. Oliveira
Department of Computer Science, The University of Hong Kong, China

Abstract

Languages with polymorphic type systems are made convenient to use by employing type inference to avoid redundant type information. Unfortunately, features such as impredicative types and subtyping make complete type inference very challenging or impossible to realize.

This paper presents a form of partial type inference called *elementary type inference*. Elementary type inference adopts the idea of inferring only monotypes from past work on predicative higher-ranked polymorphism. This idea is extended with the addition of explicit type applications that work for any polytypes. Thus easy (predicative) instantiations can be inferred, while all other (impredicative) instantiations are always possible with explicit type applications without any compromise in expressiveness. Our target is a System F extension with top and bottom types, similar to the language employed by Pierce and Turner in their seminal work on local type inference. We show a sound, complete and decidable type system for a calculus called $F^{e}_{<:}$, that targets that extension of System F. A key design choice in $F^{e}_{<:}$ is to consider top and bottom types as polytypes only. An important technical challenge is that the combination of predicative implicit instantiation and impredicative explicit type applications, in the presence of standard subtyping rules, is non-trivial. Without some restrictions, key properties, such as subsumption and stability of type substitution lemmas, break. To address this problem we introduce a variant of polymorphic subtyping called *stable subtyping* with some mild restrictions on implicit instantiation. All the results are mechanically formalized in the Abella theorem prover.

2012 ACM Subject Classification Software and its engineering → General programming languages

Keywords and phrases Type Inference

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.2

Related Version *Full Version*: `https://github.com/JimmyZJX/ElementaryTypeInference/blob/main/paper_extended.pdf`

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.5`

Funding The research is supported by the Type Inference for Complex Type Systems collaboration project (TC20210422012) between Huawei, and the University of Hong Kong and Hong Kong Research Grants Council projects number 17209520 and 17209821.

Acknowledgements We are grateful to the anonymous reviewers for their valuable comments which helped to improve the presentation of this work. We also thank Chen Cui for creating the implementation for our prototype.

1 Introduction

Many programming languages, such as Java, C#, Scala or TypeScript (among others) have type systems with parametric polymorphism, subtyping and first-class functions. For convenience, some form of type inference is desirable in those languages. Type inference avoids code being cluttered with redundant type annotations, as well as explicit type instantiations of polymorphic functions. For instance, in Java, we can write code such as:

© Jinxu Zhao and Bruno C. d. S. Oliveira;
licensed under Creative Commons License CC-BY 4.0

```
List<String> numbers = Arrays.asList("1", "2", "3", "4", "5", "6");
List<Integer> even = numbers.stream()
                            .map(s -> Integer.valueOf(s))
                            .filter(number -> number % 2 == 0)
                            .collect(Collectors.toList());
```

This code processes a list of strings representing numbers, converts the strings into numbers, and filters the even numbers. Thus the list `even` is `[2,4,6]`. The code is made practical by a form of *local type inference* [27, 21, 4, 28], which helps in two ways. Firstly, local type inference enables *synthesis of type arguments* in many cases. That is, applications of generic methods (i.e. methods parametrized by some types) will *automatically* infer the type arguments. An example of type argument synthesis is the method call `map(s -> Integer.valueOf(s))`. The polymorphic `map` function allows taking a function that converts values in the list with some type `A` into values of some other type `B`, thus producing a list with type `List<B>`. In the code above, `B` is `Integer`, and such instantiation is implicitly done by the compiler. Secondly, local type inference employs *bidirectional type-checking* [27, 10] to propagate known type-information. For example, in the snippet above the type of `numbers` is `List<String>`, therefore in `map(s -> Integer.valueOf(s))` we can deduce that `s` is of type `String`.

Without local type inference, the code above would need to explicitly provide the types for instantiation and the types of arguments for the lambda functions, making the code cluttered with type annotations. Nonetheless, some instantiations cannot be automatically inferred. For such cases, languages like Java support *explicit type applications* as well. For instance, we can write an alternative invocation of the `map` function with explicit type application as:

```
    .<Integer>map(s -> Integer.valueOf(s))
```

In their original work on local type inference, Pierce and Turner [27] considered a System F language extended with subtyping and a top and a bottom type. Such a language captures most of the essential features of interest to write code such as the above. In particular, their System F variant captures essential forms of subtyping by including top and bottom types, supports both implicit instantiation and explicit type application, and employs bidirectional type-checking for inferring types in simple cases with lambdas as arguments.

Local type inference is a pragmatic approach. It is not aimed to provide the same degree of type inference that is possible in languages with Hindley-Milner type inference [16, 19, 5]. Instead, as Pierce and Turner state, the goal is to "*exchange complete type inference for simpler methods that work well in the presence of more powerful type-theoretic features such as subtyping and impredicative polymorphism*". In local type inference, the main design decision that is made to simplify type inference is to avoid forms of global type inference that employ long-distance constraints such as unification variables.

The issues caused by both subtyping and impredicative polymorphism for more ambitious global type inference methods are well-known. There are several important undecidability results that are relevant. For instance, for System F, full type inference is undecidable [38] and an *impredicative* polymorphic subtyping relation for System F is undecidable as well [35, 3]. For a type system with top and bottom types and type variables, subtyping can also easily run into undecidable problems [34]. Despite those problems, several different approaches and restrictions have been developed for allowing type inference for: predicative versions of System F [25, 8, 9, 20, 39]; impredicative versions of System F [17, 18, 36, 33, 32, 14]; as well as several Hindley-Milner extensions with subtyping [11, 30, 6, 23].

Perhaps surprisingly, follow-up research on local type inference and type inference for type systems that include higher-ranked polymorphism, impredicativity and subtyping (with top and bottom types) has been relatively limited. In contrast, closely related global type inference approaches with higher-ranked polymorphism (HRP) for System-F-like languages

have seen quite a bit of development. Such HRP approaches [8, 17, 18, 36, 25, 33, 20] extend the classic Hindley-Milner type system, removing the restriction of top-level (let) polymorphism only. Both local type inference and HRP type inference techniques allow *synthesis of type arguments* and use type annotations to aid inference. The main difference is that HRP type inference targets a System-F-like language without subtyping, whereas the local type inference targets a System-F-like language with subtyping. Another difference is that most HRP techniques support implicit instantiation only, although there is some work on supporting visible (or explicit) type applications as well in HRP approaches [13, 32].

In this paper, we propose a form of partial type inference called *elementary type inference*. Like local type inference, we aim at having a pragmatic approach. We are willing to sacrifice some power in terms of what can be inferred, in exchange for an approach that can deal with both subtyping in the presence of top and bottom types, as well as impredicative polymorphism. Like local type inference we support synthesis of type arguments and bi-directional type-checking, but do not support Hindley-Milner style *generalization*. Unlike local type inference, our type inference is global and employs long-distance constraints. We build on recent developments in predicative type inference with HRP for System-F-like languages. The philosophy in elementary type inference is to infer only *monotypes*, but also include an explicit type application construct that can be used to instantiate *any types* (or polytypes). In other words, many programs where instantiations are monotypes can still benefit from type inference, while no expressive power is sacrificed. We can always resort to explicit type application for dealing with general polytypes.

We present a calculus with elementary type inference, called $F^e_{<:}$, that can encode all terms in a variant of System F with subtyping. The type system of $F^e_{<:}$ is a variant of the Dunfield and Krishnaswami [8] (DK) type system, extended with top and bottom types and impredicative explicit type application. The algorithmic formulation of $F^e_{<:}$ is based on the worklist algorithm by Zhao et al. [39]. We have a prototype implementation, as well as a full mechanical formalization (including results such as soundness, completeness and decidability) in the Abella theorem prover [15]. A key design choice in $F^e_{<:}$ is to consider top and bottom types as polytypes only. In other words, we avoid guessing types for implicit instantiation that use top and bottom types. This design choice avoids many of the technical challenges that would otherwise occur in the inference of terms with top and bottom types.

A key technical challenge is that the combination of predicative implicit instantiation and impredicative explicit type applications is problematic. Without some restrictions, important properties, such as subsumption and type substitution lemmas, break. To address this problem we introduce a novel polymorphic subtyping relation called *stable subtyping*. Stable subtyping has some mild restrictions, compared to the well-known Odersky and Läufer [20] formulation, but accounts for top and bottom types and impredicative instantiations. In essence, due to the presence of explicit impredicative type applications, out-of-order implicit instantiation and unused type variables are forbidden.

In summary, the contributions of this work are:

- **Elementary type inference:** A form of partial type inference that combines predicative implicit instantiation with impredicative explicit type application, in the presence of conventional subtyping rules and top and bottom types.
- **The $F^e_{<:}$ calculus:** We present a syntax-directed specification and an algorithmic version of the $F^e_{<:}$ calculus. We show that the algorithmic version is sound and complete to the specification, and its type system is also decidable. Furthermore, $F^e_{<:}$ is type-safe and complete with respect to a variant of System F with subtyping and top and bottom types.

- **Stable subtyping:** A new form of polymorphic subtyping, based on the well-known polymorphic subtyping relation by Odersky and Läufer [20], but with some restrictions. The restrictions are needed to ensure important properties such as subsumption and stability of type substitutions in the presence of impredicative type applications.
- **Implementation and mechanical formalization.** All the calculi and proofs presented in this paper are mechanically formalized in the Abella theorem prover. The formalization, an implementation and the extended version of the paper can be found in: `https://github.com/JimmyZJX/ElementaryTypeInference`

2 Overview

We start with a background on higher-ranked polymorphic (HRP) type inference and the declarative type system by Dunfield and Krishnaswami [8]. Then we discuss the challenges of extending such HRP systems with explicit type applications and top and bottom types. Finally, we illustrate the key ideas in our work to address those challenges.

2.1 Background: Higher-Ranked Type Inference and Type Applications

Our work builds on prior work on HRP type inference. In HRP type systems, universal quantification can appear in arbitrary positions in types. This lifts a restriction of Hindley-Milner type inference [16, 19, 5], where universal quantification can only appear at the top level. To introduce HRP type inference we will use examples in GHC 8, whose type inference algorithm is closely based on the work by Eisenberg et al. [13] on visible type application. The use of GHC 8 will later be helpful to illustrate some challenges of combining HRP type inference with explicit type applications and standard subtyping rules.[1]

A canonical example of an expression with an arbitrary higher-ranked type is:

```
hpoly = \(f :: forall a. a -> a) -> (f 1, f 'c')
```

The type of this function is `(forall a. a -> a) -> (Int, Char)`. A type annotation helps the type system to infer a type. In general, HRP type systems require some type annotations. In many such systems, it is enough to provide type annotations for polymorphic arguments (such as above). The use of some type annotations means that type inference is partial.

Predicativity, Polymorphic Subtyping and Explicit Type Applications. In predicative type systems, universally quantified types can only be instantiated with monotypes, which are types that do not contain universal quantifiers. For instance, the following definition of `f`:

```
f :: (forall a. Int -> a -> Int) -> Bool -> Int
f k = k 3
```

illustrates a higher-ranked function, where the argument `k` is polymorphic. In the body, implicit instantiation is used when applying `k` to an argument. In that application, the type argument of `k` is left implicit and is instantiated automatically with the monotype `Bool`.

The polymorphic subtyping relation used in GHC 8 (based on Peyton Jones et al.'s work [25]) and also by DK's type system allows implicit instantiation of type arguments in polymorphic types, which follows Hindley-Milner. This allows us, for instance, to define a function `h`, with a different but compatible type with `f`:

```
h :: (forall b a. b -> a -> b) -> Bool -> Int
h k = f k
```

[1] Type-checked with the GHC extensions: `RankNTypes`, `TypeApplications` and `ScopedTypeVariables`.

Notice that in `h`, one more universal variable is added, generalizing the argument type compared to `f`. However, since subtyping of functions is contravariant on the input types, the type of `h` is *less* general (or a supertype) of the type of `f`.

Another alternative to support instantiation is to employ an explicit type application. For example, function `g` illustrates explicit type applications in GHC Haskell:

```
g :: (forall a. Int -> a -> Int) -> Bool -> Int
g k = k @Bool 3
```

The function `g` has the same type as `f` but explicitly instantiates the type arguments of the argument `k`. The notation $e\ @\tau$ (for instance `k @Bool` in the definition of `g` above) denotes explicit type applications in Haskell. We will also adopt a similar notation in this paper.

2.2 Background: The Dunfield and Krishnaswami Type System

Our declarative type system can be viewed as a variant of the DK type system. The DK type system is predicative and supports implicit instantiation only. We review the original type system first, before proceeding with the presentation of our work.

Syntax. The syntax of DK's declarative system is shown at the top of Figure 1. A declarative type A is either the unit type 1, a type variable a, a universal quantification $\forall a.\ A$ or a function type $A \to B$. Nested universal quantifiers are allowed for types, but monotypes τ do not have any universal quantifier. Terms include a unit term $()$, variables x, lambda-functions $\lambda x.\ e$, applications $e_1\ e_2$ and annotations $(e : A)$. Contexts Ψ are sequences of type variable declarations and term variables with their types declared $x : A$.

Declarative Subtyping. The middle of Figure 1 shows DK's declarative subtyping judgment $\Psi \vdash A \leq B$, which was adopted from Odersky and Läufer [20]. This judgment compares the degree of polymorphism between types A and B in DK's implicitly polymorphic type system. If A can always be instantiated to match any instantiation of B, then A is "at least as polymorphic as" B. We also say that A is "more polymorphic than" B and write $A \leq B$. Subtyping rules $\leq$`Var`, $\leq$`Unit` and $\leq\to$ handle simple cases that do not involve universal quantifiers. The subtyping rule for function types $\leq\to$ is standard, being covariant on the return type and contravariant on the argument type. Rule $\leq\forall$`R` states that if A is more general than $\forall b.\ B$, then A must instantiate to $[\tau/b]B$ for every τ. The type variable b we introduced in the premise is implicitly fresh. We use this convention throughout the whole paper. The most interesting rule is $\leq\forall$`L`, which is where implicit instantiation can happen. If some instantiation of $\forall a.\ A$, $[\tau/a]A$, is a subtype of B, then $\forall a.\ A \leq B$. Only monotypes τ can be used to instantiate a, which is *guessed* in this declarative rule.

Declarative Typing. The bidirectional type system, shown at the bottom of Figure 1, has three judgments. The checking judgment $\Psi \vdash e \Leftarrow A$ checks expression e against the type A in the context Ψ. The synthesis judgment $\Psi \vdash e \Rightarrow A$ synthesizes the type A of expression e in the context Ψ. The application judgment $\Psi \vdash A \bullet e \Rrightarrow C$ synthesizes the type C of the application of a function of type A (which could be polymorphic) to the argument e.

Many rules are standard bidirectional type-checking rules [10], so we focus only on the more interesting and non-standard rules. Checking an expression e against a polymorphic type $\forall a.\ A$ in the context Ψ (rule `D`$\forall$`I`) succeeds if e checks against A in the extended context (Ψ, a). The subsumption rule `DSub` calls the subtyping relation, and changes the mode from checking to synthesis: if e synthesizes type A and $A \leq B$, then e checks against B. Besides a standard checking rule (`D`$\to$`I`) for lambda abstractions, rule `D`$\to$`I`$\Rightarrow$ synthesizes monotypes

$\boxed{\text{Syntax}}$

Type variables	a, b		
Types	A, B, C	$::=$	$1 \mid a \mid \forall a.\ A \mid A \to B$
Monotypes	τ, σ	$::=$	$1 \mid a \mid \tau \to \sigma$
Expressions	e	$::=$	$x \mid () \mid \lambda x.\ e \mid e_1\ e_2 \mid (e : A)$
Contexts	Ψ	$::=$	$\cdot \mid \Psi, a \mid \Psi, x : A$

$\boxed{\Psi \vdash A \leq B}$ Declarative subtyping

$$\frac{a \in \Psi}{\Psi \vdash a \leq a}\ \leq\texttt{Var} \qquad \frac{}{\Psi \vdash 1 \leq 1}\ \leq\texttt{Unit} \qquad \frac{\Psi \vdash B_1 \leq A_1 \quad \Psi \vdash A_2 \leq B_2}{\Psi \vdash A_1 \to A_2 \leq B_1 \to B_2}\ \leq\to$$

$$\frac{\Psi \vdash \tau \quad \Psi \vdash [\tau/a]A \leq B}{\Psi \vdash \forall a.\ A \leq B}\ \leq\forall\texttt{L} \qquad \frac{\Psi, b \vdash A \leq B}{\Psi \vdash A \leq \forall b.\ B}\ \leq\forall\texttt{R}$$

$\boxed{\Psi \vdash e \Leftarrow A}$ e checks against input type A.

$\boxed{\Psi \vdash e \Rightarrow A}$ e synthesizes output type A.

$\boxed{\Psi \vdash A \bullet e \Rightarrow\!\!\!\Rightarrow C}$ Applying a function of type A to e synthesizes type C.

$$\frac{(x : A) \in \Psi}{\Psi \vdash x \Rightarrow A}\ \texttt{DVar} \qquad \frac{\Psi \vdash e \Rightarrow A \quad \Psi \vdash A \leq B}{\Psi \vdash e \Leftarrow B}\ \texttt{DSub} \qquad \frac{\Psi \vdash A \quad \Psi \vdash e \Leftarrow A}{\Psi \vdash (e : A) \Rightarrow A}\ \texttt{DAnno}$$

$$\frac{}{\Psi \vdash () \Rightarrow 1}\ \texttt{D1I}\Rightarrow \qquad \frac{\Psi \vdash \tau \quad \Psi \vdash [\tau/a]A \bullet e \Rightarrow\!\!\!\Rightarrow C}{\Psi \vdash \forall a.\ A \bullet e \Rightarrow\!\!\!\Rightarrow C}\ \texttt{D}\forall\texttt{App} \qquad \frac{\Psi, a \vdash e \Leftarrow A}{\Psi \vdash e \Leftarrow \forall a.\ A}\ \texttt{D}\forall\texttt{I}$$

$$\frac{\Psi, x : A \vdash e \Leftarrow B}{\Psi \vdash \lambda x.\ e \Leftarrow A \to B}\ \texttt{D}\to\texttt{I} \qquad \frac{\Psi \vdash e \Leftarrow A}{\Psi \vdash A \to C \bullet e \Rightarrow\!\!\!\Rightarrow C}\ \texttt{D}\to\texttt{App}$$

$$\frac{\Psi \vdash \sigma \to \tau \quad \Psi, x : \sigma \vdash e \Leftarrow \tau}{\Psi \vdash \lambda x.\ e \Rightarrow \sigma \to \tau}\ \texttt{D}\to\texttt{I}\Rightarrow \qquad \frac{\Psi \vdash e_1 \Rightarrow A \quad \Psi \vdash A \bullet e_2 \Rightarrow\!\!\!\Rightarrow C}{\Psi \vdash e_1\ e_2 \Rightarrow C}\ \texttt{D}\to\texttt{E}$$

Figure 1 The Dunfield and Krishnaswami Type System.

$\sigma \to \tau$. Application $e_1\ e_2$ is handled by rule D→E, which first synthesizes the type A of the function e_1. If A is a function type $B \to C$, then rule D→App is applied. The synthesized type of function e_1 can also be polymorphic, of the form $\forall a.\ A$. In that case, we instantiate A to $[\tau/a]A$ with a monotype τ using the rule D∀App. If $[\tau/a]A$ is a function type, rule D→App is used; if $[\tau/a]A$ is another universal quantified type, rule D→I⇒ is recursively applied.

2.3 The Challenges of Explicit Type Applications

While explicit type applications seem like a natural extension to type systems with implicit instantiation, the combination can break some important properties and make programs less robust to refactoring or inlining. In particular, in many existing type systems with implicit instantiation, the order of type arguments being instantiated does not matter, whereas with explicit instantiation the order does matter (at least if arguments are positional). The design proposed by Eisenberg et al. [13] notices this difference, and distinguishes between specified and generalized type quantification. All the GHC examples that we show in this

paper employ specified type quantification, where the programmer explicitly writes the type signature, and the order of type arguments is relevant. However, there are other subtler points in the design of a subtyping relation for specified type quantification that make it hard to get certain expected properties in a language. We illustrate some concrete issues next with GHC 8 and Eisenberg et al.'s design. We remark, however, that some other issues with GHC 8's approach have already been identified and GHC 9 adopts a different approach that avoids the issues described here. A detailed discussion follows in Section 7.

Explicit Type Applications, Subsumption and Equational Reasoning. The design by Eisenberg et al. is based on bidirectional type-checking and supports a standard subsumption rule, similar to the rule `DSub` in DK's type system. Moreover, the subtyping relation employed in that design is essentially an extension of that in DK's type system. The examples that we show next involve subtyping relations that are valid in both DK's type system, as well as the subtyping relation employed by Eisenberg et al. [13]. An important lemma that holds for the DK type system is the checking subsumption lemma:

▶ **Lemma 1** (Checking Subsumption). *If* $\Psi \vdash e \Leftarrow A$ *and* $\Psi \vdash A \leq B$ *then* $\Psi \vdash e \Leftarrow B$

This lemma is quite similar to the subsumption rule. The difference is that the premise ($\Psi \vdash e \Leftarrow A$) is in checking mode, instead of synthesis mode. The lemma states that we can always change the type of an expression being checked to a supertype. A practical consequence of this lemma is that changing type annotations of an expression to a supertype is always possible, which is something that programmers would expect. For example, we could change the type annotation of `f` in Section 2.1 to that of `h` and `f` would still type-check.

In contrast to DK's type system, the work by Eisenberg et al. [13] and GHC 8 does not have the checking subsumption property. We believe that the lack of this property is undesirable, especially for a language that promotes equational reasoning like Haskell. To illustrate why the property is important, consider the Haskell functions:

```
h2 :: (forall b a. b -> a -> b) -> Bool -> Int
h2 k = g k -- type checks!

h3 :: (forall b a. b -> a -> b) -> Bool -> Int
h3 k = k @Bool 3 -- rejected!
```

Recall the examples from Section 2.1. Function `h` is defined with a simple call to `f`, although it has a different type. Here the new function `h2` has the same type as `h`, which is a supertype of the type of `f` and `g`. However, if we try to replace the call `g k` by its definition in `h2`, we get the definition `h3`, which no longer type-checks. There are two important issues to notice here. Firstly, replacing equals by equals (or inlining) results in a program that does not type-check! The problem is that now the type argument of `k` in `h3` instantiates the *wrong* type variable. Now `k` has two type arguments, and the second type argument corresponds to the first type argument of the type of the argument `k` in `g`. Because of this, the type of `k` in `h3` is not compatible with the use of `k` in the body. Secondly, the example shows that higher-ranked type arguments that use explicit type applications can break the checking subsumption property. There are some valid supertypes of functions that, if used instead of the original type, will result in an ill-typed program. Even though the type of `h2` is a supertype of the type of `g`, we cannot use that supertype to type-check `g`.

Explicit Type Applications in DK's Type System. A naive extension of the DK type system with explicit type applications would be to add the rule:

$$\frac{\Psi \vdash e \Rightarrow \forall a.\ A}{\Psi \vdash e\ @B \Rightarrow [B/a]A}\ \texttt{DTypeApp}\forall$$

This rule first synthesizes a polymorphic type $\forall a.\ A$ from e, and then outputs its instantiation $[B/a]A$ for the explicit type application $e\ @B$. Unfortunately, this rule breaks checking subsumption as well, for very similar reasons to those in the GHC examples. It is easy to port the previous GHC counter-examples to this extension of DK's type system, and get similar issues to those that we have just described.

▶ **Example 2.** Now we look at a different example that illustrates the general problem of having order-irrelevance of universally quantified type variables. Suppose that we have $e = \lambda x.\ x\ @\text{Int}\ 3\ \text{True}$, $A = (\forall a.\ \forall b.\ a \to b \to a) \to \text{Int}$, and $B = (\forall b.\ \forall a.\ a \to b \to a) \to \text{Int}$. Both conditions of the checking subsumption lemma

$$\lambda x.\ x\ @\text{Int}\ 3\ \text{True} \Leftarrow (\forall a.\ \forall b.\ a \to b \to a) \to \text{Int}$$
$$(\forall a.\ \forall b.\ a \to b \to a) \to \text{Int} \leq (\forall b.\ \forall a.\ a \to b \to a) \to \text{Int}$$

hold, yet the conclusion $\big(\lambda x.\ x\ @\text{Int}\ 3\ \text{True} \Leftarrow (\forall b.\ \forall a.\ a \to b \to a) \to \text{Int}\big)$ does not hold: the type application instantiates the wrong type variable, causing the type system to reject it. The types A and B here differ only in the order of polymorphic type variables. The order-sensitive explicit type application is not compatible with order-irrelevant subtyping relations, such as the one in DK's type system, breaking the checking subsumption lemma.

Impredicative Type Applications and Stability of Subtyping. So far, we have illustrated problems that involve only monotypes (and predicative instantiation). If impredicative type applications are supported as well, then this brings another class of problems.

▶ **Example 3.** Consider $e = \lambda x.\ x\ @C$, $A = (\forall a.\ a \to a) \to (C \to C)$, and $B = (\forall a.\ \forall b.\ b \to a) \to (C \to C)$. Here we assume $C = (\forall a.\ a \to a) \to \text{Int}$, but other polymorphic types could be used as well. Both conditions of the checking subsumption lemma:

$$\lambda x.\ x\ @C \Leftarrow (\forall a.\ a \to a) \to (C \to C)$$
$$(\forall a.\ a \to a) \to (C \to C) \leq (\forall a.\ \forall b.\ b \to a) \to (C \to C)$$

hold, but not the conclusion $\lambda x.\ x\ @C \Leftarrow (\forall a.\ \forall b.\ b \to a) \to (C \to C)$. When type checking the condition $\lambda x.\ x\ @C \Leftarrow (\forall a.\ a \to a) \to (C \to C)$, the type application $@C$ is properly applied, instantiating the universal variable a on the type of x $(\forall a.\ a \to a)$ to C. However, when we have the conclusion $\lambda x.\ x\ @C \Leftarrow (\forall a.\ \forall b.\ b \to a) \to (C \to C)$, the type application $x\ @C$ only instantiates the type argument a, but not the type argument b. Thus the type inferred for the application $x\ @C$ is $\forall b.\ b \to C$. Unfortunately, the predicative subtyping relation rejects $\forall b.\ b \to C \leq C \to C$ when C is a polymorphic type like $(\forall a.\ a \to a) \to \text{Int}$. Thus, the conclusion $\lambda x.\ x\ @C \Leftarrow (\forall a.\ \forall b.\ b \to a) \to (C \to C)$ fails to type check.

While this problem is also an instance of checking subsumption not holding, the reason why this example fails is different from the previous examples. The crux of the problem here is that some instantiations with polytypes break the polymorphic subtyping relation after instantiation. More concretely, we would like the following property to hold:

▶ **Corollary 4** (Stability of Subtyping). *If* $\Psi \vdash \forall a.\ A \leq \forall a.\ B$ *then* $\Psi \vdash [C/a]A \leq [C/a]B$ *holds for any well-formed* C *(*$\Psi \vdash C$*).*

but this property does not hold in general for polytypes in DK's type system. DK's type system has a similar property, but only for monotypes. When impredicative type applications are present, we would like to have a more general property that holds for polytypes as well.

Problem with Unused Variables. Furthermore, unused type variables in universal quantifiers are problematic, since they can also break stability of subtyping. $\forall a.\ \forall b.\ a \leq \forall a.\ a$ is accepted by most subtyping relations that support implicit instantiation, including DK and HM. If we instantiate both sides of the subtyping judgment with the polytype $C := \forall c.\ c \to c$, the judgment becomes $\forall b.\ \forall c.\ c \to c \leq \forall c.\ c \to c$, which is then problematic, because a second instantiation with $C' := \forall d.\ d \to d$ would give $\forall c.\ c \to c \leq (\forall d.\ d \to d) \to (\forall d.\ d \to d)$, which is rejected by predicative systems. Here the problem is not caused by a permutation or extra type variables in polytypes. Instead, an unused variable leads to such problem.

2.4 The Problems with Subtyping with Top and Bottom Types

Besides the problems with explicit type applications, the presence of top and bottom types raises its own class of issues.

Polymorphic Subtyping with Top and Bottom Types. To support top and bottom types in DK's type system, a first idea is to introduce those types, with their standard subtyping rules and consider them to be monotypes, leading to the following syntax for monotypes:

$$\tau ::= 1 \mid \bot \mid \top \mid \tau \to \tau \mid a$$

Unfortunately, it is known that the subtyping for a language with type variables, such as the above, can quickly become undecidable [34]. To illustrate some of the issues, consider the subtyping judgment $\forall a.\ (a \to a \to 1) \to 1 \leq (\forall b.\ b \to b) \to 1$, which reduces to the following problem: find (predicative) instantiations for $\widehat{\alpha}$ and $\widehat{\beta}$, satisfying $\widehat{\alpha} \leq \widehat{\beta}$ and $\widehat{\beta} \leq \widehat{\alpha} \to 1$. Such a problem has infinitely many solutions, where there is no best one. If we assume that existential variables $\widehat{\alpha}$ and $\widehat{\beta}$ are instantiated to the same type, there are infinite solutions:

$$\widehat{\alpha} = \widehat{\beta} = \bot \mid \top \to 1 \mid \top \to \bot \mid (\bot \to 1) \to 1 \mid \cdots$$

Additionally, $\widehat{\alpha} = \widehat{\beta}$ is not the most general unification for the judgment $\widehat{\alpha} \leq \widehat{\beta}$. Assignments like $\widehat{\alpha} = \widehat{\beta}, \widehat{\beta} = \widehat{\beta} \to 1$ also validate the subtyping judgments.

Inference of $\top$ and $\bot$ Types can Mask Type Errors errors. Consider the following expression:

$$\lambda f.\ f + f\ 1$$

Such expression can be typed with the type $\bot \to \text{Int}$. Since the input parameter f, having type $\bot$, can be converted to either an integer or to a function of type $\text{Int} \to \text{Int}$. However, inferring such a type is hardly useful in practice. Instead, the programmer might have immediately realised the bug (perhaps the argument for the first call to f is missing) if the type inference algorithm rejects the lambda expression, rather than after it has inferred a type with $\bot$. By constraining the type inference algorithm to infer types free from $\top$ and $\bot$, such type is no longer be inferred, and thus the bug is reported when defining the function.

It is mentioning that, in languages like Scala, there are several reports of issues arising because $\top$ and $\bot$ (respectively *Any* and *Nothing* in Scala) can be inferred[2]. The Scala compiler even has a flag `-Xlint:infer-any` that is used to warn whenever *Any* is inferred.

2.5 Our Solution

The form of type-inference available in $F^e_{<:}$ is inspired by current approaches employed in predicative higher-ranked type inference [25, 8, 9, 20, 39]. In terms of restrictions, $F^e_{<:}$ does not have generalization (similarly to the DK type system), and there are some restrictions that weaken the expressive power of the polymorphic subtyping relation. However, those restrictions mostly affect higher-ranked programs, and for many other Hindley-Milner style programs (with polymorphic annotations) there should be no impact from those restrictions. In terms of innovations the type system of $F^e_{<:}$ supports top and bottom types and explicit impredicative type applications. Moreover, implicit instantiation and explicit instantiation interoperate well and have important properties, such as checking subsumption and stability of subtyping. Next we show some examples that run in our implementation and illustrate the capabilities of the $F^e_{<:}$ type system. We note that our implementation contains some extra features that enable us to present more interesting examples. These features include recursive let expressions, (polymorphic) lists and case expressions on lists.

Rank-1 Polymorphism. We start with first-order polymorphism, which is the kind of polymorphism supported in Hindley-Milner. We can define the `map` function in $F^e_{<:}$ as follows:

```
let map :: forall a b. (a -> b) -> [a] -> [b]
    = \f -> \xs -> case xs of [] -> []; (x:xs) -> f x : map f xs
```

This definition is similar to a definition in a language with Hindley-Milner, except that we must explicitly provide the type of the `map` function. An explicit type is optional in languages like Haskell or ML, but must be provided in $F^e_{<:}$ for polymorphic functions like `map`. Like in Hindley-Milner, when writing the body of the function we do not need to use type binders and the recursive call implicitly instantiates the types `a` and `b`. We can use `map` conventionally, as in an HM language like Haskell or ML:

```
map (\x -> x + 1) [1,2,3]
```

or use explicit type applications if necessary or desired. For example:

```
map @Int @Top (\x -> x :: Top) [1,2,3]
```

which uses explicit type applications to instantiate `b` with the `Top` type. Since the body of the function argument uses a type annotation (`x :: Top`) to return a `Top` type, writing `map (\x -> x :: Top) [1,2,3]` would fail to type-check. The problem is that `Top` is not a monotype in our system and cannot be inferred during implicit instantiation.

While there is no generalization, we can still infer monotypes for lambdas. Therefore, the following is allowed:

```
let succ = \x -> x + 1
```

The type inferred for `succ` is the monotype `Int -> Int`. However writing `let id = \x -> x`, without an explicit annotation for `id` would fail, since generalization would be necessary to infer a polymorphic type for the identity function.

[2] See for instance: `https://riptutorial.com/scala/example/21134/preventing-inferring-nothing`.

Higher-Ranked Polymorphism. With explicit type applications, it becomes possible to perform impredicative instantiations. A simple example of this is applying the identity function to itself. In $F^e_{<:}$ we can write:

```
let id :: forall a. a -> a = \x -> x in id @(forall a. a -> a) id
```

In this case the type used to explicitly instantiate the identity function is a polymorphic type. Implicit instantiation is not possible here, since the type argument is not a monotype. Another example of impredicative instantiation is:

```
let plist :: [forall a. a -> a] = [\z -> z, \z -> z]
in map @(forall a. a -> a) (\f -> f 1) plist
```

where the `map` function takes a list with *polymorphic* functions of type `forall a. a -> a`. The function argument (`\f -> f 1`) of `map` applies a polymorphic function to `1`.

Restrictions for Higher-Ranked Polymorphism. The GHC 8 definitions in Section 2.1 for `f` and `g` also type-check in $F^e_{<:}$:

```
let f :: (forall a. Int -> a -> Int) -> Bool -> Int = \k -> k 3
let g :: (forall a. Int -> a -> Int) -> Bool -> Int = \k -> k @Bool 3
```

However, the definition of `h` fails to type-check:

```
let h :: (forall b. forall a. b -> a -> b) -> Bool -> Int = \k -> f k  -- fails!
```

This definition is rejected because $\forall b.\ \forall a.\ b \to a \to b \leq \forall a.\ \text{Int} \to a \to \text{Int}$ does not hold in $F^e_{<:}$. Our polymorphic subtyping relation does not consider the type of `k` to be a subtype of the expected argument for `f`. By preventing examples such as this one we avoid the issues discussed in Section 2.3 and we retain the checking subsumption property.

We can also type-check the expression used in our Example 2:

```
\x -> x @((forall a. a -> a) -> Int)
```

with a type annotation:

```
(forall a. a -> a) -> ((forall a. a -> a) -> Int) -> ((forall a. a -> a) -> Int)
```

owever, using the type annotation

```
(forall a. forall b. b -> a) ->
  ((forall a. a -> a) -> Int) -> ((forall a. a -> a) -> Int)
```

will fail because the type in the first annotation is not a subtype of the type in the second annotation in our polymorphic subtyping relation: the subtyping statement $\forall a.\ \forall b.\ b \to a \leq \forall a.\ a \to a$ does not hold. In the Odersky and Läufer relation, such subtyping statement holds. As discussed in Section 2.3, accepting such subtyping statements is problematic when impredicative instantiation is allowed, since we lose the stability of subtyping property.

No Inference of Top and Bottom. Finally the type system of $F^e_{<:}$ rejects:

```
let strange = \f -> f + f 1   -- fails!
```

As discussed in Section 2.4, this definition could be type-checked if we could infer the bottom type for `f`. Since bottom is not a monotype, it cannot be infered and this definition is rejected. However, the following definition, with an explicit type annotation is allowed:

```
let strange : Bot -> Int = \f -> f + f 1
```

In other words, we avoid inferring top and bottom, which can type-check programs that are likely to have type errors. However, we can always provide annotations for definitions that can type-check with polytypes, thus allowing such definitions to type-check if desired.

$\boxed{\Psi \vdash_s A \leq B}$ Stable subtyping

$$\frac{\Psi \vdash_s A}{\Psi \vdash_s A \leq A} \leq_{\mathtt{s}}\mathtt{refl} \qquad \frac{}{\Psi \vdash_s A \leq \top} \leq_{\mathtt{s}}\top \qquad \frac{}{\Psi \vdash_s \bot \leq A} \leq_{\mathtt{s}}\bot$$

$$\frac{\forall C, \Psi \vdash_s C \implies \Psi \vdash_s [C/a]A \leq [C/a]B}{\Psi \vdash_s \forall a.\ A \leq \forall a.\ B} \leq_{\mathtt{s}}\forall \qquad \frac{\Psi \vdash_s B_1 \leq A_1 \quad \Psi \vdash_s A_2 \leq B_2}{\Psi \vdash_s A_1 \to A_2 \leq B_1 \to B_2} \leq_{\mathtt{s}}\to$$

$$\frac{\Psi \vdash \tau \quad \Psi \vdash_s [\tau/a]A \leq B \quad B \neq \forall b.B'}{\Psi \vdash_s \forall a.\ A \leq B} \leq_{\mathtt{s}}\forall\mathtt{L}$$

Figure 2 Stable Subtyping.

2.6 Key Technical Ideas

Stable Polymorphic Subtyping. We address the problems with explicit type applications via a novel notion of polymorphic subtyping, which preserves both subsumption and stability of subtyping. Stable subtyping, shown in Figure 2, is a variant from DK's subtyping with some changes. Note that we use color to highlight new rules or changes, with respect to DK's type system, here and throughout the paper. The new rule $\leq_{\mathtt{s}}\mathtt{refl}$ replaces base cases in the previous system, and rule $\leq_{\mathtt{s}}\forall$ directly expresses the expected stability property. We also forbid polymorphic types that contain unused type variables. This restriction is enforced by the well-formedness relation (see details in Section 3.1). Rule $\leq_{\mathtt{s}}\forall\mathtt{L}$ has a side condition to prevent overlapping with rule $\leq_{\mathtt{s}}\forall$ and has a less priority. In addition, we also have two new (but standard) rules for top and bottom types (rules $\leq_{\mathtt{s}}\top$ and $\leq_{\mathtt{s}}\bot$).

There are 3 main differences with respect to DK's subtyping relation. Firstly, stable subtyping does not allow instantiations out-of-order. Note that, unlike DK's type system, there is no rule that corresponds to $\leq\forall\mathtt{R}$, which removes the ability to perform such instantiations. Thus, the order of type variables becomes relevant. Secondly, types with unused type variables, such as $\forall a.1$, are not allowed. Finally, top and bottom types are supported.

A Syntax-Directed System with Subtype Variables. While rule $\leq_{\mathtt{s}}\forall$ directly captures the stability property that we want, such a rule is highly declarative. Thus, an important challenge is to find an alternative equivalent set of rules that is closer to an implementation. Our solution to the problem relies on a new sort of variables, $\tilde{a}$, called subtype variables, and the following subtyping rule, which is proven to have equal effect to $\leq_{\mathtt{s}}\forall$:

$$\frac{\Psi, \tilde{a} \vdash [\tilde{a}/a]A \leq [\tilde{a}/a]B}{\Psi \vdash \forall a.\ A \leq \forall a.\ B} \leq\forall$$

The difference between a subtype variable $\tilde{a}$ and a conventional type variable a is that a subtype variable is *not a monotype*, therefore it cannot be instantiated with rule $\leq\forall\mathtt{L}$. Thus, a subtyping statement such as $\forall a.\ \forall b.\ b \to a \leq \forall a.\ a \to a$ does not hold, since it would require instantiation with a subtype variable, which is not allowed.

Explicit Type Applications for Polytype Instantiations. To avoid being overly restrictive, we still support polytype instantiations via explicit type applications. Thus, while some convenience afforded by more expressive formulations of type inference is lost, no expressive

power is lost. We can encode a variant of System $F_{<:}$ (without bounded quantification) trivially using explicit type applications. We formally verify the soundness and completeness theorems in our Abella formalization and present the results in the extended version.

Summary. With stable subtyping, Corollary 4 and the subsumption lemma hold, and the system works smoothly with the explicit type application rules in the type system. Moreover, we address the problems with top and bottom types by not considering top and bottom types as monotypes. This extends a similar idea in predicative HRP, which excludes universal types from monotypes, thus avoiding decidability issues that arise from including such types.

3 Syntax-Directed System

This section introduces a syntax-directed type system for $F^e_{<:}$, which serves as a specification for the algorithmic version that will be presented in Section 4. The type system can be viewed as a variant of the Dunfield and Krishnaswami [8] type system, adding explicit type applications and abstractions, as well as $\top$ and $\bot$ types. Furthermore, the type system supports impredicativity via explicit instantiations. The subtyping relation employed in this type system is equivalent to the *stable subtyping* relation introduced in Section 2.6, but it is syntax-directed and employs a special kind of type variables in the subtyping relation. Several important properties, such as the subsumption lemma, a generalized stability lemma for impredicative types, and transitivity of subtyping are proved.

3.1 Syntax and Well-Formedness

Compared to the syntax of the DK system presented in Figure 1, the syntax of $F^e_{<:}$:

Type variables	a, b	Subtype variables	$\tilde{a}, \tilde{b}$
Types	A, B, C	$::=$	$1 \mid a \mid \forall a.\ A \mid A \to B \mid \tilde{a} \mid \top \mid \bot$
Monotypes	τ, σ	$::=$	$1 \mid a \mid \tau \to \sigma$
Expressions	e, t	$::=$	$x \mid () \mid \lambda x.\ e \mid e_1\ e_2 \mid (e : A) \mid e\ @A \mid \Lambda a.\ e : A$
Contexts	Ψ	$::=$	$\cdot \mid \Psi, a \mid \Psi, x : A \mid \Psi, \tilde{a}$

is extended in the four directions. First, types now include $\top$ and $\bot$. Second, expressions are extended with type applications ($e\ @A$). Third, type abstractions ($\Lambda a.\ e : A$), studied by Zhao et al. [39], are adopted as well, since they are useful to express programs with *scoped type variables* [24]. Note that programmers do not have to write Λ's directly. Instead, an explicit annotation $e : \forall a.\ A$ can serve as the syntactic sugar $e : \forall a.\ A \equiv \Lambda a.e : \forall a.\ A$. Finally, there is a new syntactic sort: subtype variables, $\tilde{a}$, representing type variables that are only used in subtyping and are not monotypes. It is worth mentioning that the definition for monotypes is not changed. We do not treat $\top$, $\bot$, or subtype variables as monotypes.

Well-Formedness. The well-formedness relation is mainly used to ensure the well-scopedness of binders. Additionally, we add special free variable checks to ensure that the polymorphic type $\forall a.\ A$ is indeed polymorphic in the following two rules:

$$\frac{\Psi, a \vdash A \quad a \in \mathrm{FV}(A)}{\Psi \vdash \forall a.\ A}\ \mathtt{wf_d\forall} \qquad \frac{\Psi, a \vdash A \quad \Psi, a \vdash e \quad a \in \mathrm{FV}(A)}{\Psi \vdash \Lambda a.\ e : A}\ \mathtt{wf_dtLam}$$

The motivation for the $a \in \mathrm{FV}(A)$ restriction is discussed in Section 2.3 with examples. This has an impact on the compatibility with other systems, since we can no longer express types like $\forall a.\ 1$. Yet we argue that such types are not very useful in practice; when $a \notin \mathrm{FV}(A)$, $\forall a.\ A$ is isomorphic to A in systems without explicit instantiation.

$\boxed{\Psi \vdash A \leq B \hookrightarrow E}$ Syntax-directed subtyping

$$\frac{}{\Psi \vdash 1 \leq 1 \hookrightarrow \lambda(x : 1).\ x}\ \leq\texttt{Unit} \qquad \frac{}{\Psi \vdash A \leq \top \hookrightarrow \lambda(x : |A|).\ \mathrm{top}}\ \leq\top$$

$$\frac{}{\Psi \vdash \bot \leq A \hookrightarrow \lambda(x : \forall a.\ a).\ x\ @|A|}\ \leq\bot \qquad \frac{a \in \Psi}{\Psi \vdash a \leq a \hookrightarrow \lambda(x : a).\ x}\ \leq\texttt{Var}$$

$$\frac{\tilde{a} \in \Psi}{\Psi \vdash \tilde{a} \leq \tilde{a} \hookrightarrow \lambda(x : a).\ x}\ \leq\texttt{SVar}$$

$$\frac{\Psi \vdash \tau \quad \Psi \vdash [\tau/a]A \leq B \hookrightarrow E \quad B \neq \forall b.B' \quad B \neq \top}{\Psi \vdash \forall a.\ A \leq B \hookrightarrow \lambda(x : |\forall a.\ A|).\ E\ (x\ @\tau)}\ \leq\forall\texttt{L}$$

$$\frac{\Psi, \tilde{a} \vdash [\tilde{a}/a]A \leq [\tilde{a}/a]B \hookrightarrow E}{\Psi \vdash \forall a.\ A \leq \forall a.\ B \hookrightarrow \lambda(x : |\forall a.\ A|).\ \Lambda a.\ E\ (x\ @a)}\ \leq\forall$$

$$\frac{\Psi \vdash B_1 \leq A_1 \hookrightarrow E_1 \quad \Psi \vdash A_2 \leq B_2 \hookrightarrow E_2}{\Psi \vdash A_1 \to A_2 \leq B_1 \to B_2 \hookrightarrow \lambda(f : |A_1 \to A_2|).\ \lambda(x : |B_1|).\ E_2\ (f\ (E_1\ x))}\ \leq\to$$

$\boxed{\Psi \vdash e \Leftarrow A \hookrightarrow E}$ e checks against input type A.

$\boxed{\Psi \vdash e \Rightarrow A \hookrightarrow E}$ e synthesizes output type A.

$\boxed{\Psi \vdash A \bullet e \Rrightarrow C \hookrightarrow E_c \mid E}$ Applying a function of type A to e synthesizes type C.

$$\frac{(x : A) \in \Psi}{\Psi \vdash x \Rightarrow A \hookrightarrow x}\ \texttt{DVar} \qquad \frac{\Psi \vdash e \Rightarrow A \hookrightarrow E \quad \Psi \vdash A \leq B \hookrightarrow co \quad A \neq \forall a.\ A'}{\Psi \vdash e \Leftarrow B \hookrightarrow co\ E}\ \texttt{DSub}$$

$$\frac{\Psi \vdash A \quad \Psi \vdash e \Leftarrow A \hookrightarrow E}{\Psi \vdash (e : A) \Rightarrow A \hookrightarrow E}\ \texttt{DAnno} \qquad \frac{\Psi \vdash \sigma \to \tau \quad \Psi, x : \sigma \vdash e \Leftarrow \tau \hookrightarrow E}{\Psi \vdash \lambda x.\ e \Rightarrow \sigma \to \tau \hookrightarrow \lambda(x : \sigma).\ E}\ \texttt{D}{\to}\texttt{I}{\Rightarrow}$$

$$\frac{\Psi, a \vdash e \Leftarrow A \hookrightarrow E}{\Psi \vdash e \Leftarrow \forall a.\ A \hookrightarrow \Lambda a.\ E}\ \texttt{D}\forall\texttt{I} \qquad \frac{\Psi, x : A \vdash e \Leftarrow B \hookrightarrow E}{\Psi \vdash \lambda x.\ e \Leftarrow A \to B \hookrightarrow \lambda(x : A).\ E}\ \texttt{D}{\to}\texttt{I}$$

$$\frac{}{\Psi \vdash () \Rightarrow 1 \hookrightarrow ()}\ \texttt{D1I}{\Rightarrow} \qquad \frac{\Psi \vdash e_1 \Rightarrow A \hookrightarrow E_1 \quad \Psi \vdash A \bullet e_2 \Rrightarrow C \hookrightarrow E_c \mid E_2}{\Psi \vdash e_1\ e_2 \Rightarrow C \hookrightarrow (E_c\ E_1)\ E_2}\ \texttt{D}{\to}\texttt{E}$$

$$\frac{\Psi \vdash e}{\Psi \vdash e \Leftarrow \top \hookrightarrow \mathrm{top}}\ \texttt{D}\top \qquad \frac{\Psi \vdash e \Leftarrow A \hookrightarrow E}{\Psi \vdash A \to C \bullet e \Rrightarrow C \hookrightarrow \lambda(x : |A \to C|).\ x \mid E}\ \texttt{D}{\to}\texttt{App}$$

$$\frac{\Psi \vdash e \Rightarrow \bot \hookrightarrow E}{\Psi \vdash e\ @B \Rightarrow \bot \hookrightarrow E}\ \texttt{DTA}\bot$$

$$\frac{\Psi \vdash \tau \quad \Psi \vdash [\tau/a]A \bullet e \Rrightarrow C \hookrightarrow E_c \mid E}{\Psi \vdash \forall a.\ A \bullet e \Rrightarrow C \hookrightarrow \lambda(x : |\forall a.\ A|).\ E_c\ (x\ @\tau) \mid E}\ \texttt{D}\forall\texttt{App}$$

$$\frac{\Psi, a \vdash e \Leftarrow A \hookrightarrow E}{\Psi \vdash \Lambda a.\ e : A \Rightarrow \forall a.\ A \hookrightarrow \Lambda a.\ E}\ \texttt{DSTV} \qquad \frac{\Psi \vdash e \Rightarrow \forall a.\ A \hookrightarrow E}{\Psi \vdash e\ @B \Rightarrow [B/a]A \hookrightarrow E\ @|B|}\ \texttt{DTA}\forall$$

$$\frac{\Psi \vdash e}{\Psi \vdash \bot \bullet e \Rrightarrow \bot \hookrightarrow \lambda(x : \forall a.\ a).\ x\ @(\top \to \forall a.\ a) \mid \mathrm{top}}\ \texttt{D}\bot\texttt{App}$$

Figure 3 Syntax-directed System.

3.2 Subtyping and Typing Rules

The top of Figure 3 shows the subtyping relation. Grayed parts are $F_{<:}$ [2] expressions, which are used to prove our soundness result with respect to $F_{<:}$, can be ignored for the moment. We refer the reader to the our extended version for the details of the $F_{<:}$ soundness result. Most rules are inherited from Odersky and Läufer (OL) [20], yet there are several differences. Rule $\leq$`SVar` is a new rule for subtype variables. This rule is just like the standard rule for type variables (rule $\leq$`Var`). Rules $\leq\top$ and $\leq\bot$ are new (but standard) rules for $\top$ and $\bot$. Rule $\leq\forall$ is also new. In this rule, two forall types are subtypes if their bodies are subtypes. Importantly, the type variables in the bodies become subtype variables and are marked as such when added to the context. Rule $\leq\forall$`L` has an two additional premises to prevent overlapping with rules $\leq\forall$ and $\leq\top$, respectively. The first condition $B \neq \forall b.\ B'$ ensures that $\leq\forall$ always has priority. The second condition $B \neq \top$ can be safely omitted without changing the expressive power, but is presented here to ensure that the system is syntax-directed.

The polymorphic subtyping relation behaves slightly differently from OL's subtyping. If we ignore $\top$ and $\bot$ types, this relation is weaker than OL. In $F^e_{<:}$, the order of polymorphic variables is important. For example, in the OL's system, the subtyping statement $\cdot \vdash \forall a.\ \forall b.\ a \to b \to a \leq \forall b.\ \forall a.\ a \to b \to a$ holds, but $F^e_{<:}$ will reduce that to $\tilde{a}, \tilde{b} \vdash \tilde{a} \to \tilde{b} \to \tilde{a} \leq \tilde{b} \to \tilde{a} \to \tilde{b}$, which does not hold. Apart from the ordering of variables, instantiation works differently for subtyping between polymorphic types. OL's subtyping relation accepts the following judgment $\cdot \vdash \forall a.\ (a \to a) \to (a \to a) \leq \forall a.\ a \to a$ but $F^e_{<:}$ does not, since $\tilde{a} \vdash (\tilde{a} \to \tilde{a}) \to (\tilde{a} \to \tilde{a}) \leq \tilde{a} \to \tilde{a}$ does not hold either.

Typing. The bottom of Figure 3 shows the type system. Compared to DK's type system, there are 4 groups of changes:

1. Rules `D`$\top$ and `D`$\bot$`App` are introduced for the $\top$ and $\bot$ types. Note that in rule `D`$\top$ we employ a relation $\Psi \vdash e$ that checks for the well-formedness of expressions. We omit the definition of $\Psi \vdash e$, but it is standard, checking whether all the free variables in e are bound in Ψ. Both rules are essential for the subsumption lemma to hold. For example, rule `D`$\bot$`App` is required to type-check the expression $(\lambda x.\ x\ ()) : \bot \to 1$, where the argument x has type $\bot$ and the application $x\ ()$ synthesizes $\bot$ according to the rule, which can then check against the 1 type by rule `DSub`.
2. Rule `DSub` now requires one side-condition to prevent overlapping with Rule `D`$\forall$`I`. In presence of explicit type applications, this condition cannot be eliminated.
3. Rules `DTA`$\bot$ and `DTA`$\forall$ infer type application expressions. If the type of e synthesizes a polymorphic type $\forall a.\ A$, then $e\ @B$ has type $[B/a]A$. Any expression of type $\bot$ will synthesize $\bot$ for any type applied.
4. Rule `DSTV` enables the scoped type variables [24]. This allows flexible control of type variables by the programmer.

Note that the `D`$\top$ is peculiar in that it allows some ill-typed terms to type-check. Such rules are often needed in bi-directional type systems with top types to enable properties such as checking subsumption. For instance a similar rule is employed by Dunfield [7]. Since the top type is the supertype of all types, all well-typed expressions should be able to type-check under the top type as well. For example, we should be able to change the type annotation in function $\lambda x.x : \text{Int} \to \text{Int}$ to $\lambda x.x : \top$. However, there is not enough type information to type-check the body of the later lambda. Nevertheless, we do not need to evaluate expressions with a top type, since no information can be extracted from such type, and the elaboration to $F_{<:}$ results directly in the *top* value for such expression, preserving type-safety.

3.3 Metatheory

The type system has several desirable properties, including subsumption and a stability of type substitutions lemma in subtyping.

Reflexivity and Transitivity. Firstly, our subtyping relation is reflexive and transitive.

▶ **Lemma 5** (Subtyping Reflexivity). *If* $\Psi \vdash A$ *then* $\Psi \vdash A \leq A$.

▶ **Lemma 6** (Subtyping Transitivity). *If* $\Psi \vdash A \leq B$ *and* $\Psi \vdash B \leq C$ *then* $\Psi \vdash A \leq C$.

Equivalence to Stable Subtyping and Stability. Secondly, the syntax-directed formulation of subtyping is sound and complete with respect to the stable subtyping relation in Section 2.6. Subtype variables are used to provide an alternative formulation of the $\leq\forall$ rule, bringing subtyping closer to an algorithm. Nonetheless, syntax-directed subtyping still guesses monotypes, thus it is not algorithmic.

▶ **Theorem 7** (Soundness w.r.t stable subtyping). *If* $\Psi \vdash A \leq B$ *then* $\Psi \vdash_s A \leq B$.

▶ **Theorem 8** (Completeness w.r.t stable subtyping). *Given* $\Psi \vdash A$ *and* $\Psi \vdash B$, *if* $\Psi \vdash_s A \leq B$ *then* $\Psi \vdash A \leq B$.

The proof works by generalizing instantiations for subtype variables and existential variables (which represents monotypes to be guessed). We refer to the extended version for details. A related property of the subtyping relation is stability. The following lemma generalizes Corollary 4 by allowing the subtype variable to appear anywhere in the context.

▶ **Lemma 9** (Stability of Subtyping, Generalized). *If* $\Psi \vdash A \leq B$ *and* $\Psi \vdash C$ *then* $\Psi \vdash [C/\tilde{a}]A \leq [C/\tilde{a}]B$.

This property ensures that any subtype variable can be replaced by a polytype C in two types A and B while preserving the subtyping relation between those two types.

The Subsumption Lemma. To prove the checking subsumption lemma, we first need to generalize the statement for inference and application inference judgments, as well as introduce a *context subtyping* relation, $\Psi \leq \Psi'$, to state the most general form.

▶ **Definition 10.** $\boxed{\Psi' \leq \Psi}$ *Context Subtyping*

$$\frac{}{\cdot \leq \cdot}\ \texttt{CS\_Empty} \qquad \frac{\Psi' \leq \Psi}{\Psi', a \leq \Psi, a}\ \texttt{CS\_TV} \qquad \frac{\Psi' \leq \Psi}{\Psi', \tilde{a} \leq \Psi, \tilde{a}}\ \texttt{CS\_STV}$$

$$\frac{\Psi' \leq \Psi \quad \Psi \vdash A' \leq A}{\Psi', x : A' \leq \Psi, x : A}\ \texttt{CS\_V}$$

Context Ψ subsumes context Ψ' if they bind the same variables in the same order, but the types of variables in Ψ' must be subtypes of those in Ψ. The generalized lemma is:

▶ **Lemma 11** (Subsumption). *Given* $\Psi' \leq \Psi$:
1. *If* $\Psi \vdash e \Leftarrow A$ *and* $\Psi \vdash A \leq A'$ *then* $\Psi' \vdash e \Leftarrow A'$;
2. *If* $\Psi \vdash e \Rightarrow B$ *then there exists* B' *s.t.* $\Psi \vdash B' \leq B$ *and* $\Psi' \vdash e \Rightarrow B'$;
3. *If* $\Psi \vdash A \bullet e \Rightarrow\!\!\!\Rightarrow C$ *and* $\Psi \vdash A' \leq A$, *then* $\exists C'$ *s.t.* $\Psi \vdash C' \leq C$ *and* $\Psi' \vdash A' \bullet e \Rightarrow\!\!\!\Rightarrow C'$.

This lemma expresses that any derivation in a context Ψ has a corresponding derivation in any context Ψ' that it subsumes.

Type variables	a, b	Subtype variables	$\tilde{a}, \tilde{b}$	Existential variables $\widehat{\alpha}, \widehat{\beta}$
Algorithmic types	A, B, C	$::=$	$\ldots \mid \widehat{\alpha}$	
Judgment chain	ω	$::=$	$A \leq B \mid e \Leftarrow A \mid e \Rightarrow_a \omega \mid A \bullet e \Rrightarrow_a \omega \mid A \circ B \Rrightarrow_a \omega$	
Algorithmic worklist	Γ	$::=$	$\cdot \mid \Gamma, a \mid \Gamma, \tilde{a} \mid \Gamma, \widehat{\alpha} \mid \Gamma, x : A \mid \Gamma \Vdash \omega$	
Declarative worklist	Ω	$::=$	$\cdot \mid \Omega, a \mid \Omega, \tilde{a} \mid \Omega, x : A \mid \Omega \Vdash \omega$	

Figure 4 Extended Syntax for the Algorithmic System (Extended from Figure 1).

Relating Subtype and Type Variables. The following lemma shows that we can substitute a subtype variable with a normal type variable, while preserving the subtyping relation.

▶ **Lemma 12.** *If* $\Psi[\tilde{a}] \vdash A \leq B$ *then* $\Psi[a] \vdash [a/\tilde{a}]A \leq [a/\tilde{a}]B$.

The reason is relatively straightforward. First, the substitution does not affect the $\tilde{a} \leq \tilde{a}$ sub-judgments. Second, substituting $\tilde{a}$ to a increases the range of implicit instantiation, which means that the monotypes picked in the old context are still well-formed under the new context. Note that the reverse statement does not hold. For example, $b \vdash \forall a.\ a \to a \leq b \to b$ holds with the predicative instantiation $a := b$, but $\tilde{b} \vdash \forall a.\ a \to a \leq \tilde{b} \to \tilde{b}$ does not; one cannot instantiate a with a non-monotype, or $\tilde{b}$ in this case.

4 Algorithmic System

This section introduces an algorithmic system that implements the syntax-directed specification of $F^e_{<:}$. The new algorithm is based on Zhao et al.'s [39] worklist algorithm but extended with explicit type applications and top and bottom types. In Section 5, we show that this algorithm is sound, complete and decidable with respect to the specification presented in Section 3. We also use color throughout this section to highlight differences to the original formulation by Zhao et al. [39].

4.1 Syntax and Well-Formedness

Figure 4 shows the syntax for the algorithmic version of $F^e_{<:}$. Similarly to the syntax-directed system, the well-formedness rules are unsurprising, ensuring well-scopedness for binders as well as the free variable constraint on polymorphic types. We refer to the extended version for well-formedness relations of algorithmic types, expressions, judgments, and worklists.

Existential Variables. The algorithmic system inherits the syntax of terms from the syntax-directed system and extends types with a new sort of variables – *existential variables*. Existential variables ($\widehat{\alpha}, \widehat{\beta}$) are introduced to help find unknown monotypes τ that appear in multiple rules of the syntax-directed system. In the algorithmic worklist, the position where existential variables are declared indicates the possible monotypes they can be solved to. Formally speaking, if $\widehat{\alpha}$ is introduced right after Γ, then $\widehat{\alpha}$ can only be solved to a monotype τ where $\Gamma \vdash \tau$. This behavior is derived from the well-formedness restriction of the rule $\leq\forall\mathsf{L}$. An important remark is that subtype variables are not considered to be monotypes, therefore no existential variable can be solved to a subtype variable.

Judgment Chains. Judgment chains ω, or judgments for short, are the core components of our algorithmic type-checking. There are five kinds of judgments in our system. Four of them are inherited from [39]: subtyping ($A \leq B$), checking ($e \Leftarrow A$), inference ($e \Rightarrow_a \omega$) and application inference ($A \bullet e \Rrightarrow_a \omega$). Type application inference $A \circ B \Rrightarrow_a \omega$ is new, and it is used to help with the inference in type application expressions ($e\ @B$). This judgment plays a role similar to application inference for regular applications. In type application inference judgments, the first type A is the type inferred from the expression e. The judgment is then reduced differently depending on whether A is a polymorphic type $\forall a.\ A'$ or $\bot$.

Subtyping and checking are relatively simple, since their results are only success or failure. However, inference, application inference, and type application inference judgments return a type that is used in subsequent judgments. We use a continuation-passing-style encoding to accomplish this, following the approach by Zhao et al. [39]. For example, the judgment chain $e \Rightarrow_a (a \leq B)$ contains two judgments: first we infer the type of the expression e, and then check if the inferred type is a subtype of B. The *unknown* type of e is represented by a type variable a, which is used as a placeholder in the second judgment to denote the type of e.

Worklist Judgments. Our algorithmic context Γ, or *worklist*, combines traditional contexts and judgment(s) into a single sort. The worklist is an *ordered* collection of both variable bindings and judgments. The order captures the scope: only the objects that come after a variable's binding in the worklist can refer to it. For example, $[\cdot, a, x : a \Vdash x \Leftarrow a]$ is a valid worklist, but $[\cdot \Vdash \underline{x} \Leftarrow \underline{a}, x : \underline{a}, a]$ is not (the underlined symbols refer to out-of-scope variables). This property also affects how the algorithm behaves regarding solving existential variables. By solving an existential variable $\widehat{\alpha}$ with any monotype that does not escape the scope of $\widehat{\alpha}$ preserves well-formedness of the whole worklist.

Notation and Form of the Algorithmic Rules. The algorithmic subtyping and typing reduction rules, defined in Figures 5 and 6, have the form $\Gamma \longrightarrow \Gamma'$. Since the worklist is a stack of variable definitions and judgment chains, the algorithm pops the first element, processes according to the rules, and possibly pushes simplified judgments back. The syntax $\Gamma \longrightarrow^* \Gamma'$ denotes multiple reduction steps. A worklist Γ is accepted by the algorithm iff $\Gamma \longrightarrow^* \cdot$. In other words a program successfully type-checks if all the work has been processed. Any new variable introduced to the r.h.s of the worklist Γ' is fresh implicitly, similarly to how we treat them in the conditions of other rules. We also adopt the notation $\Gamma[\Gamma_M]$ from the DK type system to denote the worklist $\Gamma_L, \Gamma_M, \Gamma_R$, where $\Gamma[\bullet]$ is the worklist $\Gamma_L, \bullet, \Gamma_R$ with a hole ($\bullet$). Hole notations with the same name implicitly share the same structure Γ_L and Γ_R. A multi-hole notation splits the worklist into more parts. For example, $\Gamma[\widehat{\alpha}][\widehat{\beta}]$ means $\Gamma_1, \widehat{\alpha}, \Gamma_2, \widehat{\beta}, \Gamma_3$.

4.2 Garbage Collection and Algorithmic Subtyping Rules

Figure 5 defines algorithmic rules on variables (garbage collection) and subtyping. Rules 1-4 pop variable declarations that are essentially garbage. Thanks to the nature of ordered context, those variables are no longer referred to by the remaining judgments, therefore removing them does not break the well-formedness of the worklist.

Subtyping rules. We can discern 3 groups of rules for algorithmic subtyping. The first group consists of rules 5-12, where all the rules are similar to their syntax-directed system counterparts. The most interesting one is rule 12, which reflects the changes in our syntax-directed system. A subtype variable $\tilde{a}$ is used to replace the bound variable in the polymorphic types $\forall a.\ A$ and $\forall a.\ B$ for further reduction. Rule 11 differs from rule $\leq\forall\mathtt{L}$ by introducing an existential variable $\widehat{\alpha}$ instead of guessing the monotype τ instantiation.

$\boxed{\Gamma \longrightarrow \Gamma'}$ Γ reduces to Γ'.

$$\Gamma, a \longrightarrow_1 \Gamma \qquad \Gamma, \widehat{\alpha} \longrightarrow_2 \Gamma \qquad \Gamma, \tilde{a} \longrightarrow_3 \Gamma \qquad \Gamma, x : A \longrightarrow_4 \Gamma$$

$$\Gamma \Vdash 1 \leq 1 \longrightarrow_5 \Gamma \qquad \Gamma \Vdash a \leq a \longrightarrow_6 \Gamma \qquad \Gamma \Vdash \tilde{a} \leq \tilde{a} \longrightarrow_7 \Gamma$$

$$\Gamma \Vdash A \leq \top \longrightarrow_8 \Gamma \qquad \Gamma \Vdash \bot \leq A \longrightarrow_9 \Gamma$$

$$\Gamma \Vdash A_1 \to A_2 \leq B_1 \to B_2 \longrightarrow_{10} \Gamma \Vdash A_2 \leq B_2 \Vdash B_1 \leq A_1$$

$$\Gamma \Vdash \forall a.\ A \leq B \longrightarrow_{11} \Gamma, \widehat{\alpha} \Vdash [\widehat{\alpha}/a]A \leq B \quad \text{when } B \neq \forall a.\ B' \text{ and } B \neq \top$$

$$\Gamma \Vdash \forall a.\ A \leq \forall a.\ B \longrightarrow_{12} \Gamma, \tilde{a} \Vdash [\tilde{a}/a]A \leq [\tilde{a}/a]B$$

$$\Gamma \Vdash \widehat{\alpha} \leq \widehat{\alpha} \longrightarrow_{13} \Gamma$$

$$\Gamma[\widehat{\alpha}] \Vdash \widehat{\alpha} \leq A \to B \longrightarrow_{14} [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}](\Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2] \Vdash \widehat{\alpha}_1 \to \widehat{\alpha}_2 \leq A \to B) \quad \text{when } \widehat{\alpha} \notin FV(A) \cup FV(B)$$

$$\Gamma[\widehat{\alpha}] \Vdash A \to B \leq \widehat{\alpha} \longrightarrow_{15} [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}](\Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2] \Vdash A \to B \leq \widehat{\alpha}_1 \to \widehat{\alpha}_2) \quad \text{when } \widehat{\alpha} \notin FV(A) \cup FV(B)$$

$$\Gamma[\widehat{\alpha}][\widehat{\beta}] \Vdash \widehat{\alpha} \leq \widehat{\beta} \longrightarrow_{16} [\widehat{\alpha}/\widehat{\beta}](\Gamma[\widehat{\alpha}][]) \qquad \Gamma[\widehat{\alpha}][\widehat{\beta}] \Vdash \widehat{\beta} \leq \widehat{\alpha} \longrightarrow_{17} [\widehat{\alpha}/\widehat{\beta}](\Gamma[\widehat{\alpha}][])$$

$$\Gamma[a][\widehat{\beta}] \Vdash a \leq \widehat{\beta} \longrightarrow_{18} [a/\widehat{\beta}](\Gamma[a][]) \qquad \Gamma[a][\widehat{\beta}] \Vdash \widehat{\beta} \leq a \longrightarrow_{19} [a/\widehat{\beta}](\Gamma[a][])$$

$$\Gamma[\widehat{\beta}] \Vdash 1 \leq \widehat{\beta} \longrightarrow_{20} [1/\widehat{\beta}](\Gamma[]) \qquad \Gamma[\widehat{\beta}] \Vdash \widehat{\beta} \leq 1 \longrightarrow_{21} [1/\widehat{\beta}](\Gamma[])$$

Figure 5 Algorithmic Variable and Subtyping Rules.

The second group is about solving existential variables (rule 13) and existential variable decomposition (rules 14 and 15). Rule 13 is one of the base cases involving existential variables. Rules 14 and 15 are algorithmic versions of Rule $\leq\!\!\to$; they both partially instantiate $\widehat{\alpha}$ to function types. The domain $\widehat{\alpha}_1$ and range $\widehat{\alpha}_2$ of the new function type are not determined immediately: they are fresh existential variables with the same scope as $\widehat{\alpha}$. The *occurs-check* condition prevents divergence as usual. For example, without it $\widehat{\alpha} \leq 1 \to \widehat{\alpha}$ would diverge.

The final group consists of rules 16-21, where each rule solves an existential variable against a basic type. Each rule removes an existential variable and substitutes it with its solution in the remaining worklist, which preserves well-formedness in the meantime. For example, Rule 16 solves variable $\widehat{\alpha}$ with $\widehat{\beta}$ only if $\widehat{\beta}$ occurs after $\widehat{\alpha}$. It is worth noting that none of these rules solves $\widehat{\alpha}$ to a subtype variable $\tilde{b}$. As we have discussed, $\top$, $\bot$ and subtype variables are not monotypes, therefore existential variables do not unify with them.

4.3 Algorithmic Typing Rules

Figure 6 shows the algorithmic rules for typing.

Checking Judgments. Rules 22-26 deal with checking judgments. Rule 22 is `DSub` written in a continuation-passing-style. The side conditions $e \neq \lambda x.\ e'$ and $B \neq \top$ prevent overlap with all other rules. Rules 23, 24 and 26 adapt their counterparts in the syntax-directed system, where rules 23 and 26 correspond to the new/changed rules introduced in the syntax-directed system compared to DK's work. Rule 25 is a special case of `D`$\to$`I`, dealing with the case when the input type is an existential variable, representing a monotype *function* as in the syntax-directed system. The same instantiation technique as in rules 14 and 15 applies.

$\boxed{\Gamma \longrightarrow \Gamma'}$ (cont.) Γ reduces to Γ'.

$$
\begin{array}{rl}
\Gamma \Vdash e \Leftarrow B & \longrightarrow_{22} \Gamma \Vdash e \Rightarrow_a a \leq B \\
 & \quad \text{when } e \neq \lambda x.\ e' \text{ and } B \neq \forall a.\ B' \text{ and } B \neq \top \\
\Gamma \Vdash e \Leftarrow \forall a.\ A & \longrightarrow_{23} \Gamma, a \Vdash e \Leftarrow A \\
\Gamma \Vdash \lambda x.\ e \Leftarrow A \to B & \longrightarrow_{24} \Gamma, x : A \Vdash e \Leftarrow B \\
\Gamma[\widehat{\alpha}] \Vdash \lambda x.\ e \Leftarrow \widehat{\alpha} & \longrightarrow_{25} [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}](\Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2], x : \widehat{\alpha}_1 \Vdash e \Leftarrow \widehat{\alpha}_2) \\
\Gamma \Vdash e \Leftarrow \top & \longrightarrow_{26} \Gamma \\
 & \\
\Gamma \Vdash x \Rightarrow_a \omega & \longrightarrow_{27} \Gamma \Vdash [A/a]\omega \quad \text{when } (x : A) \in \Gamma \\
\Gamma \Vdash (e : A) \Rightarrow_a \omega & \longrightarrow_{28} \Gamma \Vdash ([A/a]\omega) \Vdash e \Leftarrow A \\
\Gamma \Vdash (\Lambda a.\ e : A) \Rightarrow_b \omega & \longrightarrow_{29} \Gamma \Vdash ([\forall a.\ A/b]\omega), a \Vdash e \Leftarrow A \\
\Gamma \Vdash () \Rightarrow_a \omega & \longrightarrow_{30} \Gamma \Vdash [1/a]\omega \\
\Gamma \Vdash \lambda x.\ e \Rightarrow_a \omega & \longrightarrow_{31} \Gamma, \widehat{\alpha}, \widehat{\beta} \Vdash ([\widehat{\alpha} \to \widehat{\beta}/a]\omega), x : \widehat{\alpha} \Vdash e \Leftarrow \widehat{\beta} \\
\Gamma \Vdash e_1\ e_2 \Rightarrow_a \omega & \longrightarrow_{32} \Gamma \Vdash e_1 \Rightarrow_b (b \bullet e_2 \Rrightarrow_a \omega) \\
\Gamma \Vdash e\ @C \Rightarrow_a \omega & \longrightarrow_{33} \Gamma \Vdash e \Rightarrow_b (b \circ C \Rrightarrow_a \omega) \\
 & \\
\Gamma \Vdash \forall b.\ B \circ C \Rrightarrow_a \omega & \longrightarrow_{34} \Gamma \Vdash [([C/b]B)/a]\omega \\
\Gamma \Vdash \bot \circ C \Rrightarrow_a \omega & \longrightarrow_{35} \Gamma \Vdash [\bot/a]\omega \\
 & \\
\Gamma \Vdash A \to C \bullet e \Rrightarrow_a \omega & \longrightarrow_{36} \Gamma \Vdash ([C/a]\omega) \Vdash e \Leftarrow A \\
\Gamma \Vdash \forall a.\ A \bullet e \Rrightarrow_a \omega & \longrightarrow_{37} \Gamma, \widehat{\alpha} \Vdash [\widehat{\alpha}/a]A \bullet e \Rrightarrow_a \omega \\
\Gamma \Vdash \bot \bullet e \Rrightarrow_a \omega & \longrightarrow_{38} \Gamma \Vdash [\bot/a]\omega \\
\Gamma[\widehat{\alpha}] \Vdash \widehat{\alpha} \bullet e \Rrightarrow_a \omega & \longrightarrow_{39} [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}](\Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2] \Vdash \widehat{\alpha}_1 \to \widehat{\alpha}_2 \bullet e \Rrightarrow_a \omega)
\end{array}
$$

Figure 6 Algorithmic Typing Rules.

Inference judgments. Inference judgments accept an expression and return a type. Rules 27-33 deal with type inference judgments. The algorithm uses a continuation-passing-style encoding, where the output type is passed to the next judgment. When an inference judgment succeeds with type A, the algorithm continues to work on the inner-chain ω by substituting a by A in ω. Rule 27 and 30 are base cases (variable and unit), where the inferred type is passed to its child judgment chain. Rules 28 and 29 infer an annotated expression by changing into checking mode, therefore another judgment chain is created. Rule 29 deals with scoped type variables; the type variable a is in scope in e, and corresponds to the rule `DTA`$\forall$. Rule 31 infers the type of a lambda expression by introducing $\widehat{\alpha}, \widehat{\beta}$ as the input and output types of the function, respectively. Rule 32 infers the type of an application by firstly inferring the type of the function e_1. Then the remaining work is delegated to an application inference judgment, which passes a, representing the return type of the application, to the remainder of the judgment chain ω. Rule 33 is new: it first infers the type of e, then calls the type application inference judgment to compute the return type.

Type Application and Application Inference Judgments. Rules 34 and 35 deal with the new type application inference judgments. Rule 34 accepts a polymorphic input $\forall b.\ B$ and produces its instantiation $[C/b]B$. Rule 35 returns $\bot$ as it can be used as any type. For example, if we choose to treat $\bot$ as the polymorphic type $\forall b.\ \bot$, the result after type application is $\bot$ according to rule 34. Finally, Rules 36-39 deal with application inference judgments. Rules 36, 37 and 38 behave like rules `D∀App`, `D→App` and `D⊥App`, respectively. Rule 39 instantiates $\widehat{\alpha}$ to the function type $\widehat{\alpha}_1 \to \widehat{\alpha}_2$, just like Rules 14, 15 and 25.

5 Algorithmic Metatheory

This section presents the metatheory of the algorithmic system in Section 4. We show three main results: *soundness*, *completeness* and *decidability*. Our proofs employ similar techniques to the ones by Zhao et al. [39], so we only highlight the main results and differences.

5.1 Declarative Worklist and Transfer

To aid in formalizing the correspondence between the declarative and algorithmic systems, we use a declarative worklist Ω, defined in Figure 4. A declarative worklist Ω has the same structure as an algorithmic worklist Γ, but does not contain any existential variables $\widehat{\alpha}$.

Worklist Instantiation. We instantiate an algorithmic worklist Γ to the declarative worklist Ω by instantiating all existential variables $\widehat{\alpha}$ in Γ with well-scoped monotypes τ.

▶ **Definition 13.** $\boxed{\Gamma \rightsquigarrow \Omega}$ Γ *instantiates to* Ω.

$$\frac{}{\Omega \rightsquigarrow \Omega}\ {\rightsquigarrow}\Omega \qquad \frac{\Omega \vdash \tau \qquad \Omega, [\tau/\widehat{\alpha}]\Gamma \rightsquigarrow \Omega}{\Omega, \widehat{\alpha}, \Gamma \rightsquigarrow \Omega}\ {\rightsquigarrow}\widehat{\alpha}$$

Rule $\rightsquigarrow\widehat{\alpha}$ replaces the first (left-most) existential variable with a well-scoped monotype and repeats the process on the resulting worklist until no existential variable remains and thus the algorithmic worklist has become a declarative one. In order to maintain well-scopedness, the substitution is applied to all the judgments and term variable bindings in the scope of $\widehat{\alpha}$.

Declarative Worklist Reduction. A relation $\Omega \longrightarrow \Omega'$ is defined to reduce all judgments in the declarative worklists with declarative typing rules. This relation checks that every judgment entry in the worklist holds using a corresponding conventional declarative judgment. The typing contexts of declarative judgments are recovered using an auxiliary erasure function $\|\Omega\|$. The erasure function simply drops all judgment entries from the worklist, keeping only variable and type variable declarations. Both definitions are available in the extended version.

5.2 Soundness

Our algorithm is sound with respect to the declarative system. For any worklist Γ that reduces successfully, there is a valid instantiation Ω that transfers all judgments to the declarative system.

▶ **Theorem 14** (Soundness). *If* wf Γ *and* $\Gamma \longrightarrow^* \cdot$, *then* $\exists \Omega$ *s.t.* $\Gamma \rightsquigarrow \Omega$ *and* $\Omega \longrightarrow^* \cdot$.

The proof proceeds by induction on the derivation of $\Gamma \longrightarrow^* \cdot$. Most of the proof follows Zhao et al. [39]. Algorithmic type application rules are the most interesting change, because they have a different shape compared to the declarative rules. With the help of declarative worklist reduction, we can reduce the additional form of type application syntax and therefore indirectly build a relationship with the declarative system.

5.3 Completeness

Any derivation in the declarative system has an algorithmic counterpart:

▶ **Theorem 15** (Completeness). *If* wf Γ *and* $\Gamma \rightsquigarrow \Omega$ *and* $\Omega \longrightarrow^* \cdot$, *then* $\Gamma \longrightarrow^* \cdot$.

We prove completeness by induction on the derivation of $\Omega \longrightarrow^* \cdot$ with a similar technique to the one used by Zhao et al. [39]. New rules, including the ones involve subtype variables and type applications, do not increase the difficulty of our proof significantly. It is worth noting that our system forbids the $\top$ and $\bot$ types to be instantiated by monotypes. If we did not pose such restriction, then the following lemma would not hold anymore:

▶ **Lemma 16** (Prune Transfer for Instantiation). *If* $(\Gamma \Vdash \widehat{\alpha} \leq A \to B) \rightsquigarrow (\Omega \Vdash C \leq A_1 \to B_1)$ *and* $\|\Omega\| \vdash C \leq A_1 \to B_1$, *then* $\widehat{\alpha} \notin FV(A) \cup FV(B)$.

For example, allowing instantiations like $\widehat{\alpha} := \top$ would make the algorithmic judgment $\widehat{\alpha} \to \widehat{\alpha} \leq \widehat{\alpha}$ derivable. This lemma is essential to follow the original proof to prove completeness for the occurs-check condition in rules 14 and 15.

5.4 Decidability

Finally, we show that our algorithm is decidable:

▶ **Theorem 17** (Decidability). *Given* wf Γ, *it is decidable whether* $\Gamma \longrightarrow^* \cdot$ *or not.*

Our decidability proof is based on a lexicographic group of induction measures:

$$\left\langle |\Gamma|_e, |\Gamma|_{\Leftrightarrow}, |\Gamma|_{\top\bot}, |\Gamma|_{\forall}, |\Gamma|_{\widehat{\alpha}}, |\Gamma|_{\to} + |\Gamma| \right\rangle$$

on the worklist Γ. Compared with the measures used by Zhao et al. [39], we introduce a new measure $|\cdot|_{\top\bot}$, which counts the total number of $\top$ and $\bot$ occurrences. This is required because judgments like $\widehat{\alpha} \leq \top$ now do not solve $\widehat{\alpha}$, which breaks the original proof technique. This type of judgment now reduces the new measure by at least one. The rest of the proof follows the approach closely. The extended version has detailed explanations of the measures and proofs. Combining all three main results (soundness, completeness and decidability), we conclude that the declarative system is decidable by means of our algorithm.

▶ **Corollary 18** (Decidability of Declarative Typing). *Given* wf Ω, *it is decidable whether* $\Omega \longrightarrow^* \cdot$ *or not.*

6 Discussion

Inferring Top and Bottom Types. $F^e_{<:}$ does not treat the $\top$ and $\bot$ types as monotypes, therefore these types cannot be implicitly instantiated by the type inference algorithm. However, in certain programming languages, especially OOP languages with downcasts, the $\top$ type can be useful in certain cases and implicit instantiation would be convenient to have. For example, the following Java program

```
var ns = List.of(1, 2, "3");
```

should instantiate the generic variable `A` of the `List<A>` class to `Object` (note that `Object` plays a similar role to $\top$ in Java) to type-check the program. Thus, the inferred type for `ns` is `List<Object>`. In this program, because downcasts are possible in Java, it is plausible that the programmer intended to have a heterogeneous list of values, that could later be accessed by doing some type analysis for the elements and downcasting from `Object` to `Integer` or `String`. We consider such use cases to be a practical example where instantiation with the top type would be useful in languages like Java. In contrast, in $F^e_{<:}$, we would need to explicitly instantiate the type argument. Nonetheless, in a language without downcasts (such as $F^e_{<:}$), the declaration of `ns` above would very likely be a programmer error, since there would not be much that could be done with a value of type `List<Object>`. As we have argued in Section 2.4, there is a tension between inferring types with top and bottom types and hiding programmer errors: sometimes type errors that would be caught in many type systems, are instead type-checked by inferring some types with top and bottom types. Our design decision in $F^e_{<:}$ is not to infer top and bottom types, which avoids hiding such errors as well as avoiding the technical complexities that arise from inferring such types.

It is possible to have alternative designs for $F^e_{<:}$ that infer top and bottom types as well. For instance, if we would be aiming at covering common cases that arise in practice in languages like Java, such as the inference of the type of `ns` above, we could extend our syntax-directed system and algorithmic system with the rules:

$$\frac{\Psi \vdash [\top/a]A \leq B \quad B \neq \top}{\Psi \vdash \forall a.\ A \leq B}\ \texttt{DInst}\top$$

$$\Gamma \Vdash \forall a.\ A \leq B \longrightarrow \Gamma \Vdash [\top/a]A \leq B \quad \text{when } B \neq \top$$

The two rules above support simple forms of instantiation where the type variable is directly instantiated with the $\top$ type (a similar approach could be used for $\bot$ types). Note that these rules *overlap* with the current predicative instantiation rules, and thus introduce nondeterminism. Implementing the algorithmic rule directly would require some backtracking. From the theoretical point of view the rules are quite ad-hoc, since they cover only very specific cases of instantiation with top types. A more theoretically appealing approach would be to borrow ideas from approaches such as MLSub [6], which can infer types with top and bottom. However, this would be much more technically challenging. Another direction would be to complement the global type inference approach of $F^e_{<:}$ with some more local approach to attempt to infer top and bottom types. We will discuss this approach more next.

Local Impredicative Inference. Implicit impredicative instantiation is an advanced feature in modern type systems, and it is also supported by the local type inference approach [27]. Unlike top types, which can have some practical use cases in languages like Java, no existing mainstream OOP languages support higher-ranked systems with first-class polymorphic functions/values. Thus, there is no need for impredicative instantiations in those languages today. Nonetheless, future languages may support such feature and it is worthwhile considering impredicative type inference. Local type inference algorithms are designed to support impredicative instantiations through information in the neighbor nodes of the syntax tree. The recent work on Quick Look by [32] instantiates polymorphic types through a similar local approach and falls back to a global Hindley-Milner-style unification afterwards. Currently, our system only employs global unification and ignores any local information. We believe that a promising direction would be to follow the Quick Look approach, preserve the core global

inference system of $F^e_{<:}$, and try to employ a more local approach to infer impredicative types as well as top and bottom types before introducing unification variables. The main challenge in this direction is that Serrano et al.'s [32] approach relies on invariant subtyping for function types. In contrast, we have to deal with contravariance for input types and covariance for output types.

7 Related Work

This section discusses related work, focusing on the most closely related research on higher-ranked type inference and local type inference.

Hindley-Milner. The Hindley-Milner (HM) type system [5, 19, 16] was a landmark achievement in type inference. The constraint-based presentation by Pottier and Rémy [29] for HM and ML type inference has similarities with the worklist approach and it also keeps precise scoping of variables. In HM the order of universally quantified variables is irrelevant and no annotations are required. In contrast, in our work, the order of universally quantified variables matters, and annotations are necessary for polymorphic functions. Thus, we do not support Hindley-Milner style generalization. Nevertheless if we assume annotations of polymorphic expressions, the order-relevance of universally quantified variables is not problematic. Because of its support for visible type applications [13], GHC Haskell already distinguishes between *specified* and *generalized* type quantification. Specified type quantification refers to polymorphic expressions that have explicit type annotations. Like $F^e_{<:}$, in GHC type variables in specified quantification are order relevant to be compatible with explicit type applications. In contrast to Hindley-Milner, $F^e_{<:}$ supports higher-ranked polymorphism, explicit impredicative type applications and top and bottom types.

Higher-Ranked Polymorphic Type Inference. There has been much work extending HM while preserving all of its expressive power. In particular, there are several extensions of HM to System F, which support *higher-ranked polymorphism*. Since full type inference for System F is undecidable [38], such extensions need some type annotations or restrictions to remain decidable. The work on type inference for higher-ranked polymorphism (HRP) can be divided into two main lines: predicative and impredicative type systems. In predicative type systems, only monotypes can be inferred. An advantage of predicative type systems is that the predicative polymorphic subtyping relation is decidable [20], which facilitates the design of such type systems and type inference algorithms. There are several predicative HRP type systems [25, 8, 9, 20, 39]. The work in this paper is based on DK's [8] declarative type system and the algorithmic formulation by Zhao et al. [39]. However, we support explicit impredicative type applications and top and bottom types. Such features create various challenges and, to address some of those challenges, we introduce a novel stable polymorphic subtyping relation. In contrast, DK adopt the polymorphic subtyping relation by Odersky and Läufer [20]. In essence, with stable subtyping, the order of type variables becomes relevant in universal quantification. As a consequence, some forms of subtyping that are accepted by Odersky and Läufer's relation are rejected in our type system. Nonetheless, with those restrictions we retain important properties, such as checking subsumption and stability of type substitutions, in the presence of new features that are not supported by DK.

Impredicative System F allows instantiation with polymorphic types. Unfortunately, a subtyping relation with impredicative implicit instantiation is undecidable [3, 35]. Work on partial impredicative type inference algorithms [17, 18, 36, 33, 32, 14] navigate a variety of

design tradeoffs for a decidable algorithm. Ideas from *Guarded Impredicative Polymorphism* [33] and the *Quick Look* approach [32], are being adopted in GHC 9 for enabling impredicative instantiation. They make use of local information in n-ary applications to infer polymorphic instantiations with a relatively simple specification and unification algorithm. Although not all impredicative instantiations can be handled well, these approaches are useful in practice. In contrast to this line of work, we do not attempt to infer impredicative types. Instead, all impredicative instantiations must be explicit. While explicit instantiation is less convenient, an advantage is flexibility. Approaches that only allow implicit impredicative instantiation may reject some instantiations that would be possible with explicit instantiation.

Stability. Besides the motivation of supporting a form of impredicative polymorphism, another motivation for the changes in type inference in GHC 9 has been to simplify the algorithms and address various issues surrounding subsumption. While we are not aware that the issues that we have described in Section 2 have been previously identified, there have been several discussions documenting other issues related to subsumption in GHC 8 [26]. Recently, motivated to understand what would be the best design for instantiation in GHC, Bottu and Eisenberg [1] have compared four different approaches to instantiation. They have identified stability properties as an important factor for language designers to take into consideration when designing languages with implicit instantiation. Stability also plays an important role in the Cochis calculus [31], where it ensures that the behavior of resolution (which is a mechanism employed by type classes [37] or Scala implicits [22]) is preserved after instantiation. Stable subtyping in $F^{e}_{<:}$ provides a high-level specification of polymorphic subtyping, which essentially embeds a stability property into the subtyping relation.

Type Inference with Explicit Type Applications. The work on *visible type application* (VTA) [13] adds a predicative form of explicit type application to HM and HRP type systems. This approach has been adopted in GHC 8. As discussed in detail in Section 2, a property that is not enforced in VTA is checking subsumption. We believe that checking subsumption is an important property, as it ensures that a program can always be annotated with a supertype and it can prevent situations where simply inlinings of function definitions can make a well-typed program ill-typed. The Quick Look approach [32] supports impredicative visible type application. An important difference is that in Quick Look subtyping of functions is invariant, whereas in the original VTA approach the standard subtyping rule is used. The invariant subtyping rule prevents the counter-examples to checking subsumption that we found in GHC 8, and described in Section 2. Our work shows a different way to prevent such examples, by employing stable subtyping with a standard subtyping rule for functions. We believe that there are merits in both approaches. The restrictions adopted by Quick Look do not affect backward compatibility with the HM type system. In contrast, elementary type inference does not aim at backwards compatibility with the HM type system. Instead, we are interested in backward compatibility with extensions of System F with subtyping (such as $F_{<:}$ [2]). Quick Look would not preserve backward compatibility to such type systems, which employ a standard subtyping rule for function types.

Local Type Inference. While technically speaking we are closest to predicative HRP, we are closer in spirit and in goals to local type inference [27, 21]. Like our work, local type inference does not aim to subsume the HM type system. Local type inference sacrifices some of the expressive power of type inference, in exchange for the ability to smoothly deal with features such as top and bottom types and impredicative types. Pierce and Turner

considered a language similar to the language that we consider in our work (with top and bottom types, but no bounded quantification). Like our approach, both implicit and explicit type applications are supported. Technically speaking, our approach is still a global inference approach, and thus it is quite different from local type inference. In local type inference, missing annotations are recovered using only information from adjacent nodes in the syntax tree, and there are no long-distance constraints such as unification variables. We believe that an advantage of $F^e_{<:}$ is that it has simple and clear syntax-directed specifications, whereas the specification of local type inference is more involved, and it is not obvious to programmers when instantiation works or not. Furthermore, $F^e_{<:}$ allows the inference of lambda expressions without any contextual type information, as long as the inferred type is a monotype.

Type Inference with Subtyping. Another line of work is extensions of HM with subtyping. Type systems in presence of subtyping encounter constraints that are not simply equalities as in HM. Therefore constraint solvers used in HM, where unifications are based on equality, cannot be easily extended to support subtyping. Instead, constraints are usually collected as subtyping relations and may delay resolution as the constraints accumulate. Some systems that are based on *constraint types* [12, 11], i.e. types expressed together with a set of constraints $\tau \mid \{\overline{\tau_1 \leq \tau_2}\}$. Unfortunately, such constraints can be quite large and hard to interpret by programmers. Pottier [30] proposed three methods to simplify constraints, aiming at improving the efficiency of type inference algorithms and improving the readability of the resulting types. Inspired by the simplification strategies of Pottier, MLsub [6] suggests that the data flow on the constraint graph can be reflected directly on types in a richer type system. Simple-sub [23] further simplifies the algorithm of MLsub and is implemented in 500 lines of code. While being equivalent to MLsub, it is a more efficient variant. In our work, we avoid subtyping constraints and do not infer types with top and bottom types. If instantiations with such types are needed, then an explicit type application must be used. On the other hand, we support higher-ranked polymorphism, and explicit type applications, which (as far as we know) are not supported by any extensions of HM with subtyping.

8 Conclusion

In this paper, we proposed elementary type inference: a partial form of type inference that can be used in languages with subtyping that combine implicit instantiation with explicit type applications. As type systems become more powerful, the inference problem becomes harder, quickly leading to various undecidable problems. It is clear that some form of type-inference is needed in most languages to make their use practical. However, it is not necessarily true that being able to infer more types is always better, especially if there is the possibility to resort to explicit instantiation. Attempting to infer more types may have the side-effect of hiding programmer errors, as very general types can be inferred in the presence of advanced type system features. Moreover, *predicatibility* of what can be inferred and what cannot is also an important factor for users of the programming language. Elementary type inference strikes a compromise. It chooses to infer only monotypes, which are always inferrable, and makes it easy to understand when the instantiation succeeds or fails. For polytypes (which include top and bottom), explicit type applications must be used, but no expressive power is sacrificed. More work is needed to understand what is the right balance between inference, predicatibility and usability of languages in the future.

References

1 Gert-Jan Bottu and Richard A. Eisenberg. Seeking stability by being lazy and shallow: Lazy and shallow instantiation is user friendly. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*, pages 85–97, New York, NY, USA, 2021. Association for Computing Machinery.

2 L. Cardelli, S. Martini, J.C. Mitchell, and A. Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1):4–56, 1994.

3 Jacek Chrząszcz. Polymorphic subtyping without distributivity. In Luboš Brim, Jozef Gruska, and Jiří Zlatuška, editors, *Mathematical Foundations of Computer Science 1998*, pages 346–355, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

4 Maurizio Cimadamore. Jep 101: Generalized target-type inference, 2015. URL: `http://openjdk.java.net/jeps/101`.

5 Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, 1982.

6 Stephen Dolan and Alan Mycroft. Polymorphism, subtyping, and type inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 60–72, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3009837.3009882`.

7 Jana Dunfield. Elaborating intersection and union types. *J. Funct. Program.*, 24(2-3):133–165, 2014. `doi:10.1017/S0956796813000270`.

8 Jana Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, 2013.

9 Jana Dunfield and Neelakantan R. Krishnaswami. Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. *PACMPL*, (POPL), January 2019. `arXiv:1601.05106`.

10 Jana Dunfield and Neelakantan R. Krishnaswami. Bidirectional typing. *ACM Comput. Surv.*, 54(5), May 2021.

11 Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 1995, Austin, Texas, USA, October 15-19, 1995*, OOPSLA '95, pages 169–184, New York, NY, USA, 1995. Association for Computing Machinery. `doi:10.1145/217838.217858`.

12 Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. *Electronic Notes in Theoretical Computer Science*, 1:132–153, 1995. MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference. `doi:10.1016/S1571-0661(04)80008-2`.

13 Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. Visible type application. In Peter Thiemann, editor, *Programming Languages and Systems*, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

14 Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan Coates. Freezeml: Complete and easy type inference for first-class polymorphism. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 423–437. Association for Computing Machinery, 2020.

15 Andrew Gacek. The Abella interactive theorem prover (system description). In *Proceedings of IJCAR 2008*, Lecture Notes in Artificial Intelligence, 2008.

16 Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

17 Didier Le Botlan and Didier Rémy. MLF: Raising ML to the power of system F. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, 2003.

18 Daan Leijen. HMF: Simple type inference for first-class polymorphism. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, 2008.

19 Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

20 Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, 1996.

21 Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 41–53, New York, NY, USA, 2001. Association for Computing Machinery.

22 Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360, 2010.

23 Lionel Parreaux. The simple essence of algebraic subtyping: Principal type inference with subtyping made easy (functional pearl). *Proc. ACM Program. Lang.*, 4(ICFP), August 2020. doi:10.1145/3409006.

24 Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. Draft, 2004. URL: http://research.microsoft.com/en-us/um/people/simonpj/papers/scoped-tyvars/.

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

26 Peyton Jones, Simon. Simplify Subsumption. *GHC Proposals*, 2020. URL: https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0287-simplify-subsumption.rst.

27 Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, January 2000.

28 Hubert Plociniczak. *Decrypting Local Type Inference*. PhD thesis, EPFL, 2016.

29 François Pottier and Didier Rémy. *Advanced Topics in Types and Programming Languages*, chapter The Essence of ML Type Inference, pages 387–489. The MIT Press, 2005.

30 François Pottier. *Type inference in the presence of subtyping: from theory to practice*. PhD thesis, INRIA, 1998.

31 Tom Schrijvers, Bruno C. d. S. Oliveira, Philip Wadler, and Koar Marntirosian. COCHIS: Stable and coherent implicits. *Journal of Functional Programming*, 29:e3, 2019.

32 Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. A quick look at impredicativity. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020.

33 Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. Guarded impredicative polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, 2018.

34 Zhendong Su, Alexander Aiken, Joachim Niehren, Tim Priesnitz, and Ralf Treinen. The first-order theory of subtyping constraints. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 203–216, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/503272.503292.

35 Jerzy Tiuryn and Pawel Urzyczyn. The subtyping problem for second-order types is undecidable. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, 1996.

36 Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. FPH: First-class polymorphism for Haskell. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, 2008.

37 P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. Association for Computing Machinery.

38 Joe B Wells. Typability and type checking in system F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1-3):111–156, 1999.

39 Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. A mechanical formalization of higher-ranked polymorphic type inference. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.