Greedy Implicit Bounded Quantification

CHEN CUI, The University of Hong Kong, China
SHENGYI JIANG, The University of Hong Kong, China
BRUNO C. D. S. OLIVEIRA, The University of Hong Kong, China

Mainstream object-oriented programming languages such as Java, Scala, C#, or TypeScript have polymorphic type systems with subtyping and *bounded quantification*. Bounded quantification, despite being a pervasive and widely used feature, has attracted little research work on type-inference algorithms to support it. A notable exception is *local type inference*, which is the basis of most current implementations of type inference for mainstream languages. However, support for bounded quantification in local type inference has important restrictions, and its non-algorithmic specification is complex.

In this paper, we present a variant of kernel $F_{\leq}$, which is the canonical calculus with bounded quantification, with *implicit* polymorphism. Our variant, called $F^{b}_{\leq}$, comes with a declarative and an algorithmic formulation of the type system. The declarative type system is based on previous work on bidirectional typing for predicative higher-rank polymorphism and a greedy approach to implicit instantiation. This allows for a clear declarative specification where programs require few type annotations and enables *implicit polymorphism* where applications omit type parameters. Just as local type inference, explicit type applications are also available in $F^{b}_{\leq}$ if desired. This is useful to deal with *impredicative* instantiations, which would not be allowed otherwise in $F^{b}_{\leq}$. Due to the support for impredicative instantiations, we can obtain a completeness result with respect to kernel $F_{\leq}$, showing that all the well-typed kernel $F_{\leq}$ programs can type-check in $F^{b}_{\leq}$. The corresponding algorithmic version of the type system is shown to be *sound*, *complete*, and *decidable*. All the results have been mechanically formalized in the Abella theorem prover.

CCS Concepts: • **Theory of computation → Type theory**.

Additional Key Words and Phrases: Bounded Quantification, Mechanical Formalization, Type Inference

ACM Reference Format:
Chen Cui, Shengyi Jiang, and Bruno C. d. S. Oliveira. 2023. Greedy Implicit Bounded Quantification. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 295 (October 2023), 29 pages. https://doi.org/10.1145/3622871

1 INTRODUCTION

Bounded quantification [Cardelli and Wegner 1985] is an extension of parametric polymorphism, where type variables can have subtyping bounds. Mainstream object-oriented programming languages such as Java, Scala, C#, or TypeScript have polymorphic type systems with subtyping and bounded quantification. The canonical calculus for bounded quantification is $F_{\leq}$ [Cardelli et al. 1991; Cardelli and Wegner 1985; Curien and Ghelli 1992], which is an extension of System F [Girard 1972; Reynolds 1974] with subtyping and bounded quantifiers. Like System F, polymorphism in $F_{\leq}$ is *explicit*, which means that explicit type applications are necessary to instantiate the type arguments of polymorphic functions. There are two well-known variants of $F_{\leq}$ in the literature.

Authors' addresses: Chen Cui, ccui@cs.hku.hk, The University of Hong Kong, Department of Computer Science, Hong Kong, China; Shengyi Jiang, shengyi.jiang@outlook.com, The University of Hong Kong, Department of Computer Science, Hong Kong, China; Bruno C. d. S. Oliveira, bruno@cs.hku.hk, The University of Hong Kong, Department of Computer Science, Hong Kong, China.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/10-ART295
https://doi.org/10.1145/3622871

The *full* variant of $F_{\leq}$ [Curien and Ghelli 1992] is the more expressive one but, unfortunately, it is well-known that subtyping for full $F_{\leq}$ is undecidable [Pierce 1992]. The other variant is *kernel* $F_{\leq}$ [Cardelli and Wegner 1985], which is the most well-known and used decidable variant of $F_{\leq}$. Since we are interested in decidable type systems in this paper, we will focus on kernel $F_{\leq}$.

A practical drawback of explicit polymorphism in System F or $F_{\leq}$ is that programs require too much type information to type check. Therefore, to make programming languages with (bounded) parametric polymorphism practical, *implicit* polymorphism is needed. For System F, there have been many proposals for *global* type inference with implicit polymorphism [Dunfield and Krishnaswami 2013; Le Botlan and Rémy 2003; Leijen 2008; Odersky and Läufer 1996; Peyton Jones et al. 2007; Serrano et al. 2018; Vytiniotis et al. 2008], typically designed as extensions of the Hindley-Milner type system (HM) [Hindley 1969; Milner 1978]. One well-known strand of work is type systems with *predicative* Higher Ranked Polymorphism (HRP). In the context of HRP, predicative means that instantiations can only be *monotypes*, which are types that are not polymorphic. This imposes a restriction compared to System F, where instantiations can also be polymorphic types. Nonetheless, predicative HRP has been shown to work quite well in practice for many applications since most instantiations required in practice use monotypes.

One extension to predicative HRP is to combine implicit instantiation with explicit type applications. Eisenberg et al. [2016] proposed extensions to both HM and a predictive HRP system with *predicative* explicit type applications. With explicit type applications, programmers can choose their own instantiations by using type applications similar to System F. A complication that arises in combining implicit instantiation with explicit type applications is that the usual predicative subtyping relation for HRP, due to Odersky and Läufer [1996], is incompatible with explicit type applications. Eisenberg et al. [2016] proposed modified subtyping relations that are compatible with predicative explicit type applications. More recently, Zhao and Oliveira [2022] proposed to extend a predicative HRP type system, called $F^{e}_{\leq}$, with *impredicative* explicit type applications. The impredicativity of explicit type applications requires further changes to the HRP subtyping relation.

There is little work on predicative HRP type inference with bounded quantification. Zhao and Oliveira consider a subtyping relation with top and bottom types, but $F^{e}_{\leq}$ does not have bounded quantification. Cardelli [1993] implemented a type-inference algorithm for $F_{\leq}$, but did not formally study it. The only formal study that we are aware of supporting bounded quantification is by Sequeira [1998], who has studied a calculus called $\mathrm{ML}_{\forall\leq}$: a System F variant with bounded quantification. The kind of type inference supported by Sequeira's work is quite ambitious since $\mathrm{ML}_{\forall\leq}$ supports let generalization and the inference of principal types, just as HM. However, as Sequeira admits, his algorithm is impractical because it infers subtyping constraints without simplification. For example, he notes that the type inferred for a quicksort algorithm has around 300 constraints. Thus, many inferred types are too large and difficult to understand for humans.

The de facto technique to deal with type inference for languages with bounded quantification is *local type inference* [Odersky et al. 2001; Pierce and Turner 2000]. Unlike HM and global type inference HRP approaches, local type inference does not employ long-distance constraints such as unification variables and only employs a local constraint solver to infer type arguments in polymorphic applications. This is less powerful than global approaches but, as argued by Pierce and Turner, has the advantage of scaling up to features that are hard to deal with using global type-inference approaches. For instance, local type inference techniques can deal with subtyping and impredicative instantiations. Despite the technical differences, both (predicative) HRP and local type inference target the problem of type inference for higher-rank polymorphic languages. Moreover, local type inference complements implicit polymorphism with support for explicit type applications, like some predicative HRP approaches. Programmers can explicitly provide type instantiations when automatic (implicit) inference fails to find them.

Local type inference has more modest goals regarding type inference compared to HM-style inference and many HRP type-inference approaches. In particular, local type inference does not provide any mechanism to infer the types of arguments for (top-level) functions and does not have to deal with the question of finding principal types for polymorphic functions. In this way, the issues with Sequeira's approach are a non-issue for local type inference since all polymorphic functions are given explicit type annotations. While local type inference has shown its value in practice, it has some important limitations that were identified by Hosoya and Pierce [1999]; Pierce and Turner [2000]: *hard-to-synthesize arguments*; *no best argument*; and *no support for interdependent bounds*. In particular, for bounded quantification Pierce and Turner did not know how to extend the algorithm to support interdependent bounds such as $\forall(a \leq \top).\ \forall(b \leq a).\ \cdots$ where the type variable b is bounded by another type variable a. Thus, their algorithm cannot infer such implicit instantiations. In Section 2.2 we will discuss these limitations in more detail. Especially in the presence of bounded quantification, local type inference requires a complex specification that makes it hard to predict when instantiations will be successfully inferred.

In this paper, we present a variant of kernel $F_{\leq}$, called $F_{\leq}^{b}$, with implicit predicative polymorphism as well as explicit impredicative type applications. The declarative type system of $F_{\leq}^{b}$ is simple and clear. $F_{\leq}^{b}$ extends Zhao and Oliveira's $F_{\leq}^{e}$ calculus with bounded quantification. In the design of $F_{\leq}^{e}$, the authors follow a philosophy similar to local type inference but instead advocate for a more modest form of *global* type inference: i.e., no let generalization or automatic inference of (top-level) polymorphic functions is allowed. Moreover, *easy* (predicative) instantiations can *always* be guessed automatically, but hard (impredicative) instantiations require explicit type applications. They call this a more modest form of global type inference *elementary type inference*. $F_{\leq}^{b}$ adopts a similar philosophy. Furthermore, implicit instantiations in $F_{\leq}^{b}$ support interdependent bounds and deal with other weaknesses of local type inference, and $F_{\leq}^{b}$ can type check all kernel $F_{\leq}$ programs.

In $F_{\leq}^{b}$, a critical question is what is a *monotype*? The approach for finding implicit instantiations is *greedy* [Cardelli 1993]. That is, once an instantiation is found, $F_{\leq}^{b}$ commits to that instantiation, regardless of other possible instantiations that can be found later. This greedy approach follows previous predicative HRP approaches, which have the property that monotype subtyping implies equality of monotypes. This ensures that if a monotype works for the first instantiation, it also works for the subsequent instantiations. However, without care, the property easily breaks in the presence of more expressive subtyping relations. In addition, it turns out that in $F_{\leq}^{b}$, if all the type variables are considered monotypes, the transitivity of subtyping is lost! Thus, while typically monotypes are defined in a purely syntactic way, this is not the case in $F_{\leq}^{b}$. In particular, whether or not type variables are considered to be monotypes depends on their bounds. With careful crafting of what constitutes a monotype, we can ensure that the greedy approach is always successful and the transitivity of subtyping is preserved.

The algorithmic version of the type system is shown to be *sound*, *complete*, and *decidable*. We introduce two new algorithmic techniques: *polytype splitting* and *worklist substitutions*. Polytype splitting is a simple technique where functional existential (or unification) variables are only split into two existential variables if they unify with a polytype. Polytype splitting has the property that the number of splits is statically determined, which helps derive a simpler measure for decidability. Worklist substitutions simplify the presentation and proofs of the algorithmic system by dealing with necessary reorderings that arise from widening the scope of existential variables. We also discuss a variant with a more general notion of monotype. The declarative system of this system still preserves all the desirable properties, but the greedy algorithm is no longer complete. All the results have been mechanically formalized in the Abella theorem prover.

In summary, the contributions of this paper are:

- **A declarative bidirectional type system** with predicative implicit bounded quantification and impredicative explicit type applications. The declarative type system guesses monotypes. Monotypes are unconventional in that they require a non-syntactic formulation due to the presence of bounds in type variables.
- **An algorithmic formulation of the type system**, which is shown to be *sound*, *complete* and *decidable* with respect to the declarative formulation. The algorithm employs a worklist formulation inspired by Zhao et al. [2019]'s work.
- **New algorithmic techniques:** We introduce two new algorithmic techniques: *polytype splitting* and *worklist substitutions*. Both techniques are helpful in providing a simple algorithm, and they also simplify some of the key proofs, in particular decidability.
- **A variant of $F^b_\leq$ with monotype subtyping:** In Section 5, we present a variant of $F^b_\leq$ that deals with subtyping between monotypes during implicit instantiation. This variant has a more general notion of monotypes, and infers more instantiations at the cost of completeness.
- **Completeness with respect to kernel $F_\leq$:** $F^b_\leq$'s type system is shown to be complete with respect to kernel $F_\leq$. In other words, all the programs that type check in $F_\leq$ also type check in $F^b_\leq$.
- **Mechanical formalization and implementation:** All the calculi and proofs in this paper have been mechanically formalized in the Abella theorem prover [Gacek 2008]. We also have a simple Haskell implementation, which type checks all the $F^b_\leq$ examples in this paper. The supplementary materials, including the implementation, proofs, and the extended version of the paper are available at: https://doi.org/10.5281/zenodo.8202095.

2 OVERVIEW

We start with a background on implicit (predicative) higher-rank polymorphic (HRP) type inference and elementary type inference [Zhao and Oliveira 2022]. Then, we discuss the limitations of local type inference and its variants in dealing with languages with $F_\leq$ features and bounded quantification. Finally, we illustrate the key ideas in our work that address some of these limitations and discuss technical innovations over previous work.

2.1 Background: Predicative and Elementary Type Inference

Higher-rank polymorphism for languages à la System F allows universal types to appear in arbitrary positions in types, which lifts restrictions of top-level universal types in the Hindley-Milner type system. The $F^e_\leq$ calculus [Zhao and Oliveira 2022] additionally supports impredicative instantiation by explicit type applications provided by the programmer, which allows it to type-check all the programs typable in System F. There are several important design principles of $F^e_\leq$ that are in place to preserve properties such as subsumption and inferring *all* easy instantiations.

HRP subtyping and explicit type applications. The introduction of explicit type applications invalidates some subtyping rules commonly used in predicative HRP systems, and in particular in the well-known subtyping relation for HRP by Odersky and Läufer [1996]. The two key rules in Odersky and Läufer's subtyping relation are:

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

These rules enable order-irrelevant universal quantifiers. For instance, both of the following subtyping relations hold: $\forall a.\, \forall b.\, a \to b \leq \forall b.\, \forall a.\, a \to b$ and $\forall b.\, \forall a.\, a \to b \leq \forall a.\, \forall b.\, a \to b$. Thus, $\lambda x.\, x\ 3$ can be checked against both $(\forall a.\, \forall b.\, a \to b) \to \text{Bool}$ and $(\forall b.\, \forall a.\, a \to b) \to \text{Bool}$, by instantiating a and b to Int and Bool, respectively. However, explicit type applications create problems.

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	$::=$	$1 \mid a \mid \tau \to \sigma$
Contexts	Ψ	$::=$	$\cdot \mid \Psi, a \mid \Psi, x : A \mid \Psi, \tilde{a}$

$\boxed{\Psi \vdash A \leq B}$ A is a subtype of B

$$\frac{}{\Psi \vdash 1 \leq 1}\ \leq\texttt{Unit} \qquad \frac{}{\Psi \vdash A \leq \top}\ \leq\top \qquad \frac{}{\Psi \vdash \bot \leq A}\ \leq\bot \qquad \frac{a \in \Psi}{\Psi \vdash a \leq a}\ \leq\texttt{Var}$$

$$\frac{\tilde{a} \in \Psi}{\Psi \vdash \tilde{a} \leq \tilde{a}}\ \leq\texttt{SVar} \qquad \frac{\Psi \vdash \tau \quad \Psi \vdash [\tau/a]A \leq B \quad B \text{ is not a } \forall \text{ type}}{\Psi \vdash \forall a.\, A \leq B}\ \leq\forall\texttt{L}$$

$$\frac{\Psi, \tilde{a} \vdash [\tilde{a}/a]A \leq [\tilde{a}/a]B}{\Psi \vdash \forall a.\, A \leq \forall a.\, B}\ \leq\forall \qquad \frac{\Psi \vdash B_1 \leq A_1 \quad \Psi \vdash A_2 \leq B_2}{\Psi \vdash A_1 \to A_2 \leq B_1 \to B_2}\ \leq\to$$

Fig. 1. Declarative Syntax and Subtyping Rules of $F^e_\leq$.

Consider another expression with an explicit type application: $\lambda x.\, (x \;@\text{Int}\; 3)$. This expression can be checked against $(\forall a.\, \forall b.\, a \to b) \to \text{Bool}$, but $\lambda x.\, (x \;@\text{Int}\; 3) \Leftarrow (\forall b.\, \forall a.\, a \to b) \to \text{Bool}$ does not hold. The problem is that the type application instantiates the wrong universal quantifier in the latter expression.

In addition, the possibility of impredicative explicit instantiations introduces two subtler problems. Firstly, impredicative type applications, such as $f \;@(\forall a.\, a \to a)$, also interact with the subtyping relation. As shown by Zhao and Oliveira, some impredicative type applications break important stability-of-type-substitutions lemma for subtyping if no restrictions are imposed. Secondly, unused type variables in universal quantifiers (for instance, $\forall a.\, Int$) are equally problematic. We refer to Zhao and Oliveira's work for more details on these issues.

Subtyping in $F^e_\leq$. To address abovementioned problems, $F^e_\leq$ imposes three restrictions compared to Odersky and Läufer's subtyping relation. Subtyping for $F^e_\leq$ is shown in Figure 1. The first restriction is to replace ≤∀R with a more restrictive rule (≤∀) that only allows comparing two universal quantifiers. This has the consequence of making the order of the universal quantifiers relevant, forbidding subtyping statements such as $\forall a.\, \forall b.\, a \to b \leq \forall b.\, \forall a.\, a \to b$. The second restriction is to introduce a new sort of variables, $\tilde{a}$, called subtype variables, which is used by the new rule ≤∀. A subtype variable is *not a monotype*, contrary to the conventional type variable a, therefore it cannot be instantiated with rule ≤∀L. This restriction is key to ensuring that the stability of type substitutions lemma for subtyping holds. The final restriction is to add two additional checks in well-formedness, to ensure no unused variables in universal types.

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

No inference of $\top$ and $\bot$. $F^e_\leq$ does not support implicit instantiation of type $\bot$ and $\top$, by treating them as non-monotypes. There are two reasons for this. The first reason is technical: allowing unification with $\top$ and $\bot$ introduces considerable complications in the unification procedure. Even a simple $\widehat{\alpha} \leq \widehat{\beta}$ and $\widehat{\beta} \leq \widehat{\alpha} \to 1$, produced by $\forall a.\, (a \to a \to 1) \to 1 \leq (\forall b.\, b \to b) \to 1$, can have infinitely many solutions by picking $\widehat{\alpha} = \widehat{\beta} = \bot \mid \top \to 1 \mid \top \to \bot \mid \cdots$ and no one is the best.

The second reason is practical: inference of $\top$ and $\bot$ types can often mask type errors. For instance, $\lambda f.\, (f + f\; 1)$ can be typed with the type $\bot \to \text{Int}$, but it is likely to be a programmer's

error: possibly the programmer forgot an argument in the first occurrence of f. When programmers are certain that they need a type involving $\top$ or $\bot$, they can inform the type-inference algorithm using type annotations or explicit type applications. Section 2.2 shows more examples that illustrate some undesirable consequences of enabling the inference of $\top$ and $\bot$.

$F_{\leq}^{e}$ uses the same greedy solving strategy for implicit instantiation as previous predicative HRP systems [Dunfield and Krishnaswami 2013; Odersky and Läufer 1996]. Moreover (like other HRP systems), $F_{\leq}^{e}$ guarantees that monotype instantiations can *always* be inferred. There is a folklore property for such a greedy solving of implicit instantiations on predicative HRP systems to work: subtyping between monotypes implies equality of monotypes, or formally speaking, $\tau_1 \leq \tau_2 \rightarrow \tau_1 = \tau_2$. This property is helpful in establishing the completeness of an algorithmic system with respect to its declarative specification. Suppose that there are two instantiation judgments (or unification judgments) $\widehat{\alpha} \leq \tau_1$ and $\widehat{\alpha} \leq \tau_2$ produced by the algorithm, and the declarative specification could guess a τ that satisfies $\tau \leq \tau_1$ and $\tau \leq \tau_2$. This means that $\tau_1 = \tau_2$ and the order of unification makes no difference. Note also that the fact that $\bot$ and $\top$ are not monotypes in $F_{\leq}^{e}$ is also crucial for this property to hold. However, this property requires a delicate definition of monotype after bounded quantification is introduced, and we will revisit this in Section 2.3.

2.2 A Tour of Local Type Inference

Local type inference is the most widely used technique for languages with bounded quantification. To better compare with our work, we will examine a series of examples in Scala 2, which provides a state-of-the-art implementation of (colored) local type inference [Plociniczak 2016]. Note that the implementation in Scala 2 contains some improvements compared to the algorithms formalized by calculi with local type inference [Odersky et al. 2001; Pierce and Turner 2000], enabling more programs to type-check. We will point out such cases explicitly in the following discussion. Despite such improvements, well-known limitations of local type inference can still be identified in Scala 2[1], including *hard-to-synthesize arguments*, *no best argument*, and *no support for interdependent bounds*.

For readers unfamiliar with the Scala syntax, we explain a few important pieces of syntax first. In Scala types, => denotes the function type constructor. For instance, `Int => Int` denotes a function type whose argument and output are both integers. Variables in square brackets ([A]) are universal variables, possibly with an extra bound ([A <: Int]) for bounded quantification. At the term level, => is part of the syntax of lambda abstractions. For instance, `x => x + 1` is the syntax for the lambda abstraction that increments its argument by one.

Hard-to-synthesize arguments. Local type inference employs two techniques to synthesize types: local type argument synthesis and bidirectional type checking. But circumstances exist when none of them can be applied, e.g., when a higher-order function is applied to an anonymous function.

```
def map[A, B](f: A => B, xs: List[A]): List[B] = ...
def mapPlus1: List[Int] = map(x => 1 + x, List(1, 2, 3))  // fails to type check!
```

To workaround this issue, the programmer can choose to provide type annotations to the function argument (as in `mapPlus2`) or provide all the instantiations directly (as in `mapPlus3`).

```
def mapPlus2: List[Int] = map((x: Int) => 1 + x, List(1, 2, 3))
def mapPlus3: List[Int] = map[Int, Int](x => 1 + x, List(1, 2, 3))
```

Another more language-dependent solution is to swap the order of two arguments in the definition of `map` as shown below. Because the type-inference algorithm in Scala proceeds left-to-right, the type information in the first argument helps infer the second one.

[1]We do not use Scala 3 because its approach to type inference [Martres 2023] seems to be a hybrid combination of local type inference and some other techniques, which have not been formaly studied. Thus Scala 2 type inference remains more faithful to the original work [Odersky et al. 2001; Pierce and Turner 2000].

```
def map2[A, B](xs: List[A], f: A => B): List[B] = ...
def map2Plus: List[Int] = map2(List(1, 2, 3), x => 1 + x)
```

No support for interdependent bounds. The only formalized variant of local type inference supporting bounded quantification that we are aware of is by Pierce and Turner [2000]. However, their work *forbids* the inference of type arguments with interdependent bounds because they could not find a complete algorithm that deals with such bounds. Scala 2 does provide some basic support for interdependent bounds instead of forbidding them, but type checking still fails frequently. This reveals the difficulty of achieving a complete algorithm that can always succeed in such cases. A simple example of failing to find instantiations of interdependent bounds in Scala is:

```
def idFun[A, B <: A => A](x: B): A => A = x
def idInt1: Int => Int = x => x
def idInt2 = idFun(idInt1)  // fails to type check!
```

In the above example, `A` and `B` are instantiated to $\bot$ (i.e. `Nothing` in Scala) and Int $\rightarrow$ Int. The instantiation does not conform to the bounded quantification $B \leq A \rightarrow A$ since Int $\rightarrow$ Int $\leq \bot \rightarrow \bot$ is not true. This incorrect instantiation found by Scala causes it to reject the program.

No best argument. When the type variable to instantiate appears invariantly in the output type, but the constraints are not enough to decide a unique instantiation, local type inference fails to provide any instantiations. Scala still infers a type in such cases, but the type can be meaningless. For example, the type of `id` is inferred as $\bot \rightarrow \bot$ in Scala. Thus, `id` cannot be applied further.

```
def snd[A]: Int => A => A = x => y => y
def id = snd(1)
```

Preference for uncurried functions. Local type inference prefers a fully uncurried style in polymorphic applications to obtain more constraints of type variables to obtain a precise instantiation. Sometimes, type checking will fail after making the function curried.

```
def fst1[A, B](x: A, y: B): A = x
def idInt2: Int => Int = fst1(x => x, 4)
def fst2[A, B]: A => B => A = x => y => x
def idInt3: Int => Int = fst2(x => x)(4)  // fails to type check!
```

Inference of $\top$ *and* $\bot$. Local type inference can output $\top$ and $\bot$ directly as a result of constraint solving, which can often hide possible type errors. Consider:

```
def fst3[A](x: A, y: A): A = x
def val = fst3(1, true)
```

The polymorphic function `fst3` has type $\forall a.\ a \rightarrow a \rightarrow a$ and a is instantiated to $\top$ (i.e. `Any` in Scala) in `fst3(1, true)` by Scala. It enables applying the function to two values with different types, which usually contradicts the intention of specifying two arguments with the same type A. Inferring $\bot$ may allow some programs with other possible type errors to type check. For example:

```
def fPlus[A](f: A => Int, g: A => Int): A => Int = x => f(x) + g(x)
def useless = fPlus((x: Int) => x + 1, (y: Boolean) => 5)
```

The inferred type of `useless` is $\bot \rightarrow$ `Int`[2]. Although the type of `fPlus` expects two functions of type $A \rightarrow$ Int, the application of `fPlus` in `useless` provides two functions with different domain types. In practice, this situation could correspond to a programmer unintentionally providing two library functions with different types to the function (perhaps because he did not recall the correct

[2]To be precise, here Scala 2 infers `Int & Boolean` $\rightarrow$ `Int`, since it supports intersection types. However, this intersection type is morally equivalent to $\bot$ in Scala: we cannot build a value of that type. In a system without intersection types, we would expect that $\bot \rightarrow$ `Int` is inferred.

types). Note that the instantiation of A is valid here, but it is unlikely to correspond to what the programmer would expect.

We should remark that, especially in the presence of downcasts, there are situations where inference of $\top$ and $\bot$ can be useful. So, there is also an argument for supporting the inference of such types. For instance, programmers may wish to build a generic container and performing downcast when extracting its elements. In the example below, `ls` is inferred to type `List[Any]` in Scala without any annotations.

```
val ls = List(1, 2, "3")
```

Disabling the implicit inference of $\top$ would require more boilerplate in such cases, as we would need to provide type annotations or explicit type instantiations. However, for languages without downcasts, inference of $\top$ and $\bot$ would have much more limited benefits, while hiding many possible type errors that could be caught by other type inference disciplines (such as Hindley-Milner). Thus, for languages without downcasts, forbidding inference of $\top$ and $\bot$ seems quite reasonable. Moreover, this tension between powerful type inference and early detection of errors has even led language designers to allow programmers to disable the inference of top types. For instance, Scala has a flag `-Xlint:infer-any` to disable the implicit inference of $\top$.

Higher-rank type inference. The subtyping relation employed in local type inference is basically the same as $F_{\leq}$. However, this is restrictive and prohibits type-checking many programs with higher-rank types. The definition of g is valid in Scala 3 (but not in Scala 2, which does not support higher-rank types), but the application `g(k)` is still rejected by Scala 3 because it would require $\leq\forall$L in the subtyping relation $(\mathtt{Int} \to \mathtt{Int}) \to \mathtt{Int} \leq (\forall(a \leq \mathtt{Int}).\ a \to a) \to \mathtt{Int}$.

```
def k(f: (Int => Int)) = 1
def g(f: ([A <: Int] => A => A) => Int) = 1
def f = g(k)  // fails to type check!
```

2.3 Key Ideas in our Work

Comparing Local Type Inference with $F^b_{\leq}$. Table 1 shows examples in Section 2.2 that behave differently in $F^b_{\leq}$. With global type inference, $F^b_{\leq}$ is capable of dealing with long-distance constraints. Therefore, $F^b_{\leq}$ can type the application examples mentioned in Section 2.2 without annotations, which fail in local type inference. For instance, the `map` example works well in $F^b_{\leq}$. In addition, since $F^b_{\leq}$ does not infer $\top$ and $\bot$, the applications involving `fst3` and `fPlus` fail. Higher-rank polymorphic examples, such as g, are also accepted. In $F^b_{\leq}$ interdependent bounds are not problematic, so the application of `idFun` works and finds the correct instantiation.

A restriction in $F^b_{\leq}$ is that it cannot infer instantiations using type variables with monotype bounds. The following code type checks in Scala, while it is rejected in $F^b_{\leq}$:

```
def idBnd[A <: Int](x: A) = fst1(x, x)
```

In $F^b_{\leq}$, an explicit type application (`fst1 @`$a\ x\ x$) is needed, when the instantiation is a type variable a with the monotype bound like `Int`. Section 5 shows a variant of $F^b_{\leq}$ that can accept this program, and in general can infer more instantiations, but loses completeness. However, note that if the bound of a is `Top` in `fst1` $x\ x$, the implicit instantiation still succeeds because in that case a is considered to be a monotype.

In what follows we discuss the key ideas and design decisions in $F^b_{\leq}$, starting by motivating its definition of monotypes, followed by other key technical ideas.

Non-syntactic monotypes. Previous works [Dunfield and Krishnaswami 2013; Odersky and Läufer 1996; Peyton Jones et al. 2007; Zhao and Oliveira 2022] on predicative HRP determine monotypes

Table 1. Examples used in the comparison of local type inference with $F^b_{\leq}$

$F^b_{\leq}$ **program**	**Accepted or not**
let map: $\forall(a \leq \top).\ \forall(b \leq \top).\ (a \to b) \to [a] \to [b] = \ldots$ **in** map $(\lambda x.\ x+1)\ [1,2,3]$	✓
let idFun: $\forall(a \leq \top).\ \forall(b \leq a \to a).\ b \to a \to a = \Lambda a.\ \Lambda b.\ \lambda x.\ x$, idInt: Int $\to$ Int $= \lambda x.\ x$ **in** idFun idInt	✓
let snd: $\forall(a \leq \top)$. Int $\to (a \to a) \to a \to a = \Lambda a.\ \lambda x.\ \lambda y.\ y$ **in** snd 1	✓
let fst3: $\forall(a \leq \top).\ a \to a \to a = \Lambda a.\ \lambda x.\ \lambda y.\ x$ **in** fst3 1 true	✗
let fPlus: $\forall(a \leq \top).\ (a \to$ Int$) \to (a \to$ Int$) \to$ Int $= \Lambda a.\ \lambda f.\ \lambda g.\ \lambda x.\ f\ x + g\ x$ **in** fPlus $((\lambda x.\ x+1)$: Int $\to$ Int$)$ $((\lambda y.\ 5)$: Bool $\to$ Int$)$	✗
let k: (Int $\to$ Int) $\to$ Int $= \lambda f.\ 1$, g: $((\forall(a \leq$ Int$).\ a \to a) \to$ Int$) \to$ Int $= \lambda f.\ 1$ **in** g k	✓
let fst1: $\forall(a \leq \top).\ \forall(b \leq \top).\ a \to b \to a = \Lambda a.\ \Lambda b.\ \lambda x.\ \lambda y.\ x$ **in** $(\Lambda a.\ \lambda x.$ fst1 $x\ x) : \forall(a \leq$ Int$).\ a \to a$	✗
... **in** $(\Lambda a.\ \lambda x.$ fst1 @$a\ x\ x) : \forall(a \leq$ Int$).\ a \to a$	✓
... **in** $(\Lambda a.\ \lambda x.$ fst1 $x\ x) : \forall(a \leq \top).\ a \to a$	✓

syntactically. In particular, they treat all type variables as monotypes. Simply adopting the same idea in a system extending the subtyping relation in Figure 1 with bounded quantification would break subtyping transitivity. The reader can look at the subtyping relation in Figure 2, which provides such an extension and simply assume, for the sake of our argument, that all type variables are monotypes. A counterexample that illustrates that transitivity does not hold is described next.

Let $A = \forall(a \leq \top).\ a$, $B = b$, $C = \forall(c \leq 1).\ c \to 1$ and $\Psi = \cdot, b \leq \forall(c \leq 1).\ c \to 1$. Both $\Psi \vdash A \leq B$ and $\Psi \vdash B \leq C$ hold as shown in the following derivations. But $\Psi \vdash A \leq C$ is invalid because the two bounds ($\top$ and 1) are not equivalent, which is required by kernel $F_{\leq}$ when comparing bounds.

- $\Psi \vdash A \leq B$: $\dfrac{b \leq \forall(c \leq 1).\ c \to 1 \vdash b \leq \top \qquad b \leq \forall(c \leq 1).\ c \to 1 \vdash b \leq b}{b \leq \forall(c \leq 1).\ c \to 1 \vdash \forall(a \leq \top).\ a \leq b}$ BY $\leq\forall$L
- $\Psi \vdash B \leq C$: $b \leq \forall(c <: 1).\ c \to 1 \vdash b \leq \forall(c <: 1).\ c \to 1$ BY $\leq$VarTrans
- $\Psi \vdash A \nleq C$: $\forall(a \leq \top).\ a \nleq \forall(c <: 1).\ c \to 1$

The reason is the interplay between rule $\leq\forall$L and $\leq$VarTrans: type variables are used as monotypes for instantiation in rule $\leq\forall$L, but they can be transformed into universal types using the $\leq$VarTrans and used in other subtyping judgments. Similar problems happen if a type variable with bound $\bot$ is used as a monotype. To avoid breaking transitivity, we need to regard type variables with such bounds as non-monotypes to forbid instantiating such type variables with rule $\leq\forall$L.

Can we then regard type variables with a monotype or $\top$ bound as monotypes? For transitivity, and other essential properties such as type safety and soundness of the algorithmic system, such a definition of monotypes is fine (as shown in Section 5.1). Sadly, with greedy instantiation, monotype bounds are still problematic for a different reason. Recall the important property to ensure that greedy instantiations always work: $\Psi \vdash \tau_1 \leq \tau_2$ implies $\tau_1 = \tau_2$. This property is broken if we treat type variables with a monotype bound as a monotype. The counterexample is obvious, $\cdot, a \leq 1 \vdash a \leq 1$, but $a \neq 1$. Allowing monotypes to include type variables with bounds other than $\top$ results in an incomplete greedy algorithm. To solve the abovementioned issues, all type variables with bound other than $\top$ are considered non-monotypes, i.e., they cannot be implicitly instantiated with rule $\leq\forall$L. In other words, our definition of monotypes is no longer purely syntactic.

Matching and type-application inference–judgments. An important property that our completeness result depends on is a subsumption theorem. The proof of the subsumption theorem needs us to generalize the lemma to all kinds of judgments first and then prove them simultaneously. With many new cases added due to bounded quantification, the proof becomes complex. We develop two new kinds of judgment, namely matching and type application inference, to disentangle the proof. The matching judgment is inspired by Siek et al. [2015] to replace the application inference used by Zhao and Oliveira [2022] and Dunfield and Krishnaswami [2013]. The type-application inference–judgment unifies all the cases with type application and reduces the number of cases in inference judgments. These two judgments are defined independently from checking and inference judgments, and their metatheory can be established independently. With the help of these two judgments and their independently proved metatheory, our final generalized lemma for subsumption only requires a 2-fold mutual induction with much fewer cases rather than a potential 4-fold mutual induction.

Polytype splitting and decidability. Previous works on type inference for implicit HRP [Dunfield and Krishnaswami 2013; Zhao and Oliveira 2022; Zhao et al. 2019] adopt the following (simplified) rule when comparing an existential (unification) variable with an arrow type in subtyping: split an existential variable into two fresh ones to solve the domain and codomain type separately.

$$\widehat{\alpha} \leq A \to B \quad \text{reduces to} \quad \widehat{\alpha}_1 \to \widehat{\alpha}_2 \leq A \to B \quad \text{when } \widehat{\alpha} \notin FV(A \to B)$$

This rule poses difficulties in proving decidability because it increases the number of existential variables by one without decreasing any other measures instantly. To solve this issue, previous work employs concepts such as *instantiation judgments* $\widehat{\alpha} \leq A$ and $A \leq \widehat{\alpha}$ and prove (1) each reduction procedure of these instantiation judgments is decidable because the type size decreases; (2) after an instantiation judgment is fully reduced, the overall measure will also decrease. This intermediate decision procedure, that each instantiation judgment will reduce the overall measure after an *indefinite* number of steps, creates complexity in the decidability proof.

We observe that it is unnecessary to split the existential variable when $A \to B$ is a monotype. Instead, we can always solve $\widehat{\alpha}$ using the monotype $A \to B$ directly, i.e., substitute every occurrence of $\widehat{\alpha}$ with $A \to B$, and split only when $A \to B$ is a polytype. This modification, named *polytype splitting*, allows us to develop a measure $|\cdot|_{\to}$ to track the maximum possible number of splits of a type statically. With the help of this new measure, the entire measure of our algorithm always decreases after a *constant* number of reduction steps, so the proof of decidability is greatly simplified.

Another challenge is to combine the measure used to prove the decidability of kernel $F_{\leq}$ with that of worklist algorithms. The *weight* measure for $F_{\leq}$ [Pierce 2002], is not compatible with the existential variables in worklists: the weight could increase arbitrarily after solving an existential with a monotype. To address this challenge, we develop a lexicographic group of four measures on the worklist. The first three in the group are (almost) unaffected by existential-variable solving, and the last one is a simplified version of the *weight*.

Worklist substitution. A complication that arises in HRP type-inference algorithms is that existential (unification) variables sometimes need to have their scope widened. However, changing the scope of an existential variable may require changing the scope of other existential variables recursively. To help with this process and simplify the type-inference algorithm, we employ a revised substitution procedure to deal with the variable insertion, removal, and worklist reordering after solving an existential variable with another monotype. The rules and notations for the type-inference algorithm can be presented more concisely and clearly since the scope management for existential variables is encapsulated within the worklist substitution. The lemmas related to these cases in the soundness, completeness, and decidability proof can also be stated in a unified manner to provide more intuition on the properties of our system.

3 DECLARATIVE SYSTEM

This section introduces a declarative type system for $F^b_\leq$, which serves as a specification for the algorithmic version that will be presented in Section 4. The type system extends Zhao and Oliveira [2022]'s $F^e_\leq$ type system by adding *bounded quantification* [Cardelli and Wegner 1985]. Several important properties of the subtyping and typing relation are stated and discussed, especially the transitivity of subtyping and the subsumption of typing, which are significantly complicated by the addition of bounded quantification.

3.1 Syntax, Monotypes, and Well-Formedness

Type variables	a, b		
Types	A, B, C	::=	$1 \mid a \mid \forall(a \leq B).\, A \mid A \to B \mid \top \mid \bot$
Expressions	e, t	::=	$x \mid () \mid \lambda x.\, e \mid e_1\ e_2 \mid (e : A) \mid e\ @A \mid \Lambda(a \leq B).\, e : A$
Typing contexts	Δ	::=	$\cdot \mid \Delta, x : A \mid \Delta, a \leq A$
Subtyping contexts	Ψ	::=	$\Delta \mid \Psi, a \lesssim A$

The syntax of $F^b_\leq$ is shown above. Compared to $F^e_\leq$, universal types $\forall(a \leq B).\, A$ and type abstractions $\Lambda(a \leq B).\, e : A$ now incorporate a bound B to support bounded quantification. Type application ($e\ @A$) is inherited from $F^e_\leq$. The remaining expression and type syntax is standard.

Differently from $F^e_\leq$, there are two kinds of contexts in $F^b_\leq$: typing context Δ and subtyping context Ψ. This change is to enforce some expected invariants. Firstly, subtype variables should not occur in typing and typing contexts. Secondly, only subtype variables will be added to the context in subtyping. Thus, a subtyping context should be a typing context prefix followed by a collection of subtype variables. Consequently, the bound of a type variable cannot depend on a subtype variable. Despite having no major consequences for the formalization in terms of the properties of the system, none of these invariants are enforced in $F^e_\leq$. In contrast, they are enforced in $F^b_\leq$. With this new design, $F^b_\leq$ distinguishes type variables from subtype variables by the entries in the contexts ($a \leq A$ or $a \lesssim A$) rather than by a syntactic distinction of a and $\tilde{a}$ as in $F^e_\leq$. The entries also keep track of the bound for (sub)type variables. For convenience, we sometimes use the notation $a \leqq A$ to represent both $a \leq A$ and $a \lesssim A$ entries in subtyping contexts. The key syntactic differences to $F^e_\leq$ are marked in gray.

Monotypes. The definition of monotypes plays a central role in $F^b_\leq$, and it requires significant changes in comparison to previous calculi with predicative implicit instantiation [Dunfield and Krishnaswami 2013, 2019; Odersky and Läufer 1996; Peyton Jones et al. 2007]. Monotypes are no longer defined syntactically due to the presence of bounds. Instead, we need a dedicated judgment to decide whether a type variable is a monotype or not: only unbounded type variables (the upper bound is $\top$) are allowed. The new monotype relation needs to take the context Ψ as input to perform the bound lookup. The remaining cases of the monotype definition are standard, as shown in the top part of Figure 2. For brevity, we will use conventional symbols τ, σ to represent monotypes in $\Psi \vdash^m \tau$ and sometimes omit Ψ when it is clear what the context is.

Well-formedness. Well-formedness (listed in the extended version of the paper) ensures the well-scopedness of binders. After adding bounded quantification, we need to check the well-formedness of the bounds in certain rules. We also inherit the free-variable checks in Zhao and Oliveira [2022] to ensure that the polymorphic type $\forall(a \leq B).\, A$ is indeed polymorphic. It is worth noting that if the type variable appears in the bound of a nested universal type, it also passes the free-variable check, i.e., the type $\forall(a \leq \top).\, \forall(b \leq a).\, b \to b$ is regarded as a well-formed type in $F^b_\leq$.

$\boxed{\Psi \vdash^m A}$ A is a monotype

$$\frac{}{\Psi \vdash^m 1}\ \texttt{MUnit} \qquad \frac{\Psi \vdash^m A \quad \Psi \vdash^m B}{\Psi \vdash^m A \to B}\ \texttt{M}{\to} \qquad \frac{a \le \top \in \Psi}{\Psi \vdash^m a}\ \texttt{MTVar}$$

$\boxed{\Psi \vdash A \le B}$ A is a subtype of B

$$\frac{}{\Psi \vdash 1 \le 1}\ \le\texttt{Unit} \qquad \frac{}{\Psi \vdash A \le \top}\ \le\top \qquad \frac{}{\Psi \vdash \bot \le A}\ \le\bot \qquad \frac{a \leqq B \in \Psi}{\Psi \vdash a \le a}\ \le\texttt{Var}$$

$$\frac{a \leqq B \in \Psi \quad \Psi \vdash B \le A}{\Psi \vdash a \le A}\ \le\texttt{VarTrans} \qquad \frac{\Psi \vdash B_1 \le A_1 \quad \Psi \vdash A_2 \le B_2}{\Psi \vdash A_1 \to A_2 \le B_1 \to B_2}\ \le\to$$

$$\frac{\Psi \vdash \tau \quad \Psi \vdash^m \tau \le B \quad \Psi \vdash [\tau/a]A \le C \quad C \text{ is not a } \forall \text{ type}}{\Psi \vdash \forall(a \le B).\, A \le C}\ \le\forall\texttt{L}$$

$$\frac{\Psi \vdash B_1 \le B_2 \quad \Psi \vdash B_2 \le B_1 \quad \Psi, a \lesssim B_2 \vdash A_1 \le A_2}{\Psi \vdash \forall(a \le B_1).\, A_1 \le \forall(a \le B_2).\, A_2}\ \le\forall$$

Fig. 2. Declarative Monotype and Subtyping

3.2 Subtyping

Figure 2 shows the rules of subtyping relation. Most rules are inherited from $F^e_{\le}$. Rules $\le\top$, $\le\bot$ and $\le\to$ are standard subtyping rules for calculi with $\top$, $\bot$ and function types. Rule $\le$Var is the reflexivity rule for (sub)type variables. The remaining rules are key to supporting bounded quantification. Rule $\le\forall$L states that a universal type is a subtype of another (non-universal) type C as long as we can find a monotype τ that satisfies the bound B, such that the instantiated body is the subtype of C. This rule differs from previous rules in the literature [Dunfield and Krishnaswami 2013; Zhao and Oliveira 2022] in that the monotype is not only required to be well-formed but also a subtype of the bound B. By checking that $\Psi \vdash \tau \le B$, we prevent incorrect instantiations that do not satisfy the bounds. Rule $\le\forall$ states that two universal types are subtypes if their bodies are subtypes and their bounds are mutual subtypes of each other. This rule is adopted from the *equivalent type* variant of kernel $F_{\le}$ [Pierce 2002; Zhou et al. 2023] to support more flexible bound comparison. Since we include bottom types, the subtyping relation is not antisymmetric like the original $F_{\le}$, e.g., a type variable with a bottom bound and the bottom type are subtypes of each other. Thus, the rule used here is more adequate. The extra condition "C is not a $\forall$ type" in rule $\le\forall$L ensures that rule $\le\forall$ always takes priority when both sides are universal types. Finally, rule $\le$VarTrans allows comparing a (sub)type variable with a type A as long as its bound B can be compared with this type. It is a standard rule in a type system with bounded quantification. The reflexivity and transitivity property hold for this subtyping relation:

Theorem 3.1 (Subtyping Reflexivity). *Given* $\Psi \vdash A$, $\Psi \vdash A \le A$.

Theorem 3.2 (Subtyping Transitivity). *Given* $\Psi \vdash A$, $\Psi \vdash B$, *and* $\Psi \vdash C$, *if* $\Psi \vdash A \le B$, *and* $\Psi \vdash B \le C$ *then* $\Psi \vdash A \le C$.

3.3 Typing

The top part of Figure 3 shows the declarative type system with the checking and inference judgments. Most rules are standard in existing bidirectional type systems with higher-rank predicative type inference [Dunfield and Krishnaswami 2013]. However, there are a few differences.

Firstly, we do not include a rule that checks against a universal type:

$$\frac{\Delta, a \vdash e \Leftarrow A}{\Delta \vdash e \Leftarrow \forall a.\, A}$$

Here, we omit bounded quantification for a clearer comparison with previous work. Such a rule is common in previous work [Dunfield and Krishnaswami 2013; Zhao and Oliveira 2022]. It allows implicit definitions of polymorphic functions, where type variables do not need to be explicitly bound at the term level. We drop this checking rule because it seems more appropriate for type systems where the order of bound universal variables is irrelevant, which is not the case in $F^b_\leq$. Consider the following checking judgment that holds with the above rule:

$$\cdot \vdash \Lambda a.(\Lambda b.(\lambda x.\, \lambda y.\, .x) : \forall b.a \to b \to a) : \forall a.\forall b.a \to b \to a \Leftarrow \forall b.\forall a.a \to b \to a$$

However, $\forall b.\forall a.a \to b \to a$ is not a subtype of $\forall a.\forall b.a \to b \to a$ according to the subtyping relation since the order of the binders matters in $F^b_\leq$. In $F^b_\leq$, explicit type abstractions $(\Lambda(a \leq B).\, e : A)$ are the only way to define polymorphic functions, aligning better with calculi with local type inference and with languages like Scala or Java. The consequence of removing this rule is that lambda expressions annotated with a forall type are no longer accepted, e.g., $\lambda x.\, x : \forall a.\, a \to a$. Programmers must explicitly convert such expressions to Λ expressions. Since explicit type abstraction is more powerful than the implicit one by allowing *scoped type variables* [Peyton Jones and Shields 2004; Zhao et al. 2019], no expressivity is lost.

Secondly, rule $\Leftarrow\!\to\!\top$ is added to check a lambda expression against $\top$. In $F^e_\leq$, there is a rule that allows *any* well-formed expression to check against the top type. So, some peculiar expressions, which seem nonsensical, e.g. $()\;() : \top$, can also be type checked. The $F^e_\leq$ rule is type safe, since no information can be extracted from an expression annotated with $\top$. However, we believe it is more intuitive to completely reject any nonsensical expressions in $F^b_\leq$. Therefore, in rule $\Leftarrow\!\to\!\top$, we use the greatest function type in the subtyping lattice ($\bot \to \top$) to type-check the lambda expression. This allows us to prevent nonsensical lambda expressions from being accepted when checked against $\top$. Checking the remaining expressions against $\top$, including $()$, x, $e : A$, $\Lambda(a \leq B).\, e : A$, and $e_1\; e_2$ is now handled by the subsumption rule.

Finally, there are two new judgments. The inference of application depends on the matching judgment $\Delta \vdash A \triangleright B \to C$. Note that this judgment differs from the judgment used in $F^e_\leq$ or Dunfield and Krishnaswami [2013]. In addition, the inference of type applications is unified into a single rule $\Rightarrow$TApp and depends on the new type-application inference–judgment $\Delta \vdash A \circ B \Rrightarrow C$. Both of these judgments are discussed next.

Matching. The matching judgment $\Delta \vdash A \triangleright B \to C$ in Figure 3 states that a type A can be regarded as an arrow type $B \to C$, and thus a term of type A can be applied to another term of type B and have the result type be C. It was first proposed by Siek et al. [2015] and extended to polymorphic types [Xie et al. 2019] by guessing instantiations until reaching an arrow type. The rules $\triangleright\forall$ and $\triangleright\!\to$ are similar to those in Xie et al. [2019]. In rule $\triangleright\forall$, a monotype τ that satisfies the bound is guessed, and the judgment is applied to the instantiated universal type recursively.

Rule $\triangleright\bot$ and $\triangleright$Var are newly added to support $\bot$ and bounded quantification. Since a term of type $\bot$ can be applied to any well-typed term, and the result type is still $\bot$, $\bot$ should match $\top \to \bot$, the least function type in the subtyping lattice, as stated in $\triangleright\bot$. With bounded quantification, it is also possible for type variables to match an arrow type. This case is dealt with by rule $\triangleright$Var, which states that as long as the bound of a type variable can match an arrow type, the type variable itself can match this arrow type.

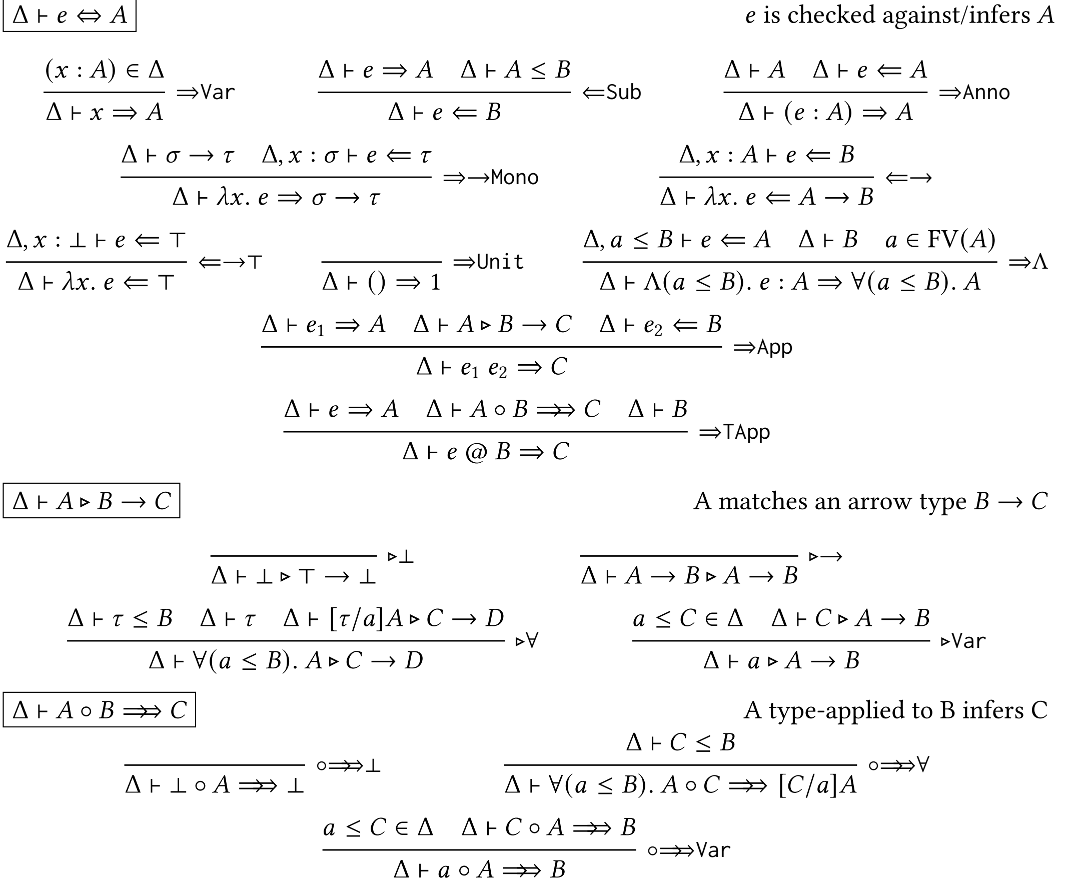

Fig. 3. Matching and Type-Application Inference.

The key reason to use matching instead of the application-inference judgment, $\Delta \vdash A \bullet e \Rrightarrow C$ due to Dunfield and Krishnaswami [2013], is that the matching judgment is a pure type-level relation and, thus, independent of the checking and inference rules. On the contrary, the application inference judgment, which is also used in $F^{e}_{\leq}$, is mutually defined together with the checking and inference judgments. Unfortunately, in a system with bounded quantification, this becomes problematic as properties such as subsumption become much more entangled with other properties, making proofs difficult. By decoupling matching from inference and checking, we are able to prove many properties for matching independently, greatly simplifying our proofs.

Type application inference. The type-application inference–judgment $\Delta \vdash A \circ B \Rrightarrow C$ at the bottom of Figure 6 states that a type A can be regarded as a universal or $\bot$ type so that it can be type-applied to type B, and the result type is C. This judgment generalizes the two rules $\bot \mathbin{@} A \Rightarrow \bot$ and $\forall a.A \mathbin{@} B \Rightarrow [B/a]A$ used by Zhao and Oliveira [2022]. It can be easily extended to support new cases, i.e. rule $\circ\Rrightarrow$Var, that is added due to bounded quantification. This rule allows type-level variables to be type-applied when their bounds can be type-applied. Type-application inference is also a pure type-level relation independent of checking and inference rules.

Checking subsumption. An important property that holds for the typing relation is:

Theorem 3.3 (Checking Subsumption). *Given* $\Delta \vdash B$, *if* $\Delta \vdash e \Leftarrow A$ *and* $\Delta \vdash A \leq B$, *then* $\Delta \vdash e \Leftarrow B$.

3.4 Metatheory

In this section, we discuss the most interesting aspects of the metatheory of the declarative type system. In particular, we discuss the challenges that emerge after the introduction of bounded quantification.

Subtyping Transitivity. The proof of subtyping transitivity proceeds by induction on the first subtyping relation. The tricky case is the $\leq\forall$ rule, i.e. $\Psi \vdash \forall(a \leq D_1).\ A \leq \forall(a \leq D_2).\ B \leq C$. When C is also a universal type $\forall(a \leq D_3).\ C'$, a *subtyping narrowing* lemma is required to replace the bound D_2 with the equivalent type D_3 in the $\leq\forall$ derivation $\Psi, a \lesssim D_2 \vdash A \leq B$. This is similar to the transitivity proof in the algorithmic version of *full* $F_{<:}$ without a built-in transitivity rule [Pierce 2002]. When C is not a universal type, with the rule $\leq\forall$L, there exists τ s.t. $\Psi \vdash \tau \leq D_3$ and $\Psi \vdash [\tau/a]B \leq C$, and thus we need a *subtyping stability* lemma to substitute a with τ in the above $\leq\forall$ derivation. Both the narrowing and stability lemmas are mutually dependent on the transitivity lemma and must be proven simultaneously. In the proofs of both lemmas, the transitivity has to be applied when A is a subtype variable due to the rule $\leq$`VarTrans`.

Lemma 3.4 (Subtyping Narrowing and Stability). *Given all types and contexts well-formed,*

(1) If $\Psi_1, a \lesssim D, \Psi_2 \vdash A \leq B$ *and* $\Psi_1 \vdash C \leq D$, *then* $\Psi_1, a \lesssim C, \Psi_2 \vdash A \leq B$.
(2) If $\Psi_1, a \lesssim D, \Psi_2 \vdash A \leq B$ *and* $\Psi_1 \vdash C \leq D$, *then* $\Psi_1, [C/a]\Psi_2 \vdash [C/a]A \leq [C/a]B$.

Subsumption. To prove the subsumption lemma, we first prove the subsumption lemmas for matching and the type-application inference–judgment independently.

Lemma 3.5 (Matching and Type-Application Inference–Subsumption). *Given all types and contexts well-formed,*

(1) If $\Delta \vdash A \triangleright B \rightarrow C$ *and* $\Delta \vdash A' \leq A$, *then* $\exists B', C'$ *s.t.* $\Delta \vdash A' \triangleright B' \rightarrow C'$ *and* $\Delta \vdash B' \rightarrow C' \leq B \rightarrow C$.
(2) If $\Delta \vdash A \circ B \Rrightarrow C$ *and* $\Delta \vdash A' \leq A$, *then* $\exists C'$ *s.t.* $\Delta \vdash A' \circ B \Rrightarrow C'$ *and* $\Delta \vdash C' \leq C$.

With the above two lemmas proved, we then adopt the proof technique used by Zhao et al. to prove the subsumption lemma, which generalizes the subsumption theorem by (1) extending the property to both checking and inference judgments; (2) introducing a sub-context relation $\Delta' <: \Delta$. The sub-context is defined on typing context only and requires the bound of corresponding type variables in Δ and Δ' to be the *same*.

$\boxed{\Delta' <: \Delta}$ Sub-context

$$\frac{}{\cdot <: \cdot} \qquad \frac{\Delta' <: \Delta}{\Delta', a \leq A <: \Delta, a \leq A} \qquad \frac{\Delta' <: \Delta \quad \Delta \vdash A \leq B}{\Delta', x : A <: \Delta, x : B}$$

The most general form of the subsumption lemma is:

Lemma 3.6 (Checking and Inference Subsumption). *Given* $\Delta' <: \Delta$

(1) if $\Delta \vdash e \Leftarrow A$, $\Delta \vdash A'$, *and* $\Delta \vdash A \leq A'$, *then* $\Delta' \vdash e \Leftarrow A'$;
(2) if $\Delta \vdash e \Rightarrow A$, *then* $\exists A'$ *s.t.* $\Delta \vdash A' \leq A$ *and* $\Delta' \vdash e \Rightarrow A'$.

It is not hard to observe that if $\Delta' <: \Delta$, any matching and type application judgment that holds in Δ also holds in Δ' and vice versa since the changes in the bound of variables x do not affect these two pure type-level judgments. Thus, the subsumption proved for matching and type-application inference can be used in the proof this generalized lemma with the premise $\Delta' <: \Delta$.

The form of our subsumption lemmas is simpler than that used by Dunfield and Krishnaswami [2013]; Zhao and Oliveira [2022]. The primary source of simplification comes from the use of the matching judgment, inspired by the work of [Xie et al. 2019]. The metatheory of the matching judgment can be developed independently of checking and inference. We initially tried to use the same approach as Zhao and Oliveira, but the proof became too intricate due to several mutual dependencies between various lemmas.

Type Safety and Completeness to kernel $F_{\leq}$. The type safety of our type system is derived via an elaboration to System F by taking a coercive interpretation [Breazu-Tannen et al. 1991]: a proof of subtyping relation $p : A \leq B$ is elaborated into a term $e : |A| \to |B|$ ($|\cdot|$ indicates type translation). The translation of types and contexts to mitigate the syntax gap is adopted from Curien and Ghelli [1992]. Specifically, $\forall(a \leq B).\, A$ is translated to $\forall a.\, (a \to |B|) \to |A|$, and $a \leq B$ in Ψ is translated to two entries a and $c : |a| \to |B|$ in $|\Psi|$. We also prove that all kernel $F_{\leq}$ programs can type check in our system after adjusting the annotations.

The extended version of the paper contains the detailed definition of elaboration, translation, and annotation decoration, and discussion.

Theorem 3.7 (Type Soundness by Elaboration to System F). *If $\Delta \vdash e \Leftrightarrow A \hookrightarrow e'$ then $|\Delta| \vdash_F e' : |A|$*

Theorem 3.8 (Type completeness w.r.t. kernel $F_{\leq}$). *If $\Delta \vdash_{F_{\leq}} e : A$ then $\exists e'$, s.t. $\lfloor e \rfloor = \lfloor e' \rfloor$, i.e., e and e' only differ on annotations, and $\Delta \vdash e' \Rightarrow A$.*

4 ALGORITHMIC SYSTEM

This section introduces an algorithmic system that implements the specification of $F^b_{\leq}$ presented in Section 3 using the worklist approach [Zhao et al. 2019]. This algorithmic system is proven to be sound and complete with respect to the declarative system and is decidable.

4.1 Syntax, Well-Formedness, and Notation

Type variables	a, b	Existential variables	$\widehat{\alpha}, \widehat{\beta}$
Algorithmic types	A, B, C	$::=$	$1 \mid a \mid \forall(a \leq B).\, A \mid A \to B \mid \top \mid \bot \mid \widehat{\alpha}$
Judgment	ω	$::=$	$A \leq B \mid e \Leftarrow A \mid e \Rightarrow_a \omega \mid A \circ B \Rrightarrow_a \omega \mid A \triangleright_{a,b} \omega \mid A \to B \bullet e \Rrightarrow_a \omega$
Algorithmic worklist	Γ	$::=$	$\cdot \mid \Gamma, a \leqq A \mid \Gamma, \widehat{\alpha} \mid \Gamma, x : A \mid \Gamma \Vdash \omega$

Compared with the declarative syntax, the syntax of algorithmic expressions remains unchanged. The syntax of algorithmic types is extended with a new sort of variable: existential variables ($\widehat{\alpha}, \widehat{\beta}$). Existential variables are used as placeholders for monotype guesses in several rules and will be unified with another monotype during the reduction of the worklist. Existential variables are also regarded as monotypes. The (sub)type variables in the worklist are modified to incorporate a bound, similar to the modifications made to the declarative context. The syntactic differences to $F^e_{\leq}$ are marked with gray shades. A worklist Γ is an *ordered* collection of both (type) variable declarations (with bindings) and judgments, so it combines traditional contexts and judgment into a single sort. The ordered nature helps with precise scope management, which simplifies mechanical formalization. Unlike the declarative system, there is only one kind of worklist that deals with both subtyping and typing judgments.

Judgments. There are six kinds of judgments in the algorithmic system. Four of them are inherited from $F^e_{\leq}$: subtyping ($A \leq B$), checking ($e \Leftarrow A$), inference ($e \Rightarrow_a \omega$) and type-application inference ($A \circ B \Rrightarrow_a \omega$). The matching judgment ($A \triangleright_{a,b} \omega$) and its accompanying application inference

$\boxed{\{\tau/\widehat{\alpha}\}\Gamma}$ Substitute $\widehat{\alpha}$ by τ in Γ

$$
\begin{array}{lll}
\{\tau/\widehat{\alpha}\}(\Gamma, \widehat{\alpha}) & =_1 \Gamma & \\
\{\tau/\widehat{\alpha}\}(\Gamma, \widehat{\beta}) & =_2 \{\tau/\widehat{\alpha}\}\Gamma, \widehat{\beta} & \text{when } \widehat{\beta} \notin FV(\tau) \\
\{\tau/\widehat{\alpha}\}(\Gamma_1, \widehat{\alpha}, \Gamma_2, \widehat{\beta}) & =_3 \{\tau/\widehat{\alpha}\}(\Gamma_1, \widehat{\beta}, \widehat{\alpha}, \Gamma_2) & \text{when } \widehat{\beta} \in FV(\tau) \\
\{\tau/\widehat{\alpha}\}(\Gamma, b \leqq A) & =_4 \{\tau/\widehat{\alpha}\}\Gamma, b \leqq [\tau/\widehat{\alpha}]A & \text{when } b \notin FV(\tau) \\
\{\tau/\widehat{\alpha}\}(\Gamma, x : A) & =_5 \{\tau/\widehat{\alpha}\}\Gamma, x : [\tau/\widehat{\alpha}]A & \\
\{\tau/\widehat{\alpha}\}(\Gamma \Vdash \omega) & =_6 \{\tau/\widehat{\alpha}\}\Gamma \Vdash [\tau/\widehat{\alpha}]\omega &
\end{array}
$$

Fig. 4. Worklist Substitution

judgment ($A \to B \bullet e \Rrightarrow_a \omega$) are new. The syntax of subtyping and checking judgments is simple since they either succeed or fail. The syntax of inference, type-application inference, matching, and application-inference judgments is represented using a continuation-passing style [Zhao et al. 2019] (i.e., with a subsequent judgment ω and a subscript type variable a). These judgments return a type that will be used in subsequent judgments ω, but the type may still be unknown when ω is created, so the type variable a serves as a placeholder in the subsequent judgment and will be substituted to a concrete type once it is known.

Miscellaneous rules and notations.

- There are several well-formedness judgments. Firstly, $\vdash \Gamma$ checks that worklist Γ is well-formed. Secondly, $\Gamma \vdash A$ checks that the type A is well-formed in Γ. Finally, $\Gamma \vdash \omega$ checks that judgment ω is well-formed in Γ. The well-formedness definitions are mostly standard, but they also enforce the distinctions between subtyping and typing contexts discussed in Section 3. The detailed rules can be found in the extended version of the paper.
- Worklist reduction $\Gamma \longrightarrow \Gamma'$: pops the last entry in the worklist Γ, processes according to the rules, and possibly pushes multiple simplified judgments back to get a new worklist Γ'. $\Gamma \longrightarrow^* \Gamma'$ denotes multiple reduction steps, and a worklist Γ is accepted by the algorithm iff the worklist processes all the work until nothing is left ($\Gamma \longrightarrow^* \cdot$).
- Worklist substitution $\{\tau/\widehat{\alpha}\}\Gamma$, replaces all the references to the existential variable $\widehat{\alpha}$ in the worklist Γ with a monotype τ and removes the existential variable $\widehat{\alpha}$. The existential variables in τ need to be moved before the original position of $\widehat{\alpha}$ to maintain the well-formedness of the worklist. Figure 4 shows the algorithmic procedure. It traverses the worklist from head to tail until the declaration of $\widehat{\alpha}$ is reached. Rule 1 is the base case where the declaration of $\widehat{\alpha}$ is reached, and $\widehat{\alpha}$ is removed to finish the worklist substitution. Rules 2 and 3 deal with reaching existential variables different from $\widehat{\alpha}$. If $\widehat{\beta}$ does not appear in τ, we continue the substitution on the remaining worklist. If $\widehat{\beta}$ appears in τ, we need to move $\widehat{\beta}$ to the front of $\widehat{\alpha}$ to keep the well-formedness of τ in the remaining substitution. Rule 4 deals with reaching (sub)type variables. Unlike existential variables, (sub)type variables cannot be moved [Dunfield and Krishnaswami 2013]. So, we need to check that the current (sub)type variable does not appear in τ. Rules 5 and 6 deal with variables and judgments by replacing every reference to $\widehat{\alpha}$ with τ and continuing on the remaining worklist.

4.2 Algorithmic Rules

All the reduction rules are defined in a single relation but for clarity of presentation, we separate them into three parts: garbage collection, subtyping, and typing. Compared with the declarative rules, there are two general differences in the algorithmic reduction rules:

- All the well-formedness checks of types in the declarative rules are dropped since we instead require the input worklist to be well-formed, which guarantees that the algorithm only deals with well-formed expressions and type annotations. Note that every reduction rule preserves the well-formedness of the worklist.
- In the declarative system, if a declarative rule has multiple judgments as premises, they are checked separately. These judgments are pushed into a single worklist and reduced jointly in the algorithmic system. The order of pushing these judgments is deterministic for each rule, though the order may sometimes make no difference.

Garbage Collection (Rules 1-3). The scoping mechanism of the ordered worklist ensures that if the worklist is well-formed, these variables can never be referred to by any entries that appear before them. Thus, it is safe to remove them if they are the last entry in the worklist.

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

The garbage collection of (type) variables is intuitive. The garbage collection of the existential variable $\widehat{\alpha}$ means that this existential variable is under-constrained and can be solved to any monotype (and trivially, to 1).

Subtyping (Rules 4-16, Figure 5). These 13 rules can be classified into two categories, where the first contains rules 4-12, and the second contains rules 13-16.

Rules in the first category are similar to their declarative counterparts. Rule 6 is added since we have a new sort of variables in the algorithmic system. The side conditions $B \neq \top$, $B \neq a$ and $A \neq \top$ in rule 7 prevent the overlapping with rules 5, 8 and 14, respectively. The most significant changes are in rule 11, where an existential variable $\widehat{\alpha}$ is introduced instead of guessing the monotype τ instantiation in its declarative counterpart rule $\leq\forall$L. The subtyping judgment $\Psi \vdash \tau \leq B$, which tests if the monotype satisfies the bounds, is also transformed accordingly to $\widehat{\alpha} \leq B$, then added to the worklist. In rules 10, 11, and 12, multiple new entries are pushed back to the worklist. The side-condition $C \neq \top$ in rule 11 prevents the overlapping with the rule 8. Though in rule 12, B_1 and B_2 are pushed back after $b \lesssim B_1$, the solving scope of the existential variables in them does not change because subtype variables are not monotypes, and cannot be used in implicit instantiation. Another subtle difference is that the idea of "equivalent type" is enriched in rule 12 compared to rule $\leq\forall$ in that two types that contain existential variables can also be equivalent but not syntactically equal, e.g., $\widehat{\alpha}_1 \to 1$ and $(1 \to 1) \to \widehat{\alpha}_2$ with $\widehat{\alpha}_1$ and $\widehat{\alpha}_2$ finally solved to $1 \to 1$ and 1.

The rules in the second category solve the existential variables. This set of rules is quite different from those used in previous work [Zhao and Oliveira 2022; Zhao et al. 2019] where: (1) they only solve existential variables to a *basic* monotype (i.e., another existential variable, type variable, and unit type); (2) they always split an existential variable into two fresh existential variables when it is compared with an arrow type. This new set of rules solves existential variables to arbitrary monotypes by employing *polytype splitting*: existential variables are only split into two when compared with a *polytype* (or non-monotype) arrow type. Since $\widehat{\alpha}_1$ and $\widehat{\alpha}_2$ are fresh, the consequence of the worklist substitution in rules 15 and 16 is equivalent to inserting $\widehat{\alpha}_1, \widehat{\alpha}_2$ before $\widehat{\alpha}$, replacing every reference to $\widehat{\alpha}$ in Γ by $\widehat{\alpha}_1 \to \widehat{\alpha}_2$, and removing $\widehat{\alpha}$. The *occurs-check* condition in all four rules prevents the possible non-termination of the algorithm caused by judgments like $\widehat{\alpha} \leq 1 \to \widehat{\alpha}$. In rules 13 and 14, this check also prevents overlapping with rule 6 since $\widehat{\alpha} \leq \widehat{\alpha}$ fails such a check. The side condition $\tau \neq \widehat{\beta}$ in rule 14 ensures that rule 13 takes priority when comparing two different existential variables. Compared with the rules of $F^e_{\leq}$, the new formulation has fewer rules and simplifies the proof of decidability, which will be discussed in detail in the Section 4.3.

$$
\begin{array}{rll}
\Gamma \Vdash 1 \leq 1 & \longrightarrow_4 & \Gamma \\
\Gamma \Vdash a \leq a & \longrightarrow_5 & \Gamma \\
\Gamma \Vdash \widehat{\alpha} \leq \widehat{\alpha} & \longrightarrow_6 & \Gamma \\
\Gamma \Vdash a \leq B & \longrightarrow_7 & \Gamma \Vdash A \leq B \quad \text{when } a \leqq A \in \Gamma \land A \neq \top \land B \neq \top \land B \neq a \\
\Gamma \Vdash A \leq \top & \longrightarrow_8 & \Gamma \\
\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 \leq B).\, A \leq C & \longrightarrow_{11} & \Gamma, \widehat{\alpha} \Vdash \widehat{\alpha} \leq B \Vdash [\widehat{\alpha}/a]A \leq C \\
 & & \quad \text{when } C \text{ is not a } \forall \text{ type} \land C \neq \top \\
\Gamma \Vdash \forall(a \leq B_1).\, A_1 \leq \forall(a \leq B_2).\, A_2 & \longrightarrow_{12} & \Gamma, a \lesssim B_1 \Vdash B_2 \leq B_1 \Vdash B_1 \leq B_2 \Vdash A_1 \leq A_2 \\
\Gamma \Vdash \widehat{\alpha} \leq \tau & \longrightarrow_{13} & \{\tau/\widehat{\alpha}\}\Gamma \quad \text{when } \Gamma \vdash^m \tau \land \widehat{\alpha} \notin FV(\tau) \\
\Gamma \Vdash \tau \leq \widehat{\alpha} & \longrightarrow_{14} & \{\tau/\widehat{\alpha}\}\Gamma \quad \text{when } \Gamma \vdash^m \tau \land \widehat{\alpha} \notin FV(\tau) \land \tau \neq \widehat{\beta} \\
\Gamma \Vdash \widehat{\alpha} \leq A \to B & \longrightarrow_{15} & \{\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 } \Gamma \nvdash^m (A \to B) \land \widehat{\alpha} \notin FV(A \to B) \\
\Gamma \Vdash A \to B \leq \widehat{\alpha} & \longrightarrow_{16} & \{\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 } \Gamma \nvdash^m (A \to B) \land \widehat{\alpha} \notin FV(A \to B)
\end{array}
$$

Fig. 5. Reduction Rules of Algorithmic Subtyping

Typing (Rules 17-36, Figure 6). Rules 17-20 deal with checking, rules 21-26 deal with inference, rules 28-32 deal with application inference and rules 34-36 deal with type inference judgments.

Rules 17-19 are the algorithmic counterparts of ⇐Sub, ⇐→ and ⇐→⊤. The premise of the inference judgment in ⇐Sub is modified to the continuation-passing style. The side condition $e \neq \lambda x.\, e'$ in 17 prevents the overlapping with the rule 19. Rule 20 is added for existential variables. An existential variable can be used to check a lambda expression iff it can finally be resolved to an arrow type, so we split it into two fresh existential variables and process the worklist with the worklist substitution.

Rules 21-27 are the algorithmic counterparts of ⇒Var, ⇒Anno, ⇒Λ, ⇒Unit, ⇒App, ⇒TApp and ⇒→Mono. Rules 21-24 are the base cases where the type is fully determined from the expression, so we replace the placeholder a in ω with the type. Rules 22 and 23 also push $e \Leftarrow A$ to the worklist to check the expression e has type A. Rules 25 and 26 infer the result of (type) application by inferring the type of e_1 and leaving the remaining work to matching and (type) application-inference judgments. Rule 27 creates two fresh existential variables $\widehat{\alpha}, \widehat{\beta}$ to solve the unknown monotype of the lambda expression by replacing the placeholder a with $\widehat{\alpha} \to \widehat{\beta}$ in ω and checking the body against $\widehat{\beta}$.

Rules 28-31 are the algorithmic counterparts of ▹→, ▹∀, ▹⊥, and ▹Var. Rules 28 and 29 are two base cases where an arrow type is known (by viewing $\bot$ as $\top \to \bot$). The placeholder a, b in ω is then replaced with the domain and codomain of this known arrow type. The modification of rule 30 of introducing an existential variable $\widehat{\alpha}$ is similar to that of rule 11. This $\widehat{\alpha}$ will finally be unified with a monotype that satisfies the bound. Rule 32 is added for existential variables. Since this existential variable must match an arrow type, we split it into two fresh variables and process the worklist with the worklist substitution. Rule 33 checks whether an expression of the arrow type $A \to B$ solved by the matching can be applied to another expression e by adding a checking

$$
\begin{array}{rll}
\Gamma \Vdash e \Leftarrow B & \longrightarrow_{17} \Gamma \Vdash e \Rightarrow_a a \leq B & \text{when } e \neq \lambda x.\, e' \\
\Gamma \Vdash \lambda x.\, e \Leftarrow A \to B & \longrightarrow_{18} \Gamma, x : A \Vdash e \Leftarrow B & \\
\Gamma \Vdash \lambda x.\, e \Leftarrow \top & \longrightarrow_{19} \Gamma, x : \bot \vdash e \Leftarrow \top & \\
\Gamma \Vdash \lambda x.\, e \Leftarrow \widehat{\alpha} & \longrightarrow_{20} \{\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}\}(\Gamma, \widehat{\alpha}_1, \widehat{\alpha}_2 \Vdash \lambda x.\, e \Leftarrow \widehat{\alpha}) & \\
\Gamma \Vdash x \Rightarrow_a \omega & \longrightarrow_{21} \Gamma \Vdash [A/a]\omega & \text{when } x : A \in \Gamma \\
\Gamma \Vdash e : A \Rightarrow_a \omega & \longrightarrow_{22} \Gamma \Vdash [A/a]\omega \Vdash e \Leftarrow A & \\
\Gamma \Vdash (\Lambda(a \leq B).\, e : A) \Rightarrow_b \omega & \longrightarrow_{23} \Gamma \Vdash ([\forall(a \leq B).\, A/b]\omega), a \leq B \Vdash e \Leftarrow A & \\
\Gamma \Vdash () \Rightarrow_a \omega & \longrightarrow_{24} \Gamma \Vdash [1/a]\omega & \\
\Gamma \Vdash e_1\, e_2 \Rightarrow_a \omega & \longrightarrow_{25} \Gamma \Vdash e_1 \Rightarrow_b (b \triangleright_{c,d} (c \to d \bullet e_2 \Rrightarrow_a \omega)) & \\
\Gamma \Vdash e\, @A \Rightarrow_a \omega & \longrightarrow_{26} \Gamma \Vdash e \Rightarrow_b (b \circ A \Rrightarrow_a \omega) & \\
\Gamma \Vdash \lambda x.\, e \Rightarrow_a \omega & \longrightarrow_{27} \Gamma, \widehat{\alpha}, \widehat{\beta} \Vdash ([\widehat{\alpha} \to \widehat{\beta}/a]\omega), x : \widehat{\alpha} \Vdash e \Leftarrow \widehat{\beta} & \\
\Gamma \Vdash A \to B \triangleright_{a,b} \omega & \longrightarrow_{28} \Gamma \Vdash [A/a, B/b]\omega & \\
\Gamma \Vdash \bot \triangleright_{a,b} \omega & \longrightarrow_{29} \Gamma \Vdash [\top/a, \bot/b]\omega & \\
\Gamma \Vdash \forall(a \leq B).\, A \triangleright_{a,b} \omega & \longrightarrow_{30} \Gamma, \widehat{\alpha} \Vdash \widehat{\alpha} \leq B \Vdash [\widehat{\alpha}/a]A \triangleright_{a,b} \omega & \\
\Gamma \Vdash a \triangleright_{b,c} \omega & \longrightarrow_{31} \Gamma \Vdash A \triangleright_{b,c} \omega & \text{when } a \leq A \in \Gamma \\
\Gamma \Vdash \widehat{\alpha} \triangleright_{b,c} \omega & \longrightarrow_{32} \{\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}\}(\Gamma, \widehat{\alpha}_1, \widehat{\alpha}_2 \Vdash \widehat{\alpha} \triangleright_{b,c} \omega) & \\
\Gamma \Vdash A \to B \bullet e \Rrightarrow_a \omega & \longrightarrow_{33} \Gamma \Vdash [B/a]\omega \Vdash e \Leftarrow A & \\
\Gamma \Vdash \forall(b \leq B).\, A \circ C \Rrightarrow_a \omega & \longrightarrow_{34} \Gamma \Vdash C \leq B \Vdash [([C/b]A)/a]\omega & \\
\Gamma \Vdash \bot \circ A \Rrightarrow_a \omega & \longrightarrow_{35} \Gamma \Vdash [\bot/a]\omega & \\
\Gamma \Vdash a \circ B \Rrightarrow_a \omega & \longrightarrow_{36} \Gamma \Vdash A \circ B \Rrightarrow_a \omega & \text{when } a \leq A \in \Gamma
\end{array}
$$

Fig. 6. Reduction Rules of Algorithmic Typing

judgment $e \Leftarrow A$ to the worklist, and then replacing the placeholder a in ω with the result of the application B. The reintroduction of the application-inference judgments in this rule is because the matching output c and d in rule 25 are used in two separate judgments: c in $e_2 \Leftarrow c$, and d in $[d/a]\omega$. Since the inner judgment ω can only pass the information about the placeholder a in our continuation passing syntax, an extra judgment is necessary to distribute c and d.

Rules 34-36 are the algorithmic counterparts of $\circ\Rrightarrow\forall$, $\circ\Rrightarrow\bot$, and $\circ\Rrightarrow\mathsf{Var}$ respectively. Rules 34 and 35 are two base cases where the result of the type application can be fully determined, so we replace the placeholder a in ω with this known result type. There is no new rule for existential variables because a monotype can never be type-applied under the monotype definition.

4.3 Metatheory

To help formalize the correspondence between the declarative and algorithmic systems, we adopt the approach used by Zhao et al. [2019], which adds another intermediate system defined by a set of reduction rules on the declarative worklist Ω. The syntax of Ω is the same as the algorithmic worklist except all types in its judgments and variable declarations are *declarative* types. Similar to the notations in the algorithmic worklist, $\vdash \Omega$, $\Omega \vdash A$ and $\Omega \vdash \omega$ denote the well-formedness.

There are two sets of reduction rules for this intermediate system, declarative reduction $\Omega \longrightarrow \Omega'$ and declarative worklist reduction $\Omega \longrightarrow_a \Omega'$, for soundness and completeness proofs, respectively. $\Omega \longrightarrow \Omega'$ is a rephrasing of the declarative judgments using the worklist style, while $\Omega \longrightarrow_a \Omega'$

mimics the algorithmic reduction rules but still guesses the monotype τ instead of introducing existential variables. The detailed rules of well-formedness and reduction of the declarative worklist can be found in the extended version of this paper. $\Omega \longrightarrow^* \cdot$ and $\Omega \longrightarrow_a^* \cdot$ are proved equivalent:

THEOREM 4.1 (EQUIVALENCE OF DECLARATIVE REDUCTION). *$\forall \Omega$, if $\vdash \Omega$, then $\Omega \longrightarrow^* \cdot$ iff $\Omega \longrightarrow_a^* \cdot$.*

Soundness and Completeness. Soundness and completeness are built on worklist instantiation [Zhao et al. 2019], which converts an algorithmic worklist Γ to a declarative worklist Ω by replacing all existential variables $\widehat{\alpha}$ in Γ with a well-formed monotype τ in Ω.

$\boxed{\Gamma \rightsquigarrow \Omega}$ Ω is instantiated to Γ

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

Their formal statement is standard as described in the following two theorems. The quantifier in the two lemmas is different because worklist instantiation is a non-deterministic relation.

THEOREM 4.2 (SOUNDNESS). *If $\vdash \Gamma$ and $\Gamma \longrightarrow^* \cdot$, then $\exists \Omega$, s.t. $\Gamma \rightsquigarrow \Omega$ and $\Omega \longrightarrow^* \cdot$.*

THEOREM 4.3 (COMPLETENESS). *If $\Omega \longrightarrow_a^* \cdot$, then $\forall \Gamma$, if $\vdash \Gamma$ and $\Gamma \rightsquigarrow \Omega$, then $\Gamma \longrightarrow^* \cdot$.*

The proof proceeds by induction of the derivation of $\Gamma \longrightarrow^* \cdot$ and $\Omega \longrightarrow_a^* \cdot$, respectively. Compared with the previous proof, there are two differences. First, existential-variable solving is all dealt with by worklist substitution. The corresponding cases on the proof are unified, all depending on the instantiation-consistency lemma that worklist substitution preserves instantiation. We prove the lemma by induction on an equivalent tail-recursive version of the worklist substitution.

LEMMA 4.4 (INSTANTIATION CONSISTENCY).

(1) If $\{\tau/\widehat{\alpha}\}\Gamma \rightsquigarrow \Omega$, then $\Gamma \rightsquigarrow \Omega$;
(2) If $\Gamma \Vdash \widehat{\alpha} \leq \tau \rightsquigarrow \Omega \Vdash \tau' \leq \tau'$ or $\Gamma \Vdash \tau \leq \widehat{\alpha} \rightsquigarrow \Omega \Vdash \tau' \leq \tau'$, then $\{\tau/\widehat{\alpha}\}\Gamma \rightsquigarrow \Omega$.

Secondly, the case $\leq\rightarrow$ of the completeness proof relies on a property to ensure the occurrence-check condition in rules 13-16 is satisfied, enabling the reduction on the algorithmic worklist to continue. The proof of this lemma in $F_{\leq}^{e}$ was based on the contradiction between the following two properties (1) if $A \rightarrow B$ contains $\widehat{\alpha}$, $A_1 \rightarrow B_1$ contains strictly more arrows that C and (2) if $\Omega \Vdash \tau \leq A$, τ has more arrows than A. Fortunately, the above two properties still hold in $F_{\leq}^{b}$ by just counting the arrows in bounds (the number of arrows of a (sub)type variable is still zero).

LEMMA 4.5 (PRUNE TRANSFER FOR INSTANTIATION). *If $(\Gamma \Vdash \widehat{\alpha} \leq A \rightarrow B) \rightsquigarrow (\Omega \Vdash C \leq A_1 \rightarrow B_1)$ and $\Omega \vdash C \leq A_1 \rightarrow B_1$, then $\widehat{\alpha} \notin FV(A) \cup FV(B)$.*

Confluence and Determinism. To demonstrate the greedy nature of the algorithmic approach (i.e., it always picks the first instantiation and does not backtrack), we show that the algorithmic system is confluent and deterministic. These properties suggest that the algorithm can be implemented in an efficient way.

THEOREM 4.6 (ALGORITHMIC-REDUCTION DETERMINISM). *Given $\vdash \Gamma$, if $\Gamma \longrightarrow \Gamma_1$ and $\Gamma \longrightarrow \Gamma_2$, then $\Gamma_1 = \Gamma_2$ (up to α-equivalance).*

COROLLARY 4.7 (CONFLUENCE). *Given $\vdash \Gamma$, if $\Gamma \longrightarrow^* \Gamma_1$ and $\Gamma \longrightarrow^* \Gamma_2$, then $\exists \Gamma_3, \Gamma_4$ s.t. $\Gamma_1 \longrightarrow^* \Gamma_3$ and $\Gamma_2 \longrightarrow^* \Gamma_4$ and $\Gamma_3 = \Gamma_4$ (up to α-equivalance).*

Confluence is a direct corollary of Lemma 4.6, proved by induction on the derivation of $\Gamma \longrightarrow^* \Gamma_1$. Removing some side-conditions would make the algorithm non-deterministic, but could still retain confluence. Most of the restrictions that can be dropped, while retaining confluence, are checks that a type cannot be $\top$. For instance, we do not need $B \neq \top$ in rule 7 or $C \neq \top$ in rule 11. Besides, $\tau \neq \widehat{\beta}$ in rule 14 can also be dropped. The more interesting restriction that we believe can be dropped is the monotype restriction in rules 15 and 16. All the other conditions seem necessary not to get into any stuck state. We conjecture that the removal of restrictions preserves confluence (while losing determinism), but we did not formally study that version of the algorithm.

Decidability. The decidability proof is based on a lexicographic group of four measures on the worklist Γ: $\langle |\Gamma|_e, |\Gamma|_\omega, 2|\Gamma|_\rightarrow + |\Gamma|_\forall + |\Gamma|_{\widehat{\alpha}}, |\Gamma|_w \rangle$. $|\Gamma|_e$, $|\Gamma|_\omega$, $|\Gamma|_\rightarrow$, $|\Gamma|_\forall$, $|\Gamma|_{\widehat{\alpha}}$, $|\Gamma|_w$ are the term size, number of judgments, number of splits, number of $\forall$, number of existential-variable declarations, and a simplified version of weight, of the worklist, respectively. Compared with the measure used by $F^e_\leq$, the key innovation is $|\Gamma|_\rightarrow$, which statically computes the number of maximum splits of a type A, assuming it is compared with an existential variable in subtyping. All the detailed measure definitions are in the extended version of this paper. The $|\cdot|_\rightarrow$ and $|\cdot|_\forall$ measures enjoy the desirable property of remaining unchanged after substituting an existential variable with *any* monotype, implying that both existential-variable solving and worklist instantiation will not affect them.

Using $|\cdot|_\rightarrow$ enables us to reduce the number of measures required for the decidability proof. In $F^e_\leq$, a tuple of 6 different measures (instead of 4) is needed. It also solves the complication in the decidability proof when comparing existential variables with an arrow type in subtyping. Zhao et al. uses an *instantiation decidability* lemma to state that their measure will finally decrease after an *indefinite* number of steps after they split an existential variable by observing how worklist reduction behaves on instantiation judgments. Instead, in our rules 15 and 16, $|\Gamma|_\rightarrow$ decreases by one, and $|\Gamma|_{\widehat{\alpha}}$ increases by one after a two-step reduction, so the whole measure decreases.

The most interesting case in our decidability proof is the rule 12, where the rebounding happens for the a in A_2 during the reduction from $|\forall(a \leq B_2).\, A_2|^\Gamma_\rightarrow$ to $|A_2|^{\Gamma, a \lesssim B_1}_\rightarrow$. To proceed with the proof, we need to first find some connections between the measures before and after the rebounding based on the condition that B_1 and B_2 are equivalent.

According to the definition of $|\cdot|^\Gamma_\rightarrow$, we can prove that if two declarative types A and B are equivalent, then $|A|^\Omega_\rightarrow = |B|^\Omega_\rightarrow$ and $|A|^\Omega_\forall = |B|^\Omega_\forall$ both hold.

Lemma 4.8 ($|\cdot|_\rightarrow$ and $|\cdot|_\forall$ Equality of Equivalent Declarative Type). *Given* $\vdash \Omega \Vdash A \leq B \Vdash B \leq A$, *if* $\Omega \Vdash A \leq B \Vdash B \leq A \longrightarrow^* \cdot$, *then* $|A|^\Omega_\rightarrow = |B|^\Omega_\rightarrow$ *and* $|A|^\Omega_\forall = |B|^\Omega_\forall$.

Using soundness, completeness, and the property that $|\cdot|_\rightarrow$ and $|\cdot|_\forall$ remains unchanged under worklist instantiation, we can transfer this property to the algorithmic system.

Lemma 4.9 ($|\cdot|_\rightarrow$ and $|\cdot|_\forall$ Equality of Equivalent Algorithmic Type). *Given* $\vdash \Gamma \Vdash A \leq B \Vdash B \leq A$, *if* $\Gamma \Vdash A \leq B \Vdash B \leq A \longrightarrow^* \cdot$, *then* $|A|^\Gamma_\rightarrow = |B|^\Gamma_\rightarrow$ *and* $|A|^\Gamma_\forall = |B|^\Gamma_\forall$.

Then, we apply the induction hypothesis on $\Gamma_1 : \Gamma, a \lesssim B_1 \Vdash B_2 \leq B_1 \Vdash B_1 \leq B_2$ to get that Γ_1 reduces or not. If Γ_1 does not reduce, based on the algorithmic-reduction weakening–lemma (++ is the worklist concatenation), $\Gamma_1 \Vdash A_1 \leq A_2$ does not reduce either. The algorithmic-reduction weakening–lemma is also proved by transferring the declarative-reduction weakening–lemma using soundness and completeness.

Lemma 4.10 (Algorithmic-Reduction Weakening). *Given* $\vdash \Gamma_1 \mathbin{++} \Gamma_2$, *if* $\Gamma_1 \mathbin{++} \Gamma_2 \longrightarrow^* \cdot$, *then* $\Gamma_1 \longrightarrow^* \cdot$

Otherwise, if Γ_1 reduces, we can now utilize the equality property of $|\cdot|_\rightarrow$ and $|\cdot|_\forall$ to know these two measures of the body A_2 are not affected by the rebounding on equivalent types and the

overall measure decreases. For the remaining cases, the measures are also guaranteed to decrease after a *constant* number of steps, so the decidability proof of our system can be finalized in a straightforward manner. We can now establish the decidability theorem for both algorithmic and declarative systems.

THEOREM 4.11 (ALGORITHMIC DECIDABILITY). *Given* $\vdash \Gamma$, *it is decidable whether* $\Gamma \longrightarrow^* \cdot$ *or not.*

COROLLARY 4.12 (DECLARATIVE DECIDABILITY). *Given* $\vdash \Omega$, *it is decidable whether* $\Omega \longrightarrow^* \cdot$ *or not.*

One remark is that the dependence of the decidability proof on soundness and completeness limits the applicability of the proof technique to other variants or extensions of the system where completeness is lost. We discuss one such variant in Section 5. The dependence on the measure equality of algorithmic types can be possibly removed by modifying the algorithm to avoid re-bounding in the first place. However, such a change to the algorithm seems non-trivial. Another choice is to prove the equality properties directly on the algorithmic system. The dependence on the algorithmic-reduction weakening is difficult to remove and proving the lemma directly seems to require a simultaneous proof with the decidability itself. Abella's lack of handy mathematical reasoning already resulted in a complicated decidability proof, so we did not attempt a direct proof for either of the properties. Exploring solutions to simplifying the proof and removing the dependence on soundness and completeness is left for future work.

5 EXPLORING THE DESIGN SPACE

In this section, we revisit some design decisions of $F^b_\leq$ and present a sound variant of $F^b_\leq$ that supports subtyping between monotypes, at the cost of completeness.

5.1 $F^b_\leq$ with Monotype Subtyping

The inference algorithm of $F^b_\leq$, as presented in the earlier sections of the paper, relies on the equality of monotypes. Consequently, implicit instantiation cannot employ subtyping between monotypes, which can be limiting, since subtyping between monotypes is pervasive in object-oriented languages. Equality of monotypes is basically imposed by the desire to obtain a *complete* type-inference algorithm, and the choice of greedy instantiation. Gladly, it is possible to generalize our definition of monotypes to allow for monotype subtyping, while still having a sound type inference algorithm.

More inference without completeness. We can broaden our definition of monotypes to get more type inference. We consider type variables bounded by monotypes to be themselves monotypes in our new variant of $F^b_\leq$. This is achieved by adding the following rule to the definition of monotypes in Figure 2:

$$\frac{a \leq A \in \Psi \quad \Psi \vdash^m A}{\Psi \vdash^m a} \texttt{MTVarRec}$$

With this rule, we can accept more programs where implicit instantiation can use type variables with (monotype) bounds. This rule also makes the definition of monotypes even less syntactic as now we need to recursively analyze the bounds of type variables to determine whether a type variable is a monotype or not. In this design, we still obtain a sound algorithm with respect to the declarative specification, but completeness is lost. The algorithmic system behaves similarly to the implementation by Cardelli [1993], where the success or failure of type inference becomes less predictable. An example that illustrates the loss of completeness is the subtyping judgment $\cdot, b \leq 1 \Vdash \forall(a \leq 1).\, a \to a \to a \leq 1 \to b \to b$, which is accepted by the new declarative system by

instantiating a with b (as b is regarded as a monotype in the new definition). However, it is rejected by algorithmic reduction because the algorithm will solve $\widehat{\alpha}$ to 1 after a two-step reduction.

$$\cdot, b \leq 1 \Vdash \forall(a \leq 1).\, a \to a \to a \leq 1 \to b \to b \longrightarrow^* \cdot, b \leq 1, \widehat{\alpha} \Vdash \widehat{\alpha} \to \widehat{\alpha} \leq b \to b \Vdash 1 \leq \widehat{\alpha}$$

Nominal subtyping. The previous example hints for how our algorithm can be extended to deal with nominal subtyping, which is common in OOP languages. In mainstream OOP languages, class hierarchies can be defined, introducing subtyping relations between classes. For instance we can add `Circle` and `Shape` classes, where `Circle <: Shape`. Here `Circle` and `Shape` would be two monotypes, and it would be desirable to account for the subtyping relation between those types to enable implicit instantiation.

We can emulate what would happen after extending the calculus with nominal subtyping via bounded quantification. With the new `MTVarRec` rule, we can model examples involving a `Circle` and `Shape` type with a subtyping relation between them. For instance, consider the following two subtyping statements:

(1) $\cdot, \texttt{Shape} \leq \top, \texttt{Circle} \leq \texttt{Shape} \Vdash \forall(a \leq \texttt{Shape}).\, a \to a \to a \leq \texttt{Shape} \to \texttt{Circle} \to \texttt{Shape}$
(2) $\cdot, \texttt{Shape} \leq \top, \texttt{Circle} \leq \texttt{Shape} \Vdash \forall(a \leq \texttt{Shape}).\, a \to a \to a \leq \texttt{Circle} \to \texttt{Shape} \to \texttt{Shape}$

By viewing `Shape` and `Circle` just as regular (bounded) type variables, the first statement succeeds as greedy instantiation picks `Shape`, leading to $\texttt{Shape} \to \texttt{Shape} \to \texttt{Shape} \leq \texttt{Shape} \to \texttt{Circle} \to \texttt{Shape}$, which is valid. In contrast, the second statement fails as greedy instantiation results in $\texttt{Circle} \to \texttt{Circle} \to \texttt{Circle} \leq \texttt{Circle} \to \texttt{Shape} \to \texttt{Shape}$, which is invalid. This behavior closely parallels the behavior of instantiation in Scala 2 concerning curried functions:

```
trait Shape
trait Circle extends Shape
def poly2[A] : A => A => A = x => y => x
def ex1 = poly2 (new Shape {}) (new Circle{})  // Accepted
def ex2 = poly2 (new Circle{}) (new Shape{})   // Fails: instantiation picks Circle
```

In Scala 2, `ex1` is accepted, while `ex2` fails, since Scala (greedily) picks `Circle` for the instantiation of `A`. This example illustrates the left-to-right bias of Scala 2 type inference. Scala 3 seems to employ a hybrid algorithm that combines local type inference with other techniques and can accept `ex2`. Unfortunately, this algorithm is undocumented, and it is hard to make a precise comparison against it. The behavior in Scala 2 should be consistent with local type inference and algorithms employed in other OOP languages. Of course, for uncurried functions, Scala 2 employs local type inference techniques, which collect subtyping constraints for all the arguments. Thus, both of the examples with *uncurried* functions below succeed:

```
def poly[A](x : A, y : A) : A = x
def ex3 = poly(new Shape {}, new Circle{})  // Accepted
def ex4 = poly(new Circle{}, new Shape {})  // Accepted
```

The primary limitation of our greedy algorithm, compared to the algorithms employed in Scala 2, is the absence of specialized support for uncurried functions. This limitation is still important, given that mainstream OOP languages use uncurried functions by default. However, it is worth mentioning that many implicit instantiations should still work correctly with our algorithm, even with uncurried functions.

Metatheory results and practicality. The algorithmic rules of this variant are the same, except that the side condition $A \neq \top$ in rule 7 is changed to $(\Gamma \nvdash^m a) \vee (B$ is not a existential variable) for determinism. We have formalized this variant and proved all the same results that we have presented for $F^b_{\leq}$, except for completeness and decidability. Completeness does not hold, as our earlier example shows. We conjecture that the algorithm will terminate, but since our current proof

of decidability relies on the completeness, we cannot adapt it to obtain a proof of termination for the algorithm. Despite being incomplete, we believe that this variant of $F^b_\leq$ would work well in practice. Cardelli [1993] has observed that his greedy algorithm works well in practice, and other researchers [Mercer et al. 2022; Pierce and Turner 2000] corroborated Cardelli's opinion on the greedy approach. In fact, Scala's type-inference algorithm is itself (partly) greedy, and the "left-to-right" bias of Scala in doing type inference and instantiation is well-known. Seasoned Scala programmers are aware of the left-to-right bias of the algorithm and exploit it in the design of Scala programs[3]. Thus, we think that this variant of $F^b_\leq$ could work well, without causing much confusion to programmers, despite its bias in the instantiation order.

5.2 Other Possible Extensions

Complete type inference with monotype subtyping. One alternative way to obtain more type inference while preserving completeness would be to use a non-greedy algorithm instead. The greediness of our algorithm shows up in rules such as:

$$\Gamma \Vdash \widehat{\alpha} \leq \tau \longrightarrow_{13} \{\tau/\widehat{\alpha}\}\Gamma \qquad \text{when } \widehat{\alpha} \notin FV(\tau)$$

Here, once an existential variable is found to be a subtype of a monotype τ, we immediately solve it to that monotype by substituting $\widehat{\alpha}$ by τ in the remaining worklist. A non-greedy algorithm cannot employ substitution directly and should, instead, collect subtyping constraints. For instance, instead of an entry $\Gamma, \widehat{\alpha}$ in the worklist, we could have, $\Gamma, A \leq \widehat{\alpha} \leq B$, where A and B are lower and upper bounds, respectively, of $\widehat{\alpha}$. In a rule such as the above, instead of substituting $\widehat{\alpha}$ by τ, we would need to update the upper bound of $\widehat{\alpha}$ in the worklist. This change in the algorithm would be non-trivial and require many modifications, as well as changes in the metatheory. With this change, we would be able to delay instantiation until all the constraints have been collected. In turn, this would avoid the bias of the (incomplete) greedy algorithm in Section 5.1. We are interested in exploring this idea in the future, but have not developed an algorithm and metatheory yet.

Let polymorphism. Another way to get more inference with the current design of $F^b_\leq$ is to add let polymorphism. We believe that $F^b_\leq$ has interesting properties that would make this easier than previous designs attempting to add HM-style type inference in languages with subtyping. Our definition of monotypes in Figure 2 forbids all bounded type variables (i.e., the bound can only be $\top$). While this is quite conservative for some programs with subtyping, it does have some advantages. If generalization is applied to a monotype containing existential type variables, then all those existential type variables would have to be *unbounded* if generalized. This would mean that no simplification of polymorphic constraints would be needed, since there could be no bounds for generalized type variables.

A problem in adapting this idea to $F^b_\leq$ is that, in the current design, the order of type variables in universal quantification is relevant. But, for let generalization, we should have order-irrelevant universal quantification. Fortunately, Eisenberg et al. [2016] has studied a related problem and proposed a solution that enables both order-relevant quantifiers (which can use explicit type applications), as well as order-irrelevant quantifiers, which enable HM-style let polymorphism (but cannot use explicit type applications). Eisenberg et al.'s approach is currently implemented in the GHC Haskell compiler. We believe that a similar design could be adopted for $F^b_\leq$.

6 RELATED WORK

Local type inference. Local type inference has been shown to work in the presence of bounded quantification by Pierce and Turner [2000], albeit with some limitations as discussed in Section 2.

[3] See, for example: https://discuss.daml.com/t/scala-inference-rules-of-thumb/2242.

Despite its success and impact, local type inference has only been studied formally in restricted settings, which are basically variants of System F and $F_{\leq}$. Many of the extensions of local type inference used in practical implementations have not been formally studied. *Colored* local type inference [Odersky et al. 2001] enables the propagation of partial type information by coloring inherited and synthesized types according to their directions of propagation in the syntax tree. The algorithm was also adopted by Plociniczak [2016] to analyze the type errors and it is the basis of the type-inference algorithm in Scala 2. Section 2 gives an extensive overview of (colored) local type inference, and compares it to $F^b_{\leq}$.

The treatment of curried applications is limited in local type inference, possibly hindering its usability for (functional) languages where curried applications are common. *Spine-local type inference* [Jenkins and Stump 2018] takes an approach that utilizes the type information from the application spine to address this limitation, improving type inference for curried applications. An example that illustrates this limitation of local type inference, due to Jenkins and Stump, is:

$$\texttt{pair} : \forall a.\, \forall b.\, a \to b \to (a \times b) \Vdash \texttt{pair}\ (\lambda x.x)\ 0 : (\texttt{Int} \to \texttt{Int}) \times \texttt{Int} \Rightarrow (\texttt{Int} \to \texttt{Int}) \times \texttt{Int}$$

Even if full types $(\texttt{Int} \to \texttt{Int}) \times \texttt{Int}$ are provided in a type annotation, the above program fails to type-check with classic local type inference because the type of the first argument $\lambda x.x$ cannot be inferred. Jenkins and Stump propose a form of contextual type-argument inference, which allows the propagation of constraints in curried-function applications, accepting the above example. $F^b_{\leq}$ treats curried and uncurried applications uniformly with existential variables, naturally solving the problem. In the above example, $F^b_{\leq}$ will generate $\widehat{\alpha}$ and $\widehat{\beta}$, which later will be instantiated to $\texttt{Int} \to \texttt{Int}$ and $\texttt{Int}$ during constraint solving. In addition, in $F^b_{\leq}$, even the expression without the annotation ($\texttt{pair}\ (\lambda x.x)\ 0$) can type-check due to $F^b_{\leq}$'s global approach to type-inference. In contrast, without type annotations, $\texttt{pair}\ (\lambda x.x)\ 0$ would be rejected using (spine) local type inference approaches. However, since $F^b_{\leq}$ only solves existential variables to monotypes, spine-local type inference accepts some examples that are rejected in $F^b_{\leq}$:

$$\texttt{pair} : \forall a.\, \forall b.\, a \to b \to (a \times b) \Vdash \texttt{pair}\ (\lambda x.x)\ 0 : (\forall a.\, a \to a) \times \texttt{Int} \Rightarrow (\forall a.\, a \to a) \times \texttt{Int}$$

$F^b_{\leq}$ requires explicit instantiation for the above example since the type that should be inferred is a polytype, $\forall a.\, a \to a$. Mercer et al. [2022] proposed a local type-inference algorithm for an impredicative polarized call-by-push-value variant of System F. The proposed algorithm is formally proved to be decidable. *Quick Look* [Serrano et al. 2020], adopted in GHC 9, provides impredicative inference, but subtyping of function types is treated invariantly.

Unlike our work, these recent developments adopting ideas of local type inference focus on traditional System F, and do not consider bounded quantification or top and bottom types.

Global type inference with bounded quantification. Global type inference employs long-distance constraints via unification variables. Cardelli [1993]'s $F_{\leq}$ implementation provides a heuristic global type-inference algorithm that is not formally studied. It introduces unification variables to support long-distance constraints. Impredicative instantiation is implemented by unifying the unification variable with the first type constraint it encounters. In other words, Cardelli adopts a *greedy* approach, like our work. Type variables are directly instantiated to their bounds. However, the algorithm is not complete in the sense that the success of its greedy solving scheme depends on the order of candidate instantiations. On the other hand, $F^b_{\leq}$ is guaranteed to find the correct solution meeting our declarative specification, yielding a more predictable behavior. $F^b_{\leq}$ achieves this by distinguishing monotypes, which can always be guessed, from more general polytypes that require explicit type applications.

Sequeira [1998] studied a predicative variant of System F with bounded quantification, named $ML_{\forall\leq}$. $\mathrm{ML}_{\forall\leq}$ is not an extension of $F_{\leq}$ and employs some different rules. In particular, there

are at least two important restrictions. Firstly, $ML_{\forall\leq}$ is predicative and does not support explicit type applications. Thus, it cannot encode impredicative instantiations of $F_{\leq}$. Secondly, the bounds for type variables are restricted to monotypes, whereas $F_{\leq}$ supports unrestricted bounds. $ML_{\forall\leq}$ is more ambitious than our work in the sense that it aims at complete Hindley-Milner style inference, whereas we do not support let generalization or inference of principal types. Another important issue, which was left unsolved in $ML_{\forall\leq}$, is that it does not address the issue of simplifying constraints. This results in an impractical system where the inferred types can be too hard for humans to understand due to the number of constraints on types.

Type inference with subtyping and MLSub. The conventional approach to tackle type inference in HM extended with subtyping is collecting the constraint set and simplifying it using finite automata [Eifrig et al. 1995a,b; Pottier 1998]. ML^F [Rémy and Yakobowski 2008] extends the syntax of System F with *flexible quantification* $\forall(a \geq B).\ A$ and *rigid quantification* $\forall(a = B).\ A$ to help track the information for possible instantiations, requiring annotations on variables with divergent impredicative instantiations only. Similarly to *Quick Look*, HMF [Leijen 2008] also restricts the subtyping of the function type to be invariant. MLsub [Dolan and Mycroft 2017; Parreaux 2020] has the more ambitious goal to have *complete* inference of principal types with (first-order, or rank-1) polymorphic functions. To obtain the principality of inference and subsumption in the presence of subtyping, MLsub builds an initial distributive lattice over type with a specific group of type constructors: equi-recursive, union, and intersection types. MLstruct [Parreaux and Chau 2022] extends MLsub by supporting first-class union and intersection types with a new boolean algebra over types. $F^b_{\leq}$, has more modest goals in terms of type inference. Full inference for System F and $F_{\leq}$ without type annotations, is undecidable, so the techniques of MLsub and its variants cannot be used directly. More generally MLsub/struct address first-order polymorphism only, whereas $F^b_{\leq}$ deals with higher-order polymorphism, and also supports impredicative explicit type applications. Nevertheless, we expect that several of the techniques developed in MLSub/struct will prove to be useful in the future to improve partial type inference methods.

7 CONCLUSION

Type inference is vital in polymorphic programming languages to alleviate the burden of writing type annotations everywhere. However, it remains controversial how to balance the expressiveness of the language and the ability to obtain a practical and effective type inference algorithm. In this paper, we propose $F^b_{\leq}$, which extends elementary type inference and $F^e_{\leq}$ to bounded quantification, and inherits $F^e_{\leq}$'s modest approach to global type inference. With a careful design of what should be regarded as monotypes, we obtain a predictable and easy-to-implement algorithm that is sound and complete with respect to the declarative system. Meanwhile, instantiations involving polytypes are still available using explicit type applications. Thus, $F^b_{\leq}$ can express all kernel $F_{\leq}$ programs. We also study a variant of $F^b_{\leq}$ that can infer types for more programs at the cost of completeness. This variant supports monotype subtyping, and has a left-to-right bias with respect to instantiation, which is similar to the bias that exists in languages like Scala. Thus we believe that, despite incompleteness, this variant can be used in practical type-inference algorithms.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their valuable comments, to Yaoda Zhou for his helpful feedback on the draft, and especially to Jinxu Zhao for his help in the earlier designs of $F^b_{\leq}$. The research is supported by the Practical Type Inference with Bounded Quantification and Union and Intersection Types collaboration project (TC20230508031) between Huawei and The University of Hong Kong and project number 17209821 of Hong Kong Research Grants Council.

REFERENCES

Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. 1991. Inheritance as implicit coercion. *Information and Computation* 93, 1 (1991), 172–221. https://doi.org/10.1016/0890-5401(91)90055-7 Selections from 1989 IEEE Symposium on Logic in Computer Science.

Luca Cardelli. 1993. *An Implementation of $F_{<:}$.* Technical Report. SRC Research Reports, Digital Equipment Corporation.

Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. 1991. An Extension of System F with Subtyping. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS '91)*. Springer-Verlag, Berlin, Heidelberg, 750–770.

Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.* 17, 4 (dec 1985), 471–523. https://doi.org/10.1145/6041.6042

Pierre-Louis Curien and Giorgio Ghelli. 1992. Coherence of subsumption, minimum typing and type-checking in F≤. *Mathematical Structures in Computer Science* 2, 1 (1992), 55–91. https://doi.org/10.1017/S0960129500001134

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 60–72. https://doi.org/10.1145/3009837.3009882

Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-rank Polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*.

Jana Dunfield and Neelakantan R. Krishnaswami. 2019. Sound and Complete Bidirectional Typechecking for Higher-Rank Polymorphism with Existentials and Indexed Types. *PACMPL* POPL (Jan. 2019). http://arxiv.org/abs/1601.05106.

Jonathan Eifrig, Scott Smith, and Valery Trifonov. 1995a. Sound Polymorphic Type Inference for Objects *(OOPSLA '95)*. Association for Computing Machinery, New York, NY, USA, 169–184. https://doi.org/10.1145/217838.217858

Jonathan Eifrig, Scott Smith, and Valery Trifonov. 1995b. Type Inference for Recursively Constrained Types and its Application to OOP. *Electronic Notes in Theoretical Computer Science* 1 (1995), 132–153. https://doi.org/10.1016/S1571-0661(04)80008-2 MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference.

Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. 2016. Visible Type Application. In *European Symposium on Programming (ESOP)*. 229–254.

Andrew Gacek. 2008. The Abella Interactive Theorem Prover (System Description). In *Proceedings of IJCAR 2008 (Lecture Notes in Artificial Intelligence)*.

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et elimination des coupures de l'arithmétique d'ordre supérieur.* Thèse d'état. Université de Paris 7.

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

Haruo Hosoya and Benjamin C. Pierce. 1999. *How Good is Local Type Inference?* Technical Report MS-CIS-99-17. University of Pennsylvania.

Christopher Jenkins and Aaron Stump. 2018. Spine-Local Type Inference. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages* (Lowell, MA, USA) *(IFL 2018)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/3310232.3310233

Didier Le Botlan and Didier Rémy. 2003. MLF: Raising ML to the Power of System F. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*.

Daan Leijen. 2008. HMF: Simple Type Inference for First-class Polymorphism. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*.

Guillaume André Fradji Martres. 2023. Type-Preserving Compilation of Class-Based Languages. (2023), 153. https://doi.org/10.5075/epfl-thesis-8218

Henry Mercer, Cameron Ramsay, and Neel Krishnaswami. 2022. Implicit Polarized F: local type inference for impredicativity. arXiv:2203.01835 [cs.PL]

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

Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*.

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

Lionel Parreaux. 2020. The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 124 (Aug. 2020), 28 pages. https://doi.org/10.1145/3409006

Lionel Parreaux and Chun Yin Chau. 2022. MLstruct: Principal Type Inference in a Boolean Algebra of Structural Types. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 141 (oct 2022), 30 pages. https://doi.org/10.1145/3563304

Simon Peyton Jones and Mark Shields. 2004. Lexically scoped type variables. (March 2004). Draft.

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

Benjamin C. Pierce. 1992. Bounded Quantification is Undecidable. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico, USA) *(POPL '92)*. Association for Computing Machinery, New York, NY, USA, 305–315. https://doi.org/10.1145/143165.143228

Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 1–44.

Hubert Plociniczak. 2016. *Decrypting Local Type Inference.* Ph. D. Dissertation. EPFL.

François Pottier. 1998. *Type inference in the presence of subtyping: from theory to practice.* Ph. D. Dissertation. INRIA.

Didier Rémy and Boris Yakobowski. 2008. From ML to MLF: Graphic Type Constraints with Efficient Type Inference. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/1411204.1411216

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium*, B. Robinet (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 408–425.

Dilip Sequeira. 1998. *Type inference with bounded quantification.* Ph. D. Dissertation. University of Edinburgh.

Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A Quick Look at Impredicativity. *Proc. ACM Program. Lang.* 4, ICFP, Article 89 (Aug. 2020), 29 pages.

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

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32)*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

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

Ningning Xie, Xuan Bi, Bruno C. D. S. Oliveira, and Tom Schrijvers. 2019. Consistent Subtyping for All. *ACM Trans. Program. Lang. Syst.* 42, 1, Article 2 (nov 2019), 79 pages. https://doi.org/10.1145/3310339

Jinxu Zhao and Bruno C. d. S. Oliveira. 2022. Elementary Type Inference. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.2

Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. A Mechanical Formalization of Higher-Ranked Polymorphic Type Inference. *Proc. ACM Program. Lang.* 3, ICFP, Article 112 (July 2019).

Litao Zhou, Yaoda Zhou, and Bruno C. d. S. Oliveira. 2023. Recursive Subtyping for All. *Proc. ACM Program. Lang.* 7, POPL, Article 48 (jan 2023), 30 pages. https://doi.org/10.1145/3571241

Received 2023-04-14; accepted 2023-08-27