

Bidirectional Type Checking for Existential Types with Higher-Rank Polymorphism

Anonymous

Abstract. This paper presents a new type system that combines strong existential types, higher-rank polymorphism, and polymorphic subtyping. We begin by presenting a declarative type system, and contributing several metatheoretic results. We then detail an algorithmic type inference system featuring bidirectional type checking. We prove that the algorithm is sound. Moreover, we propose two design variants of the declarative system with respect to which the algorithmic system is complete. This work can serve as a basis for further studies on strong existentials, their corresponding type inference algorithms, and their combinations with higher-rank polymorphism and subtyping.

1 Introduction

Modern functional programming languages employ powerful forms of type inference, with growing interest in more expressive forms of types, such as *higher-rank polymorphism* [2, 14, 16], *impredicativity* [5, 15, 20], *generalized algebraic datatypes* (GADTs) [17, 24], and forms of dependent types [27].

With such more expressive types, users can write programs with stronger guarantees. A common example with GADTs is the length-indexed vectors¹:

```
data Nat = Zero | Succ Nat
data Vec n a where
  Nil :: Vec Zero a
  (:>) :: a → Vec n a → Vec (Succ n) a
  -- (:>) : ∀n a. a → Vec n a → Vec (Succ n) a
```

This definition enables a safe *head* function that only accepts non-empty vectors:

```
head :: ∀n a. Vec (Succ n) a → a
```

While defining *head* is straightforward, other vector-processing functions like *filter* pose a challenge. The *filter* function takes a predicate function ($a \rightarrow \text{Bool}$) and a vector ($\text{Vec } n \ a$), returning a vector with elements that satisfy the predicate. However, since the predicate function and the concrete vector are passed in only at runtime, we cannot statically know the length of the output vector. Fortunately, with *existential types*, we can give *filter* the following type signature:

```
filter :: ∀n a. (a → Bool) → Vec n a → ∃m. Vec m a
```

Here, the output type $\exists m. \text{Vec } m \ a$ denotes a vector of some unknown length m .

¹ We use GADTs for illustration, though they are not yet supported in our language.

Weak existential types via packing and unpacking. A common presentation of existential types uses `pack` and `unpack` [18, §24]. For example:

```
e ≡ pack Int, { new = 0, get = λi : Int. i, incr = λi : Int. i + 1 }
as ∃a. { new : counter, get : counter → Int, incr : counter → counter }
```

The syntax $(\text{pack } \tau_1, e \text{ as } \exists a. \tau_2)$ packages up type τ_1 and expression $e : [\tau_1 / a] \tau_2$, and produces an existential type $\exists a. \tau_2$. The type above hides the internal representation of `counter`, so that later the program can change the representation (e.g. from `Int` to `Nat`) without breaking any user code. Such a mechanism is important for modularity and *data abstraction* [12].

Unpacking the expression allows us to use its fields:

```
unpack e as a, c in get (incr (incr (new c))) -- ok: Int
```

Given $e_1 : \exists a. \tau_1$, the expression $(\text{unpack } e_1 \text{ as } a, x \text{ in } e_2)$ opens up the package and binds the type component to a and the term component to $x : \tau_1$, which can then be used inside e_2 . If e_2 has type τ , then the whole expression has type τ . However, we must ensure that the hidden type does not escape:

```
unpack e as a, c in new c -- error: a escapes in the return type
```

Existential types with `pack` and `unpack` are sometimes considered *weak* [7], in the sense that they require pattern-matching (i.e. `unpack`) to bring the type component of an existential type into scope. Since the abstract type a cannot escape the scope of `unpack`, programming with weak existentials can be inconvenient in practice: to access an existential type and define datatypes on top of its hidden type, it is necessary to `unpack` the existential type at the top of a program, to create a scope where the abstract type is available to all its clients [11].

Strong existential types via packing and opening. Strong existential types [1], in contrast, allow projection of the hidden type. Strong existentials are introduced using `pack`, similarly to weak existentials, but they are eliminated using `open` instead of `unpack`.

With strong existentials, we can directly open the previous expression e :

```
open e -- ok: { new : |e|, get : |e| → Int, incr : |e| → |e| }
new (open e) -- ok: |e|
get (incr (incr (new (open e)))) -- ok: Int
```

Specifically, if $e : \exists a. \tau$, then $(\text{open } e) : [|e| / a] \tau$, directly projecting out the term component of an existential. The *existential projection* syntax $|e|$ denotes the hidden type component of e . Such a type is opaque, in the sense that $|e|$ is equal only to itself. As such, the notation preserves abstraction, and no computation is needed on the type level. With this notation, one can refer to the hidden type or project out the term from existentials without necessarily creating a scope.

More recently, Eisenberg et al. [4] observed that strong existential types also increase laziness, compared to weak existentials. With strong existentials, we can implement `filter` as follows:

```
filter :: ∀n a. (a → Bool) → Vec n a → ∃m. Vec m a
filter = Λn a. λ(p :: a → Bool) (vec :: Vec n a) → case vec of
  Nil → pack Zero, Nil as ∃m. Vec m a
```

```
(:>) n1 a (x :: a) (xs :: Vec n1 a)
| p x → let ys = filter n1 a p xs -- a lazy let, with ys : ∃n2. Vec n2 a
    pack (Succ [ys]), (:>) [ys] a x (open ys) as ∃m. Vec m a
| otherwise → filter n1 a p xs
```

The program matches on *vec*. Of particular interest is when the vector has a head *x* and a tail *xs*, and *p x* holds. In this case, we first recursively call *filter* on *xs*, producing *ys*. Notably, we can directly access the length of *ys* using *[ys]*, without unpacking it. In contrast, an implementation based on weak existentials would require one to unpack the result of the recursive *filter*. Finally, we pack the incremented length and the extended vector. This definition implements a lazy filter: getting the head element of a filtered list will return a result immediately after seeing the first element that satisfies *p*.

However, writing programs such as *filter* with explicit type annotations can be tedious, as the types are almost as long as the terms! Instead, we would like for the compiler to automatically infer *pack*, *open*, as well as type instantiations. Early work [9, 22] on type inference with existential types mostly focused on existential datatypes [12]. Recently, Dunfield and Krishnaswami [3] infer weak existentials through subtyping. Eisenberg et al. [4] infer strong existentials, but present only a syntax-directed declarative specification, lacking a corresponding algorithmic type inference system or its metatheory.

This work. This paper introduces a novel type system that combines strong existential types, higher-rank polymorphism, and polymorphic subtyping. To our knowledge, this is the first such combination. While the combination appears intuitive, integrating these features presents subtle consequences when establishing key properties of the type system.

We begin by presenting a *bidirectional* [19] declarative type system, which exhibits several interesting properties. We then introduce an algorithmic type system, employing *ordered contexts* [2, 6] to manage unification variables, their scopes, and their solutions. This is the first algorithmic type inference system for strong existentials. The precise way that the type inference algorithm works turns out to be non-trivial, requiring us to address several challenges in both algorithm design and its corresponding proofs. We offer the following contributions:

- **A declarative type system:** We present a declarative, bidirectional, syntax-directed type system with strong existential types, higher-rank polymorphism, and subtyping (§3), and study its properties (§4).
- **A type inference algorithm:** We contribute an algorithmic type system (§5) that adapts Dunfield and Krishnaswami [2]’s *ordered contexts* approach. This system addresses several technical obstacles in applying this method to type inference for strong existentials.
- **Soundness:** We prove that the algorithm is *sound* with respect to the declarative system (§6). Thus, when the algorithmic type system infers a type for an expression, that type assignment is also valid in the declarative type system.
- **Completeness:** However, establishing *completeness* presents significant challenges. In fact, we report that the type inference algorithm is incomplete, and analyze the precise cause of this incompleteness (§7). To restore complete-

ness, we propose two design variants of the declarative system, with *restricted existential inference* (§7.1) and *restricted existential instantiation* (§7.2), respectively, and establish completeness for both variants.

- **Elaboration:** We provide an elaboration of the declarative type system and prove elaboration soundness, establishing soundness of the declarative system.
- **Implementation:** We provide a Haskell implementation of the algorithmic type checker in the supplementary materials.

For space reasons, the complete set of rules, the elaboration, and proofs of all stated lemmas and theorems, along with the implementation, are provided in the supplementary materials.

2 Overview

In this section, we outline features of our system, specifically type inference for strong existential types (§2.1), higher-rank polymorphism (§2.2), and polymorphic subtyping (§2.3). We use Haskell-like syntax for examples.

2.1 Type Inference for Strong Existential Types

While strong existential types are useful, writing fully-annotated programs such as *filter* can be tedious. Ideally, we would like to write *filter* as follows:

$$\begin{aligned} \textit{filter} &:: \forall n. a. (a \rightarrow \textit{Bool}) \rightarrow \textit{Vec } n \ a \rightarrow \exists m. \textit{Vec } m \ a \\ \textit{filter} &= \lambda p \ \textit{vec} \rightarrow \text{case } \textit{vec} \ \text{of } \textit{Nil} \rightarrow \textit{Nil} \\ &\quad (:>) \times \textit{xs} \mid p \ x \rightarrow (:>) \times (\textit{filter } p \ \textit{xs}) \\ &\quad \mid \text{otherwise} \rightarrow \textit{filter } p \ \textit{xs} \end{aligned}$$

and let compilers automatically infer existential types and type arguments.

To this end, we follow the approach of Eisenberg et al. [4] (EDWL hereafter), which infers existential types when typing lambda expressions, and opens them through type instantiation. Consider inferring the type of the following program:
 $f = \lambda p \ \textit{xs} \rightarrow (:>) \ 2 \ (\textit{filter } p \ \textit{xs})$

We begin by assigning types based on *filter* and 2: we have $p : \textit{Int} \rightarrow \textit{Bool}$ and $\textit{xs} : \textit{Vec } n \ \textit{Int}$ for some n . Moreover, we know that $(\textit{filter } p \ \textit{xs}) : \exists m. \textit{Vec } m \ \textit{Int}$. When constructing $(:>) \ 2 \ \textit{ys}$, since 2 has type \textit{Int} , the $(:>)$ constructor requires that $(\textit{filter } p \ \textit{xs})$ should be of type $(\textit{Vec } \dots \ \textit{Int})$. Therefore, we instantiate its existential type by opening it, yielding $(\textit{filter } p \ \textit{xs}) : \textit{Vec } [\textit{filter } p \ \textit{xs} : \exists m. \textit{Vec } m \ \textit{Int}] \ \textit{Int}$. Here, instantiation replaces m with $[\textit{filter } p \ \textit{xs} : \exists m. \textit{Vec } m \ \textit{Int}]$. In other words, *open* is inferred through type instantiations. We thus have

$$((:>) \ 2 \ (\textit{filter } p \ \textit{xs})) : \textit{Vec } (\textit{Succ } [\textit{filter } p \ \textit{xs} : \exists m. \textit{Vec } m \ \textit{Int}]) \ \textit{Int}$$

Now, what is the type of f ? A naive inference might yield:

$$f :: \forall n. (\textit{Int} \rightarrow \textit{Bool}) \rightarrow \textit{Vec } n \ \textit{Int} \rightarrow \textit{Vec } (\textit{Succ } [\textit{filter } p \ \textit{xs} : \exists m. \textit{Vec } m \ \textit{Int}]) \ \textit{Int}$$

However, this type is problematic because \textit{xs} is no longer in the scope! Fortunately, existential types provide a solution. When returning from a lambda expression, we can traverse the function’s return type, and replace any type that

mentions the bound variable with fresh existential variables, which are then quantified in the return type. This leads to:

$$f :: \forall n. (\text{Int} \rightarrow \text{Bool}) \rightarrow \text{Vec } n \text{ Int} \rightarrow \exists m. \text{Vec } m \text{ Int}$$

EDWL specified exactly when existentials should be inferred, but only presented a syntax-directed *declarative* system. A declarative type system provides a specification of desired typing behavior that allows for the free guessing of types. For instance, in an expression $\lambda x. e$, the type of x can be non-deterministically chosen. To actually implement type inference, we need an algorithmic type system where we create *unification variables* to be solved later. Our algorithm effectively provides such an algorithmic system and studies its properties. Additionally, we study the combination of strong existentials, higher-rank polymorphism, and polymorphic subtyping.

2.2 Higher-Rank Polymorphism

Higher-rank polymorphism [2, 14, 16], where a universal quantifier can appear contravariantly, has now appeared quite regularly in polymorphic type systems.

As an example, we adapt the *mapM* program from Peyton Jones et al. [16], which defines an explicit data structure for *Monad* as a record of two functions:

$$\begin{aligned} \text{data } \text{Monad } m = \text{Mon} \{ & \text{return} :: \forall a. a \rightarrow m a, \\ & \text{bind} :: \forall a b. m a \rightarrow (a \rightarrow m b) \rightarrow m b \} \end{aligned}$$

The data structure contains polymorphic functions *return* and *bind*, and *Mon* has a rank-2 type: $(\forall a. a \rightarrow m a) \rightarrow (\forall a b. m a \rightarrow (a \rightarrow m b) \rightarrow m b) \rightarrow \text{Monad } m$.

The function *mapM* takes an explicit argument record of type *Monad* m , and maps a monadic function f over a vector *Vec* $n a$:

$$\begin{aligned} \text{mapM} :: \text{Monad } m \rightarrow (a \rightarrow m b) \rightarrow \text{Vec } n a \rightarrow m (\text{Vec } n b) \\ \text{mapM } m @ (\text{Mon} \{ \text{return} = ret, \text{bind} = bnd \}) f xs = \text{case } xs \text{ of} \\ Nil \rightarrow ret \text{ Nil} \\ (:) x xs \rightarrow f x `bnd` \lambda y \rightarrow \text{mapM } m f xs `bnd` \lambda ys \rightarrow ret ((:) y ys) \end{aligned}$$

Here, *bnd* is used with two different types, which is possible since it is polymorphic. This demonstrates how constructors with higher-rank types can be used to simulate type classes, which is precisely how GHC implements type classes.

We can now combine *mapM* and *filter*, for example, to map over a filtered vector, assuming *Maybe* is a monad with *return* and $\gg=$:

$$\text{mapM } (\text{Mon} (\text{return}) (\gg=)) \text{ Just } (\text{filter even} ((:) 1 ((:) 2 ((:) 3 \text{ Nil}))))$$

The program has type *Maybe* (*Vec* [*filter even* ((:) 1 ((:) 2 ((:) 3 *Nil*))]) *Int*).

2.3 Polymorphic Subtyping

Our design builds upon systems with *predicative* higher-rank polymorphism, with both polymorphic subtyping and contravariant function types. Consider:

$$f : (\forall a. \text{Int} \rightarrow a \rightarrow a) \rightarrow \text{Bool} \quad g : \text{Int} \rightarrow \forall b. b \rightarrow b$$

The application $(f\ g)$ type-checks in most higher-rank polymorphic type systems [2, 3, 14, 16], where functions are contravariant in their argument types, and in this case $(\text{Int} \rightarrow \forall b. b \rightarrow b)$ is a subtype of $(\forall a. \text{Int} \rightarrow a \rightarrow a)$.

However, type-checking programs with higher-rank polymorphism also requires care. Assuming $(h : \forall a. b. a \rightarrow b \rightarrow b)$, consider the following program:

$$(\lambda(x :: \forall c. c \rightarrow \forall d. d \rightarrow d) \rightarrow x) h \text{ -- rejected}$$

Here, the function expects an argument of type $(\forall c. c \rightarrow \forall d. d \rightarrow d)$, but h has type $(\forall a. b. a \rightarrow b \rightarrow b)$. Therefore, we need to check the subtyping relation $(\forall a. b. a \rightarrow b \rightarrow b) <: (\forall c. c \rightarrow \forall d. d \rightarrow d)$, which, unfortunately, does not hold. To illustrate why, following polymorphic subtyping, we will first *skolemize* c , transforming the constraint to: $(\forall a. b. a \rightarrow b \rightarrow b) <: (c \rightarrow \forall d. d \rightarrow d)$. Next, we instantiate a with c , yielding: $(\forall b. c \rightarrow b \rightarrow b) <: (c \rightarrow \forall d. d \rightarrow d)$. At this point, we would need to instantiate b before skolemizing d . However, since d is outside the scope of b , this subtyping relation does not hold.

To correctly manage the scope of variables, our algorithmic type system employs the mechanism of *ordered contexts* [2, 6]. Additionally, the algorithmic system addresses several technical obstacles when applying this design to type inference for strong existentials.

3 Declarative Type System

This section presents our declarative type system. The syntax is given in Fig. 1. Expressions e includes literals n , variables x , lambdas $\lambda x. e$, annotated lambdas $\lambda x : \sigma. e$, applications $e_1 e_2$, and annotated expressions $(e : \sigma)$.

Types are stratified: σ represents types with top-level universal quantifiers, ϵ represents types with top-level existential quantifiers, ρ allows for quantifiers nested inside functions, and τ denotes strictly monomorphic types.² An existential projection $[e : \exists a. \epsilon]$ contains an expression e as well as a type $\exists a. \epsilon$, which is needed in order to avoid ambiguity when there are nested existentials. A context Γ maps variables to their types, and stores type variables.

3.1 Type Inference and Checking

Fig. 1 presents selected bidirectional declarative typing rules. The typing judgment has two modes: an expression e either infers (\Rightarrow) a type σ , or is checked against (\Leftarrow) a type σ . The typing rules use well-formedness judgments, where $\Gamma \vdash \sigma$ indicates that σ is well-formed under Γ , meaning all type and term variables in σ are bound in Γ ; and $\vdash \Gamma$ means that all types within Γ are well-formed. The definitions of the well-formedness judgments can be found in the appendix.

Type inference. Rule D-I-INT and rule D-I-VAR are standard. Rule D-I-ABS begins by adding $x : \tau$ to the context and inferring the type of e , which produces σ . It then uses the instantiation judgment (§3.2) to instantiate σ to a ϵ type, so

² Stratification simplifies the challenge of comparing types with mixed quantifiers [3].

expression	$e := n \mid x \mid \lambda x. e \mid \lambda x : \sigma. e \mid e_1 e_2 \mid (e : \sigma)$
universally quantified type	$\sigma := \epsilon \mid \forall a. \sigma$
existentially quantified type	$\epsilon := \rho \mid \exists b. \epsilon$
top-level monomorphic type	$\rho := \tau \mid \sigma_1 \rightarrow \sigma_2$
monomorphic type	$\tau := a \mid \text{Int} \mid \tau_1 \rightarrow \tau_2 \mid [e : \exists a. \epsilon]$
typing context	$\Gamma := \bullet \mid \Gamma, x : \sigma \mid \Gamma, a$

$\boxed{\Gamma \vdash e \Rightarrow \sigma}$	<i>(Inference: under Γ, e infers type σ)</i>
$\frac{\text{D-I-INT} \quad \vdash \Gamma}{\Gamma \vdash n \Rightarrow \text{Int}}$ $\frac{\text{D-I-VAR} \quad \vdash \Gamma \quad x : \sigma \in \Gamma}{\Gamma \vdash x \Rightarrow \sigma}$ $\frac{\text{D-I-APP} \quad \begin{array}{c} \Gamma \vdash e \Rightarrow \sigma \\ \Gamma \vdash e : \sigma \rightsquigarrow \sigma_1 \rightarrow \sigma_2 \\ \Gamma \vdash e_1 \Leftarrow \sigma_1 \end{array}}{\Gamma \vdash e e_1 \Rightarrow \sigma_2}$	$\frac{\text{D-I-ABS} \quad \begin{array}{c} \Gamma, x : \tau \vdash e \Rightarrow \sigma \\ \bar{a} \text{ fresh} \\ \epsilon' = [\bar{a}/[\epsilon]_x]\epsilon \end{array}}{\Gamma \vdash \lambda x. e \Rightarrow \tau \rightarrow \exists \bar{a}. \epsilon'}$ $\frac{\text{D-I-ANN} \quad \begin{array}{c} \Gamma \vdash e \Leftarrow \sigma \\ \Gamma \vdash \sigma \quad \text{ftv}(\sigma) = \emptyset \end{array}}{\Gamma \vdash (e : \sigma) \Rightarrow \sigma}$
$\boxed{\Gamma \vdash e \Leftarrow \sigma}$	<i>(Checking: under Γ, e checks against type σ)</i>
$\frac{\text{D-C-ABS} \quad \Gamma, x : \sigma_1 \vdash e \Leftarrow \sigma_2}{\Gamma \vdash \lambda x. e \Leftarrow \sigma_1 \rightarrow \sigma_2}$ $\frac{\text{D-C-FORALL} \quad \Gamma, a \vdash e \Leftarrow \sigma}{\Gamma \vdash e \Leftarrow \forall a. \sigma}$ $\frac{\text{D-C-SUB} \quad \begin{array}{c} \Gamma \vdash e \Rightarrow \sigma \\ \Gamma \vdash e : \sigma <: \rho \end{array}}{\Gamma \vdash e \Leftarrow \rho}$	$\frac{\text{D-C-EXISTS} \quad \begin{array}{c} \Gamma \vdash e \Leftarrow [\tau/b]\epsilon \\ \Gamma \vdash \tau \end{array}}{\Gamma \vdash e \Leftarrow \exists b. \epsilon}$

Fig. 1: Declarative typing rules

that we can introduce existential quantifiers. We describe instantiation in §3.2. Within ϵ , there may be existential projections $[e : \exists a. \epsilon']$ that refer to x . The notation $[\epsilon]_x$ denotes the set of all such existential projections:

$$[\epsilon]_x = \{[e : \exists a. \epsilon'] \mid ([e : \exists a. \epsilon'] \text{ is a sub-expression of } \epsilon) \wedge (x \in \text{fv}(e))\}$$

Since we are getting out of the scope of x , these projections are replaced with fresh variables \bar{a} , by setting $\epsilon' = [\bar{a}/[\epsilon]_x]\epsilon$. Finally, the rule infers $\tau \rightarrow \exists \bar{a}. \epsilon'$.

Rule D-I-APP handles an application. First, it infers the type of e to be σ , and then instantiates σ to a function type $\sigma_1 \rightarrow \sigma_2$. The rule then switches to checking mode, by checking the argument e_1 against the expected type σ_1 . Similarly, rule D-I-ANN transitions from inference to checking mode, by simply checking the expression against the provided type annotation σ . Note that annotations must not contain any free term variables (as enforced by $\text{ftv}(\sigma) = \emptyset$), though they may contain term variables, for example in $\lambda x. \lambda y. (y : [x : \exists a. a])$.

Type checking. In checking mode, rules check the expression against an input type. Rule D-C-ABS type-checks an abstraction, by adding $x : \sigma_1$ to the context and checking the lambda body against σ_2 . This rule highlights the advantages of bidirectional type checking: unlike rule D-I-ABS where x is restricted to a monomorphic type, here x can have a polymorphic type σ_1 .

$\Gamma \vdash \sigma \rightsquigarrow_{\forall} \epsilon$	$\Gamma \vdash e : \sigma \rightsquigarrow \sigma_1 \rightarrow \sigma_2$	(Instantiation: under Γ , σ , as the type of e ,) (instantiates to ϵ ($\sigma_1 \rightarrow \sigma_2$))
$\frac{\text{D-INSTF-FORALL}}{\Gamma \vdash [\tau/a]\sigma \rightsquigarrow_{\forall} \epsilon}$	$\frac{\text{D-INSTF-REFL}}{\vdash \Gamma}$	$\frac{\text{D-INST-REFL}}{\vdash \Gamma}$
$\frac{\Gamma \vdash [\tau/a]\sigma \rightsquigarrow_{\forall} \epsilon \quad \Gamma \vdash \tau}{\Gamma \vdash \forall a.\sigma \rightsquigarrow_{\forall} \epsilon}$	$\frac{}{\Gamma \vdash \epsilon \rightsquigarrow_{\forall} \epsilon}$	$\frac{}{\Gamma \vdash e : \sigma_1 \rightarrow \sigma_2 \rightsquigarrow \sigma_1 \rightarrow \sigma_2}$
$\frac{\text{D-INST-FORALL}}{\Gamma \vdash e : [\tau/a]\sigma \rightsquigarrow \sigma_1 \rightarrow \sigma_2}$	$\frac{\text{D-INST-EXISTS}}{\Gamma \vdash e : [[e : \exists a.\epsilon]/a]\epsilon \rightsquigarrow \sigma_1 \rightarrow \sigma_2}$	$\frac{\Gamma \vdash e : \forall a.\sigma \rightsquigarrow \sigma_1 \rightarrow \sigma_2}{\Gamma \vdash e : \exists a.\epsilon \rightsquigarrow \sigma_1 \rightarrow \sigma_2}$

Fig. 2: Declarative instantiation

$\Gamma \vdash e_1 : \sigma_1 <: \sigma_2$	(Subtyping: under Γ , σ_1 , as the type of e_1 , is a subtype of σ_2)
$\frac{\text{D-S-VAR}}{\vdash \Gamma \quad a \in \text{dom}(\Gamma)}$	$\frac{\text{D-S-INT}}{\vdash \Gamma \quad \text{Int} <: \text{Int}}$
$\frac{}{\Gamma \vdash e : a <: a}$	$\frac{}{\Gamma \vdash e : \text{Int} <: \text{Int}}$
$\frac{\text{D-S-ARR}}{\Gamma, x : \sigma'_1 \vdash x : \sigma'_1 <: \sigma_1 \quad \Gamma, x : \sigma'_1 \vdash e : \sigma_2 <: \sigma'_2}$	$\frac{\text{D-S-PROJ}}{\Gamma \vdash [e : \exists a.\epsilon] <: [e : \exists a.\epsilon]}$
$\frac{}{\Gamma \vdash e : \sigma_1 \rightarrow \sigma_2 <: \sigma'_1 \rightarrow \sigma'_2}$	$\frac{\text{D-S-FORALL}}{\Gamma \vdash e : [\tau/a]\sigma <: \epsilon \quad \Gamma \vdash \tau}$
$\frac{\text{D-S-FORALLR}}{\Gamma, a \vdash e : \sigma_1 <: \sigma_2}$	$\frac{\text{D-S-EXISTSL}}{\Gamma \vdash e : [[e : \exists a.\epsilon_1]/a]\epsilon_1 <: \epsilon_2}$
$\frac{}{\Gamma \vdash e : \sigma_1 <: \forall a.\sigma_2}$	$\frac{\text{D-S-EXISTSR}}{\Gamma \vdash e : \rho <: [\tau/a]\epsilon \quad \Gamma \vdash \tau}$
	$\frac{}{\Gamma \vdash e : \exists a.\epsilon_1 <: \epsilon_2}$
	$\frac{}{\Gamma \vdash e : \rho <: \exists a.\epsilon}$

Fig. 3: Declarative subtyping

Rule D-C-FORALL checks an expression e against a polymorphic type $\forall a.\sigma$. The rule skolemizes the universal quantifier, by introducing a skolem type variable a into the context, and proceeding to check e with σ . Rule D-C-EXISTS checks an expression e against an existential type $\exists b.\epsilon$, by checking e against $[\tau/b]\epsilon$, where τ is a well-formed type that is non-deterministically guessed.

Rule D-C-SUB transitions from checking to inference. To check e against a type ρ , the rule infers the type of e to be σ , and requires σ to be a subtype of ρ .

3.2 Instantiation

Fig. 2 presents the two instantiation judgments. Given type σ , $\rightsquigarrow_{\forall}$ instantiates top-level universal quantifiers and returns a ϵ . Rule D-INSTF-FORALL instantiates a with a well-formed type τ , and proceeds recursively with $[\tau/a]\sigma$.

The judgment \rightsquigarrow instantiates a type σ to a function type. Rule D-INST-EXISTS is specific to strong existential types [4]. The rule opens an existential type, by instantiating the existential variable a with the existential projection $[e : \exists a.\epsilon]$.³ This direct opening is why e must be provided as an input to the instantiation judgment. The rule then proceeds recursively with $[[e : \exists a.\epsilon]/a]\epsilon$.

³ Here, the annotation in the projection is needed to avoid ambiguity: it specifies which existential is being opened when e 's type contains nested existentials.

3.3 Subtyping

Fig. 3 presents the declarative subtyping judgment. Given $\Gamma \vdash e : \sigma_1 <: \sigma_2$, we assume $\Gamma \vdash e \Rightarrow \sigma_1$, which holds when referring to subtyping from typing.

The first three rules are straightforward. Rule D-S-ARR handles function subtyping with contravariance: it checks that σ'_1 is a subtype of σ_1 , and that σ_2 is a subtype of σ'_2 . In the premises, we put a fresh variable $x : \sigma'_1$ in the context, which effectively means that if existential projections involving x appear in the derivation tree, they will not be equivalent to any existing types. This is desired, as such an existential projection refers to an argument unknown to this function.

The remaining four rules address subtyping in the presence of universal and existential types. Given the restriction that universal quantifiers can only appear outside of existential ones, these rules are designed to be applied in a specific order: when a universal quantifier appears on the right-hand side of the subtyping relation, rule D-S-FORALLR skolemizes the type variable. On the other hand, if a universal quantifier is present on the left-hand side, rule D-S-FORALLL non-deterministically guesses a well-formed monotype τ to instantiate the type. In the case of an existential type on the left-hand side, rule D-S-EXISTSL directly opens the existential type by substituting the type variable with an existential projection $[e : \exists a. \epsilon]$, similar to the instantiation rule D-INST-EXISTS. This rule is why the expression e must be provided as an input to subtyping. Finally, with an existential type on the right-hand side, rule D-S-EXISTSR instantiates the existential quantifier with a well-formed type τ .

4 Metatheory

This section studies properties of the declarative systems, with a particular focus on how strong existentials interact with the subtyping relation.

Robustness of typing. We show that the typing is robust under substitution:

Lemma 4.1 (Substitution). *Given $\vdash \Gamma_1, x : \sigma', \Gamma_2$ and $\Gamma_1 \vdash e' \Rightarrow \sigma'$,*

1. *If $\Gamma_1, x : \sigma', \Gamma_2 \vdash e \Rightarrow \sigma$, then $\Gamma_1, [e'/x]\Gamma_2 \vdash e \Rightarrow [e'/x]\sigma$.*
2. *If $\Gamma_1, x : \sigma', \Gamma_2 \vdash e \Leftarrow \sigma$, then $\Gamma_1, [e'/x]\Gamma_2 \vdash e \Leftarrow [e'/x]\sigma$.*

Subtyping and instantiation. The subtyping relation is a key extension to the declarative system of EDWL. Notably, the subtyping rule D-S-EXISTSL for existential types closely resembles the instantiation rule D-INST-EXISTS. In both cases, opening an existential type involves substituting the type variable with the corresponding existential projection. Rule D-S-FORALLL corresponds to rule D-INST-FORALL, which non-deterministically guesses a monomorphic type. This similarity is not coincidental; instead, it reflects a connection established by the following lemma:

Lemma 4.2 ($<$: subsumes \rightsquigarrow). *If $\Gamma \vdash e : \sigma \rightsquigarrow \rho$, then $\Gamma \vdash e : \sigma <: \rho$.*

That is, if a type σ can be instantiated to ρ , then σ is a subtype of ρ . This lemma suggests that instantiation can be viewed as a specialized form of subtyping, where the more general type σ is related to its more specific instantiation ρ .

Subtyping reflexivity and transitivity. Conventionally, reflexivity and transitivity of subtyping are expected. Indeed, we prove that subtyping is reflexive:

Lemma 4.3 (Reflexivity of subtyping). *If $\Gamma \vdash \sigma$ and $\text{fv}(e) \subseteq \text{dom}(\Gamma)$, then $\Gamma \vdash e : \sigma <: \sigma$.*

Unfortunately, rule D-S-EXISTS_L makes subtyping non-transitive. Consider:

$$(1) \quad \Gamma \vdash (\lambda x : \text{Int}. 1) : \text{Int} \rightarrow \text{Int} <: \exists a.a \rightarrow \text{Int} \quad (\text{rule D-S-EXISTSR})$$

$$(2) \quad \Gamma \vdash (\lambda x : \text{Int}. 1) : \exists a.a \rightarrow \text{Int} \\ <: [(\lambda x : \text{Int}. 1) : \exists a.a \rightarrow \text{Int}] \rightarrow \text{Int} \quad (\text{rule D-S-EXISTSL})$$

However, the derivation

$$(3) \quad \Gamma \vdash (\lambda x : \text{Int}. 1) : \text{Int} \rightarrow \text{Int} <: [(\lambda x : \text{Int}. 1) : \exists a.a \rightarrow \text{Int}] \rightarrow \text{Int}$$

does *not* hold. In other words, the subtyping relation is non-transitive.⁴

Instead, we establish a limited form of transitivity as follows:

Lemma 4.4 (Transitivity of subtyping (limited)). *If $\Gamma \vdash e : \sigma_1 <: \sigma_2$, $\Gamma \vdash e : \sigma_2 <: \sigma_3$ and σ_2 does not contain any \exists , then we have $\Gamma \vdash e : \sigma_1 <: \sigma_3$.*

That is, transitivity holds when the intermediate type σ_2 is free of existential types. This restriction rules out the problematic example above, indicating that non-transitivity fundamentally arises from the presence of existential types.

Note that this non-transitivity is not a designed feature but arises naturally as a direct consequence of building on EDWL, where existential projections are considered opaque, and our subtyping rules are directly based on instantiations.⁵ To our knowledge, this non-transitivity of subtyping in the presence of strong existentials has not been previously recognized, and our observation offers a valuable contribution.⁶ One may wonder if we could simply add a subtyping transitivity rule to the declarative system. However, challenges remain when designing a corresponding algorithmic type system. More fundamentally, it is questionable whether such a transitive relation would be desirable, since any such relation would require us to look inside existential projections. We leave the development of such a system that restores complete subtyping transitivity—which will likely entail non-trivial design choices—to future work.

Subtyping of weak existentials. In systems with weak existentials, a subtyping constraint $\exists a.\epsilon <: \sigma$ skolemizes a and check $\epsilon <: \sigma$. The key difference is in the use of rule D-SS-EXISTSL, as given on the right, instead of rule D-S-EXISTSL. We write $\Gamma \vdash \sigma_1 \leq \sigma_2$ for such a subtyping relation; this judgment does not need to take expressions. It is known that such a subtyping relation is transitive. We

$$\frac{\Gamma, a \vdash \epsilon_1 \leq \epsilon_2}{\Gamma \vdash \exists a.\epsilon_1 \leq \epsilon_2} \quad \text{D-SS-EXISTSL}$$

⁴ While unusual, non-transitive subtyping has been observed in practical languages like Scala [8, 13].

⁵ In EDWL, this means that (1) $\Gamma \vdash e \Leftarrow \sigma$ and (2) $\Gamma \vdash e : \sigma \rightsquigarrow \sigma_1 \rightarrow \sigma_2$ do not necessarily imply (3) $\Gamma \vdash e \Leftarrow \sigma_1 \rightarrow \sigma_2$.

⁶ Cardelli and Leroy [1, §4.4] discussed a somewhat related issue: whether $\lfloor \text{pack } t, \text{eas } t' \rfloor$ should be equivalent to t . They noted that such an equivalence would lead to a weaker notion of abstraction.

prove that every subtyping relationship valid under the standard weak existential rules also holds within our system, establishing our subtyping relation as an extension of such a subtyping relation:

Lemma 4.5 ($<:$ subsumes \leq). *If $\Gamma \vdash \sigma_1 \leq \sigma_2$, then $\Gamma \vdash e : \sigma_1 <: \sigma_2$.*

5 Algorithmic Type System

This section presents our algorithmic type system. Fig. 4 presents the syntax of types and contexts. The algorithmic system has the same definition of expressions and most types as the declarative system, with monotypes τ extended with unification variables \hat{a} , highlighted in gray.

Algorithmic contexts Δ store, in addition to the variables $x:\sigma$ and the skolem type variables a , unsolved unification variables \hat{a} , which are placeholders for monotypes τ . When a unification variable is solved, the entry \hat{a} in the context is replaced by $\hat{a} = \tau$, which records \hat{a} 's solution. Markers $\blacktriangleright_{\hat{a}}$ are used to indicate the scope of \hat{a} , which we discuss along with the algorithmic rules. Complete contexts Ω have all unification variables solved.

Importantly, an algorithmic context is *ordered*, following Dunfield and Krishnaswami [2] (DK hereafter). Thus, in a context $(\Delta_1, \hat{a} = \tau, \Delta_2)$, τ can only refer to skolem and unification variables that appear in Δ_1 , not Δ_2 . This requirement is important to guarantee that skolem variables do not escape their scope.

Since Δ stores solutions of unification variables, the notation $[\Delta]\sigma$ represents the type obtained by substituting all solutions from Δ into σ . Moreover, the *hole* notation $\Delta[\Delta']$ means that Δ has the form $\Delta_L, \Delta', \Delta_R$. For example, if $\Delta[\hat{a}] = \Delta_L, \hat{a}, \Delta_R$, then $\Delta[\hat{a} = \tau] = \Delta_L, \hat{a} = \tau, \Delta_R$.

5.1 Type Inference and Checking

Fig. 4 presents selected typing rules, with the typing rule for lambdas being the most crucial. We first discuss below the typing rule for lambdas used in DK, before explaining our rule A-I-ABS.

The DK rule for typing functions. The DK algorithm uses rule DK-ABS below for typing lambdas. The rule not only creates a fresh unification variable \hat{a} for x 's type, but it also creates a fresh unification variable \hat{b} for the function's return type. Then, in the premise, the rule checks (\Leftarrow) e against the \hat{b} . This design allows the rule to discard both $x:\hat{a}$ and the context Δ_2 in the final output context, since the return type $\hat{a} \rightarrow \hat{b}$ has all its information in the context before x .

$$\text{DK-ABS} \quad \frac{\Delta, \hat{a}, \hat{b}, x : \hat{a} \vdash e \Leftarrow \hat{b} \dashv \Delta_1, x : \hat{a}, \Delta_2}{\Delta \vdash \lambda x. e \Rightarrow \hat{a} \rightarrow \hat{b} \dashv \Delta_1}$$

However, this approach is no longer feasible when typing strong existentials, where variables like x can appear within types, and, as a result, a function's result type may actually need to refer to x , *before* it is wrapped inside an existential type. Consider the following program:

monomorphic type $\tau := a \mid \hat{a} \mid \text{Int} \mid \tau_1 \rightarrow \tau_2 \mid [e : \exists a. \epsilon]$
algorithmic context $\Delta := \bullet \mid \Delta, x : \sigma \mid \Delta, a \mid \Delta, \hat{a} \mid \Delta, \hat{a} = \tau \mid \Delta, \blacktriangleright_{\hat{a}}$
complete context $\Omega := \bullet \mid \Omega, x : \sigma \mid \Omega, a \mid \Omega, \hat{a} = \tau$

$\Delta_1 \vdash e \Rightarrow \sigma \dashv \Delta_2$	(Inference: under input Δ_1 , e infers type σ , with output Δ_2)
--	--

$$\frac{\begin{array}{c} \text{A-I-ABS} \\ \Delta_1, \hat{a}, x : \hat{a} \vdash e \Rightarrow \sigma \dashv \Delta_2 \\ \epsilon_1 = [\Delta_3, x : \hat{a}, \Delta_4]\epsilon \\ \bar{a} \text{ fresh} \end{array}}{\Delta_1 \vdash \lambda x. e \Rightarrow \hat{a} \rightarrow \exists \bar{a}. \epsilon_2 \dashv \Delta_3, \langle \Delta_4 \rangle}$$

$$\frac{\begin{array}{c} \text{A-I-APP} \\ \Delta_1 \vdash e \Rightarrow \sigma \dashv \Delta_2 \\ \Delta_2 \vdash e : [\Delta_2]\sigma \rightsquigarrow \sigma_1 \rightarrow \sigma_2 \dashv \Delta_3 \\ \Delta_3 \vdash e_1 \Leftarrow \sigma_1 \dashv \Delta_4 \end{array}}{\Delta_1 \vdash e e_1 \Rightarrow \sigma_2 \dashv \Delta_4}$$

$\Delta_1 \vdash e \Leftarrow \sigma \dashv \Delta_2$	(Checking: under input Δ_1 , e checks against type σ , with output Δ_2)
---	---

$$\frac{\begin{array}{c} \text{A-C-ABS} \\ \Delta_1, x : \sigma_1 \vdash e \Leftarrow \sigma_2 \dashv \Delta_2, x : \sigma_1, \Delta_3 \end{array}}{\Delta_1 \vdash \lambda x. e \Leftarrow \sigma_1 \rightarrow \sigma_2 \dashv \Delta_2} \quad \frac{\begin{array}{c} \text{A-C-SUB} \\ \Delta_1 \vdash e \Rightarrow \sigma \dashv \Delta_2 \\ \Delta_2 \vdash e : [\Delta_2]\sigma <: [\Delta_2]\rho \dashv \Delta_3 \end{array}}{\Delta_1 \vdash e \Leftarrow \rho \dashv \Delta_3}$$

Fig. 4: Algorithmic typing

$(\lambda x. f x \text{ True})$ -- assume $(f : \text{Int} \rightarrow \exists a. \text{Bool} \rightarrow a)$

From the application $(f x)$, we know $x : \text{Int}$ and $(f x) : \exists a. \text{Bool} \rightarrow a$. To apply f to True , we instantiate $(f x) : \text{Bool} \rightarrow [f x]$. Therefore, $(f x \text{ True}) : [f x]$. Lastly, we substitute $[f x]$ with a fresh existential variable, ultimately obtaining $(\lambda x. f x \text{ True}) : \text{Int} \rightarrow \exists b. b$. As shown by this example, a function's return type during the intermediate derivation steps (in this case, $(f x \text{ True}) : [f x]$) must be able to refer to x , even though the final type $(\text{Int} \rightarrow \exists b. b)$ does not. Therefore, the approach of placing \hat{b} as the function's return type before x does not work.

Rule DK-ABS also has another limitation: it restricts a function's return type to be monomorphic (since \hat{b} can only be solved with a monomorphic type). This limitation, not discussed in the original presentation of the DK algorithm, has also been adopted without explicit consideration in other higher-rank type systems (e.g. Zhao et al. [26]) that follow the DK approach. This restriction leads to the rejection of seemingly valid programs, especially programs with higher-rank types. For example, the following function fails to type-check in DK:

$\lambda x. g$ -- rejected; assume $g : (\forall a. a \rightarrow a) \rightarrow \text{Bool}$

even though the function simply wraps g inside a lambda abstraction where the variable x is not even used. This is because rule DK-ABS enforces a monomorphic return type by checking g against a monomorphic type. However, g 's type cannot be instantiated to a monotype, leading to a type error.

Our rule for typing functions. In our system, we instead use a typing rule that infers the type of the function body, rather than checking it against a

monomorphic type. Such a design immediately fixes both limitations, and we show that the rule works effectively in the presence of existential types and a form of context drop.

Specifically, rule A-I-ABS uses a fresh unification variable \hat{a} as the type of x , and adds \hat{a} and $x : \hat{a}$ to the context to infer the type of e to be σ . It then instantiates $(\rightsquigarrow_{\forall}) \sigma$ to ϵ . The output context must contain $x : \hat{a}$, thus we can write it as $\Delta_3, x : \hat{a}, \Delta_4$. We obtain $\epsilon_1 = [\Delta_3, x : \hat{a}, \Delta_4]\epsilon$. As in the declarative system, we replace ϵ_1 's projections that mention x with fresh variables.

We discard x in the output context. However, we cannot discard Δ_4 , since the return type may refer to unsolved unification variables in Δ_4 (while solved unification variables have already been applied). Thus, we use $\langle \Delta_4 \rangle$ to denote unsolved unification variables in Δ_4 , producing the output context $\Delta_3, \langle \Delta_4 \rangle$.⁷

The rest of the rules. Rule A-I-APP first infers the type of e , getting σ . The rule then uses the instantiation judgment (\rightsquigarrow) to instantiate $[\Delta_2]\sigma$ to a function type $\sigma_1 \rightarrow \sigma_2$. Lastly, it checks the argument e_1 against the expected type.

Rule A-C-ABS checks the function against a given type. In this case, we throw away the trailing context after x . Rule A-C-SUB first infers the type of e , getting σ . It then checks that $[\Delta_2]\sigma$ is a subtype of $[\Delta_2]\rho$, where both types contain no solved unification variables after context application.

5.2 Instantiation and Subtyping

Fig. 5 and Fig. 6 present rules for algorithmic instantiation and subtyping.

Instantiation. The instantiation rules $(\rightsquigarrow_{\forall})$ instantiate top-level universal quantifiers: rule A-INSTF-FORALL creates a new unification variable \hat{a} when instantiating a polymorphic type. Instantiation does not require the input context to be fully substituted, since only top-level quantifiers matter.

The \rightsquigarrow instantiation judgment instantiates both top-level universal (rule A-INST-FORALL) and existential quantifiers (rule A-INST-EXISTS), and matches the result to a function. Rule A-INST-REFL applies if the type is already a function, by simply returning the type. Otherwise, the input type must be an unsolved unification variable \hat{a} , and it must be solved with a function type. Rule A-INST-UVAR solves \hat{a} by setting $\hat{a} = \hat{a}_1 \rightarrow \hat{a}_2$, where \hat{a}_1 and \hat{a}_2 are fresh type variables inserted before \hat{a} , so that they can be used in \hat{a} 's solution, and returns $\hat{a}_1 \rightarrow \hat{a}_2$.

Subtyping. The subtyping judgment $\Delta_1 \vdash e : \sigma_1 <: \sigma_2 \dashv \Delta_2$ checks that σ_1 is a subtype of σ_2 . Rule A-S-MONO handles monotypes by unifying the types (§5.3).

When comparing a unification variable with a function type (rule A-S-ARRL), we know that \hat{a} 's solution must be a function, and thus we solve \hat{a} by $\hat{a}_1 \rightarrow \hat{a}_2$, where both \hat{a}_1 and \hat{a}_2 are fresh unification variables inserted right before \hat{a} . We

⁷ The output context can be made more precise, by replacing $\langle \Delta_4 \rangle$ with the intersection of $\langle \Delta_4 \rangle$ and unification variables in the output type.

$\Delta_1 \vdash \sigma \rightsquigarrow_{\forall} \epsilon \dashv \Delta_2$	$\Delta_1 \vdash e : \sigma \rightsquigarrow \sigma_1 \rightarrow \sigma_2 \dashv \Delta_2$
(Instantiation: under input Δ_1 , σ , as type of e , instantiates to ϵ ($\sigma_1 \rightarrow \sigma_2$),) with output Δ_2)	
A-INSTF-FORALL $\Delta_1, \hat{a} \vdash [\hat{a}/a]\sigma \rightsquigarrow_{\forall} \rho \dashv \Delta_2$	A-INSTF-REFL $\Delta \vdash \epsilon \rightsquigarrow_{\forall} \epsilon \dashv \Delta$
$\Delta_1 \vdash \forall a. \sigma \rightsquigarrow_{\forall} \rho \dashv \Delta_2$	$\Delta_1 \vdash e : [\hat{a}/a]\sigma \rightsquigarrow \sigma_1 \rightarrow \sigma_2 \dashv \Delta_2$
A-INST-EXISTS $\Delta_1 \vdash e : [e : \exists a. \epsilon /a]\epsilon \rightsquigarrow \sigma_1 \rightarrow \sigma_2 \dashv \Delta_2$	A-INST-REFL $\Delta \vdash e : \sigma_1 \rightarrow \sigma_2 \rightsquigarrow \sigma_1 \rightarrow \sigma_2 \dashv \Delta$
$\Delta_1 \vdash e : \exists a. \epsilon \rightsquigarrow \sigma_1 \rightarrow \sigma_2 \dashv \Delta_2$	
A-INST-UVAR $\Delta[\hat{a}] \vdash e : \hat{a} \rightsquigarrow \hat{a}_1 \rightarrow \hat{a}_2 \dashv \Delta[\hat{a}_1, \hat{a}_2, \hat{a} = \hat{a}_1 \rightarrow \hat{a}_2]$	

Fig. 5: Algorithmic instantiation

$\Delta_1 \vdash e : \sigma_1 <: \sigma_2 \dashv \Delta_2$	Algorithmic subtyping: under input Δ_1 , σ_1 , (as type of e , is a subtype of σ_2 , with output Δ_2)
A-S-MONO $\Delta_1 \vdash \tau_1 \approx \tau_2 \dashv \Delta_2$	A-S-ARR $\Delta_1, x : \sigma'_1 \vdash x : \sigma'_1 <: \sigma_1 \dashv \Delta_2$
$\Delta_1 \vdash e : \tau_1 <: \tau_2 \dashv \Delta_2$	$\Delta_2 \vdash e x : [\Delta_2]\sigma_2 <: [\Delta_2]\sigma'_2 \dashv \Delta_3, x : \sigma'_1, \Delta_4$
A-S-ARRL $\Delta_1[\hat{a}_2, \hat{a}_1, \hat{a} = \hat{a}_1 \rightarrow \hat{a}_2], x : \sigma_1 \vdash x : \sigma_1 <: \hat{a}_1 \dashv \Delta_2$	$\Delta_2 \vdash e x : \hat{a}_2 <: [\Delta_2]\sigma_2 \dashv \Delta_3, x : \sigma_1, \Delta_4$
$\Delta_1[\hat{a}] \vdash e : \hat{a} <: \sigma_1 \rightarrow \sigma_2 \dashv \Delta_3$	
A-S-FORALL $\Delta_1, \blacktriangleright_{\hat{a}}, \hat{a} \vdash e : [\hat{a}/a]\sigma <: \epsilon \dashv \Delta_2, \blacktriangleright_{\hat{a}}, \Delta_3$	A-S-FORALLR $\Delta_1, a \vdash e : \sigma_1 <: \sigma_2 \dashv \Delta_2, a, \Delta_3$
$\Delta_1 \vdash e : \forall a. \sigma <: \epsilon \dashv \Delta_2$	$\Delta_1 \vdash e : \sigma_1 <: \forall a. \sigma_2 \dashv \Delta_2$
A-S-EXISTSL $\Delta_1 \vdash e : [e : \exists a. \epsilon_1 /a]\epsilon_1 <: \epsilon_2 \dashv \Delta_2$	A-S-EXISTSR $\Delta_1, \blacktriangleright_{\hat{a}}, \hat{a} \vdash e : \rho <: [\hat{a}/a]\epsilon \dashv \Delta_2, \blacktriangleright_{\hat{a}}, \Delta_3$
$\Delta_1 \vdash e : \exists a. \epsilon_1 <: \epsilon_2 \dashv \Delta_2$	$\Delta_1 \vdash e : \rho <: \exists a. \epsilon \dashv \Delta_2$

Fig. 6: Algorithmic subtyping

then decompose $\sigma_1 \rightarrow \sigma_2$, and proceed with $\sigma_1 <: \hat{a}_1$, and then $\hat{a}_2 <: [\Delta_2]\sigma_2$, where any solutions found by Δ_2 are substituted into σ_2 .

Rule A-S-FORALLL instantiates the polymorphic type with a fresh unification variable. Notably, we insert a marker as the scope for \hat{a} . When we get out of the scope of \hat{a} , we remove the marker and the trailing context Δ_3 . The marker makes the scope of \hat{a} more precise, as additional unification variables can be inserted before \hat{a} (e.g. during instantiation). Rule A-S-FORALLR skolemizes the type variable a . Similarly, when we get out of the scope of a , we remove a and the trailing context.

Rule A-S-EXISTSL simply opens the existential type. Rule A-S-EXISTSR creates a unification variable for the existential type and a marker for its scope, and removes the marker and the trailing context in the conclusion.

5.3 Unification and Promotion

Fig. 7 presents selected rules for unification. Most of these rules, including rule A-UE-ANN, A-UT-EXISTS, and A-UT-PROJ, recursively unify subcomponents.

To explain rules A-UT-UVARL and A-UT-UVARR, let us first discuss how unification works under ordered contexts. This will help us understand the *promotion* process, as in Fig. 8.

The DK rule for unification. The sequential order in the algorithmic context implies that unification needs to consider the order of variables. For example, consider $\hat{a}, \hat{b} \vdash \hat{a} \approx \hat{b} \rightarrow \text{Int}$. In this case, we cannot simply solve \hat{a} with $\hat{b} \rightarrow \text{Int}$, since \hat{b} is not in the scope of \hat{a} . Instead, we may solve \hat{a} by $\hat{a}_1 \rightarrow \hat{a}_2$ with fresh \hat{a}_1, \hat{a}_2 put before \hat{a} , and recursively unify $\hat{a}_1 \approx \hat{b}$ and $\hat{a}_2 \approx \text{Int}$, respectively. This produces the final context $\hat{a}_1, \hat{a}_2 = \text{Int}, \hat{a} = \hat{a}_1 \rightarrow \hat{a}_2, \hat{b} = \hat{a}_1$.

To enforce this ordering, DK's unification rules decompose the type being unified with a unification variable, until rule DK-UNIF (on the right) applies. This rule checks that the type τ is well-formed under the context before \hat{a} , and thus all variables inside τ are in the scope of the unification variable, so we can solve $\hat{a} = \tau$.

Applying the same idea of examining each sub-component of a type, we would expect a rule like the one on the right for projections. A projection $[e : \exists a. \epsilon]$ allows an expression to appear inside types. Since unification variables can appear inside $[e : \exists a. \epsilon]$, we must ensure that when unifying \hat{a} with $[e : \exists a. \epsilon]$, all variables in $[e : \exists a. \epsilon]$ appear to the left of \hat{a} in the context. Thus, the rule sets $\hat{a} = [\hat{b}]$, and unifies \hat{b} with e .

However, we have now introduced \hat{b} to stand for *an expression*, rather than a type! Allowing unification variables to stand for expressions would significantly complicate almost all key judgments, including subtyping, well-formedness of algorithmic contexts, as well as the *context extension* notion (§6).

Our rule for unification. To avoid this complexity, we observe that *the relative order between unification variables does not matter for solving a constraint*⁸. Thus we adapt the process of *promotion* first applied to ordered contexts by Xie et al. [25], who employed promotion to handle unification variables appearing in *kinds* in a *higher-kinded* type system. We apply promotion to expressions.

Specifically, consider unifying \hat{a} with a type τ . Instead of decomposing τ down to its primitive types, we directly promote τ . This involves promoting all of its unification variables (but not skolem variables!) that are not in the scope of \hat{a} , by creating fresh in-scope unification variables to the left of \hat{a} in the context, and solving the out-of-scope ones with these fresh in-scope ones. Consider

$$\hat{a}, \hat{b} \vdash \hat{a} \approx [\lambda x : \hat{b} \rightarrow \text{Int}. e]$$

$$\frac{\text{DK-UNIF}}{\Delta_1, \hat{a}, \Delta_2 \vdash \hat{a} \approx \tau \dashv \Delta_1, \hat{a} = \tau, \Delta_2}$$

$$\frac{\text{PROJRBROKEN}}{\Delta[\hat{b}, \hat{a} = [\hat{b}]] \vdash \hat{b} \approx e \dashv \Delta_1}{\Delta[\hat{a}] \vdash \hat{a} <: [e] \dashv \Delta_2}$$

⁸ The relative order between unification variables and skolem variables still matters.

$$\boxed{\Delta_1 \vdash e_1 \approx e_2 \dashv \Delta_2} \quad \boxed{\Delta_1 \vdash \sigma_1 \approx \sigma_2 \dashv \Delta_2}$$

(Unification: under input Δ_1 , unifying e_1 (or σ_1) and e_2 (or σ_2) returns output Δ_2)

$$\begin{array}{c} \text{A-UE-ANN} \\ \frac{\Delta_1 \vdash e_1 \approx e_2 \dashv \Delta_2 \quad \Delta_2 \vdash [\Delta_2]\sigma_1 \approx [\Delta_2]\sigma_2 \dashv \Delta_3}{\Delta_1 \vdash (e_1 : \sigma_1) \approx (e_2 : \sigma_2) \dashv \Delta_3} \end{array} \quad \begin{array}{c} \text{A-UT-REFL} \\ \frac{}{\Delta \vdash \sigma \approx \sigma \dashv \Delta} \end{array} \quad \begin{array}{c} \text{A-UT-PROJ} \\ \frac{\Delta_1 \vdash e_1 \approx e_2 \dashv \Delta_2 \quad \Delta_2 \vdash [\Delta_2](\exists a.\epsilon_1) \approx [\Delta_2](\exists a.\epsilon_2) \dashv \Delta_3}{\Delta_1 \vdash [e_1 : \exists a.\epsilon_1] \approx [e_2 : \exists a.\epsilon_2] \dashv \Delta_2} \end{array}$$

$$\begin{array}{c} \text{A-UT-EXISTS} \\ \frac{\Delta_1, a \vdash \epsilon_1 \approx \epsilon_2 \dashv \Delta_2, a, \Delta_3}{\Delta_1 \vdash \exists a.\epsilon_1 \approx \exists a.\epsilon_2 \dashv \Delta_2} \end{array} \quad \begin{array}{c} \text{A-UT-UVARL} \\ \frac{\Delta_1 \vdash_{\hat{a}} \tau \rightsquigarrow \tau' \dashv \Delta_2}{\Delta_1[\hat{a}] \vdash_{\hat{a}} \tau \rightsquigarrow \tau' \dashv \Delta_2[\hat{a} = \tau']} \end{array} \quad \begin{array}{c} \text{A-UT-UVARR} \\ \frac{\Delta_1 \vdash_{\hat{a}} \tau \rightsquigarrow \tau' \dashv \Delta_2}{\Delta_1[\hat{a}] \vdash \tau \approx \hat{a} \dashv \Delta_2[\hat{a} = \tau']} \end{array}$$

Fig. 7: Unification

$$\boxed{\Delta_1 \vdash_{\hat{a}} e_1 \rightsquigarrow e_2 \dashv \Delta_2} \quad \boxed{\Delta_1 \vdash_{\hat{a}} \sigma_1 \rightsquigarrow \sigma_2 \dashv \Delta_2}$$

(Promotion: under input Δ_1 , promoting e_1 (or σ_1) to e_2 (or σ_2) returns output Δ_2)

$$\begin{array}{c} \text{A-PRT-UVARL} \\ \frac{\vdash \Delta \quad \hat{b} \neq \hat{a}}{\Delta[\hat{b}][\hat{a}] \vdash_{\hat{a}} \hat{b} \rightsquigarrow \hat{b} \dashv \Delta[\hat{b}][\hat{a}]} \end{array} \quad \begin{array}{c} \text{A-PRT-UVARR} \\ \frac{\vdash \Delta \quad \hat{b} \neq \hat{a}}{\Delta[\hat{a}][\hat{b}] \vdash_{\hat{a}} \hat{b} \rightsquigarrow \hat{b}_1 \dashv \Delta[\hat{b}_1, \hat{a}][\hat{b} = \hat{b}_1]} \end{array}$$

$$\begin{array}{c} \text{A-PRT-PROJ} \\ \frac{\text{A-PRT-VAR} \quad \vdash \Delta}{\Delta[a][\hat{a}] \vdash_{\hat{a}} a \rightsquigarrow a \dashv \Delta[a][\hat{a}]} \end{array} \quad \begin{array}{c} \text{A-PRE-ANN} \\ \frac{\Delta_1 \vdash_{\hat{a}} e_1 \rightsquigarrow e_2 \dashv \Delta_2 \quad \Delta_2 \vdash_{\hat{a}} [\Delta_2](\exists a.\epsilon_1) \rightsquigarrow \exists a.\epsilon_2 \dashv \Delta_3}{\Delta_1 \vdash_{\hat{a}} [e_1 : \exists a.\epsilon_1] \rightsquigarrow [e_2 : \exists a.\epsilon_2] \dashv \Delta_3} \end{array}$$

$$\begin{array}{c} \text{A-PRT-EXISTS} \\ \frac{\Delta_1[a, \hat{a}] \vdash_{\hat{a}} \epsilon_1 \rightsquigarrow \epsilon_2 \dashv \Delta_2, a, \Delta_3}{\Delta_1[\hat{a}] \vdash_{\hat{a}} \exists a.\epsilon_1 \rightsquigarrow \exists a.\epsilon_2 \dashv \Delta_2, \Delta_3} \end{array} \quad \begin{array}{c} \text{A-PRT-UVARL} \\ \frac{\Delta_1 \vdash_{\hat{a}} e_1 \rightsquigarrow e_2 \dashv \Delta_2 \quad \Delta_2 \vdash_{\hat{a}} [\Delta_2]\sigma_1 \rightsquigarrow \sigma_2 \dashv \Delta_3}{\Delta_1 \vdash_{\hat{a}} (e_1 : \sigma_1) \rightsquigarrow (e_2 : \sigma_2) \dashv \Delta_3} \end{array}$$

Fig. 8: Promotion

We first promote $\lambda x : \hat{b} \rightarrow \text{Int. } e$, resulting in an updated context and constraint
 $\hat{b}_1, \hat{a}, \hat{b} = \hat{b}_1 \vdash \hat{a} \approx [\lambda x : \hat{b}_1 \rightarrow \text{Int. } e]$

Now we can solve \hat{a} with the existential projection, producing the output context

$$\hat{b}_1, \hat{a} = [\lambda x : \hat{b}_1 \rightarrow \text{Int. } e], \hat{b} = \hat{b}_1$$

This way, we handle expressions using promotion, and eliminate the need for unification variables to stand for expressions. We also extend promotion to deal with binders, as we may need to promote, e.g. lambdas, when promoting expressions, in which case the bound variables will be introduced before the unification variable, and then removed afterwards.

Rules A-UT-UVARL and A-UT-UVARR unify a unification variable with a monotype. We cannot simply solve $\hat{a} = \tau$, since τ may not be a well-formed solution for \hat{a} . Instead, these rules promote τ to τ' , solving $\hat{a} = \tau'$.

Promotion. Fig. 8 presents selected promotion rules for expressions and types. The key idea of promotion is to “move” the unification variables that are out of scope of \hat{a} to be before \hat{a} , by solving those variables by fresh ones appearing before \hat{a} . If promotion succeeds, the output expression (or type) will be well-defined in the output context before \hat{a} .

Rule A-PRT-UVARL and rule A-PRT-UVARR are of particular interest. If the unification variable \hat{b} to be promoted appears before \hat{a} in the context, then rule A-PRT-UVARL simply returns \hat{b} . Otherwise, if \hat{b} appears later than \hat{a} , rule A-PRT-UVARR creates a fresh \hat{b}_1 before \hat{a} , setting $\hat{b} = \hat{b}_1$. Note that we cannot promote \hat{a} itself (i.e. the occurs-check).

Notably, a skolem variable cannot be promoted. Thus, rule A-PRT-VAR applies only when a appears before \hat{a} . Promoting an out-of-scope skolem variable simply leads to type errors. When dealing with binders, such as promoting an existential type, rule A-PRT-EXISTS puts the skolem variable a to be before \hat{a} , by putting it at the beginning of the context, so that promoting a will not fail. The rule then removes the variable in the output context.

6 Soundness

This section proves that the algorithmic type system is sound with respect to the declarative type system; we discuss completeness in §7. For better clarity, we use **blue** for declarative definitions and judgments, and **orange** for algorithmic ones in the remaining sections, except when the meaning is clear from the context.

Context extension. First, we define *context extension* in Fig. 9. The judgment $\Delta_1 \rightarrow \Delta_2$ reads “context Δ_1 is extended to Δ_2 ”. Context extension expresses a form of information increase: if $\Delta_1 \rightarrow \Delta_2$, then Δ_2 may include additional unification variables, and/or more information about existing unification variables.

All algorithmic judgments produce extended contexts. In particular:

Lemma 6.1 (Context extension). *If $\Delta_1 \vdash e \Rightarrow \sigma \dashv \Delta_2$ or $\Delta_1 \vdash e \Leftarrow \sigma \dashv \Delta_2$, then $\Delta_1 \rightarrow \Delta_2$.*

Recall that Ω (Fig. 4) is a complete context which has all unification variables solved. Therefore, $\Delta_2 \rightarrow \Omega$ solves all unification variables in Δ_2 . Applying a complete context to a type that is well-formed under the context, i.e., $[\Omega]\sigma$, produces declarative types.

Given $\Delta \rightarrow \Omega$, Fig. 10 defines the notion of context application. Intuitively, since Ω contains no unsolved unification variables, applying Ω to Δ (i.e., $[\Omega]\Delta$) produces a context with no unification variables, i.e., a declarative context.

Soundness of typing. We now prove soundness of promotion and unification for expressions and types.

Lemma 6.2 (Soundness of promotion).

- a) *If $\Delta_1[\hat{a}] \vdash_{\hat{a}} \sigma_1 \rightsquigarrow \sigma_2 \dashv \Delta_2$, then $[\Delta_2]\sigma_1 = [\Delta_2]\sigma_2$ and $\Delta_2 = \Delta_3, \hat{a}, \Delta_4$ where $\Delta_3 \vdash \sigma_2$.*
- b) *If $\Delta_1[\hat{a}] \vdash_{\hat{a}} e_1 \rightsquigarrow e_2 \dashv \Delta_2$, then $[\Delta_2]e_1 = [\Delta_2]e_2$ and $\Delta_2 = \Delta_3, \hat{a}, \Delta_4$ where $\text{fv}(e_2) \subseteq \text{dom}(\Delta_3)$.*

$$\boxed{\Delta_1 \longrightarrow \Delta_2} \quad (\text{Context extension: } \Delta_1 \text{ is extended by } \Delta_2)$$

$$\begin{array}{c}
\frac{}{\bullet \longrightarrow \bullet} \\
\frac{\Delta_1 \longrightarrow \Delta_2 \quad [\Delta_2]\sigma_1 = [\Delta_2]\sigma_2}{\Delta_1, x : \sigma_1 \longrightarrow \Delta_2, x : \sigma_2} \\
\frac{\Delta_1 \longrightarrow \Delta_2}{\Delta_1, \hat{a} \longrightarrow \Delta_2, \hat{a}} \quad \frac{\Delta_1 \longrightarrow \Delta_2}{\Delta_1, \hat{a} \longrightarrow \Delta_2, \hat{a} = \tau} \quad \frac{\Delta_1 \longrightarrow \Delta_2 \quad [\Delta_2]\tau_1 = [\Delta_2]\tau_2}{\Delta_1, \hat{a} = \tau_1 \longrightarrow \Delta_2, \hat{a} = \tau_2} \\
\frac{\Delta_1 \longrightarrow \Delta_2}{\Delta_1 \longrightarrow \Delta_2, \hat{a}} \quad \frac{\Delta_1 \longrightarrow \Delta_2}{\Delta_1 \longrightarrow \Delta_2, \hat{a} = \tau} \quad \frac{\Delta_1 \longrightarrow \Delta_2}{\Delta_1, \blacktriangleright_{\hat{a}} \longrightarrow \Delta_2, \blacktriangleright_{\hat{a}}}
\end{array}$$

Fig. 9: Context extension

$$\boxed{[\Omega]\Delta} \quad (\text{Context application: complete context } \Omega \text{ applied to context } \Delta)$$

$$\begin{array}{lll}
[\bullet]\bullet & = \bullet & [\Omega, x : \sigma](\Delta, x : \sigma') = [\Omega]\Delta, x : [\Omega]\sigma \text{ if } [\Omega]\sigma = [\Omega]\sigma' \\
[\Omega, a](\Delta, a) & = [\Omega]\Delta, a & [\Omega, \hat{a} = \tau](\Delta, \hat{a} = \tau') = [\Omega]\Delta \text{ if } [\Omega]\tau = [\Omega]\tau' \\
[\Omega, \hat{a} = \tau](\Delta, \hat{a}) & = [\Omega]\Delta & [\Omega, \hat{a} = \tau]\Delta = [\Omega]\Delta \text{ if } \hat{a} \notin \text{dom}(\Delta)
\end{array}$$

Fig. 10: Context application

This lemma states that after applying the output context, the output type remains identical to the input type after applying the output context, and this output type is well-formed under the output context before \hat{a} . Similarly, for expressions, applying the output context yields identical expressions, with all free term, type, and unification variables of the output expression bound in the output context before \hat{a} . With that, we prove soundness of unification:

Lemma 6.3 (Soundness of unification).

- a) If $\Delta_1 \vdash \sigma_1 \approx \sigma_2 \dashv \Delta_2$, then $[\Delta_2]\sigma_1 = [\Delta_2]\sigma_2$.
- b) If $\Delta_1 \vdash e_1 \approx e_2 \dashv \Delta_2$, then $[\Delta_2]e_1 = [\Delta_2]e_2$.

We then prove soundness of subtyping.

Lemma 6.4 (Soundness of subtyping). Given $\Delta_2 \longrightarrow \Omega$, if $\Delta_1 \vdash e_1 : \sigma_1 <: \sigma_2 \dashv \Delta_2$ where $[\Delta_1]\sigma_1 = \sigma_1$, $[\Delta_1]\sigma_2 = \sigma_2$, then $[\Omega]\Delta_2 \vdash e_1 : [\Omega]\sigma_1 <: [\Omega]\sigma_2$.

In this statement, soundness is established by applying the complete context Ω to both contexts and types, and the result is thus the corresponding declarative subtyping judgment. The statement of the soundness of instantiation is similar.

With that, we prove soundness of algorithmic typing:

Theorem 6.1 (Soundness of typing). Given $\Delta_2 \longrightarrow \Omega$,

- (inference) If $\Delta_1 \vdash e \Rightarrow \sigma \dashv \Delta_2$, then $[\Omega]\Delta_2 \vdash e \Rightarrow [\Omega]\sigma$.
- (checking) If $\Delta_1 \vdash e \Leftarrow \sigma \dashv \Delta_2$, then $[\Omega]\Delta_2 \vdash e \Leftarrow [\Omega]\sigma$.

7 Towards Completeness of Typing

Next, we move to completeness. First, we prove completeness of instantiation and subtyping:

Theorem 7.1 (Completeness of instantiation). If $[\Omega_1]\Delta_1 \vdash e : [\Omega_1]\sigma \rightsquigarrow \sigma_1 \rightarrow \sigma_2$ where $\Delta_1 \rightarrow \Omega_1$, then there exist Δ_2 , Ω_2 , σ'_1 , and σ'_2 such that $\Delta_2 \rightarrow \Omega_2$, $\Omega_1 \rightarrow \Omega_2$, $\sigma_1 = [\Omega_2]\sigma'_1$, $\sigma_2 = [\Omega_2]\sigma'_2$, and $\Delta_1 \vdash e : [\Delta_1]\sigma \rightsquigarrow \sigma'_1 \rightarrow \sigma'_2 \dashv \Delta_2$.

Theorem 7.2 (Completeness of subtyping). If $[\Omega_1]\Delta_1 \vdash e_1 : [\Omega_1]\sigma_1 <: [\Omega_1]\sigma_2$, where $\Delta_1 \rightarrow \Omega_1$, $\Delta_1 \vdash \sigma_1$, and $\Delta_1 \vdash \sigma_2$, then there exist Δ_2 and Ω_2 such that $\Delta_2 \rightarrow \Omega_2$, $\Omega_1 \rightarrow \Omega_2$, and $\Delta_1 \vdash e_1 : [\Delta_1]\sigma_1 <: [\Delta_1]\sigma_2 \dashv \Delta_2$.

Note that Ω_1 may not contain all unification variables in Δ_2 ; as an example, Ω_1 may contain $\hat{a} = \text{Int}$, while Δ_2 contains $\hat{b} = \text{Int}$, $\hat{a} = \hat{b}$. Thus, we need to show that there exists a Ω_2 such that both Ω_1 and Δ_2 extend to Ω_2 .

Unfortunately, establishing completeness of typing proves difficult. We demonstrate this with two illustrative examples below.

Example 7.1 (“Unnecessary” existential projections). Consider the expression $\lambda x : \text{Int}. \lambda y. y$. In this case, y ’s type is not constrained. Thus, the declarative system can non-deterministically pick a type for y . If the declarative system picks the type $[x : \exists a.a]$ (which we write as $[x]$ below for brevity), we can derive the type $\text{Int} \rightarrow \exists a.a \rightarrow a$ following the derivation on the left:

$$\frac{\begin{array}{c} \text{D-I-ABS} \\ x : \text{Int} \vdash \lambda y. y \Rightarrow [x] \rightarrow [x] \\ \rho' = [a/[x]]([x] \rightarrow [x]) = a \rightarrow a \end{array}}{\bullet \vdash \lambda x : \text{Int}. \lambda y. y \Rightarrow \text{Int} \rightarrow \exists a.a \rightarrow a} \quad \frac{\begin{array}{c} \text{A-I-ABS} \\ x : \text{Int}, \hat{a}, y : \hat{a} \vdash y \Rightarrow \hat{a} \dashv x : \text{Int}, \hat{a}, y : \hat{a} \\ x : \text{Int} \vdash \lambda y. y \Rightarrow \hat{a} \rightarrow \hat{a} \dashv x : \text{Int} \end{array}}{\bullet \vdash \lambda x : \text{Int}. \lambda y. y \Rightarrow \text{Int} \rightarrow \hat{a} \rightarrow \hat{a}}$$

However, the algorithm picks a unification variable for y , and derives $\text{Int} \rightarrow \hat{a} \rightarrow \hat{a}$, following the derivation on the right. To establish completeness, we must relate the declarative type $\text{Int} \rightarrow \exists a.a \rightarrow a$ and the algorithmic type $\text{Int} \rightarrow \hat{a} \rightarrow \hat{a}$.

Worse, expanding on this example, we show that type inference is *incomplete*.

Example 7.2 (Instantiation with “unnecessary” existential projections). Consider the following expression:

$$e \triangleq \lambda w : \text{Int}. \lambda z : [w : \exists a.a]. \lambda x : \text{Int}. \lambda y. y$$

The declarative system may derive the type $\text{Int} \rightarrow \exists a.a \rightarrow \text{Int} \rightarrow \exists b.b \rightarrow b$ as above by picking the type $[x : \exists a.a]$ for y , whereas the algorithmic system derives $\text{Int} \rightarrow \exists a.a \rightarrow \text{Int} \rightarrow \hat{a} \rightarrow \hat{a}$. Now consider $e 1 x$ for some x , which requires instantiating $e 1$ ’s type in the declarative and algorithmic systems respectively:

$$\begin{aligned} &\bullet \vdash e 1 : \exists a.a \rightarrow \text{Int} \rightarrow \exists b.b \rightarrow b \\ &\rightsquigarrow [e 1 : \exists a.a \rightarrow \text{Int} \rightarrow \exists b.b \rightarrow b] \rightarrow \text{Int} \rightarrow \exists b.b \rightarrow b \\ &\bullet \vdash e 1 : \exists a.a \rightarrow \text{Int} \rightarrow \hat{a} \rightarrow \hat{a} \rightsquigarrow [e 1 : \exists a.a \rightarrow \text{Int} \rightarrow \hat{a} \rightarrow \hat{a}] \rightarrow \text{Int} \rightarrow \hat{a} \rightarrow \hat{a} \end{aligned}$$

However, these two instantiated types are not comparable! That is, if we would like to relate the two types, we would need to look inside the existential projections, but existential projections are opaque. Now assuming a context with $x : [e 1 : \exists a.a \rightarrow \text{Int} \rightarrow \exists b.b \rightarrow b]$, we have:

$$x : [e 1 : \exists a.a \rightarrow \text{Int} \rightarrow \exists b.b \rightarrow b] \vdash e 1 x \Rightarrow \text{Int} \rightarrow \exists b.b \rightarrow b \text{ — accepted}$$

$$x : [e 1 : \exists a.a \rightarrow \text{Int} \rightarrow \exists b.b \rightarrow b] \vdash e 1 x \Rightarrow \text{— rejected}$$

since unifying $[e : \exists a.a \rightarrow \text{Int} \rightarrow \exists b.b \rightarrow b]$ with $[e : \exists a.a \rightarrow \text{Int} \rightarrow \hat{a} \rightarrow \hat{a}]$ fails.

We have provided a program ($\lambda x : [e_1 : \exists a.a \rightarrow \text{Int} \rightarrow \exists b.b \rightarrow b]. e_1 x$) that is accepted by the declarative system but rejected by the algorithmic one. Thus, we can conclude that the type inference algorithm is incomplete. Note that none of these examples use higher-rank polymorphism. Indeed, the incompleteness result, along with all examples examined in this section, would apply similarly to a DK-style algorithmic system corresponding directly to the declarative system in EDWL. Thus, we report the first type inference incompleteness result for an algorithm with EDWL-style strong existentials.

This incompleteness stems primarily from the interplay of two factors: the non-deterministic type selection for lambda-bound variables in the declarative system, and the opaque nature of existential projections. Preserving the opacity of existential projections is crucial for data abstraction. Therefore, to restore completeness, we consider restricting the non-determinism of the declarative system. We consider two variants of the declarative system: one with restricted existential inference (§7.1), and the other with restricted existential instantiation (§7.2). In the former, the system is more restrictive, but completeness can be stated in a straightforward way, while latter system requires additional sophisticated mechanisms to state and prove completeness.

7.1 Completeness with Restricted Existential Inference

A key observation is that incompleteness arises when the declarative system guesses “unnecessary” existential projections, leading to additional existential quantification in the inferred type. Thus, we consider a declarative system which disallows unnecessary existential quantification. We first define a predicate \min_{\exists} over declarative typing, where $|\exists(\sigma)|$ denotes the number of existential quantifiers (\exists) in σ , excluding those occurring within existential projections.

Definition 7.1. $\min_{\exists}(\Gamma \vdash e \Rightarrow \sigma) \triangleq \forall \sigma'. \Gamma \vdash e \Rightarrow \sigma' \implies |\exists(\sigma)| \leq |\exists(\sigma')|$.

This predicate enforces that the number of \exists s in the type σ is less than or equal to the number of \exists s in any possible inferred type of e .

We replace rule D-I-ABS (Fig. 1) with the rule D-I-ABS1 on the right, where \Rightarrow_1 denotes the type inference relation with rule D-I-ABS1 replacing rule D-I-ABS. Note that the last premise refers to the original judgement \Rightarrow , since we want to compare the type to all possible inferred types without restriction. The last premise ensures that the number of \exists s in the inferred type $\tau \rightarrow \exists \bar{a}. \epsilon'$ is less than or equal to the number of \exists s in any possible inferred type of $\lambda x. e$. In particular, this ensures that the number of elements in $[\epsilon]_x$ is minimal when compared to all other instances of rule D-I-ABS applied to $\lambda x. e$.⁹

$$\begin{array}{c} \text{D-I-ABS1} \\ \frac{\Gamma, x : \tau \vdash e \Rightarrow_1 \sigma \quad \Gamma, x : \tau \vdash \sigma \rightsquigarrow_{\forall} \epsilon \quad \bar{a} \text{ fresh} \quad \epsilon' = [\bar{a}/[\epsilon]_x]\epsilon}{\min_{\exists}(\Gamma \vdash \lambda x. e \Rightarrow \tau \rightarrow \exists \bar{a}. \epsilon')} \\ \hline \Gamma \vdash \lambda x. e \Rightarrow_1 \tau \rightarrow \exists \bar{a}. \epsilon' \end{array}$$

Fig. 11: Restricted inference
Rule D-I-ABS1 quantifies over all possible derivations. This use of quantification is not uncommon in declarative systems which serve as specifications [5, 10, 23].

⁹ Rule D-I-ABS1 quantifies over all possible derivations. This use of quantification is not uncommon in declarative systems which serve as specifications [5, 10, 23].

Although the min_{\exists} predicate is explicitly enforced only in rule D-I-ABS1, this restriction effectively ensures that the predicate holds for any inferred type, since abstraction is the only place where existentials can be introduced nondeterministically. The algorithm remains sound with respect to \Rightarrow_1 , as the algorithm indeed infers only the minimum number of existential quantifiers.

Revisiting the expression $\lambda x : \text{Int}. \lambda y. y$ from Example 7.1, we recall that the original declarative system can infer $\text{Int} \rightarrow \exists a.a \rightarrow a$ by guessing the type $[x : \exists a.a]$ for y , while the algorithmic system infers $\text{Int} \rightarrow \hat{a} \rightarrow \hat{a}$. The restricted system will not allow for the type $\text{Int} \rightarrow \exists a.a \rightarrow a$ to be inferred, while still allowing types like $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$. Now completeness is restored, as the latter can be obtained by applying a complete context $\hat{a} = \text{Int}$ to the type $\text{Int} \rightarrow \hat{a} \rightarrow \hat{a}$ inferred by the algorithmic system. Note that rule D-I-ABS1 can still introduce \exists s in the inferred type, but it does so only when strictly necessary, e.g. for $(\lambda x : \text{Int}. \lambda y. (y : [x : \exists a.a]))$.

We can now also establish the completeness of typing:

Theorem 7.3 (Completeness of typing). *Given $\Delta_1 \rightarrow \Omega_1$,*

(inference) *If $[\Omega_1]\Delta_1 \vdash e \Rightarrow_1 \sigma$, then there exist Δ_2 , Ω_2 , and σ' such that $\Delta_2 \rightarrow \Omega_2$, $\Omega_1 \rightarrow \Omega_2$, $\Delta_1 \vdash e \Rightarrow \sigma' \dashv \Delta_2$, and $[\Omega_2]\sigma' = \sigma$.*

(checking) *If $[\Omega_1]\Delta_1 \vdash e \Leftarrow [\Omega_1]\sigma$ where $\Delta_1 \vdash \sigma$, then there exist Δ_2 and Ω_2 such that $\Omega_1 \rightarrow \Omega_2$, $\Delta_2 \rightarrow \Omega_2$, and $\Delta_1 \vdash e \Leftarrow [\Delta_1]\sigma \dashv \Delta_2$.*

Specifically, for any expression, if the declarative system infers type σ , the algorithm can infer a type σ' , with a context Ω_2 that extends both Δ_2 and Ω_1 , such that $[\Omega_2]\sigma' = \sigma$. The context Ω_2 is necessary since Ω_1 and Δ_2 may contain equivalent but not syntactically identical entries.

7.2 Completeness with Restricted Existential Instantiation

The previous section established completeness by disallowing unnecessary existential quantifiers. In this section, we explore how additional existential quantifiers are, in fact, permissible under certain conditions.

Consider again Example 7.1 with $\lambda x : \text{Int}. \lambda y. y$, where the declarative system and the algorithmic system infer the following types, respectively:

declarative: $\text{Int} \rightarrow \exists a.a \rightarrow a$ algorithmic: $\text{Int} \rightarrow \hat{a} \rightarrow \hat{a}$

A closer examination reveals that even though $\text{Int} \rightarrow \exists a.a \rightarrow a$ contains an unnecessary existential quantifier, it is “less general” than $\text{Int} \rightarrow \hat{a} \rightarrow \hat{a}$: any solution $\hat{a} = \tau$ makes $\text{Int} \rightarrow \tau \rightarrow \tau$ a subtype of $\text{Int} \rightarrow \exists a.a \rightarrow a$. Thus, while we cannot apply the stated completeness result (Theorem 7.3), the typing does not present the same fundamental issue as observed in Example 7.2. The problem only arises when these types are put inside existential projections through instantiation or subtyping. Example 7.2 demonstrates the problem with instantiation. Given that subtyping subsumes instantiation (§4), a similar issue can occur with subtyping.

Therefore, we can refine the restriction to check the min_{\exists} predicate only when we know a type will be placed within an existential projection, rather than applying it to any type inference. This allows the declarative type system

to infer unnecessary existential types, provided these existential types do not appear within existential projections. Thus, the declarative system can infer the type in Example 7.1, while still preventing Example 7.2.

To this end, we check \min_{\exists} immediately before any instantiation or subtyping.¹⁰ We present rules D-I-APP2 and D-C-SUB2 on the right, where we write \Rightarrow_2 and \Leftarrow_2 to denote the typing inference relation and checking relations respectively, with rules D-I-APP2 and D-C-SUB2 replacing rules D-I-APP and D-C-SUB.

The algorithm remains sound with respect to this system, since this system subsumes the one in §7.1. We now establish completeness for this system.

$$\frac{\begin{array}{c} \text{D-I-APP2} \\ \Gamma \vdash e \Rightarrow_2 \sigma \\ \min_{\exists}(\Gamma \vdash e \Rightarrow \sigma) \end{array}}{\Gamma \vdash e : \sigma \rightsquigarrow \sigma_1 \rightarrow \sigma_2} \quad \frac{\begin{array}{c} \text{D-C-SUB2} \\ \Gamma \vdash e \Rightarrow_2 \sigma \\ \min_{\exists}(\Gamma \vdash e \Rightarrow \sigma) \\ \Gamma \vdash e_1 \Leftarrow_2 \sigma_1 \\ \Gamma \vdash e : \sigma <: \rho \end{array}}{\Gamma \vdash e e_1 \Rightarrow_2 \sigma_2} \quad \frac{}{\Gamma \vdash e \Leftarrow_2 \rho}$$

Fig. 12: Restricted instantiation

Challenges of the completeness statement. To state completeness, we need to formulate a relation between the declarative and algorithmic types, taking note of the fact that the declarative type may contain more existentials (as in Example 7.1). Therefore, Theorem 7.3 does not apply; instead, for type inference completeness, we may require the algorithmic type to be a subtype of the declarative one, given a complete context.

Unfortunately, such a theorem is not true. Take $\lambda x : \text{Int}. \lambda z : [x]. \lambda y. y$ as an example. The declarative system can again pick y 's type to be $[x]$. Then:

declarative: $\text{Int} \rightarrow \exists a.(a \rightarrow a \rightarrow a)$ algorithmic: $\text{Int} \rightarrow \exists b.(b \rightarrow \hat{a} \rightarrow \hat{a})$
and the algorithmic type is not a subtype of the declarative type. Thus, we must state a stronger relation between the declarative and algorithmic types.

Key ideas. To establish a stronger relation between the declarative and algorithmic types, we rely on the following observation: *any additional existential variables in the declarative system correspond to unsolved unification variables in the algorithmic system*. Moreover, the declarative rule D-I-ABS can only introduce additional existential quantifiers in the return type of a function. The challenge now is to precisely formulate this observation.

From Example 7.1, it may seem that there is a one-to-one correspondence between an existentially quantified variable a and the unification variable \hat{a} :

declarative: $\text{Int} \rightarrow \exists a.a \rightarrow a$ algorithmic: $\text{Int} \rightarrow \hat{a} \rightarrow \hat{a}$

However, the declarative system can also pick y 's type to be $\text{Int} \rightarrow [x]$, and we now have to relate $\text{Int} \rightarrow a$ with \hat{a} :

declarative: $\text{Int} \rightarrow \exists a.((\text{Int} \rightarrow a) \rightarrow (\text{Int} \rightarrow a))$ algorithmic: $\text{Int} \rightarrow \hat{a} \rightarrow \hat{a}$

As another example, for $\lambda x : \text{Int}. \lambda z : [x]. \lambda y. y$, we need to relate a to both \hat{a} and b :

declarative: $\text{Int} \rightarrow \exists a.(a \rightarrow a \rightarrow a)$ algorithmic: $\text{Int} \rightarrow \exists b.(b \rightarrow \hat{a} \rightarrow \hat{a})$

¹⁰ This restriction could be refined by checking \min_{\exists} only when we know rule D-INST-EXIST or D-S-EXISTS-L will be applied.

$$\begin{array}{c}
 \boxed{\bar{c} \vdash \sigma_1 \lesssim \sigma_2} \\
 \text{D-CMT-REFL} \quad \text{D-CMT-ARR} \quad \text{D-CMT-EXISTS} \quad \text{D-CMT-EXISTSR} \\
 \frac{}{\bar{c} \vdash \sigma \lesssim \sigma} \quad \frac{\bar{c} \vdash \sigma_2 \lesssim \sigma'_2}{\bar{c} \vdash \sigma_1 \rightarrow \sigma_2 \lesssim \sigma_1 \rightarrow \sigma'_2} \quad \frac{\bar{c} \vdash [c/a]\epsilon_1 \lesssim [c/b]\epsilon_2}{c, \bar{c} \vdash \exists a.\epsilon_1 \lesssim \exists b.\epsilon_2} \quad \frac{\bar{c} \vdash \epsilon_1 \lesssim [c/a]\epsilon_2}{c, \bar{c} \vdash \epsilon_1 \lesssim \exists a.\epsilon_2}
 \end{array}$$

(Compatibility: σ_1 and σ_2 are compatible,)
using \bar{c} to instantiate existentials)

Fig. 13: Compatibility

Based on these examples, we can relate the declarative and algorithmic types through the following two steps:

1. Instantiating existential quantifiers in the return type of the declarative type with fresh type variables;
2. Using the same set of variables to solve unification variables in the algorithmic type, as well as to instantiate its existential quantifiers.

Following these steps, we should obtain the same type.

As an example, with fresh c , we can first instantiate $\text{Int} \rightarrow \exists a.a \rightarrow a$ to $\text{Int} \rightarrow c \rightarrow c$. We then solve $\text{Int} \rightarrow \hat{a} \rightarrow \hat{a}$ with $\hat{a} = c$, and derive $\text{Int} \rightarrow c \rightarrow c$.

Similarly, we can use fresh c to instantiate $\text{Int} \rightarrow \exists a.(\text{Int} \rightarrow a) \rightarrow (\text{Int} \rightarrow a)$ to $\text{Int} \rightarrow (\text{Int} \rightarrow c) \rightarrow (\text{Int} \rightarrow c)$. We then solve $\hat{a} = \text{Int} \rightarrow c$ in order to derive $\text{Int} \rightarrow (\text{Int} \rightarrow c) \rightarrow (\text{Int} \rightarrow c)$.

Lastly, we can first instantiate $\text{Int} \rightarrow \exists a.(a \rightarrow a \rightarrow a)$ to $\text{Int} \rightarrow (c \rightarrow c \rightarrow c)$. We then solve $\hat{a} = c$, and also instantiate b with c , getting $\text{Int} \rightarrow (c \rightarrow c \rightarrow c)$.

In the rest of the section, we provide formal definitions to capture these steps.

Formal definitions. Fig. 13 defines a *compatibility* relation between declarative types. The judgment $\bar{c} \vdash \sigma_1 \lesssim \sigma_2$ reads that under the list of variables \bar{c} , the types σ_1 and σ_2 are compatible. Rule D-CMT-REFL establishes reflexivity. Note that this rule is the only one that relates two universally quantified types, which is sufficient since such types are derived deterministically only through annotations. Rule D-CMT-ARR relates two function types, by requiring the same argument types, and recursively checking the return types, given the declarative system can only introduce additional existentials in the return type.

Of particular interest are rules D-CMT-EXISTS and D-CMT-EXISTSR. The former requires both existential quantifiers to be instantiated with the same variable c from the list, while the latter handles the case where the type on the right contains an additional existential quantifier to be instantiated.

To see how the definition is useful, consider our previous example:

declarative: $\text{Int} \rightarrow \exists a.((\text{Int} \rightarrow a) \rightarrow (\text{Int} \rightarrow a))$ algorithmic: $\text{Int} \rightarrow \hat{a} \rightarrow \hat{a}$

We now first generate a fresh variable c , and choose an appropriate complete context, in this case $c, \hat{a} = \text{Int} \rightarrow c$. Then, we have

$$c \vdash [c, \hat{a} = \text{Int} \rightarrow c](\text{Int} \rightarrow \hat{a} \rightarrow \hat{a}) \lesssim \text{Int} \rightarrow \exists a.((\text{Int} \rightarrow a) \rightarrow (\text{Int} \rightarrow a))$$

which uses c to instantiate a , resulting in the same type.

Compatibility is a stronger relation than subtyping. Intuitively, it indicates that one type may have more existentials, but is otherwise equivalent to the other type. The variables \bar{c} in the judgment represent the unknown existential projections when opening existential types. While the declarative type may

contain more existentials, we know that upon opening them, we can solve the unification variables in the algorithmic type with the corresponding existential projections to obtain an equivalent type.

Crucially, we can prove completeness of typing using compatibility, because of its useful properties. First, under the \min_{\exists} restriction (as in rule D-I-APP2), instantiation preserves compatibility. In fact, we can prove a stronger result:

Lemma 7.1 (\exists Inequality of Compatibility). *If $\bar{c} \vdash \sigma_1 \lesssim \sigma_2$, then $|\exists(\sigma_1)| \leq |\exists(\sigma_2)|$. Moreover, $|\exists(\sigma_1)| = |\exists(\sigma_2)|$ if and only if $\sigma_1 = \sigma_2$.*

Thus, when performing instantiation with the \min_{\exists} restriction, the declarative type is equivalent to the algorithmic one under a complete context substitution.

Moreover, compatibility is preserved when inferring existential types as in rule D-I-ABS. Since compatible types are equivalent modulo existentials, the $[\bar{a}/[\epsilon]_x]\epsilon$ operator will generalize the same set of existential projections across two compatible types, and thus the resulting types are still compatible.

Completeness of typing. We prove typing completeness using the \lesssim relation:

Theorem 7.4 (Completeness of typing). *Given $\Delta_1 \rightarrow \Omega_1$,*

(inference) *If $[\Omega_1]\Delta_1 \vdash e \Rightarrow_2 \sigma$, then there exist fresh $\bar{c} \notin \text{dom}(\Omega_1) \cup \text{dom}(\Delta_2)$, and $\Delta_2, \Omega_2, \sigma'$, such that $(\bar{c}, \Delta_2) \rightarrow \Omega_2$, $(\bar{c}, \Omega_1) \rightarrow \Omega_2$, $\Delta_1 \vdash e \Rightarrow \sigma' \dashv \Delta_2$, and $\bar{c} \vdash [\Omega_2]\sigma' \lesssim \sigma$.*

(checking) *If $[\Omega_1]\Delta_1 \vdash e \Leftarrow_2 [\Omega_1]\sigma$ where $\Delta_1 \vdash \sigma$, then there exist Δ_2 and Ω_2 such that $\Omega_1 \rightarrow \Omega_2$, $\Delta_2 \rightarrow \Omega_2$, and $\Delta_1 \vdash e \Leftarrow [\Delta_1]\sigma \dashv \Delta_2$.*

Specifically, for type inference, we prove that the inferred algorithmic type and the declarative type satisfy $\bar{c} \vdash [\Omega_2]\sigma' \lesssim \sigma$, while for type checking, we can check the expression against the $[\Delta_1]\sigma$ in the algorithmic system.

As a useful corollary, we give below a special case of this theorem for type inference where the inferred algorithmic type is a monotype, in which case the algorithmic type (after context application) is equivalent to the declarative type:

Corollary 7.1 (Completeness of monotype inference). *Given $\Delta_1 \rightarrow \Omega_1$, if $[\Omega_1]\Delta_1 \vdash e \Rightarrow_2 \tau$, then there exist $\Delta_2, \Omega_2, \tau'$, such that $\Delta_2 \rightarrow \Omega_2$, $\Omega_1 \rightarrow \Omega_2$, $\Delta_1 \vdash e \Rightarrow \tau' \dashv \Delta_2$, and $[\Omega_2]\tau' = \tau$.*

8 Related Work

Fig. 14 presents a concise summary of the most relevant work, namely Eisenberg et al. [4] and Dunfield and Krishnaswami [2, 3], as discussed throughout.

Existential types and abstract datatypes. Mitchell and Plotkin [12] pioneered the use of existential types as a foundation for abstract datatypes. Cardelli and Leroy [1] then observed the inconvenience of *pack*, and proposed *open* to model the *dot notation* in *module systems*. Assuming a module m has one hidden type component and one function set, they proposed $m.\text{Fst}$ to project out the hidden type and $m.\text{Snd}$ to access the function set. Cardelli and Leroy [1, §4] extended this to arbitrary expressions $e.\text{Fst}$. The $[e : \epsilon]$ notation used by Eisenberg et al. [4] and in this work can be seen as an alternative to $e.\text{Fst}$.

	DK 2013	DK 2019	EDWL	This Work
Higher-rank polymorphism	✓	✓	✓	✓
Polymorphic subtyping	✓	✓	✗	✓
Existential types	✗	weak	strong	strong
Declarative specification	✓	✓	✓	✓
Algorithmic specification	✓	✓	✗	✓

Fig. 14: Comparison of our work with EDWL and DK

Type inference for existential types. Early work on type inference for existential types (e.g. [9, 22]) mostly focused on existential datatypes with (weak) existentials. A more recent, closely related work is Dunfield and Krishnaswami [3], which developed a bidirectional type system for higher-rank polymorphism, (weak) existentials, and indexed types. Their system supported types with mixed quantifiers and uses a particularly fixed order when comparing quantifiers for subtyping. In our type system, types are simplified as we only allow universal quantifiers to precede existential ones. Our type inference algorithm for existential types directly builds on Eisenberg et al. [4]. However, their system did not present an algorithmic type system, nor did it support higher-rank polymorphism with polymorphic subtyping.

Higher-rank polymorphism and impredicativity. Higher-rank polymorphism has been studied extensively, with most work supporting polymorphic subtyping and contravariant functions [2, 14, 16]. The Glasgow Haskell Compiler (GHC) has seen various implementations. Peyton Jones et al. [16] supported contravariant function types and introduced *deep skolemisation*, deriving $\forall a. b. a \rightarrow b \rightarrow b$ as a subtype of $\forall a. a \rightarrow \forall b. b \rightarrow b$. Serrano et al. [20, 21] support invariant functions for inferring more impredicative types.

Ordered contexts. Gundry et al. [6] first used ordered contexts for type inference in a dependently typed language to address scoping challenges. Dunfield and Krishnaswami [2] contributed a particularly elegant algorithm for higher-rank polymorphism using this technique. It has since been employed in various works [3, 25, 26].

Bibliography

- [1] Cardelli, L., Leroy, X.: Abstract types and the dot notation. In: IFIP TC2 working conference on programming concepts and methods, pp. 479–504, North-Holland (1990)
- [2] Dunfield, J., Krishnaswami, N.R.: Complete and easy bidirectional type-checking for higher-rank polymorphism. In: Proceedings of the 18th ACM SIGPLAN international conference on Functional programming, pp. 429–442 (2013)
- [3] Dunfield, J., Krishnaswami, N.R.: Sound and complete bidirectional type-checking for higher-rank polymorphism with existentials and indexed types. Proceedings of the ACM on Programming Languages **3**(POPL), 1–28 (2019)
- [4] Eisenberg, R.A., Duboc, G., Weirich, S., Lee, D.: An existential crisis resolved: Type inference for first-class existential types. Proceedings of the ACM on Programming Languages **5**(ICFP), 1–29 (2021)
- [5] Emrich, F., Lindley, S., Stolarek, J., Cheney, J., Coates, J.: Freezeml: Complete and easy type inference for first-class polymorphism. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 423–437 (2020)
- [6] Gundry, A., McBride, C., McKinna, J.: Type inference in context. In: Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming, p. 43–54, MSFP ’10, Association for Computing Machinery, New York, NY, USA (2010), ISBN 9781450302555, <https://doi.org/10.1145/1863597.1863608>, URL <https://doi.org/10.1145/1863597.1863608>
- [7] Howard, W.A., et al.: The formulae-as-types notion of construction. To HB Curry: essays on combinatory logic, lambda calculus and formalism **44**, 479–490 (1980)
- [8] Hu, J.Z., Lhoták, O.: Undecidability of $d <$: and its decidable fragments. Proceedings of the ACM on Programming Languages **4**(POPL), 1–30 (2019)
- [9] Läufer, K., Odersky, M.: An extension of ml with first-class abstract types. In: ACM SIGPLAN Workshop on ML and its Applications, pp. 78–91 (1992)
- [10] Leijen, D.: Hmf: simple type inference for first-class polymorphism. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, p. 283–294, ICFP ’08, Association for Computing Machinery, New York, NY, USA (2008), ISBN 9781595939197, <https://doi.org/10.1145/1411204.1411245>, URL <https://doi.org/10.1145/1411204.1411245>
- [11] MacQueen, D.B.: Using dependent types to express modular structure. In: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 277–286 (1986)
- [12] Mitchell, J.C., Plotkin, G.D.: Abstract types have existential type. ACM Transactions on Programming Languages and Systems (TOPLAS) **10**(3), 470–502 (1988)

- [13] Nieto, A.: Towards algorithmic typing for dot. arXiv preprint arXiv:1708.05437 (2017)
- [14] Odersky, M., Läufer, K.: Putting type annotations to work. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, p. 54–67, POPL ’96, Association for Computing Machinery, New York, NY, USA (1996), ISBN 0897917693, <https://doi.org/10.1145/237721.237729>, URL <https://doi.org/10.1145/237721.237729>
- [15] Parreaux, L., Boruch-Gruszecki, A., Fan, A., Chau, C.Y.: When subtyping constraints liberate: A novel type inference approach for first-class polymorphism. Proceedings of the ACM on Programming Languages **8**(POPL), 1418–1450 (2024)
- [16] Peyton Jones, S., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. Journal of functional programming **17**(1), 1–82 (2007)
- [17] Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for gadts. ACM SIGPLAN Notices **41**(9), 50–61 (2006)
- [18] Pierce, B.C.: Types and programming languages. MIT press (2002)
- [19] Pierce, B.C., Turner, D.N.: Local type inference. ACM Transactions on Programming Languages and Systems (TOPLAS) **22**(1), 1–44 (2000)
- [20] Serrano, A., Hage, J., Peyton Jones, S., Vytiniotis, D.: A quick look at impredicativity. Proceedings of the ACM on Programming Languages **4**(ICFP), 1–29 (2020)
- [21] Serrano, A., Hage, J., Vytiniotis, D., Peyton Jones, S.: Guarded impredicative polymorphism. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 783–796 (2018)
- [22] Simonet, V.: An extension of hm (x) with bounded existential and universal data-types. ACM SIGPLAN Notices **38**(9), 39–50 (2003)
- [23] Vytiniotis, D., Weirich, S., Peyton Jones, S.: Boxy types: inference for higher-rank types and impredicativity. In: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming, p. 251–262, ICFP ’06, Association for Computing Machinery, New York, NY, USA (2006), ISBN 1595933093, <https://doi.org/10.1145/1159803.1159838>, URL <https://doi.org/10.1145/1159803.1159838>
- [24] Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 224–235 (2003)
- [25] Xie, N., Eisenberg, R.A., Oliveira, B.C.d.S.: Kind inference for datatypes. Proc. ACM Program. Lang. **4**(POPL) (dec 2019), <https://doi.org/10.1145/3371121>, URL <https://doi.org/10.1145/3371121>
- [26] Zhao, J., Oliveira, B.C.d.S., Schrijvers, T.: A mechanical formalization of higher-ranked polymorphic type inference. Proceedings of the ACM on Programming Languages **3**(ICFP), 1–29 (2019)
- [27] Ziliani, B., Sozeau, M.: A unification algorithm for coq featuring universe polymorphism and overloading. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, pp. 179–191 (2015)