Bidirectional Higher-Rank Polymorphism with Intersection and Union Types

ANONYMOUS AUTHOR(S)

Modern mainstream programming languages, such as TypeScript, Flow, and Scala, have polymorphic type systems enriched with intersection and union types. These languages implement variants of *bidirectional higher-rank polymorphic* type inference, which was previously studied mostly in the context of functional programming. However, existing type inference implementations lack solid theoretical foundations when dealing with non-structural subtyping and intersection and union types, which were not studied before.

In this paper, we study bidirectional higher-rank polymorphic type inference algorithms with intersection and union types. We introduce both a bidirectional specification and algorithmic versions of the type system. We first study a type inference algorithm that has good theoretical properties: it is sound, complete, and decidable with respect to the corresponding bidirectional specification. This is helpful to identify a class of types that can always be inferred. We also explore algorithmic variants that are sound but not necessarily complete. These variants incorporate practical features, such as handling records, which align with real-world implementations. These variants enable inference for a broader range of types, enhancing the expressiveness of the type systems. Our findings provide insights for improving current implementations and inspire the design of novel type inference algorithms. To ensure rigor, all results are formalized in the Coq proof assistant.

1 INTRODUCTION

Programming languages such as TypeScript, Flow, and Scala, embrace polymorphic type systems with intersection and union types [Bierman et al. 2014; Chaudhuri et al. 2017; Rompf and Amin 2016]. These languages started to also incorporate *higher-rank polymorphism* (HRP) [Odersky and Läufer 1996] into object-oriented programming. HRP enables polymorphic types to appear anywhere in nested positions inside function types. In TypeScript or Flow, it is also possible to have types such as $(\forall a.\ a \to a)$ & $(\texttt{string} \to (\forall b.\ b \to \texttt{boolean}))$ where polymorphic types appear nested under other type constructors, such as intersection types. These features enhance expressiveness and flexibility, enabling developers to write concise and robust code.

For programming languages to be practical, they must support type inference, enabling automatic deduction of type information with only a small amount of explicit type annotations. Type inference in object-oriented programming (OOP) languages has predominantly relied on local type inference [Pierce and Turner 2000]. Local type inference scales well to the forms of non-structural subtyping employed in OOP. Furthermore, local type inference enables both *implicit* and *explicit* instantiation of polymorphic functions. Therefore, even if it is not possible to automatically infer some instantiation, programmers have the possibility to explicitly specify the instantiation themselves via *explicit type applications* [Eisenberg et al. 2016a; Pierce and Turner 2000].

HRP type inference has been extensively studied in the context of functional programming [Dunfield and Krishnaswami 2013; Leijen 2008; Peyton Jones et al. 2007]. However, there is little work on HRP techniques dealing with intersection and union types. Since existing OOP type inference implementations have traditionally relied on techniques inspired by local type inference, which employ bidirectional typing [Dunfield and Krishnaswami 2021; Pierce and Turner 2000], bidirectional HRP techniques [Cui et al. 2023; Dunfield and Krishnaswami 2013; Peyton Jones et al. 2007; Zhao and Oliveira 2022] seem to fit well with those implementations. However, a significant difference between traditional HRP algorithms and local type inference is that HRP algorithms typically support *polymorphic subtyping* [Mitchell 1988; Odersky and Läufer 1996]. The most distinctive and noteworthy rule in polymorphic subtyping is the $\forall$L rule:

$$\frac{\Gamma \vdash \tau \quad \Gamma \vdash [\tau/a]A <: B}{\Gamma \vdash \forall a.A <: B}$$

The $\forall L$ rule expresses the relationship between polymorphic types and their more specific (instantiated) counterparts. For example, the statement $\forall a.a \to a <: \texttt{Int} \to \texttt{Int}$ is a valid subtyping assertion that can be derived by selecting $\tau = \texttt{Int}$, and subsequently substituting the type variable on the left-hand side of subtyping, as dictated by the $\forall L$ rule. In contrast, traditional local type inference lacks polymorphic subtyping and does not allow relating polymorphic types to their instantiated counterparts.

A key challenge in developing type inference techniques for HRP for languages such as TypeScript and Flow lies on the interaction between polymorphic subtyping and intersection and union types, which is non-trivial. To complicate matters further, languages like TypeScript and Flow support explicit type applications, which is known to have non-trivial interactions with HRP as well [Eisenberg et al. 2016a; Zhao and Oliveira 2022].

In this paper we study the integration and interaction of three features: 1) *higher-rank polymorphism*; 2) *intersection and union types*; 3) *explicit type applications*. This interaction poses challenges to existing type inference implementations, which lack solid theoretical foundations for handling these features together. For instance, careful consideration is needed to preserve desirable properties such as transitivity of subtyping, decidability, or completeness with respect to a type system specification. As we show in our work, current type inference implementations of TypeScript and Flow have some issues related to these interactions.

We present a calculus and type system, called $F^e_{\sqcup\sqcap}$, with three features. $F^e_{\sqcup\sqcap}$ has both a bidirectional specification and algorithmic versions of the type system, and extends previous work by Zhao and Oliveira [2022]. The key novelty of $F^e_{\sqcup\sqcap}$ is the addition of intersection and union types. We first study a type inference algorithm that has good theoretical properties: it is sound, complete with respect to the corresponding bidirectional specification and decidable. This is helpful to identify a class of types that can always be inferred. In addition, we study some variants of the algorithm that are only sound, but not complete. These variants incorporate practical features such as records, and they also enable inference for a larger class of types. In particular, it is possible, in many cases, to infer types with intersection and union types. These variants are closer to practical implementations like TypeScript or Flow.

In summary, the contributions of our work are:

- We study **bidirectional HRP with intersection and union types and explicit type applications.** The three features are important in practice and adopted by languages like TypeScript and Flow.
- **Type inference algorithms using a worklist approach.** We adopt the worklist formulation for algorithmic type inference by Zhao et al. [2019] and show how the worklist approach can be extended with intersection and union types.
- As part of the development of $F^e_{\sqcup\sqcap}$, we introduced a few **technical innovations**. Our syntax-directed transfer and defunctionalized representation of continuation-passing style are suitable for mechanical formalization in any proof assistants without built-in binder support. We also improve the polytype-splitting technique [Cui et al. 2023] by dropping unnecessary checks and improving the decidability reasoning related to it.
- We prove several results about the **metatheory:** of $F^e_{\sqcup\sqcap}$. These include soundness and completeness between the bidirectional type systems and the algorithmic formulations, and decidability of the latter. Furthermore, we also prove important properties of the bidirectional type systems, such as transitivity of subtyping and subsumption.
- All the results are **mechanically formalized** in the Coq proof assistant. Furthermore, we have a simple prototype implementation that can run the examples presented in the paper. The Coq formalization, the implementation, and our examples can be found in the supplementary materials.

2 OVERVIEW

We start with a background on implicit (predicative) higher-rank polymorphism with explicit impredicative type applications [Zhao and Oliveira 2022] as our work adopts a similar framework and design principles. Then we revisit the support of HRP and intersection and union types in TypeScript. Finally, we provide an overview of the features of $F^{e}_{\sqcup\sqcap}$.

2.1 Background: Higher-Rank Polymorphism with Explicit Type Applications

Higher-rank polymorphism for languages à la System F allows universal types to appear deeply inside function types, generalizing the Hindley-Milner (HM) polymorphism [Hindley 1969; Milner 1978] where universal types must be top-level. HRP with implicit predicative instantiation has been thoroughly studied [Dunfield and Krishnaswami 2013; Odersky and Läufer 1996; Peyton Jones et al. 2007]. The $F^{e}_{\leq}$ calculus [Zhao and Oliveira 2022], originally based on Dunfield and Krishnaswami [2013]'s type system, additionally supports impredicative instantiations to be explicitly provided by the programmer with a type application syntax. All programs typable in System F can be type-checked in $F^{e}_{\leq}$. We provide a quick overview of the important design choices in $F^{e}_{\leq}$ next.

HRP subtyping and explicit type applications. In traditional HRP systems [Odersky and Läufer 1996] *without* explicit type applications, the two key rules in subtyping relation are:

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

These rules enable order-irrelevant universal quantifiers, i.e., two universal types with the same body but a different order of quantifiers are considered equivalent. For example $\forall a.\, \forall b.\, a \to b$ and $\forall b.\, \forall a.\, a \to b$ are subtypes of each other. However, order-irrelevant quantifiers are incompatible with explicit type applications in general [Eisenberg et al. 2016a] because the provided instantiation is supposed to bind to a certain quantifier. Consider $\lambda x.\, x\ 3$, which can be checked against both $(\forall a.\, \forall b.\, a \to b) \to \mathsf{Bool}$ and $(\forall b.\, \forall a.\, a \to b) \to \mathsf{Bool}$, by instantiating a and b to Int and Bool, respectively. Now, suppose that we provide an explicit type instantiation `Int` to x as $\lambda x.\, (x\ @\mathtt{Int}\ 3)$. This expression can be checked against $(\forall a.\, \forall b.\, a \to b) \to \mathtt{Bool}$, but it cannot be checked against $(\forall b.\, \forall a.\, a \to b) \to \mathtt{Bool}$. Thus the types $\forall a.\, \forall b.\, a \to b$ and $\forall b.\, \forall a.\, a \to b$ do not behave equivalently in the presence of explicit type applications and should not be in a subtyping relation. Zhao and Oliveira also identify other subtler problems when explicit instantiations are impredicative. Interested readers can refer to their paper for concrete examples and other details.

To address all the problems, $F^{e}_{\leq}$ adopts a different subtyping relation compared to Odersky and Läufer's, as shown in Figure 1. The $\leq\forall\mathsf{R}$ rule is replaced with a more restrictive rule ($\leq\forall$) where both sides must be universal types. This makes the order of the universal quantifiers relevant, forbidding subtyping statements such as $\forall a.\, \forall b.\, a \to b \leq \forall b.\, \forall a.\, a \to b$. The second restriction is introducing a new sort of variable, subtype variables ($\tilde{a}$), used by the new rule $\leq\forall$. Subtype variables are *not monotypes*, so the implicit instantiation τ in rule $\leq\forall\mathsf{L}$ cannot contain them. The third restriction is adding two checks in well-formedness, to ensure no unused variables in universal types.

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

The latter two restrictions are key to retain stability under explicit type application instantiation.

Theorem 2.1 (Stability under Instantiation). *Given all contexts and types well-formed, if $\Psi \vdash \forall a.\, A \leq \forall a.\, B$, then $\Psi \vdash [C/a]A \leq [C/a]B$.*

Greedy solving strategy. Most previous HRP systems with implicit predicative instantiation [Cui et al. 2023; Dunfield and Krishnaswami 2013; Odersky and Läufer 1996; Zhao and Oliveira 2022]

Type variables	a, b		Subtype variables $\tilde{a}, \tilde{b}$
Types	A, B, C	$::=$	$1 \mid a \mid \tilde{a} \mid \forall a.\, A \mid A \to B \mid \top \mid \bot$
Monotypes	τ, σ	$::=$	$1 \mid a \mid \tau \to \sigma$
Contexts	Ψ	$::=$	$\cdot \mid \Psi, a \mid \Psi, \tilde{a} \mid \Psi, x : A$

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

$$\frac{}{\Psi \vdash \mathbb{1} \leq \mathbb{1}}\ \leq\mathbb{1} \qquad \frac{}{\Psi \vdash A \leq \top}\ \leq\top \qquad \frac{}{\Psi \vdash \bot \leq A}\ \leq\bot \qquad \frac{}{\Psi \vdash a \leq a}\ \leq\text{TVar} \qquad \frac{}{\Psi \vdash \tilde{a} \leq \tilde{a}}\ \leq\text{STVar}$$

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

Fig. 1. Type syntax and subtyping rules of $F^e_\leq$.

employ a greedy approach to solve the instantiation: existential variables are solved to the first monotype they are compared against in subtyping. The completeness of such a strategy relies on an important property among the monotypes [Cui et al. 2023]: subtyping between monotypes implies equality of monotypes, or formally speaking, $\tau_1 \leq \tau_2 \to \tau_1 = \tau_2$. The fact that $\bot$ and $\top$ are not monotypes in $F^e_\leq$ is also crucial for this property to hold.

Bidirectional typing. $F^e_\leq$ chooses bidirectional typing to describe its specification and implement the algorithm. The $F^b_\leq$ calculus [Cui et al. 2023], adds some general improvements to the typing rules of $F^e_\leq$, which we will also adopt in our work. An important property for a bidirectional type system is checking subsumption. This property expresses the intuition that if a program type-checks under some type A then it should remain well-typed after changing A to its supertypes.

Theorem 2.2 (Checking Subsumption). *Given $\vdash \Psi$, $\Psi \vdash B$, $\Psi \vdash e$, if $\Psi \vdash e \Leftarrow A$ and $\Psi \vdash A \leq B$, then $\Psi \vdash e \Leftarrow B$.*

2.2 HRP and Intersection and Union Types in TypeScript

We briefly introduce TypeScript's syntax used in our examples. Base types include `number` and `boolean`. `A & B` and `A | B` denote intersection ($A \sqcap B$) and union ($A \sqcup B$) types. Function types $A \to B$ in TypeScript require an argument name. Thus a function type is written as `(x:A) => B`. Since this argument name makes no difference, we usually write it as `(_:A) => B` in the following examples. Universal types $\forall a.\, A$ are represented as `<a>A` (though the standard naming convention usually picks capital letters for the type variable). Record types are denoted as `{m: A, ...}` where `m` is a label. As an example, the type $(\forall a.\, a \to \texttt{Int}) \to (\texttt{Int} \sqcup \texttt{Bool})$ is written as `<A>(_:A)=>number)=>(number|boolean)` in TypeScript. The expression syntax is more standard, including applications `f(x)`, record projections `e.l` and function definitions `function f(x: A): B {...}`.

Intersection and union types. The following example shows how intersections are used to model objects that implement 2 interfaces (`{m: number}` and `{n: boolean}`). The variable `o1` is an object that implements both interfaces `m` and `n` and has the type `{m:number}&{n:boolean}`. Functions that only require either interface can be safely applied to `o1`. Analogously, unions can be used for specifying objects that implement either one of two interfaces. In such cases, it is safe to apply the common operations supported by both interfaces to the object, as illustrated by `h1`.

```
function f1(o: {m: number}): number { return o.m }
function g1(o: {n: boolean}): boolean { return o.n }
var o1 = {m: 1, n: true}; var ex1_1 = f1(o1); var ex1_2 = g1(o1)
function h1(o: {m: number, n: boolean} | {k: string, m: number}): number { return o.m }
```

Function overloading and backtracking. TypeScript supports a general type system with intersections and union types, with few restrictions. With this general support, function overloading can also be modeled using intersection types.

```
function f2(g: ((_:number)=>number) & ((_:boolean)=>boolean)): boolean { return g(true) }
```

Because of this backtracking is needed. When applying g, since the argument is a boolean, the first function type in the intersection cannot be used to type the application. Thus, we must try the next function, which accepts a boolean argument. TypeScript does avoid some backtracking by employing a committed choice [Shapiro 1989] for overloaded functions: once a certain branch is partially matched, it will commit to that branch and reject if some mismatch happens later, as shown by `f3_2`.

```
function f3_1(f: ((_:number) => (_:number) => number) &
                 ((_:number) => (_:boolean) => number)) { return f(1)(2) }
function f3_2(f: ((_:number) => (_:number) => number) &
                 ((_:number) => (_:boolean) => number)) { return f(1)(true) } // rejected
```

Polymorphism and type applications. TypeScript also supports parametric polymorphism. The identity function `f4` below is given a polymorphic type $\forall a.\ a \to a$. The function `f4` is applied to `1` directly, as `ex4_1` shows, with `A` *implicitly* instantiated to **`number`**. In `ex4_2`, the function is *explicitly* instantiated: `A` is first explicitly instantiated to **`number`** and the function is applied to `1`.

```
function f4<A>(x: A): A { return x }
var ex4_1 = f4(1); var ex4_2 = f4<number>(1)
```

Types inferred for implicit instantiation. TypeScript avoids inferring some supertypes/subtypes. The following example `ex5_1` is rejected by TypeScript, though common supertypes of **`number`** and **`boolean`**, like **`number | boolean`** or `Any`, are valid instantiations (as illustrated by `ex5_2`). Similarly, `ex5_3` gets rejected though **`number & boolean`** or `never` are valid instantiations.

A possible justification for this behavior is that such patterns often correspond to an error instead of intended behavior. Inferring union/intersection or top/bottom types too eagerly would hide errors. Still, with explicit type applications, programmers can write such a program.

```
function f5<A>(x: A, y: A): A { return x }
var ex5_1 = f5(1, true) // rejected!
var ex5_2 = f5<boolean|number>(1, true)
function g5<A>(g1: (_:A)=>number, g2: (_:A)=>number): (_:A)=>number { return x => 1 }
var ex5_3 = g5((x: number) => 1, (y: boolean) => 2) // rejected!
var ex5_4 = g5<number&boolean>((x: number) => 1, (y: boolean) => 2)
```

Higher-rank polymorphism and polymorphic subtyping. Besides top-level polymorphism, TypeScript also supports HRP, where polymorphic types can appear nested inside a type. Moreover, polymorphic subtyping is allowed: TypeScript also supports the polymorphic subtyping for higher-rank types, demonstrated by the following example `ex6`. The function `f6` expects an argument with type $(\forall a.\ a \to a) \to \mathtt{Int}$ and `g6` is of type $(\mathtt{Int} \to \mathtt{Int}) \to \mathtt{Int}$. Since $(\mathtt{Int} \to \mathtt{Int}) \to \mathtt{Int} \leq (\forall a.\ a \to a) \to \mathtt{Int}$, applying `f6` to `g6` is valid.

```
function f6(f: (_:<A>(_:A) => A) => number) { return 2 }
function g6(f: (_:number) => number) { return 1 }
var ex6 = f6(g6)
```

The following example provides an example of combining HRP with explicit instantiation. In `ex7_1`, the `A` of `f7` is explicitly instantiated to $\forall a.\ a \to a$. The type of `f7` then becomes $(\forall a.\ a \to a) \to (\forall a.\ a \to a)$, so `f7` can be applied to itself. The type application could also contain intersection and union types, as demonstrated by `ex7_2`, where the instantiation is $\forall a.\ (a \sqcap a) \to (a \sqcup a)$.

```
function f7<A>(x: A): A { return x }
var ex7_1 = f7<<A>(_:A)=>A>(f7)
var ex7_2 = f7<<A>(_:A&A)=>(A|A)>(f7)
```

Greedy solving strategy. Like most previous work on HRP, TypeScript adopts a *greedy* instantiation approach in polymorphic subtyping. This means that the first candidate for instantiating a universal variable is always chosen, even if this is not the best choice. This behavior can be demonstrated by the following examples. Example `ex8_1` is accepted because the first candidate is **`number`** and $\mathtt{Int} \to \mathtt{Int} \to \mathtt{Int} \leq \mathtt{Int} \to (\mathtt{Int} \sqcap \mathtt{Bool}) \to \mathtt{Int}$. In contrast, `ex8_3` is rejected because the first candidate is **`number&boolean`** and $(\mathtt{Int} \sqcap \mathtt{Bool}) \to (\mathtt{Int} \sqcap \mathtt{Bool}) \to \mathtt{Int} \not\leq (\mathtt{Int} \sqcap \mathtt{Bool}) \to \mathtt{Int} \to \mathtt{Int}$. The situation for `ex_2` and `ex_4` is dual except the correct instantiation is $\mathtt{Int} \sqcup \mathtt{Bool}$. The use of greedy instantiation is understandable and justifiable in practice. Adopting non-greedy instantiation, which could always pick the best choice, easily runs into some fundamental open problems in the presence of non-structural subtyping [Dudenhefner et al. 2016; Su et al. 2002].

```
function f8(x: (_:<A>(_:A)=>(_:A)=>number)=>number): number { return 1 }
var g8_1: (_:(_:number)=>(_:number&boolean)=>number) => number = f => 1
var g8_2: (_:(_:(number|boolean))=>(_:number)=>number) => number = f => 1
var g8_3: (_:(_:(number&boolean))=>(_:number)=>number) => number = f => 1
var g8_4: (_:(_:number)=>(_:number|boolean)=>number) => number = f => 1
var ex8_1 = f8(g8_1); var ex8_2 = f8(g8_2)
var ex8_3 = f8(g8_3); var ex8_4 = f8(g8_4) // both rejected!
```

Order-irrelevant quantifiers and broken checking subsumption. TypeScript adopts order-irrelevant universal quantifiers with explicit type application. A possible explanation for this choice is that the initial support for HRP in TypeScript was done around 2015, which was before the interaction between HRP and explicit type applications was first studied [Eisenberg et al. 2016a]. The use of order-irrelevant quantifiers with explicit type applications leads to problem because explicit type applications use the order of type arguments to decide which arguments to instantiate. A consequence of this choice in TypeScript is that it is not always possible to replace the type of an expression with a supertype. In other words, checking subsumption is broken. The function `h9` demonstrates the broken checking subsumption. $\forall a.\ \forall b.\ a \to b \leq \forall a.\ \mathtt{number} \to a$ is accepted, but after explicitly instantiating the first quantifier to boolean, $\forall a.\ \mathtt{boolean} \to b \leq \mathtt{number} \to \mathtt{boolean}$ does not hold, leading to the rejection of `ex9_2`.

```
var f9: (k: <A>(_:number)=>(_:A)=>number) => (_:boolean) => number = k => k(3)
var h9: (k: <A,B>(_:B)=>(_:A)=>B) => (b: boolean) => number = k => f9(k)
var g9: (k: <A>(_:number)=>(_:A)=>number) => (_:boolean) => number = k => k<boolean>(3)

var ex9_1: (k: <A,B>(_:B)=>(_:A)=>B) => (_:boolean) => number = k => g9(k)
var ex9_2: (k: <A,B>(_:B)=>(_:A)=>B) => (_:boolean) => number = k => k<boolean>(3) // rejected!
```

In $F^e_{\sqcup\sqcap}$, we adopt order-relevant quantifiers instead, which interact well with explicit type applications, and lead to a more principled design with HRP and explicit type applications. However, this change impacts the subtyping relation considerably and is a non-trivial change compared to adopting order-irrelevant quantifiers.

2.3 Our Approach

In this section, we provide an overview of the contributions of our work. Our work studies how to combine HRP with explicit type applications, and intersection and union types in a type system. We study three type systems in this paper: $F^e_{\sqcup\sqcap}$, and its two extensions $F^e_{\sqcup\sqcap}$ with records, and $F^e_{\sqcup\sqcap}$ with implicitly instantiable intersection and union types. All the examples in Section 2.2 and this section (except for those relying on order irrelevant quantification) are also encodable in $F^e_{\sqcup\sqcap}$. The examples using records require the extensions of $F^e_{\sqcup\sqcap}$, whereas the other examples work on the base formulation of $F^e_{\sqcup\sqcap}$. Appendix A illustrates all the examples written in our prototype implementation, which are omitted here and just presented in TypeScript syntax for space reasons.

Base $F^e_{\sqcup\sqcap}$. Our base type system $F^e_{\sqcup\sqcap}$ combines HRP with explicit type application and intersection and union types. It supports unrestricted intersection and union types, while still retaining good properties, including subtyping transitivity and checking subsumption. This system is equipped with an algorithmic system adopting a greedy strategy to find instantiations. We aim at a modest inference for this system where intersections and unions are excluded from monotypes (i.e. implicitly instantiable types). The algorithm is sound and complete with such a specification: it can infer all the possible monotypes under such a definition. The soundness and completeness imply that all the good properties also hold for the algorithm. Besides a careful treatment of the interaction among HRP, explicit type application, and intersection and union types to build a type system with desirable properties, $F^e_{\sqcup\sqcap}$ has the following features.

Inference of unannotated functions with monomorphic types. $F^e_{\sqcup\sqcap}$ supports inference of unannotated functions with monomorphic types. TypeScript makes no effort in inferring the types for unannotated functions. It simply outputs the most general type `Any` for arguments and return type. The following example `ex10` is accepted in TypeScript, but this is problematic since TypeScript infers `Any => Any` for `h10`, which it is not a subtype of `number => number`. In contrast, the same program is accepted in $F^e_{\sqcup\sqcap}$ with the correct type for `h10`.

```
function f10(g: ((_:number)=>number)): number { return 1 }
var g10 = x => y => y; var h10 = g10(1)
var ex10 = f10(h10) // accepted, but with wrong type inferred for h10
```

Intersection introduction rule. In standard type systems with intersection types and $F^e_{\sqcup\sqcap}$, if an expression can be checked by two types, it can be checked by the intersection of these two types, as formulated by the following informal rule $\frac{e \Leftarrow A_1 \quad e \Leftarrow A_2}{e \Leftarrow A_1 \sqcap A_2}$. TypeScript does not support this rule and it rejects `ex11`, which means that it becomes much harder to actually build an expression with an intersection type in TypeScript. Example `ex11` is accepted in $F^e_{\sqcup\sqcap}$.

```
function f11(g: ((_:number)=>number) & ((_:boolean)=>boolean)): number { return 1 }
var ex11 = f11(x => x) // rejected!
```

Less syntactic restriction for polymorphic types. TypeScript requires a function type inside the polymorphic quantifier, i.e., it must be of the shape $\forall a.\ A \to B$. So it can not express types like $\forall a.a$ or $\forall a.(a \to \texttt{number}) \sqcap (a \to \texttt{boolean})$. There is no way to define functions that take the argument with the above type. $F^e_{\sqcup\sqcap}$ has a more complete support for polymorphic types and it puts no other restriction on the type inside the quantifier as long as it is a true polymorphic type.

```
function ex12_1(x : <A>A) {...} // rejected
function ex12_2(x : <A>(((_:A)=>boolean) & ((_:A)=>number))) {...}  // rejected
```

More complete overlapping. $F^e_{\sqcup\sqcap}$ explores the best option from the theoretical perspective to choose the correct branch if there is one from the overloaded function. This choice does come with the cost of more backtracking. We believe the "committed choice" principle of TypeScript can be adapted to $F^e_{\sqcup\sqcap}$, as a more pragmatic option.

Compared with TypeScript on the behavior of examples presented in the previous section, our implementation for $F^e_{\sqcup\sqcap}$ additionally accepts `f3_2` and `f8_3`, while rejects `ex8_2`, `ex8_4` and `h_9`, plus all the expressions in the first example (`f1`, `g1`, ...) since they cannot be expressed in this system. The rejections are for good reasons. For `ex8_2` and `ex8_4`, with the greedy solving strategy, inferring intersection and union types cannot be complete and can lead to inconsistent behavior. For `h9`, with order-relevant quantifiers, it should not be accepted. The acceptance of `f3_2` demonstrates $F^e_{\sqcup\sqcap}$'s better support of overloading. The acceptance of `f8_3` demonstrates $F^e_{\sqcup\sqcap}$'s completeness in finding solutions. The examples in this section are also accepted in our implementation.

$F^e_{\sqcup\sqcap}$ with records. To illustrate the expressive power of $F^e_{\sqcup\sqcap}$ and provide support for records, we add a simple extension that allows modeling records via overloading [Castagna et al. 1995]. Since intersection and union types are first-class in $F^e_{\sqcup\sqcap}$, they can be used to encode other practical features. We demonstrate one of such encoding: records as intersection types and reuse the type system of $F^e_{\sqcup\sqcap}$ to support record extension and projection. A record entry $\{m : \texttt{number}\}$ is encoded as a function type `Label m` $\rightarrow$ `Int`, and record extension is simply the intersection of the type of the new entry with the type of the remaining record. Record projection is encoded as a function application. The inference of record projection can be handled in the same way as that of function application and reuse the rules in $F^e_{\sqcup\sqcap}$. Under such encoding, `o1` has type (`Label m` $\rightarrow$ `Int`) $\sqcap$ (`Label n` $\rightarrow$ `Bool`) and `h1` has type (((`Label m` $\rightarrow$ `Int`)$\sqcap$(`Label n` $\rightarrow$ `Bool`))$\sqcup$((`Label k` $\rightarrow$ `String`)$\sqcap$(`Label m` $\rightarrow$ `Int`))) $\rightarrow$ `Int`. Since the label type itself is also monotype, `var ex13 = (x => x.m)({m: 1})` can also be accepted without any annotations.

This extension is also equipped with a sound and complete algorithm. It should be decidable as well, with a simple modification to our decidability proof of $F^e_{\sqcup\sqcap}$. Our implementation for this type system additionally accepts all the expressions in the first example in the previous section.

$F^e_{\sqcup\sqcap}$ with record and intersection/union type inference. We also study a variant where the intersection and union of monotypes are considered monotypes. The bidirectional type system itself still has various desirable properties. With this extended monotype definition, more programs can be type-checked. But our greedy algorithm cannot be complete in this case: it cannot infer *all* such monotypes. The incompleteness means that the algorithmic system may not enjoy all the properties of the bidirectional type system, but it is more aligned with the practice of TypeScript and does infer more programs which reduces the burden of the programmer. For instance, it is not hard to find a counter-example of subtyping transitivity, exploiting the incompleteness of greedy instantiation, for both this extended algorithmic type system and TypeScript.

```
var f14: <A>(x: A) => (y: A) => number = x => y => 1
var g14: (x: number | boolean) => (y: number | boolean) => number = f14
var h14_1: (x: number) => (y: boolean) => number = g14
var h14_2: (x: number) => (y: boolean) => number = f14 // rejected!
```

This system strictly infers more types than the base $F^e_{\sqcup\sqcap}$, at the cost of more backtracking, meaning all monotype instantiation without intersection and union types are guaranteed to be found. So, it still accepts `ex8_2`, where `Int` is a valid instantiation. Our implementation for this type system additionally accepts `ex8_4`, by picking `Int` $\sqcup$ `Bool` as the instantiation. A limitation of the type system is that it does not infer any impredicative instantiation, i.e. a polymorphic type. Thus `ex15` is accepted by TypeScript but rejected by this type system.

```
function f15(x: (_:<A>(_:A)=>A) => number) : number {return 1}
function h15(x:((_:<A>(_:A)=>A) => (<A>(_:A)=>A))) : number {return 1}
var ex15 = f15(h15)
```

We do not have the decidability proof for this system either, and it seems to require a new proof technique since intersection and union types can be introduced by solving the instantiation. Nonetheless, we believe that our incomplete algorithm is still terminating.

Technical innovation. Besides the investigation of the type systems, our work also provides the following technical innovations.

Syntax-directed transfer and defunctionalized continuation representation. We develop a new syntax-directed transfer relation to relate the bidirectional type system and its algorithmic version for the soundness and completeness proof. This transfer is built upon a defunctionalized representation of continuations so that they can be inductively defined independently. These new techniques not only reduce substitution operations but also encode stronger invariants of the system and simplify the formal reasoning: e.g. the continuation and the whole continuation chain must be of certain shapes; the corresponding work in the bidirectional type system and the algorithmic worklist must be of the same kind, etc. Syntax-directed transfer is discussed in Section 5 and the details of defunctionalization is introduced in Appendix B.4.

Decidability proof, which gets very intricate due to intersections and unions (due to a duplication problem and intersection introduction). The measure definitions are highly non-trivial as they now incorporate multiplication operations and recursion over the entire chain of continuations. We believe our work proves the feasibility of decidability proofs for worklist-based algorithms with complex branching, and provides some general recipes in designing measures.

Coq formalization. Most previous mechanically formalized type inference algorithms based on worklists [Cui et al. 2023; Zhao and Oliveira 2022; Zhao et al. 2019] choose Abella [Gacek 2008] as the proof assistant. As mentioned by Cui et al. [2023], the lack of proof automation in Abella has already outweighed the benefits of the built-in support of the abstract binding tree. For better reusability and scalability, the formalization of $F^e_{\sqcup\sqcap}$ starts fresh using Coq. We build a new framework based on locally nameless representation [Charguéraud 2012] for worklist reasoning. Since $F^e_{\sqcup\sqcap}$ has complex reduction behavior like gathering results from multiple branches, we believe this framework could scale well to the formalization of a wide range of systems. Coq also serves as a more recognized trust base compared to Abella.

3 BIDIRECTIONAL TYPE SYSTEM

This section introduces a bidirectional type system for $F^e_{\sqcup\sqcap}$, which serves as a specification for the algorithmic version that will be presented in Section 4. The type system extends Zhao and Oliveira [2022]'s $F^e_{\leq}$ type system by adding *intersection and union* types and inherits several general improvements in the $F^b_{\leq}$ type system [Cui et al. 2023], which adds bounded quantification to $F^e_{\leq}$. This type system enjoys several desirable properties, including subtyping transitivity and subsumption.

3.1 Syntax and Well-formedness

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

The syntax for $F^e_{\sqcup\sqcap}$ is shown above. $F^e_{\sqcup\sqcap}$ includes usual types such as the unit type, type variables a, universal types ($\forall a.A$), function types ($A \to B$) and the top ($\top$) and bottom ($\bot$) type. In addition $F^e_{\sqcup\sqcap}$ also has subtype variables and intersection ($A \sqcap B$) and union ($A \sqcup B$) types. Subtype variables are needed to support order-relevant quantifiers and intersection and union types are our new extensions. The expression syntax is standard, supporting variables (x), unit value ($()$), abstractions ($\lambda x.e$), annotations ($e : A$), explicit type applications ($e\ @A$) and type abstractions ($\Lambda a.\ e : A$). The support of intersection and union types in $F^e_{\sqcup\sqcap}$ is first-class: they can appear in any part of the type and can be used in annotations for any expressions.

Uniform type-level variable representation. $F^e_{\sqcup\sqcap}$ still has the concept of subtype variables, to implement order-relevant quantifiers. However, subtype variables are represented uniformly in types as type variables to simplify reasoning. Type variables and subtype variables are distinguished by their binding in the context: if $a \in \Psi$ then a is a type variable; if $\tilde{a} \in \Psi$ then a is a subtype variable. This means a type must be associated with a context to be interpreted.

Monotypes and greedy instantiation. Since we would like a greedy algorithm that infers all the implicitly instantiable type arguments, we have to restrict the subtyping relation on implicitly instantiable types to be an equivalence relation, i.e. $\tau \leq \sigma$ implies $\tau = \sigma$ [Cui et al. 2023]. This design choice forces us to exclude intersection and union types from monotypes, which are the class of types that can be inferred/instantiated. Otherwise, we can trivially have $\tau := \mathbb{1} \sqcap (\mathbb{1} \to \mathbb{1})$ and $\sigma := \mathbb{1}$, breaking such property. So the monotypes in $F^e_{\sqcup\sqcap}$ still consists of 3 cases: unit type, type variables (i.e., $a \in \Psi$), and function types of monotypes.

Even if we could have taken a non-greedy approach, introducing intersection, union, top and bottom type in the solution domain is problematic. This would lead us to the *non-structural subtyping entailment problem*, where types with different shapes can be related, bringing fundamental obstacles for bounding the depth of substitutions via any kind of standard occurs-check. Specifically, the decidability of non-structural subtyping entailment both for top and bottom types [Su et al. 2002] and for intersection types [Dudenhefner et al. 2016] remains an open problem.

Well-formedness. Well-formedness ensures that all references to type variables are valid from the context. To retain the subsumption property under explicit type application, besides order relevant quantifiers, we also have to ensure that the universal type $\forall a.\ A$ and type abstraction $\Lambda a.\ e : A$ are indeed polymorphic, i.e., a is actually used by A. However, after adding intersection and union types, the free-variable check also requires an update. Specifically, for intersection, we need the variable to appear in both branches, as formulated by the following rules.

$$\frac{}{a \in^s a} \qquad \frac{a \in^s A}{a \in^s \forall b.\ A} \qquad \frac{a \in^s A}{a \in^s A \to B} \qquad \frac{a \in^s B}{a \in^s A \to B} \qquad \frac{a \in^s A \quad a \in^s B}{a \in^s A \sqcap B} \qquad \frac{a \in^s A}{a \in^s A \sqcup B} \qquad \frac{a \in^s B}{a \in^s A \sqcup B}$$

Otherwise, with the original free variable check and allowing types like $\forall a.\forall b.(a \sqcap b)$, the stability under polytype substitutions is broken. We can modify the example with an unused type variables by Zhao and Oliveira, $\forall a.\forall b.a \leq \forall a.a$ to $\forall a.\forall b.(a \sqcap b) \leq \forall a.a$ where both a and b appear in the body of the polymorphic type. By explicitly applying $\forall a.a \to a$ twice, we will first get $\forall b.(\forall a.a \to a \sqcap b) \leq \forall a.a \to a$, and then $(\forall a.a \to a) \sqcap (\forall a.a \to a) \leq (\forall a.a \to a) \to (\forall a.a \to a)$. The last subtyping fails since it would require impredicative instantiation. The intuition behind this stronger free variable check is that the subtyping of intersection types may completely discard one branch. Thus, we need to be conservative enough so that no matter which branch remains, a should be in it. Due to the asymmetry of $\leq\forall$ and $\leq\forall$L, union types can use the normal free variable check.

3.2 Subtyping

Figure 2 shows the rules of subtyping relation. The basic rules are standard, including rules $\leq\to$, $\leq\top$, and $\leq\bot$. Rules $\leq\sqcap$R, $\leq\sqcap\text{L}_1$, $\leq\sqcap\text{L}_2$, $\leq\sqcup$L, $\leq\sqcup\text{R}_1$, $\leq\sqcup\text{R}_2$ are newly added to support subtyping in the presence of intersection and union types. These six rules are standard. The remaining rules are key to supporting universal types. Rule $\leq\forall$L and $\leq\forall$ deal with HRP. Rule $\leq\forall$L states that a universal type is a subtype of another (non-universal) type B as long as the subtyping relation holds after instantiating the universal type with a monotype τ. Rule $\leq\forall$ states that two universal types are subtypes if their bodies are subtypes. The two rules dealing with universal types ensure that the order of the universal quantifiers is relevant [Eisenberg et al. 2016a].

The interaction between higher-rank polymorphism and intersection and union types requires a careful treatment of the prioritization between rule $\leq\forall$ and $\leq\forall$L. In previous systems with these two rules [Cui et al. 2023; Zhao and Oliveira 2022], a simple syntactic check "$B \neq \forall.*$" in rule $\leq\forall$L ensures that rule $\leq\forall$ always takes priority when both sides are universal types (this prioritization

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

$$\frac{}{\Psi \vdash \mathbb{1} \leq \mathbb{1}} \leq\mathbb{1} \qquad \frac{}{\Psi \vdash A \leq \top} \leq\top \qquad \frac{}{\Psi \vdash \bot \leq A} \leq\bot \qquad \frac{}{\Psi \vdash a \leq a} \leq\mathsf{TVar}$$

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

$$\frac{\Psi \vdash A \leq B_1 \quad \Psi \vdash A \leq B_2}{\Psi \vdash A \leq B_1 \sqcap B_2} \leq\sqcap\mathsf{R} \qquad \frac{\Psi \vdash A_1 \leq B}{\Psi \vdash A_1 \sqcap A_2 \leq B} \leq\sqcap\mathsf{L}_1 \qquad \frac{\Psi \vdash A_2 \leq B}{\Psi \vdash A_1 \sqcap A_2 \leq B} \leq\sqcap\mathsf{L}_2$$

$$\frac{\Psi \vdash A_1 \leq B \quad \Psi \vdash A_2 \leq B}{\Psi \vdash A_1 \sqcup A_2 \leq B} \leq\sqcup\mathsf{L} \qquad \frac{\Psi \vdash A \leq B_1}{\Psi \vdash A \leq B_1 \sqcup B_2} \leq\sqcup\mathsf{R}_1 \qquad \frac{\Psi \vdash A \leq B_2}{\Psi \vdash A \leq B_1 \sqcup B_2} \leq\sqcup\mathsf{R}_2$$

Fig. 2. Subtyping

is essential to support explicit type application). This check becomes inappropriate in the presence of intersection and union types, because it treats $(\forall a.\ A) \sqcap (\forall a.\ A)$ and $\forall a.\ A$ differently while these two types should be considered equivalent. Adopting the same syntactic check would break the transitivity of subtyping as the following counterexample demonstrates:

- $A \leq B : \forall a.\ \forall b.\ b \to a \leq (\forall a.\ a \to \mathbb{1}) \sqcap (\forall a.\ a \to \mathbb{1})$
- $B \leq C : (\forall a.\ a \to \mathbb{1}) \sqcap (\forall a.\ a \to \mathbb{1}) \leq (\forall a.\ a \to \mathbb{1})$
- $A \nleq C : \forall a.\ \forall b.\ b \to a \nleq \forall a.\ a \to \mathbb{1}$

This counterexample is because $A \leq B$ can use the $\leq\forall$L while $A \leq C$ cannot: the outer intersection over the inner universal type in B bypasses the syntactic check. A similar situation also happens when RHS contains some union type (e.g., change B to $(\forall a.\ a \to \mathbb{1}) \sqcup (\forall a.\ a \to \mathbb{1})$) and when B is not idempotent to C (e.g., change B to $(\forall a.\ a \to \mathbb{1}) \sqcup \bot$). The direct solution to avoid such a bypass is to always prioritize the rules for intersection and union types (rules $\leq\sqcap$R, $\leq\sqcup\mathsf{R}_1$ and $\leq\sqcup\mathsf{R}_2$) over rule $\leq\forall$L when LHS is a $\forall$ and RHS is an intersection or union type, leading to the following syntactic check for rule $\leq\forall$L.

$$\frac{\Psi \vdash \tau \quad \Psi[\tau/a]A \leq B \quad B \neq \forall.* \text{ or } * \sqcap * \text{ or } * \sqcup *}{\Psi \vdash \forall a.\ A \leq B}$$

This version of rule $\leq\forall$L indeed yields a subtyping relation with reflexivity, transitivity, and stability under polytype instantiation. However, it rejects many reasonable examples where we should expect the implicit instantiation to work, e.g. $\forall a.\ (a \to \mathbb{1}) \sqcup (a \to (\mathbb{1} \to \mathbb{1})) \leq (\mathbb{1} \to \mathbb{1}) \sqcup (\mathbb{1} \to (\mathbb{1} \to \mathbb{1}))$. To solve this deficiency of expressive power, we employ a syntactic check $B^{\neq\forall}$ that examines deeply into the type structure of B. This check requires both branches of intersection types and either branch of union types to be not a universal type, bottom type, or subtype variable. The exclusion of universal type and bottom type is crucial for transitivity, and the exclusion of subtype variables is necessary for stability under polytype instantiation. The complete rules are:

$$\frac{}{\mathbb{1}^{\neq\forall}} \qquad \frac{}{\top^{\neq\forall}} \qquad \frac{a \in \Psi}{a^{\neq\forall}} \qquad \frac{}{A \to B^{\neq\forall}} \qquad \frac{A_1^{\neq\forall} \quad A_2^{\neq\forall}}{A_1 \sqcap A_2^{\neq\forall}} \qquad \frac{A_1^{\neq\forall}}{A_1 \sqcup A_2^{\neq\forall}} \qquad \frac{A_2^{\neq\forall}}{A_1 \sqcup A_2^{\neq\forall}}$$

This syntactic check enjoys a good semantic interpretation that if $A^{\neq\forall}$ then forall $\tilde{a} \in \Psi, B, C$, $\Psi \vdash [C/a]A \nleq \forall a.\ B$. Thus, $\leq\forall$L with this syntactic check is strictly more expressive than $F^e_\leq$ or the simpler syntactic check discussed earlier. Another subtle interaction that happens between universal types and intersection types is that a universal quantifier could be instantiated multiple times when the RHS is an intersection type. For instance, the following example is accepted: $\forall a.\ (a \to \mathbb{1}) \sqcap (a \to (\mathbb{1} \to \mathbb{1})) \leq (\mathbb{1} \to \mathbb{1}) \sqcap ((\mathbb{1} \to \mathbb{1}) \to (\mathbb{1} \to \mathbb{1}))$.

$\boxed{\Psi \vdash e \Leftrightarrow A}$ *e* is checked against/infers *A*

$$\frac{\Psi, x : A \vdash e \Leftarrow B}{\Psi \vdash \lambda x.\ e \Leftarrow A \to B} \Leftarrow\!\to \qquad \frac{\Psi, x : \bot \vdash e \Leftarrow \top}{\Psi \vdash \lambda x.\ e \Leftarrow \top} \Leftarrow\!\to\!\top \qquad \frac{\Psi \vdash e \Rightarrow A \quad \Psi \vdash A \le B}{\Psi \vdash e \Leftarrow B} \Leftarrow\mathsf{Sub}$$

$$\frac{\Psi \vdash e \Leftarrow A \quad \Psi \vdash e \Leftarrow B}{\Psi \vdash e \Leftarrow A \sqcap B} \Leftarrow\!\sqcap \qquad \frac{\Psi \vdash e \Leftarrow A}{\Psi \vdash e \Leftarrow A \sqcup B} \Leftarrow\!\sqcup_1 \qquad \frac{\Psi \vdash e \Leftarrow B}{\Psi \vdash e \Leftarrow A \sqcup B} \Leftarrow\!\sqcup_2$$

$$\frac{(x : A) \in \Psi}{\Psi \vdash x \Rightarrow A} \Rightarrow\mathsf{Var} \qquad \frac{\Psi \vdash e \Leftarrow A}{\Psi \vdash (e : A) \Rightarrow A} \Rightarrow\mathsf{Anno} \qquad \frac{\Psi \vdash \sigma \to \tau \quad \Psi, x : \sigma \vdash e \Leftarrow \tau}{\Psi \vdash \lambda x.\ e \Rightarrow \sigma \to \tau} \Rightarrow\!\to\!\mathsf{Mono}$$

$$\frac{}{\Psi \vdash () \Rightarrow \mathbb{1}} \Rightarrow\mathtt{Unit} \qquad \frac{\Psi, a \vdash e \Leftarrow A}{\Psi \vdash \Lambda a.\ e : A \Rightarrow \forall a.\ A} \Rightarrow\Lambda$$

$$\frac{\Psi \vdash e_1 \Rightarrow A \quad \Psi \vdash A \triangleright B \to C \quad \Psi \vdash e_2 \Leftarrow B}{\Psi \vdash e_1\ e_2 \Rightarrow C} \Rightarrow\mathsf{App} \qquad \frac{\Psi \vdash e \Rightarrow A \quad \Psi \vdash A \circ B \Rrightarrow C}{\Psi \vdash e\ @B \Rightarrow C} \Rightarrow\mathsf{TApp}$$

$\boxed{\Psi \vdash A \triangleright B \to C}$ A matches an arrow type $B \to C$

$$\frac{}{\Psi \vdash A \to B \triangleright A \to B} \triangleright\!\to \qquad \frac{}{\Psi \vdash \bot \triangleright \top \to \bot} \triangleright\bot \qquad \frac{\Psi \vdash \tau \quad \Psi \vdash ([\tau/a]A) \triangleright B \to C}{\Psi \vdash \forall a.\ A \triangleright B \to C} \triangleright\forall$$

$$\frac{\Psi \vdash A_1 \triangleright B \to C}{\Psi \vdash (A_1 \sqcap A_2) \triangleright B \to C} \triangleright\sqcap_1 \qquad \frac{\Psi \vdash A_2 \triangleright B \to C}{\Psi \vdash (A_1 \sqcap A_2) \triangleright B \to C} \triangleright\sqcap_2 \qquad \frac{\Psi \vdash A_1 \triangleright B_1 \to C_1 \quad \Psi \vdash A_2 \triangleright B_2 \to C_2}{\Psi \vdash (A_1 \sqcup A_2) \triangleright (B_1 \sqcap B_2) \to (C_1 \sqcup C_2)} \triangleright\sqcup$$

$\boxed{\Psi \vdash A \circ B \Rrightarrow C}$ A type-applied to B infers C

$$\frac{}{\Psi \vdash \forall a.\ A \circ B \Rrightarrow [B/a]A} \circ\!\Rrightarrow\!\forall \qquad \frac{}{\Psi \vdash \bot \circ A \Rrightarrow \bot} \circ\!\Rrightarrow\!\bot \qquad \frac{\Psi \vdash A_1 \circ B \Rrightarrow C}{\Psi \vdash (A_1 \sqcap A_2) \circ B \Rrightarrow C} \circ\!\Rrightarrow\!\sqcap_1$$

$$\frac{\Psi \vdash A_2 \circ B \Rrightarrow C}{\Psi \vdash (A_1 \sqcap A_2) \circ B \Rrightarrow C} \circ\!\Rrightarrow\!\sqcap_2 \qquad \frac{\Psi \vdash A_1 \circ B \Rrightarrow C_1 \quad \Psi \vdash A_2 \circ B \Rrightarrow C_2}{\Psi \vdash (A_1 \sqcup A_2) \circ B \Rrightarrow (C_1 \sqcup C_2)} \circ\!\Rrightarrow\!\sqcup$$

Fig. 3. Checking, Inference, Matching and Type Application

Metatheory. With such design, reflexivity, and transitivity hold for this subtyping relation. As shown below, stability under polytype-instantiation (Theorem 2.1) holds as well. Some proof details, including the generalized theorems to conduct inductions, are discussed in Section 5.

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

Theorem 3.2 (Subtyping Transitivity). *Given all contexts and types well-formed, if* $\Psi \vdash A \le B$, *and* $\Psi \vdash B \le C$ *then* $\Psi \vdash A \le C$.

3.3 Typing

Figure 3 shows the bidirectional type system with the checking and inference judgments. Variables, annotations, unit value, type abstractions are directly inferable, and they are dealt with by rules $\Rightarrow$Var, $\Rightarrow$Anno, $\Rightarrow$`Unit` and $\Rightarrow\Lambda$ respectively. We also allow the inference of abstractions as long as their types are monotypes, as in rule $\Rightarrow\to$Mono. The inference of application and type application is delegated to two modular judgments, matching and type application. Checking judgments allow abstractions to be checked by a function type ($\Leftarrow\to$) or $\top$ ($\Leftarrow\to\top$). An expression can also be checked against a supertype, if it infers a subtype ($\Leftarrow$Sub). $\Leftarrow\sqcap$, $\Leftarrow\sqcup_1$, and $\Leftarrow\sqcup_2$ are the checking introduction rules of intersection and union types. Rule $\Leftarrow\sqcap$ allows a term that has uniform behavior over several types to be given an intersection type of all these types, as

$\lambda x.x : (\mathbb{1} \to \mathbb{1}) \sqcap ((\mathbb{1} \to \mathbb{1}) \to (\mathbb{1} \to \mathbb{1}))$. Rules for union introduction are needed since the subsumption rule $\Leftarrow$Sub does not cover the case when e cannot infer either branch.

Matching. The matching judgment $\Psi \vdash A \triangleright B \to C$ in Figure 3 states that a type A can be regarded as a subtype of function type $B \to C$, so that a term of type A can be applied to another term of type B and the result will be of type C. We adopt this design from $F^b_{\leq}$ [Cui et al. 2023], which uses a similar judgment to deal with HRP with bounded quantification. Rules $\triangleright\!\to$, $\triangleright\bot$ are the base cases, since a function type or $\bot$ can be regarded as a function type directly. Rule $\triangleright\forall$ converts a universal type to a function type by guessing a monotype τ for instantiation and continuing the conversion to the instantiated body. Since intersection and union types can also be subtypes of function types, we need to extend the matching relation to deal with such cases, as shown in rules $\triangleright\sqcap_1$, $\triangleright\sqcap_2$ and $\triangleright\sqcup$. We cannot output the intersection of both function types, since the intersection of function types cannot be regarded as a function type due to the lack of a distributivity rules between the function type and intersection type in $F^e_{\sqcup\sqcap}$: $(A_1 \to B_1) \sqcap (A_2 \to B_2) \leq (A_1 \sqcup A_2) \to (B_1 \sqcap B_2)$.

Matching can be viewed as a syntax-directed transformation to get a function-type like supertype of A, as formulated by the following lemma.

Lemma 3.3 (Specification of Matching). *Given all contexts and types well-formed,*

(1) if $\Psi \vdash A \triangleright B \to C$, then $\Psi \vdash A \leq B \to C$;
(2) if $\Psi \vdash A \leq B \to C$, then exists B', C' s.t. $\Psi \vdash A \triangleright B' \to C'$ and $\Psi \vdash B' \to C' \leq B \to C$.

Type application. The type-application judgment $\Psi \vdash A \circ B \Rrightarrow C$ in Figure 3 states that a type A can be regarded as a subtype of a universal type so that it can be type-applied to type B, and the result type is C. Rule $\circ\!\Rrightarrow\!\forall$ and $\circ\!\Rrightarrow\!\bot$ are the base cases where the type-application result is known: $[B/a]A$ and $\bot$. Similarly to matching, we also need to extend the type application to deal with intersection and union types. Rule $\circ\!\Rrightarrow\!\sqcap_1$ and $\circ\!\Rrightarrow\!\sqcap_2$ choose one branch of the intersection type to proceed. $\forall a.\ A$ or $\forall a.\ B$ is the best approximation of $\forall a.\ A \sqcap B$ with a universal-type shape since $F^e_{\sqcup\sqcap}$ does not have a distributivity rule between the universal type and intersection type: $\forall a.\ A \sqcap \forall a.\ B \leq \forall a.\ (A \sqcap B)$. Rule $\circ\!\Rrightarrow\!\sqcup$ traverses both branches and combines the result. Type-application can be viewed as a syntax-directed transformation to get a universal-type like supertype of A and type-apply it to B (without considering $\bot$), as formulated by the following lemma. A more general version, where A containing $\bot$ is considered, is shown in Appendix C.

Lemma 3.4 (Specification of Type-Application). *Given all contexts and types well-formed,*

(1) if $A^{\not\bot}$ and $\Psi \vdash A \circ B \Rrightarrow C$, then exists A', $\Psi \vdash A \leq \forall a.A'$, $C = [B/a]A'$;
(2) if $\Psi \vdash A \leq \forall a.\ A'$, then exists C s.t. $\Psi \vdash A \circ B \Rrightarrow C$ and $\Psi \vdash C \leq [B/a]A'$.

4 ALGORITHMIC SYSTEM

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

4.1 Syntax

The syntax of the algorithmic system is shown below. A new type of variables, existential variables, are introduced as placeholders for unknown implicit arguments and will finally be solved to a concrete monotype. Existential variables themselves are also monotypes. Type variables, subtype variables, and existential variables are represented uniformly at the type level and distinguished by their bindings in the algorithmic worklist, $a, \tilde{a}, \widehat{a}$, respectively. Due to this uniform representation, algorithmic types have the same syntax as declarative types. The expression syntax remains unchanged as well. Works w are judgments to process. Two new works are added to $F^e_{\sqcup\sqcap}$ compared with $F^b_\leq$: union matching $(A_1 \rightarrow B_1) \Cup_{\triangleright} (A_2 \rightarrow B_2) \blacktriangleright \omega$ and union type application $A_1 \Cup_\circ A_2 \blacktriangleright \omega$ to combine the results for union types in matching and type application. Their function will be introduced in detail in rules related to them. A worklist Γ is an *ordered* list of both (type) variable declarations (with bindings) and works.

$$
\begin{array}{llll}
\text{Work} & w & ::= & A \leq B \mid e \Leftarrow A \mid e \Rightarrow_\alpha \omega \mid A \circ B \Rrightarrow_\alpha \omega \mid A \triangleright_{\alpha,\beta} \omega \mid \\
& & & A \rightarrow B \bullet e \Rrightarrow_\alpha \omega \mid (A_1 \rightarrow B_1) \Cup_{\triangleright} (A_2 \rightarrow B_2) \blacktriangleright_\alpha \omega \mid \\
& & & A_1 \Cup_\circ A_2 \blacktriangleright_\alpha \omega \\
\text{Algorithmic worklist} & \Gamma & ::= & \cdot \mid \Gamma, a \mid \Gamma, \tilde{a} \mid \Gamma, \widehat{a} \mid \Gamma, x : A \mid \Gamma \Vdash \omega
\end{array}
$$

Unlike previous work [Cui et al. 2023; Zhao and Oliveira 2022; Zhao et al. 2019], we present the continuation-passing style using a higher-order abstract syntax rather than the substitution-based syntax. The first difference is that continuations are given a new syntactic symbol ω, which is a syntactic sugar for $A \supset w$, i.e., a meta function from type(s) to work. The argument for a continuation is represented with Greek letters α, β, γ. The continuation application is now presented as $\omega \diamond A$. This representation is more aligned with our formalization and more details are covered in the appendix. $\Gamma \vdash A$, $\Gamma \vdash e$, $\Gamma \vdash w$, and $\vdash \Gamma$ denote the well-formedness of each syntactic category, and $\Gamma \longrightarrow^* \Gamma'$ denotes the worklist reduction.

4.2 Algorithmic Rules

All the reduction rules are defined in a single relation but, for clarity of presentation, we separate them into three parts: garbage collection, subtyping, and typing. The scoping mechanism of the worklist ensures that variables can never be referred to by any entries that appear before them. Thus, it is safe to remove them if they are the last entry in the worklist.

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

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

Subtyping (Rules 5-21, Figure 4). These 19 rules can be classified into two categories, where the first contains rules 5-17, and the second contains rules 18-21. Most rules in the first category are similar to their declarative counterparts. Rule 6 also deals with the reflexivity of existential variables. The most significant changes are in rule 10, where an existential variable $\widehat{a}$ is introduced instead of guessing the monotype τ instantiation in its declarative counterpart rule $\leq\forall$L. We abuse the notation for the side condition $B^{\neq\forall}$ as it is now defined for algorithmic types. The check is almost the same with one new base case $\frac{\widehat{a} \in \Gamma}{a^{\neq\forall}}$. Rules 12-17 are new, but they are aligned with their declarative counterparts. In rules 9, 12, and 15, multiple new entries are pushed back to the worklist, while their declarative counterparts check each new entry separately.

The rules in the second category solve the existential variables, i.e., substituting the existential variable with the monotype found. We adopt the polytype splitting technique [Cui et al. 2023], which solves an existential variable to an arbitrary monotype (instead of just base monotypes like a and $\mathbb{1}$) and only splits it to two fresh existential variables when it is compared with a polymorphic function type (instead of an arbitrary function type). The *occurs-check* condition in rules 18 and 19 prevents the possible non-termination of the algorithm caused by judgments like $\widehat{a} \leq 1 \rightarrow \widehat{a}$.

Compared to the original polytype splitting rule used by Cui et al., we find that it is unnecessary (in the sense of keeping the decidability of the algorithm) to add the occurs-check condition in rule 20 and 21. This helps pre-rejecting incorrect subtyping relations in earlier type systems, but the same naive occurs-check ($a \notin \mathrm{fv}(A \rightarrow B)$) would cause the loss of completeness in $F^e_{\sqcup\sqcap}$. For instance, $a \leq (a \sqcap \mathbb{1}) \rightarrow \mathbb{1}, \widehat{a} \in \Gamma$ would be rejected but $\mathbb{1} \rightarrow \mathbb{1}$ is a valid solution for a.

$\boxed{\Gamma \longrightarrow \Gamma'}$ Γ worklist-reduces to Γ'

$$
\begin{array}{rcll}
\Gamma \Vdash \mathbb{1} \le \mathbb{1} & \longrightarrow_5 & \Gamma & \\
\Gamma \Vdash a \le a & \longrightarrow_6 & \Gamma & \\
\Gamma \Vdash A \le \top & \longrightarrow_7 & \Gamma & \\
\Gamma \Vdash \bot \le A & \longrightarrow_8 & \Gamma & \\
\Gamma \Vdash A_1 \to A_2 \le B_1 \to B_2 & \longrightarrow_9 & \Gamma \Vdash B_1 \le A_1 \Vdash A_2 \le B_2 & \\
\Gamma \Vdash \forall a.\ A \le B & \longrightarrow_{10} & \Gamma, \widehat{a} \Vdash A \le B & \text{when } B^{\neq\forall} \\
\Gamma \Vdash \forall a.\ A \le \forall a.\ B & \longrightarrow_{11} & \Gamma, \tilde{a} \Vdash A \le B & \\
\Gamma \Vdash A \le B_2 \sqcap B_2 & \longrightarrow_{12} & \Gamma \Vdash A \le B_1 \Vdash A \le B_2 & \\
\Gamma \Vdash A_1 \sqcap A_2 \le B & \longrightarrow_{13} & \Gamma \Vdash A_1 \le B & \\
\Gamma \Vdash A_1 \sqcap A_2 \le B & \longrightarrow_{14} & \Gamma \Vdash A_2 \le B & \\
\Gamma \Vdash A_1 \sqcup A_2 \le B & \longrightarrow_{15} & \Gamma \Vdash A_1 \le B \Vdash A_2 \le B & \\
\Gamma \Vdash A \le B_1 \sqcup B_2 & \longrightarrow_{16} & \Gamma \Vdash A \le B_1 & \\
\Gamma \Vdash A \le B_1 \sqcup B_2 & \longrightarrow_{17} & \Gamma \Vdash A \le B_2 & \\
\Gamma \Vdash a \le \tau & \longrightarrow_{18} & \{\tau/a\}\Gamma & \text{when } \widehat{a} \in \Gamma \land a \notin \mathrm{fv}(\tau) \\
\Gamma \Vdash \tau \le a & \longrightarrow_{19} & \{\tau/a\}\Gamma & \text{when } \widehat{a} \in \Gamma \land a \notin \mathrm{fv}(\tau) \\
\Gamma \Vdash a \le A \to B & \longrightarrow_{20} & \{a_1 \to a_2/a\}(\Gamma, \widehat{a}_1, \widehat{a}_2, a \le A \to B) & \\
 & & & \text{when } \widehat{a} \in \Gamma \land \Gamma \nvdash^m A \to B \\
\Gamma \Vdash A \to B \le a & \longrightarrow_{21} & \{a_1 \to a_2/a\}(\Gamma, \widehat{a}_1, \widehat{a}_2, A \to B \le a) & \\
 & & & \text{when } \widehat{a} \in \Gamma \land \Gamma \nvdash^m A \to B
\end{array}
$$

Fig. 4. Algorithmic Worklist Reduction (Subtyping)

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

$$
\begin{array}{lll}
\{\tau/\widehat{a}\}(\Gamma, \widehat{a}) & =_1 \Gamma & \\
\{\tau/\widehat{a}\}(\Gamma, \widehat{\beta}) & =_2 \{\tau/\widehat{a}\}\Gamma, \widehat{\beta} & \text{when } \widehat{\beta} \notin FV(\tau) \\
\{\tau/\widehat{a}\}(\Gamma_1, \widehat{a}, \Gamma_2, \widehat{\beta}) & =_3 \{\tau/\widehat{a}\}(\Gamma_1, \widehat{\beta}, \widehat{a}, \Gamma_2) & \text{when } \widehat{\beta} \in FV(\tau) \\
\{\tau/\widehat{a}\}(\Gamma, b) & =_4 \{\tau/\widehat{a}\}\Gamma, b & \text{when } b \notin FV(\tau) \\
\{\tau/\widehat{a}\}(\Gamma, \tilde{b}) & =_5 \{\tau/\widehat{a}\}\Gamma, \tilde{b} & \\
\{\tau/\widehat{a}\}(\Gamma, x : A) & =_6 \{\tau/\widehat{a}\}\Gamma, x : [\tau/\widehat{a}]A & \\
\{\tau/\widehat{a}\}(\Gamma \Vdash w) & =_7 \{\tau/\widehat{a}\}\Gamma \Vdash [\tau/\widehat{a}]w &
\end{array}
$$

Fig. 5. Worklist Substitution

The substitution operations in these 4 rules are encapsulated in the worklist substitution $\{\tau/\widehat{a}\}\Gamma$. The $\{\tau/\widehat{a}\}\Gamma$ operation not only substitutes every occurence of $\widehat{a}$ to τ in Γ and removes $\widehat{a}$, but also performs necessary reordering of other existential variables. Reordering is needed since the monotype may contain some existential variables that originally appear after the target $\widehat{a}$. To keep the well-formedness of the worklist, we need to move these referred existential variables in front of the target. Note this process always narrows the scope of existential variables but never widens it. The concrete process of worklist substitution is shown in Figure 5.

Typing (Rules 22-50, Figure 6). These 29 rules can be further split into 4 categories. Rules 22-28 for checking , rules 29-35 for inference, rules 36-42 for matching and rules 45-50 for type application.

$\boxed{\Gamma \longrightarrow \Gamma'}$ Γ worklist-reduces to Γ'

$$
\begin{array}{rll}
\Gamma \Vdash e \Leftarrow B & \longrightarrow_{22} \Gamma \Vdash e \Rightarrow_\alpha \alpha \leq B & \\
\Gamma \Vdash \lambda x.\, e \Leftarrow A \to B & \longrightarrow_{23} \Gamma, x : A \Vdash e \Leftarrow B & \\
\Gamma \Vdash \lambda x.\, e \Leftarrow \top & \longrightarrow_{24} \Gamma, x : \bot \Vdash e \Leftarrow \top & \\
\Gamma \Vdash e \Leftarrow A_1 \sqcap A_2 & \longrightarrow_{25} \Gamma \Vdash e \Leftarrow A_1 \Vdash e \Leftarrow A_2 & \\
\Gamma \Vdash e \Leftarrow A_1 \sqcup A_2 & \longrightarrow_{26} \Gamma \Vdash e \Leftarrow A_1 & \\
\Gamma \Vdash e \Leftarrow A_1 \sqcup A_2 & \longrightarrow_{27} \Gamma \Vdash e \Leftarrow A_2 & \\
\Gamma \Vdash \lambda x.\, e \Leftarrow a & \longrightarrow_{28} \{a_1 \to a_2/a\}(\Gamma, \widehat{a}_1, \widehat{a}_2 \Vdash \lambda x.e \Leftarrow a) \text{when } \widehat{a} \in \Gamma & \\
\Gamma \Vdash x \Rightarrow_\alpha \omega & \longrightarrow_{29} \Gamma \Vdash \omega \diamond A & \text{when } x : A \in \Gamma \\
\Gamma \Vdash e : A \Rightarrow_\alpha \omega & \longrightarrow_{30} \Gamma \Vdash \omega \diamond A \Vdash e \Leftarrow A & \\
\Gamma \Vdash (\Lambda a.\, e : A) \Rightarrow_\alpha \omega & \longrightarrow_{31} \Gamma \Vdash \omega \diamond (\forall a.\, A), a \Vdash e \Leftarrow A & \\
\Gamma \Vdash () \Rightarrow_\alpha \omega & \longrightarrow_{32} \Gamma \Vdash \omega \diamond \mathbb{1} & \\
\Gamma \Vdash e_1\ e_2 \Rightarrow_\alpha \omega & \longrightarrow_{33} \Gamma \Vdash e_1 \Rightarrow_\beta (\beta \triangleright_{\gamma_1} (\gamma_1 \to \gamma_2 \bullet e_2 \Rrightarrow_\alpha \omega)[\gamma_2]) & \\
\Gamma \Vdash e\ @A \Rightarrow_\alpha \omega & \longrightarrow_{34} \Gamma \Vdash e \Rightarrow_\beta (\beta \circ A \Rrightarrow_\alpha \omega) & \\
\Gamma \Vdash \lambda x.\, e \Rightarrow_\alpha \omega & \longrightarrow_{35} \Gamma, \widehat{a}_1, \widehat{a}_2 \Vdash \omega \diamond (a_1 \to a_2), x : a_1 \Vdash e \Leftarrow a_2 & \\
\Gamma \Vdash A \to B \triangleright_{\alpha,\beta} \omega & \longrightarrow_{36} \Gamma \Vdash \omega \diamond A \diamond B & \\
\Gamma \Vdash \bot \triangleright_{\alpha,\beta} \omega & \longrightarrow_{37} \Gamma \Vdash \top \to \bot \triangleright_{\alpha,\beta} \omega & \\
\Gamma \Vdash \forall a.\, A \triangleright_{\alpha,\beta} \omega & \longrightarrow_{38} \Gamma, \widehat{a} \Vdash A \triangleright_{\alpha,\beta} \omega & \\
\Gamma \Vdash A_1 \sqcap A_2 \triangleright_{\alpha,\beta} \omega & \longrightarrow_{39} \Gamma \Vdash A_1 \triangleright_{\alpha,\beta} \omega & \\
\Gamma \Vdash A_1 \sqcap A_2 \triangleright_{\alpha,\beta} \omega & \longrightarrow_{40} \Gamma \Vdash A_2 \triangleright_{\alpha,\beta} \omega & \\
\Gamma \Vdash A_1 \sqcup A_2 \triangleright_{\alpha_1,\alpha_2} \omega & \longrightarrow_{41} \Gamma \Vdash A_1 \triangleright_{\beta_1,\beta_2} (A_2 \triangleright_{\gamma_1,\gamma_2} (\beta_1 \to \beta_2 \Cup_{\triangleright} \gamma_1 \to \gamma_2 \blacktriangleright_{\alpha_1,\alpha_2} \omega)) & \\
\Gamma \Vdash a \triangleright_{\alpha,\beta} \omega & \longrightarrow_{42} \{a_1 \to a_2/a\}(\Gamma, \widehat{a}_1, \widehat{a}_2 \Vdash a \triangleright_{\alpha,\beta} \omega) & \text{when } \widehat{a} \in \Gamma \\
\Gamma \Vdash (A_1 \to B_1) \Cup_{\triangleright} (A_2 \to B_2) \blacktriangleright_{\alpha,\beta} \omega & \longrightarrow_{43} \Gamma \Vdash \omega \diamond (A_1 \sqcap A_2) \diamond (B_1 \sqcup B_2) & \\
\Gamma \Vdash A \to B \bullet e \Rrightarrow_\alpha \omega & \longrightarrow_{44} \Gamma \Vdash \omega \diamond B \Vdash e \Leftarrow A & \\
\Gamma \Vdash \forall a.\, A \circ B \Rrightarrow_\alpha \omega & \longrightarrow_{45} \Gamma \Vdash \omega \diamond ([B/a]A) & \\
\Gamma \Vdash \bot \circ A \Rrightarrow_\alpha \omega & \longrightarrow_{46} \Gamma \Vdash \omega \diamond \bot & \\
\Gamma \Vdash A_1 \sqcap A_2 \circ B \Rrightarrow_\alpha \omega & \longrightarrow_{47} \Gamma \Vdash A_1 \circ B \Rrightarrow_\alpha \omega & \\
\Gamma \Vdash A_1 \sqcap A_2 \circ B \Rrightarrow_\alpha \omega & \longrightarrow_{48} \Gamma \Vdash A_2 \circ B \Rrightarrow_\alpha \omega & \\
\Gamma \Vdash A_1 \sqcup A_2 \circ B \Rrightarrow_\alpha \omega & \longrightarrow_{49} \Gamma \Vdash A_1 \circ B \Rrightarrow_{\beta_1} (A_2 \circ B \Rrightarrow_{\beta_2} (\beta_1 \Cup_{\circ} \beta_2 \blacktriangleright \omega)) & \\
\Gamma \Vdash A_1 \Cup_{\circ} A_2 \blacktriangleright \omega & \longrightarrow_{50} \Gamma \Vdash \omega \diamond (A_1 \sqcup A_2) &
\end{array}
$$

Fig. 6. Algorithmic Worklist Reduction (Typing)

Rules 22-24 are the algorithmic counterparts of $\Leftarrow$Sub, $\Leftarrow\to$ and $\Leftarrow\to\top$. Rules 25-27 are new, corresponding to declarative rules $\Leftarrow\sqcap$, $\Leftarrow\sqcup_1$ and $\Leftarrow\sqcup_2$. The premise of the inference judgment in $\Leftarrow$Sub is modified to the continuation-passing style, whose LHS operand is unknown before the inference judgment finishes. Rule 28 is added for existential variables. An existential variable can be used to check an abstraction if it can finally be resolved to a function type, so we substitute it with two fresh existential variables $a_1 \to a_2$ using the worklist substitution.

Rules 29-35 are the algorithmic counterparts of $\Rightarrow$Var, $\Rightarrow$Anno, $\Rightarrow\Lambda$, $\Rightarrow$Unit, $\Rightarrow$App, $\Rightarrow$TApp and $\Rightarrow\to$Mono. Rules 29-32 are the base cases where the type is fully determined from the expression so that we can apply the continuation, waiting for the result to this known type. Rules 30 and 31 also push $e \Leftarrow A$ to the worklist to check the expression e with type A. Rule 33 infers the result of the application by inferring the type of e_1 and creating a matching and an application inference continuation. The matching continuation waits for the inference result and passes its result to the application inference continuation. The inference application, after becoming a work, is processed by rule 44 to check the expression e_2 with the first received result (i.e., the domain type) and pass the second result (i.e., the codomain type) to the original continuation ω. Rule 34 is similar: it infers the type of e and passes the result to the type application continuation. Rule 35 creates two fresh existential variables a_1, a_2 as the placeholder of the function type of the abstraction $\lambda x.\ e$ by applying the ω to $a_1 \rightarrow a_2$ and checking the body e against a_2.

Rules 36-38 are the algorithmic counterparts of $\triangleright\!\rightarrow$, $\triangleright\!\bot$ and $\triangleright\!\forall$. Rules 36 and 37 are two base cases where a function type is known (by lifting $\bot$ to $\top \rightarrow \bot$). The continuation is first applied to the domain, then to the codomain type. The modification of rule 38 of introducing an existential variable $\widehat{a}$ is similar to that of rule 10. Rules 39 and 40 proceed by choosing a branch of the intersection type. Rule 41 is interesting because its declarative counterpart requires matching both branches and then combining the results. The algorithmic rule creates a nested continuation that first matches A_1, and passes the result to the first two arguments (β_1 and β_2) of $\beta_1 \rightarrow \beta_2 \Cup_{\triangleright} \gamma_1 \rightarrow \gamma_2 \blacktriangleright \omega$. Then A_2 is matched and passes the result to the latter two arguments (γ_1 and γ_2) of the continuation. When this continuation is fully applied, rule 43 combines the result by taking the intersection of the domain type and the union of the codomain type. At first glance, the scope management for $A_2 \triangleright \cdots$ is a bit concerning since it sees an extended context due to the process of $A_1 \triangleright \cdots$. However, it does not cause real trouble since no type variables can be introduced by matching. The actual solution domain of existential variables that could be created during $A_2 \triangleright \cdots$ remains correct. Rule 42 is added for existential variables. Since the monotype of the existential variable a must match a function type, we generate two fresh existential variables a_1 and a_2 and replace a with $a_1 \rightarrow a_2$ using worklist substitution. Rule 44 checks whether an expression of the function type $A \rightarrow B$, solved by matching, can be applied to another expression e by adding a checking judgment $e \Leftarrow A$ to the worklist. The result of application B is fed to the continuation.

Rules 45-49 are the algorithmic counterparts of $\circ\!\Rrightarrow\!\forall$, $\circ\!\Rrightarrow\!\bot$, $\circ\!\Rrightarrow\!\sqcap_1$, $\circ\!\Rrightarrow\!\sqcap_2$, and $\circ\!\Rrightarrow\!\sqcup$. Rules 45 and 46 are two base cases where the result of the type application can be fully determined, so we apply the continuation to the known result type. Rule 47 and 48 proceed by choosing a branch of the intersection type. Similar to matching, rule 49 also combines the results of both branches by creating a nested continuation, and the actual combination is done by rule 50. The scope management is not a problem here since type application does not even change the scope. There is no new rule for existential variables because a monotype can never be type-applied.

Metatheory. This algorithmic system is sound and complete with respect to the declarative system. It is also decidable. The formal statement is shown below. Some proof details, including the generalized theorems to conduct inductions, are discussed in more detail in Section 5.

Theorem 4.1 (Soundness and Completeness). *Given $\cdot \vdash e$ and $\cdot \vdash A$, then*

(1) $\cdot \vdash e \Leftarrow A$, iff $\cdot \vDash e \Leftarrow A \longrightarrow^* \cdot$.

(2) $\cdot \vdash e \Rightarrow A$, iff $\cdot \vDash e \Rightarrow_\alpha \alpha \leq \top \longrightarrow^* \cdot$.

Theorem 4.2 (Decidability). *Given $\vdash \Gamma$, it is decidable whether $\Gamma \longrightarrow^* \cdot$ or not.*

5 METATHEORY

This section discusses interesting aspects of the metatheory. We first discuss interesting properties of the bidirectional type system, including properties about subtyping, subsumption, and type safety. Then we discuss the soundness and completeness of the algorithmic type system. For soundness

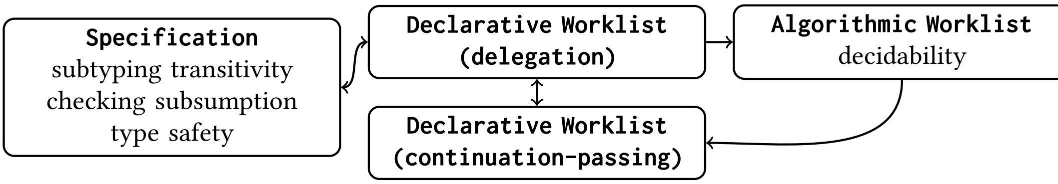

Fig. 7. Overall Proof Structure

and completeness, an important innovation in our work is a new style of transfer relation that connects the algorithmic and the bidirectional type systems. Finally, we discuss decidability, which is subtle due to the branching caused by intersection and union types and requires a complex measure based on a tuple with 9 components. Figure 7 shows the overview of each system and the properties proved for them.

5.1 Type System Properties

Subtyping reflexivity, transitivity and stability under polytype instantiation. The proof of subtyping reflexivity is straightforward by induction on the well-formedness of A. Transitivity is proved by first defining an equivalent step-indexed subtyping relation $\Psi \vdash A \leq B \mid n$ and induction on the sum of steps $(n_1 + n_2)$, and the number of $\forall$ of the middle type B. In the proof, an inductively defined ordinary-type relation B° [Davies and Pfenning 2000; Huang et al. 2021] is useful to decompose intersection and union types to a base type that is no longer an intersection or union type. Stability under polytype instantiation is proved by generalizing the theorem to a substitution-based form.

Lemma 5.1 (Stability under Polytype Substitution). *Let $\Psi := \Psi_1, \tilde{a}, \Psi_2$, given $\vdash \Psi$, $\Psi \vdash A$, $\Psi \vdash B$, and $\Psi_1 \vdash C$, if $\Psi \vdash A \leq B$, then $\Psi_1, [C/a]\Psi_2 \vdash [C/a]A \leq [C/a]B$.*

Subsumption. To prove the subsumption lemma, we first prove the subsumption lemmas for matching and the type application judgment independently. Stability under polytype instantiation is used in the proof of type-application subsumption.

Lemma 5.2 (Matching and Type-Application Subsumption). *Given everything well-formed,*

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

The general subsumption requires two lemmas, for both checking and inference, and a sub-context relation $\Psi' <: \Psi$ that allows variables x to be rebound to a subtype in Ψ'.

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

Then, by induction on the size of A, the size of e and the size of mode ($\Rightarrow$ is 0 and $\Leftarrow$ is 1), we can prove the general subsumption lemma.

Lemma 5.3 (Subsumption). *Given all contexts, expressions, and types wellformed and $\Psi' <: \Psi$,*

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

Type Safety. The type safety of our type system is derived via an elaboration to System F with sum and product types. This elaboration is similar to other elaborations used in the past to prove type safety of calculi with intersection and union types [Dunfield 2014; Pierce 1992]. The elaboration fills all the implicit instantiation and takes a coercive interpretation of each type-level conversion, including subtyping, matching, and type application.

Theorem 5.4 (Type Safety). *Given everything well-formed, if $\Psi \vdash e \Leftrightarrow A \hookrightarrow e'$ then $|\Psi| \vdash_F e' : |A|$*

5.2 Soundness and Completeness

To help formalize the correspondence between the specification and algorithmic systems, we build an intermediate system: declarative worklist [Zhao et al. 2019]. The declarative worklist has a similar syntax to Γ but it does not have existential variables. Similar to the notations in the algorithmic worklist, $\vdash \Omega$, $\Omega \vdash A$, $\Omega \vdash e$ and $\Omega \vdash w$ denote various well-formedness relations.

There are two sets of reduction rules for this intermediate system, declarative reduction $\Omega \longrightarrow \Omega'$ and declarative worklist reduction $\Omega \longrightarrow_{\omega} \Omega'$, for soundness and completeness proofs, respectively. $\Omega \longrightarrow \Omega'$ is a rephrasing of the declarative judgments using the worklist syntax, while $\Omega \longrightarrow_{\omega} \Omega'$ mimics the algorithmic reduction using the continuation-passing style but still guesses the monotype τ instead of introducing existential variables. The detailed rules of well-formedness and reduction of the declarative worklist can be found in the extended version of this paper. $\Omega \longrightarrow^{*} \cdot$ and $\Omega \longrightarrow_{\omega}^{*} \cdot$ are proved equivalent:

Theorem 5.5 (Equivalence of Declarative Reduction). *If $\vdash \Omega$, then $\Omega \longrightarrow^{*} \cdot$ iff $\Omega \longrightarrow_{\omega}^{*} \cdot$.*

To relate the algorithmic and declarative worklist, previous works [Cui et al. 2023; Zhao and Oliveira 2022; Zhao et al. 2019] use the following transfer relation, which relates a worklist Ω with all algorithmic worklists Γ if Γ is equal to Ω under a substitution of all existential variables in Γ to a well-formed declarative monotype.

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

This transfer relation is too declarative. Thus, it requires a large number of inversion lemmas to reason about the correspondence between the shape of type, expression, and work of two related worklists Γ and Ω (e.g. if the last work in Ω is a subtyping judgment, so should be for Γ). It also erases the original structure of the algorithmic worklist before the current substitution, making the reasoning of certain properties complicated. Last but not least, its pervasive use of substitution makes its adaptation to Coq very difficult to deal with.

Such drawbacks motivate us to develop a new relation, more syntax-directed, to describe the transfer. The new transfer relation is inductively defined for each syntactic category: types ($\theta \vDash A \rightsquigarrow A'$), expressions ($\theta \vDash e \rightsquigarrow e'$), continuations ($\theta \vDash \omega^{s} \rightsquigarrow \omega^{s\prime}$ and $\theta \vDash \omega^{d} \rightsquigarrow \omega^{d\prime}$)[1] works ($\theta \vDash w \rightsquigarrow w'$), and worklists ($\theta \vDash \Gamma \rightsquigarrow \Omega \dashv \theta'$). θ is a substitution set $\theta := \cdot \mid \theta, a \mid \theta, \tilde{a} \mid \theta, \widehat{a} : \tau$ that keeps track of each existential variable a and its instantiation τ, which is a well-formed declarative monotype. θ also keeps track of type variables so that the well-formedness of each instantiation τ can be checked just by inspecting θ.

The transfer relation of types, expressions, continuations, and works should be read as: under the current substitution set θ, an algorithmic type, expression, continuation, or work is transferred to a declarative type, expression, continuation, or work by replacing all the existential variables in the types to its declarative monotype instantiation bound τ in θ, respectively. The transfer relation for type, expression, and worklist is shown in Figure 8. The transfer relation for continuations and works follows the same routine, as they always get decomposed to transfer types and expressions. The concrete rules are shown in Figure S7 in the appendix. The transfer of the worklist ($\theta \vDash \Gamma \rightsquigarrow \Omega \dashv \theta'$) should be read as, the worklist Γ is transferred to Ω under substitution set θ and θ is extended to θ' with more type variables and existential variables in Γ. When the last entry of the algorithmic worklist is a type variable, subtype variable, or existential variable, θ gets updated. Type variables and subtype variables are kept in the transferred declarative worklist and existential variables get erased. The last two cases (variables and works) of worklist transfer are straightforward. It is not

[1]Instead of using HOAS, we adopt a defunctionalized style for continuation, with details shown in the appendix. So the continuation has its own syntax categories.

$\boxed{\theta \vDash A \rightsquigarrow A'}$ A is transferred to A' under θ

$$\frac{}{\theta \vDash \mathbb{1} \rightsquigarrow \mathbb{1}} \quad \frac{}{\theta \vDash \top \rightsquigarrow \top} \quad \frac{}{\theta \vDash \bot \rightsquigarrow \bot} \quad \frac{a \in \theta \lor \tilde{a} \in \theta}{\theta \vDash a \rightsquigarrow a} \quad \frac{\widehat{a} : \tau \in \theta}{\theta \vDash a \rightsquigarrow \tau} \quad \frac{\theta, a \vDash A \rightsquigarrow A'}{\theta \vDash \forall a.\, A \rightsquigarrow \forall a.\, A'}$$

$$\frac{\theta \vDash A_1 \rightsquigarrow A_1' \quad \theta \vDash A_2 \rightsquigarrow A_2'}{\theta \vDash A_1 \to A_2 \rightsquigarrow A_1' \to A_2'} \quad \frac{\theta \vDash A_1 \rightsquigarrow A_1' \quad \theta \vDash A_2 \rightsquigarrow A_2'}{\theta \vDash A_1 \sqcap A_2 \rightsquigarrow A_1' \sqcap A_2'} \quad \frac{\theta \vDash A_1 \rightsquigarrow A_1' \quad \theta \vDash A_2 \rightsquigarrow A_2'}{\theta \vDash A_1 \sqcup A_2 \rightsquigarrow A_1' \sqcup A_2'}$$

$\boxed{\theta \vDash e \rightsquigarrow e'}$ e is transferred to e' under θ

$$\frac{}{\theta \vDash x \rightsquigarrow x} \quad \frac{}{\theta \vDash () \rightsquigarrow ()} \quad \frac{\theta \vDash e \rightsquigarrow e'}{\theta \vDash \lambda x.e \rightsquigarrow \lambda x.\, e'} \quad \frac{\theta \vDash e_1' \rightsquigarrow e_1' \quad \theta \vDash e_2' \rightsquigarrow e_2'}{\theta \vDash e_1\ e_2 \rightsquigarrow e_1'\ e_2'}$$

$$\frac{\theta \vDash e \rightsquigarrow e' \quad \theta \vDash A \rightsquigarrow A'}{\theta \vDash e : A \rightsquigarrow e' : A'} \quad \frac{\theta \vDash e \rightsquigarrow e' \quad \theta \vDash A \rightsquigarrow A'}{\theta \vDash e\ @A \rightsquigarrow e'\ @A'} \quad \frac{\theta, a \vDash e \rightsquigarrow e' \quad \theta, a \vDash A \rightsquigarrow A'}{\theta \vDash \Lambda a.e : A \rightsquigarrow \Lambda a.e' : A'}$$

$\boxed{\theta \vDash \Gamma \rightsquigarrow \Omega \dashv \theta'}$ Γ is transferred to Ω with θ updated to θ'

$$\frac{}{\theta \vDash \cdot \rightsquigarrow \cdot \dashv \theta} \quad \frac{\theta \vDash \Gamma \rightsquigarrow \Omega \dashv \theta'}{\theta \vDash \Gamma, a \rightsquigarrow \Omega, a \dashv \theta', a} \quad \frac{\theta \vDash \Gamma \rightsquigarrow \Omega \dashv \theta'}{\theta \vDash \Gamma, \tilde{a} \rightsquigarrow \Omega, \tilde{a} \dashv \theta', \tilde{a}} \quad \frac{\theta \vDash \Gamma \rightsquigarrow \Omega \dashv \theta' \quad \lfloor \theta' \rfloor \vdash \tau}{\theta \vDash \Gamma, \widehat{a} \rightsquigarrow \Omega \dashv \theta', \widehat{a} : \tau}$$

$$\frac{\theta \vDash \Gamma \rightsquigarrow \Omega \dashv \theta' \quad \theta' \vDash A \rightsquigarrow A'}{\theta \vDash \Gamma, x : A \rightsquigarrow \Omega, x : A' \dashv \theta'} \quad \frac{\theta \vDash \Gamma \rightsquigarrow \Omega \dashv \theta' \quad \theta' \vDash w \rightsquigarrow w'}{\theta \vDash \Gamma \Vdash w \rightsquigarrow \Omega \Vdash w' \dashv \theta'}$$

Fig. 8. Syntax-directed Transfer for Type and Expression and Worklist

hard to informally verify that the syntax-directed transfer is equivalent to the original transfer relation used by Zhao et al. [2019].

Soundness and completeness are built upon this new syntax-directed transfer to relate a declarative worklist with multiple algorithmic worklists. The quantifiers in the two lemmas are different because worklist transfer is a non-deterministic relation.

Theorem 5.6 (Soundness). *If* $\vdash \Gamma$ *and* $\Gamma \longrightarrow^* \cdot$, *then exists* θ, Ω, *s.t.* $\cdot \vDash \Gamma \rightsquigarrow \Omega \dashv \theta$, *and* $\Omega \longrightarrow^* \cdot$.

Theorem 5.7 (Completeness). *If* $\Omega \longrightarrow_\omega^* \cdot$, $\vdash \Gamma$, *and* $\cdot \vDash \Gamma \rightsquigarrow \Omega \dashv \theta$, *then* $\Gamma \longrightarrow^* \cdot$.

The proof proceeds by induction of the derivation of $\Gamma \longrightarrow^* \cdot$ and $\Omega \longrightarrow_\omega^* \cdot$, respectively. There are three interesting points about this new proof, which we discuss next.

Existential-variable solving. First, existential-variable solving is all dealt with by worklist substitution. The corresponding cases on the proof are unified, all depending on the instantiation-consistency lemma that worklist substitution preserves the worklist transfer. The lemma now requires reasoning about the properties of the substitution set θ as well. With a stronger conclusion, the induction hypothesis is then strong enough to be applied: (1) the substitution set is well-formed ($\vdash \theta'$) (2) the resulting substitution set before and after the worklist substitution contains the same entries except $\widehat{a} : \tau$ ($\theta \overset{/\widehat{a}}{\equiv} \theta'$). These lemmas capture the core invariance of worklist substitution: instantiation for other existential variables remain valid after and before it.

Lemma 5.8 (Instantiation Consistency).

(1) *If* $\cdot \vDash \{\tau/a\}\Gamma \rightsquigarrow \Omega \dashv \theta$, *then exists* θ', τ' *s.t.* $\cdot \vDash \Gamma \rightsquigarrow \Omega \dashv \theta'$, $\theta \vDash \tau \rightsquigarrow \tau'$, $\theta' \vDash \tau \rightsquigarrow \tau'$, *and* $\widehat{a} : \tau' \in \theta'$, $\theta \overset{/\widehat{a}}{\equiv} \theta'$, *and* $\vdash \theta'$;

(2) *If* $\cdot \vDash \Gamma \rightsquigarrow \Omega \dashv \theta, \theta \vDash \tau \rightsquigarrow \tau', \widehat{a} : \tau' \in \theta$, *and* $a \notin fv(\tau)$, *then exists* θ', *s.t.* $\cdot \vDash \{\tau/a\}\Gamma \rightsquigarrow \Omega \dashv \theta'$, $\theta \overset{/\widehat{a}}{\equiv} \theta'$, *and* $\vdash \theta'$.

Occurs-check. Second, the case $\leq\rightarrow$ of the completeness proof relies on a property to ensure the occurs-check condition in rules 18 and 19 is always satisfied when one side is transferred from an existential variable and the other is transferred from a function type. The satisfiability is guaranteed by the following lemma. Compared with the lemmas used by Cui et al. [2023], this lemma only needs to cover mono function types, since the occurs-check is dropped for rules 20 and 21 about comparing existential variables with non-mono function types.

Lemma 5.9 (Satisfiability of Occurs-check).

(1) *If* $\cdot \vDash \Gamma \Vdash a \leq \sigma_1 \rightarrow \sigma_2 \rightsquigarrow \Omega \Vdash \tau_1 \rightarrow \tau_2 \leq \sigma_1' \rightarrow \sigma_2' \dashv \theta, \widehat{a} \in \Gamma$, *and* $\lfloor\Omega\rfloor \vdash \tau_1 \rightarrow \tau_2 \leq \sigma_1' \rightarrow \sigma_2'$, *then* $a \notin fv(\sigma_1 \rightarrow \sigma_2)$;

(2) *If* $\cdot \vDash \Gamma \Vdash \sigma_1 \rightarrow \sigma_2 \leq a \rightsquigarrow \Omega \Vdash \sigma_1' \rightarrow \sigma_2' \leq \tau_1 \rightarrow \tau_2 \dashv \theta, \widehat{a} \in \Gamma$, *and* $\lfloor\Omega\rfloor \vdash \sigma_1' \rightarrow \sigma_2' \leq \tau_1 \rightarrow \tau_2$, *then* $a \notin fv(\sigma_1 \rightarrow \sigma_2)$.

This lemma is proved by proof by contradiction: if a occurs in $\sigma_1 \rightarrow \sigma_2$ and a is transferred to $\tau_1 \rightarrow \tau_2$, then $\sigma_1' \rightarrow \sigma_2'$ must have a deeper function type than $\tau_1 \rightarrow \tau_2$. However, for the subtyping relation $\lfloor\Omega\rfloor \vdash \tau_1 \rightarrow \tau_2 \leq \sigma_1' \rightarrow \sigma_2'$ to hold in the type system specification, $\sigma_1' \rightarrow \sigma_2'$ and $\tau_1 \rightarrow \tau_2$ must have function type of the same depth, since monotype subtyping is structural. Note that it is very hard to generalize this lemma to the non-monotype case, since the subtyping relation of intersection and union types are highly non-structural.

Transfer relation inversion. Third, since the existential variables get erased during the transfer, the syntactic correspondence between Γ and Ω is not completely trivial: $\Gamma, \widehat{a}_1, \ldots, \widehat{a}_n$ (an algorithmic worklist with trailing existential variables) is transferred to the same declarative worklist Ω as Γ does. Because of this, in the proof of completeness, if we naively invert the transfer relation $\cdot \vDash \Gamma \rightsquigarrow \Omega \dashv \theta$, there is always an extra case saying that Γ could be some $\Gamma', \widehat{a}$ and Γ' is still transferred to Ω. To get rid of this, we first show that an algorithmic worklist with trailing existential variables can always reduce to an algorithmic worklist whose last entry is no longer an existential variable, and then perform the inversion.

5.3 Decidability

The decidability proof is based on a lexicographic group of 9 measures on the worklist Γ: $(|\Gamma|_e, |\Gamma|_\omega, |\Gamma|_{\Rrightarrow}, |\Gamma|_{\Rrightarrow_\omega}, |\Gamma|_{\triangleright}, |\Gamma|_{\triangleright_\omega}, |\Gamma|_{\rightarrow}, |\Gamma|_{\widehat{a}}, |\Gamma|_{\leq})$. These measures are: the term size ($|\Gamma|_e$), number of judgments ($|\Gamma|_\omega$), type size of type-application judgments ($|\Gamma|_{\Rrightarrow}$), number of type-application judgments ($|\Gamma|_{\Rrightarrow_\omega}$), type size of matching judgments ($|\Gamma|_{\triangleright}$), number of matching judgments ($|\Gamma|_{\triangleright_\omega}$), the sum of ranks of non-mono components ($|\Gamma|_{\rightarrow}$), number of existential variables ($|\Gamma|_{\widehat{a}}$), and type size of subtyping of the worklist ($|\Gamma|_{\leq}$), respectively. The general design principles is that the sizes related to expressions and checking and inference judgments should be put in the early positions of the lexicographic group, while measures about the reduction of subtyping judgments can be put in latter positions. This group of measure always decreases after 1 or 2 steps of reduction, so the reasoning itself is straightforward. Nonetheless, the measure gets significantly more complicated because of intersection and union types. The detailed measure definitions and the rationales behind their design are in Appendix D.2. We highlight some challenges and solutions to them here.

Theorem 5.10 (Decidability). *Given* $\vdash \Gamma$, $(|\Gamma|_e, |\Gamma|_\omega, |\Gamma|_{\Rrightarrow}, |\Gamma|_{\Rrightarrow_\omega}, |\Gamma|_{\triangleright}, |\Gamma|_{\triangleright_\omega}, |\Gamma|_{\rightarrow}, |\Gamma|_{\widehat{a}}, |\Gamma|_{\leq}) < (n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9)$, $\Gamma \longrightarrow^* \cdot$ *is decidable.*

New sum of ranks measure. The measure $|\Gamma|_{\rightarrow}$ computes the sum of rank (i.e. depth inside function type) of non-monotype components of each type. It captures the essence of polytype splitting proposed by $F^b_{\leq}$ [Cui et al. 2023]: in every split destructing one polytype function type $A \to B$, every non-monotype component in it will have a smaller rank. This generalizes the split measure used in $F^b_{\leq}$ to provide a uniform treatment for all non-mono types, and avoids the unintuitive post-computation needed by the original measure. This measure decreases whether $a \in \mathrm{fv}(A \to B)$ or not in rule 20 and 21, so the occurs-check is not needed.

Duplication and recursively computed measures. The major complexity of the decidability proof is the duplication caused by intersection and union types. In previous systems, when multiple judgments are created by some algorithmic reduction rules, every component of the original judgment is used at most once. For example, in rule 9, A_1, A_2, B_1, B_2 is used once by two new judgments. However, in $F^e_{\sqcup\sqcap}$, certain types and expressions may just get duplicated twice without any change. Type duplication happens in subtyping (rule 12, $A \leq B_1 \sqcap B_2$ and 15, $A_1 \sqcup A_2 \leq B$). Therefore, the maximum number of duplications for each type should be considered. In the subtyping work $A \leq B$, A may be duplicated up to $|B|_{\sqcup\sqcap}$ times and B may be duplicated up to $|A|_{\sqcup\sqcap}$ times ($|\cdot|_{\sqcup\sqcap}$ means the number of intersection and union types). Consequently, the type size of subtyping of $A \leq B$ ($|A \leq B|_{\leq}$) is calculated as $|A|_{\leq} \cdot (|B|_{\sqcup\sqcap} + 1) + |B|_{\leq} \cdot (|A|_{\sqcup\sqcap} + 1)$. Expression duplication happens in checking (rule 25, $e \Leftarrow A_1 \sqcap A_2$). The the situation in typing is even more complex: though the duplication only happens in checking, checking can be created from the inference of application $e_1\ e_2 \Rightarrow \omega$, where the domain type matched from e_1 is used to check e_2. Since the type to check e_2 is not known yet, we must develop an over-estimation by syntactically analyzing e_1 and the whole chain of continuation to accumulate all the information.

Modular reasoning for matching and type-application. Since rule 49 creates one more type-application judgment without decreasing the expression size, the size of the type-application judgment must be counted as 0 in $|\Gamma|_{\omega}$, otherwise the overall measure must increase. We develop two new measures: $|\Gamma|_{\Rrightarrow}$ and $|\Gamma|_{\Rrightarrow_\omega}$ to reason about the reduction behavior of type application. Since type-application only creates new type-application judgments and the type being type-applied always gets smaller, $|\Gamma|_{\Rrightarrow}$ counts the size of the type being type-applied, and $|\Gamma|_{\Rrightarrow_\omega}$ counts the number of the type-application judgment. So the termination of each type-application: for any type $A, B, A \circ B \Rrightarrow \omega$ will reduce to $\omega \diamond C$ or not after a certain number of steps. Termination can be established using these two measures. Similarly, $|\Gamma|_{\triangleright}$ and $|\Gamma|_{\triangleright_\omega}$ are designed to reason about the termination of each matching judgment: for any type A, $A \triangleright_{\alpha,\beta} \omega$ will always reduce to $\omega \diamond B \diamond C$ or not after a certain number of steps. The situation of matching is almost the same as that of type application: rule 41 creates more matching judgments, and the type being matched always gets smaller. The reason to put $|\Gamma|_{\Rrightarrow}$, $|\Gamma|_{\Rrightarrow_\omega}$ in front of $|\Gamma|_{\triangleright}$, $|\Gamma|_{\triangleright_\omega}$ is that the result of matching can never be type-applied but the result of type application can possibly be matched.

6 EXTENSIONS

In this section, we present two extensions of $F^e_{\sqcup\sqcap}$, which are useful in practice. The first extension adds a form of labels to $F^e_{\sqcup\sqcap}$, and enables an encoding of records. The second extension widens monotypes to include intersections and union types. Both extensions have also been formalized in Coq. The soundness of the corresponding algorithmic system with respect to the bidirectional specification is proved. However, allowing intersection and union types as monotypes makes the algorithm incomplete.

6.1 Encoding Records as Intersection Types

The type system of $F^e_{\sqcup\sqcap}$ is quite general as it supports unrestricted intersection and union types. Thus, it can be extended to support other useful language features easily. In this section, we demonstrate one of such extensions: supporting records by encoding record types as intersection types. In this extension, we rely on the expressive power of the matching relation in $F^e_{\sqcup\sqcap}$, which is capable of dealing with record subtyping without any additional rules or changes. This extension is partly inspired by existing encodings of records using first-class labels and functions [Castagna et al. 1995]. The syntax for this extension is shown below. Label name is a set of available names that can be used as a label for records, or other structures. The label is first-class at the type level and represented by the new $\texttt{Label}\ l$ syntax. Expressions are extended with three new forms: empty record $\langle\rangle$, record extension $\langle l_1 \mapsto e_1, e_2\rangle$, and record projection $e.l$. The label at the expression level can only be used in such expressions, and thus it is not first-class.

$$
\begin{array}{llll}
\ldots & \text{Label name} ::= l & & \\
\text{Types} & A, B, C & ::= & \cdots \mid \texttt{Label}\ l \\
\text{Monotypes} & \tau, \sigma & ::= & \cdots \mid \texttt{Label}\ l \\
\text{Expressions} & e, t & ::= & \cdots \mid \langle\rangle \mid \langle l_1 \mapsto e_1, e_2\rangle \mid e.l
\end{array}
$$

Four rules are added to the type system: one in subtyping ($\leq$Label); and three in typing ($\Rightarrow\langle\rangle$Nil, $\Rightarrow\langle\rangle$Cons and $\Rightarrow\langle\rangle$Proj). The subtyping rule is simply a reflexive rule for the new type constructor. The typing rules for record are all inference rules. The empty record can infer the unit type. The record extension rule infers an intersection type whose first branch is a function type from $\texttt{Label}\ l$ to A_1, the type of e_1. For example, $\langle l_1 \mapsto (), \langle l_2 \mapsto (), \langle\rangle\rangle\rangle$ infers the type $(\texttt{Label}\ l_1 \to \mathbb{1}) \sqcap (\texttt{Label}\ l_2 \to \mathbb{1}) \sqcap \mathbb{1}$. The $\Rightarrow\langle\rangle$Cons rule is quite flexible in the sense that it does not even require e_2 to be a record (if such a restriction, it can be easily added to the expression well-formedness). With such an encoding, the record projection can be completely dealt with by the existing matching relation, as shown in the rule $\Rightarrow\langle\rangle$Proj. Compared with rule $\Rightarrow$App, the only difference is that the last premise is changed from $\Psi \vdash e_2 \Leftarrow B$ to $\Psi \vdash \texttt{Label}\ l \leq B$.

$$
\frac{}{\Psi \vdash \langle\rangle \Rightarrow \mathbb{1}}\ \Rightarrow\langle\rangle\texttt{Nil}
\qquad
\frac{\Psi \vdash e_1 \Rightarrow A_1 \quad \Psi \vdash e_2 \Rightarrow A_2}{\Psi \vdash \langle l_1 \mapsto e_1, e_2\rangle \Rightarrow (\texttt{Label}\ l_1 \to A_1) \sqcap A_2}\ \Rightarrow\langle\rangle\texttt{Cons}
$$

$$
\frac{\Psi \vdash e \Rightarrow A \quad \Psi \vdash A \triangleright B \to C \quad \Psi \vdash \texttt{Label}\ l \leq B}{\Psi \vdash e.l \Rightarrow C}\ \Rightarrow\langle\rangle\texttt{Proj}
\qquad
\frac{}{\Psi \vdash \texttt{Label}\ l \leq \texttt{Label}\ l}\ \leq\texttt{Label}
$$

This extension still keeps all the desirable properties of the base system and it is also equipped with a sound and complete algorithmic system. The algorithm is extended with 2 new works and 6 new rules to deal with this new feature, with details in Appendix B.5. We did not prove decidability for the algorithm, but we believe it is a simple modification of the existing proof. In addition to records, it should be possible to add some other features quite easily as well. For instance, adding variants and variant subtyping can be done with a similar approach, by relying on matching and first-class labels.

6.2 Inferring Intersection and Union Types

While $F^e_{\sqcup\sqcap}$ has good properties, completeness can only be achieved with a fairly restrictive definition of monotypes that excludes intersection and union types. In practice though, we may want to infer intersection and union types. It is possible to extend our definition of monotypes to include intersection and union types, while still employing our algorithm. The extended monotype definition is $\tau, \sigma ::= \mathbb{1} \mid a \mid \tau \to \sigma \mid \boxed{\tau \sqcap \sigma} \mid \boxed{\tau \sqcup \sigma}$ (a is still a type variable). With this new definition, $\mathbb{1} \sqcap (\mathbb{1} \to \mathbb{1})$ and $(\texttt{Label}\ l_1 \to \mathbb{1}) \sqcap (\texttt{Label}\ l_2 \to \mathbb{1})$ now become monotypes. This enables a larger class of types to be inferred. The rules of the bidirectional and the algorithmic type systems are the same as in the previous extension.

In this extension, the bidirectional type system itself retains all the properties. The greedy algorithm remains sound but it is not complete, as we explained in Section 3. Despite incompleteness to the specification, it does accept strictly more programs compared with previous systems, at a cost of more backtracking in existential-variable solving. For example, given $\Gamma \Vdash \mathbb{1} \sqcap (\mathbb{1} \to \mathbb{1}) \leq a$ and $\widehat{a} \in \Gamma$, our algorithm will first try to solve a to $\mathbb{1} \sqcap (\mathbb{1} \to \mathbb{1})$. If this solution fails, the algorithm will continue to try $\mathbb{1}$ and $\mathbb{1} \to \mathbb{1}$. It is still greedy, but it tries to make more use of the first instantiation found: try it and its approximations. Due to the incompleteness, the algorithmic type system may not enjoy the same properties of the bidirectional specification. We do not have the decidability for this algorithm either, since the intersection and union types could now be introduced by solving monotypes. For example, $\lambda x.x \Leftarrow a$ could become $\lambda x.x \Leftarrow (\texttt{Int} \to \texttt{Int}) \sqcap (\texttt{Bool} \to \texttt{Bool})$ if a is solved to $(\texttt{Int} \to \texttt{Int}) \sqcap (\texttt{Bool} \to \texttt{Bool})$. Thus, it violates the principle that our decidability proof builds on: all intersection and union types are from the annotations of expression, so the duplication can be statically computed. Nonetheless, we still conjecture that the algorithm will terminate, but proving this would require a new proof strategy.

7 RELATED WORK

Polymorphic type inference with intersections and union types. Most works on type inference with polymorphism, as well as intersection and union types, support only a restrictive form of intersection and union types. Jim [2000] introduces a polar type system, called **P**, which comes with a decidable type inference algorithm. Intersection types and polymorphism are restricted. In **P**, quantifiers must only appear in positive positions, while intersection types are restricted to appear in negative positions. MLSub [Dolan and Mycroft 2017] has sound and complete inference for principal types based on a algebraic subtyping lattice. The types in the system are also polarized: intersection types are constrained to appear in negative positions; while union types are constrained to appear in positive positions. Simple-sub [Parreaux 2020] provides an implementation-oriented reinterpretation of MLsub that still preserves the principality of inference and ignores the algebraic property for simplicity. MLstruct [Parreaux and Chau 2022] extends MLsub by introducing first-class intersections and union types and negation. However, abstractions cannot be assigned with intersections of arrow types. Castagna et al. [2024] proposes a set-theoretic type system with first-order polymorphism, intersection and union types. The type reconstruction algorithm is sound, terminating, but incomplete. The type system has the intersection introduction rule and can express overloaded functions. All of these works support only first-order (or rank-1) polymorphism and do not support explicit type applications, unlike $F^e_{\sqcup\sqcap}$.

SuperF [Parreaux et al. 2024] is a type inference approach based on multi-bounded polymorphism [Cretin 2014], supporting higher-rank polymorphism, intersection and union types. The type inference algorithm is terminating but incomplete. SuperF employs a similar restriction to MLSub, with intersection types limited to appear in negative positions, and union types limited to appear in positive positions. Dunfield [2009] presents a bidirectional type system with higher-rank polymorphism, as well as intersection and union types. Like our work, Dunfield employs greedy instantiation. The corresponding type inference algorithm is sound. However, as observed by Dunfield and Krishnaswami [2013, 2021], the completeness and decidability proofs provided in Dunfield work are flawed and have not been tackled yet. Unlike our work, none of the previous works consider the interaction with explicit type applications, and only the work by Dunfield considers HRP with unrestricted intersection and union types as we do. However we have complete proofs, which are mechanically formalized and verified in the Coq proof assistant.

Other work in higher-rank polymorphic type inference. Explicit type applications allow programmers to specify their own instantiations. Eisenberg et al. [2016b] proposes extensions to both HM and a predictive HRP system with *predicative* explicit type applications, which has been implemented in GHC 8. Zhao and Oliveira [2022] proposes $F^e_{\leq}$ to extend a predicative HRP type system with *impredicative* explicit type applications. Cui et al. [2023] extends $F^e_{\leq}$ with bounded quantification, resulting a variant of kernel $F_{\leq}$ [Cardelli et al. 1994; Cardelli and Wegner 1985]. Our work incorporates the idea of $F^e_{\leq}$ for explicit type applications. The main addition over $F^e_{\leq}$ are intersection and union types, which have non-trivial interactions with HRP and also complicate the interaction with explicit type applications. There are also other lines of work on *impredicative* HRP for System F like languages (without intersections and union types) [Emrich et al. 2020; Le Botlan and Rémy 2003; Leijen 2008; Serrano et al. 2020, 2018; Vytiniotis et al. 2008]. In those type systems, implicit instantiations can also be polytypes. These type systems are notably more complex due to the undecidability of the natural subtyping relation [Chrząszcz 1998; Tiuryn and Urzyczyn 1996]. Thus, they must impose some restrictions to ensure decidability.

Local type inference. Local type inference [Odersky et al. 2001; Pierce and Turner 2000] has shown to be a great success in practice and forms the foundation of type inference implementations in many mainstream programming languages. Its success is largely attributed to its adaptability to various programming language features. However, many practical extensions of local type inference, such as intersection and union types, have not been formally studied. For example, Java and Scala 2, whose type inference is based on local type inference, incorporate intersection types. Local type inference prefers uncurried applications, where all the arguments must be given at once. With uncurried applications it is possible to exploit the type information of the arguments to improve the results of type inference. For example, consider the following program in TypeScript, which also adopts some local type inference techniques:

```
var r1: {m : number, n : boolean} = { m: 1, n: true }; var r2: {m : number} = { m: 2 }
function f<A>(x: A, y: A): A { return x }
var ex1 = f(r1, r2); var ex2 = f(r2, r1);
function g: <A>(x: A) => (y: A) => A = x => y => { return y }
var ex3 = g(r2)(r1);
var ex4 = g(r1)(r2) // rejected!
```

The type of `r1` is a subtype of the type of `r2`. With uncurried functions, both `ex1` and `ex2` are accepted, which suggests that a non-greedy constraint solving approach, similar to what is used in local type inference, is adopted in this case. In contrast, for curried functions, only `ex3` is accepted, and `ex4` is rejected, indicating that the type argument `A` is committed to the type of the first argument directly. While local type inference has been extended in mainstream programming languages with both intersection and union types, we do not know of work formally studying such extensions. In addition we also do not know any work on local type inference that supports polymorphic subtyping. In Scala 3 HRP is also supported, but polymorphic subtyping is not.

8 CONCLUSION

As programming languages evolve, features that once belonged to functional programming and OOP begin to intersect. Higher-rank polymorphism, intersection and union types, and explicit type applications are examples of such features. In this paper, we formally study the interaction of these features in the context of the bidirectional type system $F^e_{\sqcup\sqcap}$, that supports unrestricted forms of intersection and union types. $F^e_{\sqcup\sqcap}$ is equipped with a sound, complete, and decidable algorithm that infers monotype instantiations and unannotated functions. Some variants of $F^e_{\sqcup\sqcap}$ that incorporate more practical features, such as handling records, and providing more inference at a cost of completeness are also discussed. Our findings provide insights for improving current implementations in languages with those features and can help with the design of novel type inference algorithms. As a byproduct of our work, we develop several new techniques in formalizing worklist-based approaches, and our formalization in Coq can be used as a general framework for studying other type systems.

REFERENCES

Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 257–281.

L. Cardelli, S. Martini, J.C. Mitchell, and A. Scedrov. 1994. An Extension of System F with Subtyping. *Information and Computation* 109, 1 (1994), 4–56. https://doi.org/10.1006/inco.1994.1013

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

G. Castagna, G. Ghelli, and G. Longo. 1995. A calculus for overloaded functions with subtyping. *Information and Computation* 117, 1 (Feb. 1995), 115–135.

Giuseppe Castagna, Mickaël Laurent, and Kim Nguyen. 2024. Polymorphic Type Inference for Dynamic Languages. *Proc. ACM Program. Lang.* 8, POPL (2024).

Arthur Charguéraud. 2012. The Locally Nameless Representation. *Journal of Automated Reasoning* 49, 3 (01 Oct 2012), 363–408. https://doi.org/10.1007/s10817-011-9225-2

Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. 2017. Fast and precise type checking for JavaScript. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 48 (oct 2017), 30 pages. https://doi.org/10.1145/3133872

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

Julien Cretin. 2014. *Erasable coercions: a unified approach to type systems. (Coercions effaçables : une approche unifiée des systèmes de types)*. Ph. D. Dissertation. Paris Diderot University, France. https://tel.archives-ouvertes.fr/tel-00940511

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

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

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL, 60–72. https://doi.org/10.1145/3009837.3009882

Andrej Dudenhefner, Moritz Martens, and Jakob Rehof. 2016. The Intersection Type Unification Problem. In *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 52)*, Delia Kesner and Brigitte Pientka (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 19:1–19:16. https://doi.org/10.4230/LIPIcs.FSCD.2016.19

Jana Dunfield. 2009. Greedy Bidirectional Polymorphism. In *ML Workshop (ML '09)*. 15–26. http://www.cs.queensu.ca/~jana/papers/poly/.

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

Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-rank Polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP, 429–442. https://doi.org/10.1145/2500365.2500582

Jana Dunfield and Neelakantan R. Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5, Article 98 (May 2021), 38 pages.

Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. 2016a. Visible Type Application. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg.

Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. 2016b. Visible Type Application. In *European Symposium on Programming*. ESOP, 229–254. https://doi.org/10.1007/978-3-662-49498-1_10

Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan Coates. 2020. FreezeML: Complete and Easy Type Inference for First-Class Polymorphism. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI, 423–437.

Andrew Gacek. 2008. The Abella Interactive Theorem Prover (System Description). In *Automated Reasoning*, Alessandro Armando, Peter Baumgartner, and Gilles Dowek (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 154–161.

Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* 146 (1969), 29–60. https://doi.org/10.2307/1995158

Xuejing Huang, Jinxu Zhao, and Bruno C. d. S. Oliveira. 2021. Taming the Merge Operator. *Journal of Functional Programming* 31 (2021), e28. https://doi.org/10.1017/S0956796821000186

Trevor Jim. 2000. A Polar Type System. In *ICALP Workshops 2000, Proceedings of the Satelite Workshops of the 27th International Colloquium on Automata, Languages and Programming, Geneva, Switzerland, July 9-15, 2000*, José D. P. Rolim, Andrei Z. Broder, Andrea Corradini, Roberto Gorrieri, Reiko Heckel, Juraj Hromkovic, Ugo Vaccaro, and J. B. Wells (Eds.). Carleton Scientific, Waterloo, Ontario, Canada, 323–338.

Didier Le Botlan and Didier Rémy. 2003. ML^F: Raising ML to the Power of System F. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*. ICFP, 27–38. https://doi.org/10.1145/944705.944709

Daan Leijen. 2008. HMF: Simple Type Inference for First-class Polymorphism. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP, 283–294. https://doi.org/10.1145/1411204.1411245

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

John C. Mitchell. 1988. Polymorphic type inference and containment. *Information and Computation* 76, 2 (1988), 211–249. https://doi.org/10.1016/0890-5401(88)90009-0

Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL, 54–67. https://doi.org/10.1145/237721.237729

Martin Odersky, Christoph Zenger, and Matthias Zenger. 2001. Colored Local Type Inference. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL, 41–53. https://doi.org/10.1145/360204.360207

Lionel Parreaux. 2020. The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl). In *Proceedings of the ACM on Programming Languages*, Vol. 4. ICFP, Article 124. https://doi.org/10.1145/3409006

Lionel Parreaux, Aleksander Boruch-Gruszecki, Andong Fan, and Chun Yin Chau. 2024. When Subtyping Constraints Liberate: A Novel Type Inference Approach for First-Class Polymorphism. *Proc. ACM Program. Lang.* 8, POPL, Article 48 (jan 2024), 33 pages. https://doi.org/10.1145/3632890

Lionel Parreaux and Chun Yin Chau. 2022. MLstruct: Principal Type Inference in a Boolean Algebra of Structural Types. In *Proceedings of the ACM on Programming Languages*, Vol. 6. OOPSLA2, Article 141. https://doi.org/10.1145/3563304

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical Type Inference for Arbitrary-Rank Types. *Journal of Functional Programming* 17, 1 (2007), 1–82. https://doi.org/10.1017/s0956796806006034

Benjamin Crawford Pierce. 1992. *Programming with intersection types and bounded polymorphism.* Ph. D. Dissertation. USA. UMI Order No. GAX92-16028.

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL, 252–265. https://doi.org/10.1145/268946.268967

Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 624–641. https://doi.org/10.1145/2983990.2984008

Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A Quick Look at Impredicativity. In *Proceedings of the ACM on Programming Languages*, Vol. 4. ICFP, Article 89. https://doi.org/10.1145/3408971

Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded Impredicative Polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI, 783–796. https://doi.org/10.1145/3192366.3192389

Ehud Shapiro. 1989. The family of concurrent logic programming languages. *ACM Comput. Surv.* 21, 3 (sep 1989), 413–510. https://doi.org/10.1145/72551.72555

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

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

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2008. FPH: First-class Polymorphism for Haskell. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP, 295–306. https://doi.org/10.1145/1411204.1411246

Jinxu Zhao and Bruno C. d. S. Oliveira. 2022. Elementary Type Inference. In *36th European Conference on Object-Oriented Programming*. ECOOP, 2:1–2:28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.2

Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. A Mechanical Formalization of Higher-Ranked Polymorphic Type Inference. In *Proceedings of the ACM on Programming Languages*, Vol. 3. ICFP, Article 112. https://doi.org/10.1145/3341716