

# Bidirectional Higher-Rank Polymorphism with Intersection and Union Types

SHENGYI JIANG, University of Hong Kong, China

CHEN CUI, University of Hong Kong, China

BRUNO C. D. S. OLIVEIRA, University of Hong Kong, China

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 with explicit type applications, and intersection and union types and demonstrate that these features have non-trivial interactions. We first present a type system, described by a bidirectional specification, with good theoretical properties and a sound, complete, and decidable algorithm. This is helpful to identify a class of types that can always be inferred. We also explore variants incorporating practical features, such as handling records and inferring a larger class of types, which align better with real-world implementations. Though some variants no longer have a complete algorithm, they still enhance the expressiveness of the type system. To ensure rigor, all results are formalized in the Coq proof assistant.

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

Additional Key Words and Phrases: Intersection and Union Types, Polymorphism, Type Inference, Bidirectional Typing, Mechanical Formalization

## ACM Reference Format:

Shengyi Jiang, Chen Cui, and Bruno C. d. S. Oliveira. 2025. Bidirectional Higher-Rank Polymorphism with Intersection and Union Types. *Proc. ACM Program. Lang.* 9, POPL, Article 71 (January 2025), 58 pages. <https://doi.org/10.1145/3704907>

## 1 Introduction

Programming languages such as TypeScript, Flow, and Scala, embrace type systems with intersection and union types [Bierman et al. 2014; Chaudhuri et al. 2017; Rompf and Amin 2016]. Intersection and union types are important features for many languages. For typed variants of dynamic languages – such as TypeScript, Flow and Typed Scheme [Tobin-Hochstadt and Felleisen 2008] – intersection and union types are useful to type a lot of programming patterns including heterogeneous lists, function overloading [Castagna et al. 1995], type narrowing, mixin patterns [Bessai et al. 2014], etc. For Scala, intersection and union types are part of the DOT calculus [Rompf and Amin 2016], and are considered to be key features of the language.

TypeScript, Flow, and Scala have also started to incorporate *higher-rank polymorphism* (HRP) [Odersky and Läufer 1996] into object-oriented programming. HRP enables polymorphic types to appear

---

Authors' Contact Information: Shengyi Jiang, University of Hong Kong, Hong Kong, China, syjiang@cs.hku.hk; Chen Cui, University of Hong Kong, Hong Kong, China, ccui@cs.hku.hk; Bruno C. d. S. Oliveira, University of Hong Kong, Hong Kong, China, bruno@cs.hku.hk.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2025 Copyright held by the owner/author(s).

ACM 2475-1421/2025/1-ART71  
<https://doi.org/10.1145/3704907>

anywhere in nested positions inside function types. In TypeScript or Flow, it is also possible to have types such as  $(\forall a. a \rightarrow a) \& (\text{string} \rightarrow (\forall b. b \rightarrow \text{boolean}))$  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. [Castagna et al. \[2024\]](#) points out that parametric polymorphism with intersection and unions is a good combination to type dynamic languages.

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. 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. 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 some instantiations cannot be automatically inferred, programmers have the possibility to explicitly specify them via *explicit type applications* [[Eisenberg et al. 2016a](#); [Pierce and Turner 2000](#)].

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  $\forall a.a \rightarrow a <: \text{Int} \rightarrow \text{Int}$  is a valid subtyping assertion that can be derived by selecting  $\tau = \text{Int}$ , 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 in 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 are 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 languages with these features, like TypeScript, have some issues related to these interactions.

We present a calculus and type system, called  $F_{\sqcup, \cap}^e$ , with three features.  $F_{\sqcup, \cap}^e$  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  $F_{\sqcup, \cap}^e$  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 first type system 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:

- **Bidirectional HRP with intersection and union types and explicit type applications.** These features are important in practice and adopted by TypeScript, Flow, and Scala.
- **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 to systems with intersection and union types.
- As part of the development of  $F_{\sqcup\cap}^e$ , we introduced a few **technical innovations**. We design new work and continuation to deal with the new reasoning required by intersection and union types. Our syntax-directed transfer and defunctionalized representation of continuation-passing style simplify reasoning and 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 proof related to it.
- We prove several results about the **metatheory** of  $F_{\sqcup\cap}^e$ . 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 subtyping transitivity and checking subsumption.
- All the results are **mechanically formalized** in Coq. Furthermore, we have a simple prototype implementation capable of running the examples presented in the paper. The Coq formalization, the implementation, and our examples can be found in the supplementary materials available at <https://doi.org/10.5281/zenodo.13922446>.

## 2 Overview

We start with a background on implicit (predicative)<sup>1</sup> 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  $F_{\sqcup\cap}^e$ .

### 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  $F_{\leq}^e$  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  $F_{\leq}^e$ . We provide a quick overview of the important design choices in  $F_{\leq}^e$  next.

<sup>1</sup>We slightly abuse the term "predicative instantiations". It originally meant instantiations not containing universal types in System  $F$ , while we use it to mean all monotype instantiations. Accordingly, "impredicative instantiations" means polytype instantiations in the remaining paper.

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

$\Psi \vdash A \leq B$

A is a subtype of B

$$\begin{array}{c}
\frac{}{\Psi \vdash \perp \leq \perp} \leq \perp \quad \frac{}{\Psi \vdash A \leq \top} \leq \top \quad \frac{}{\Psi \vdash \perp \leq A} \leq \perp \quad \frac{}{\Psi \vdash a \leq a} \leq \text{TVar} \quad \frac{}{\Psi \vdash \tilde{a} \leq \tilde{a}} \leq \text{STVar} \\
\Psi \vdash B_1 \leq A_1 \quad \Psi \vdash A_2 \leq B_2 \quad \Psi \vdash \tau \quad \Psi \vdash [\tau/a]A \leq B \quad B \neq \forall.* \quad \Psi, \tilde{a} \vdash [\tilde{a}/a]A \leq [\tilde{a}/a]B \\
\Psi \vdash A_1 \rightarrow A_2 \leq B_1 \rightarrow B_2 \quad \leq \rightarrow \quad \frac{}{\Psi \vdash \forall a. A \leq B} \leq \forall L \quad \frac{}{\Psi \vdash \forall a. A \leq \forall a. B} \leq \forall R
\end{array}$$

Fig. 1. Type syntax and subtyping rules of  $F_{\leq}^e$ .

*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:

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

These rules enable order-irrelevant universal quantifiers: two universal types with the same body but a different order of quantifiers are considered equivalent. For example  $\forall a. \forall b. a \rightarrow b$  and  $\forall b. \forall a. a \rightarrow b$  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  $(\forall a. \forall b. a \rightarrow b) \rightarrow \text{Bool}$  and  $(\forall b. \forall a. a \rightarrow b) \rightarrow \text{Bool}$ , by instantiating  $a$  and  $b$  to  $\text{Int}$  and  $\text{Bool}$ , respectively. Now, suppose that we provide an explicit type instantiation  $\text{Int}$  to  $x$  as  $\lambda x. (x \ @\ \text{Int} \ 3)$ . This expression can be checked against  $(\forall a. \forall b. a \rightarrow b) \rightarrow \text{Bool}$ , but it cannot be checked against  $(\forall b. \forall a. a \rightarrow b) \rightarrow \text{Bool}$ . Thus the types  $\forall a. \forall b. a \rightarrow b$  and  $\forall b. \forall a. a \rightarrow b$  do not behave equivalently in the presence of explicit type applications and should not be in a subtyping relation. Zhao and Oliveira [2022] 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,  $F_{\leq}^e$  adopts a different subtyping relation compared to Odersky and Läufer's, as shown in Figure 1. The  $\leq \forall R$  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  $\forall a. \forall b. a \rightarrow b \leq \forall b. \forall a. a \rightarrow b$ . 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  $\tau$  in rule  $\leq \forall L$  cannot contain them. The third restriction is adding two checks in well-formedness, to ensure no unused variables in universal types.

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

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

**THEOREM 2.1 (STABILITY UNDER INSTANTIATION).** *Given  $\vdash \Psi, \Psi \vdash \forall a. A, \Psi \vdash \forall a. B$  and  $\Psi \vdash C$ , 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] employ a greedy approach to solve the instantiation: existential variables are solved to the first monotype (i.e., a non-polymorphic type) 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,  $\tau_1 \leq \tau_2 \rightarrow \tau_1 = \tau_2$ . The fact that  $\perp$  and  $\top$  are not monotypes in  $F_{\leq}^e$  is also crucial for this property to hold.

*Bidirectional typing.*  $F_{\leq}^e$  chooses bidirectional typing to describe its specification and implement the algorithm. The  $F_{\leq}^b$  calculus [Cui et al. 2023], adds some general improvements to the typing rules of  $F_{\leq}^e$ , 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  $A \rightarrow B$  in TypeScript require an argument name. Thus a function type is written as  $(x:A) \Rightarrow B$ . Since this argument name makes no difference, we usually write it as  $(_:A) \Rightarrow B$  in the following examples. Universal types  $\forall a. A$  are represented as  $\langle a \rangle A$  (though the standard naming convention usually picks capital letters for the type variable). Record types are denoted as  $\{m: A, \dots\}$  where  $m$  is a label. As an example, the type  $(\forall a. a \rightarrow \text{Int}) \rightarrow (\text{Int} \sqcup \text{Bool})$  is written as  $\langle A \rangle (\_ : A) \Rightarrow \text{number} \Rightarrow (\text{number} \ | \ \text{boolean})$  in TypeScript. The expression syntax is more standard, including applications  $f(x)$ , record projections  $e.l$ , and function definitions **function**  $f(x: A) \{ \dots \}$ .

*Intersection and union types.* The following example shows how intersections are used to model objects that implement 2 interfaces ( $\{m: \text{number}\}$  and  $\{n: \text{boolean}\}$ ). The variable  $o1$  is an object that implements both interfaces  $m$  and  $n$  and has the type  $\{m: \text{number}\} \& \{n: \text{boolean}\}$ . 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) => ( _: 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  $\forall a. a \rightarrow a$ . 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 as instantiations. 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  $(\forall a. a \rightarrow a) \rightarrow \text{Int}$  and `g6` is of type  $(\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}$ . Since  $(\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \leq (\forall a. a \rightarrow a) \rightarrow \text{Int}$ , 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 combines HRP with explicit instantiation. In `ex7_1`, the `A` of `f7` is explicitly instantiated to  $\forall a. a \rightarrow a$ . The type of `f7` then becomes  $(\forall a. a \rightarrow a) \rightarrow (\forall a. a \rightarrow a)$ , 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  $\forall a. (a \sqcap a) \rightarrow (a \sqcup a)$ .

```
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 through the following examples. Example `ex8_1` is accepted because the first candidate is `number` and  $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \leq \text{Int} \rightarrow (\text{Int} \sqcap \text{Bool}) \rightarrow \text{Int}$ . In contrast, `ex8_3` is rejected because the first candidate is `number&boolean` and  $(\text{Int} \sqcap \text{Bool}) \rightarrow (\text{Int} \sqcap \text{Bool}) \rightarrow \text{Int} \not\leq (\text{Int} \sqcap \text{Bool}) \rightarrow \text{Int} \rightarrow \text{Int}$ . The situation for `ex8_2` and `ex8_4` is dual except that the correct instantiation is  $\text{Int} \sqcup \text{Bool}$ . The use of greedy instantiation is understandable and justifiable in practice. Adopting non-greedy instantiation could provide more precise instantiation in certain scenarios, but complicates the algorithm. Moreover, trying to always infer the best instantiation 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!

```

### 2.3 Our Approach

In this section, we provide an overview 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:  $F_{\sqcup\cap}^e$ , and its two extensions  $F_{\sqcup\cap}^e$  with records, and  $F_{\sqcup\cap}^e$  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  $F_{\sqcup\cap}^e$ . The examples using records require the extensions of  $F_{\sqcup\cap}^e$ , whereas the other examples work with the base formulation of  $F_{\sqcup\cap}^e$ . 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  $F_{\sqcup\cap}^e$ .** Our base type system  $F_{\sqcup\cap}^e$  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.  $F_{\sqcup\cap}^e$  has an algorithmic formulation that adopts a greedy strategy to find instantiations. We aim at a modest inference for base  $F_{\sqcup\cap}^e$  where intersections and unions are excluded from monotypes (i.e. implicitly instantiable types). The algorithm is sound and complete with respect to 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.  $F_{\sqcup\cap}^e$  has a careful treatment of the interaction among HRP, explicit type application, and intersection and union types to build a type system with desirable properties. Next, we highlight several differences to TypeScript, some of which are preferable, some of which could be arguable, but at least provide design alternatives for languages wanting to incorporate such features. These differences provide useful alternative design choices for both the developers of existing languages, as well as language designers interested in modeling programming languages with similar features.

*Order-relevant quantifiers.* 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 problems 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.  $\forall a. \forall b. a \rightarrow b \rightarrow a \leq \forall a. \text{Int} \rightarrow a \rightarrow \text{Int}$  is accepted. However, explicit instantiation with `Bool` changes the LHS type to  $\forall b. \text{Bool} \rightarrow b \rightarrow \text{Bool}$ , but the RHS type to  $\text{Int} \rightarrow \text{Bool} \rightarrow \text{Int}$ . The subtyping relation does not hold for these two new types, 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  $F_{\sqcup\cap}^e$ , we adopt order-relevant quantifiers instead. Thus, the order of quantifiers in types affects their behavior in subtyping, interacting well with explicit type applications, and leading to a

more principled design with HRP and explicit type applications. With order-relevant quantifiers,  $\forall a. \forall b. b \rightarrow a \rightarrow b \leq \forall a. \text{Int} \rightarrow a \rightarrow \text{Int}$  holds, but  $\forall a. \forall b. a \rightarrow b \rightarrow a \leq \forall a. \text{Int} \rightarrow a \rightarrow \text{Int}$  does not. However, this change impacts the subtyping relation considerably and is a non-trivial change compared to the original order-irrelevant quantifiers in HRP [Odersky and Läufer 1996].

*Inference of unannotated functions with monomorphic types.*  $F_{\sqcup}^e$  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  $F_{\sqcup}^e$  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 the type inferred for h10 is wrong
```

*Intersection introduction rule.* In standard type systems with intersection types and  $F_{\sqcup}^e$ , 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<sup>2</sup>. Example `ex11` is accepted in  $F_{\sqcup}^e$ .

```
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  $\forall a. A \rightarrow B$ . So it can not express types like  $\forall a. a$  or  $\forall a. (a \rightarrow \text{Int}) \sqcap (a \rightarrow \text{Bool})$ . There is no way to define functions that take the argument with the above type.  $F_{\sqcup}^e$  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 overloading.*  $F_{\sqcup}^e$  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  $F_{\sqcup}^e$ , as a more pragmatic option.

Compared with TypeScript on the behavior of examples presented in the previous section, our implementation for  $F_{\sqcup}^e$  additionally accepts `f3_2` and `ex8_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  $F_{\sqcup}^e$ 's better support of overloading. The acceptance of `ex8_3` demonstrates  $F_{\sqcup}^e$ 's completeness in finding solutions. The examples in this section are also accepted in our implementation.

$F_{\sqcup}^e$  **with records.** To illustrate the expressive power of  $F_{\sqcup}^e$  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  $F_{\sqcup}^e$ , they can be used to encode other practical features. We demonstrate one of such encodings: records as intersection types and reuse the

<sup>2</sup>TypeScript offers a mechanism called `overload signature` that can partially emulate the intersection introduction rule for functions.



type system of  $F_{\sqcup}^e$  to support record extension and projection. A record entry  $\{m : \text{number}\}$  is encoded as a function type  $\text{Label } m \rightarrow \text{Int}$ , 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  $F_{\sqcup}^e$ . Under such encoding,  $\text{o1}$  has type  $(\text{Label } m \rightarrow \text{Int}) \sqcap (\text{Label } n \rightarrow \text{Bool})$  and  $\text{h1}$  has type  $((\text{Label } m \rightarrow \text{Int}) \sqcap (\text{Label } n \rightarrow \text{Bool})) \sqcup ((\text{Label } k \rightarrow \text{String}) \sqcap (\text{Label } m \rightarrow \text{Int})) \rightarrow \text{Int}$ . Since the label type itself is also monotype,  $\text{var } \text{ex13} = (x \Rightarrow x.m) (\{m: 1\})$  can also be accepted without any annotations.

This extension has a sound and complete algorithm. It should be decidable as well, with a simple modification to our decidability proof of  $F_{\sqcup}^e$ . Our implementation for this type system additionally accepts all the expressions in the first example ( $\text{f1}, \text{g1}, \dots$ ) in the previous section.

**$F_{\sqcup}^e$  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>(<_:A> => (<_:A> => number = x => y => 1)
var g14: (<_:number | boolean> => (<_:number | boolean> => number = f14)
var h14_1: (<_:number> => (<_:boolean> => number = g14)
var h14_2: (<_:number> => (<_:boolean> => number = f14 // rejected!
```

This system strictly infers more types than the base  $F_{\sqcup}^e$ , at the cost of more backtracking, meaning all monotype instantiation without intersection and union types are guaranteed to be found. So, it still accepts  $\text{ex8\_2}$ , where  $\text{Int}$  is a valid instantiation. Our implementation for this type system additionally accepts  $\text{ex8\_4}$ , by picking  $\text{Int} \sqcup \text{Bool}$  as the instantiation. A limitation of the type system is that it does not infer any impredicative instantiation, i.e. a polymorphic type. Thus  $\text{ex15}$  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 to worklist-based algorithm and their formalization.

*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.

*New types of works and continuations to gather information.* The original worklist algorithm [Zhao et al. 2019] can be viewed as a linearized approximation of the tree-structure reasoning. However, it is not completely obvious that this linearized approach can be extended to systems that rely extensively on branching. Intersection and union types have heavy branching behavior. In particular, the rules related to matching and type application relations in  $F_{\sqcup\cap}^e$  are much more complex than in previous work, and previous techniques alone are not sufficient. We design new works and continuations to combine results from multiple branches needed by the matching ( $A_1 \sqcup A_2 \triangleright \omega$ ) and type application rule ( $A_1 \sqcup A_2 \circ B \Rightarrow \omega$ ) for union types by transforming them into nested continuations, and adding a final "combination" continuation to post-process the result. Their defunctionalized version also requires new techniques to handle the nested continuation by creating auxiliary works and continuations to explicitly relay the result of the first branch. These new designs prove that worklist-based algorithms can handle such situations and are also scalable to other (type) systems requiring similar reasoning.

*Decidability proof,* which gets very intricate due to intersections and unions (specifically, due to a duplication problem and the intersection introduction rule). The measure definitions are highly non-trivial as they now incorporate multiplication operations and recursive computation over the entire chain of continuations. We believe our work demonstrates the feasibility of decidability proofs for worklist-based algorithms with complex branching and provides some general recipes for 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., 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  $F_{\sqcup\cap}^e$  starts fresh using Coq. We build a new framework based on locally nameless representation [Charguéraud 2012] for worklist reasoning. Since the algorithm of  $F_{\sqcup\cap}^e$  has complex reduction behavior and our framework handles it well, we believe this framework could scale to the formalization of a wide range of systems. Last but not least, Coq also serves as a more recognized trust base compared to Abella.

### 3 Bidirectional Type System

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

#### 3.1 Syntax and Well-formedness

Type variables	$a, b$	
Types	$A, B, C$	$::= \mathbb{1} \mid a \mid \forall a. A \mid A \rightarrow B \mid \top \mid \perp \mid A \sqcap B \mid A \sqcup B$
Monotypes	$\tau, \sigma$	$::= \mathbb{1} \mid a \mid \tau \rightarrow \sigma$
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	$\Psi$	$::= \cdot \mid \Psi, x : A \mid \Psi, a \mid \Psi, \tilde{a}$

The syntax for  $F_{\sqcap}^e$  is shown above.  $F_{\sqcap}^e$  includes usual types such as the unit type, type variables  $a$ , universal types  $(\forall a. A)$ , function types  $(A \rightarrow B)$  and the top ( $\top$ ) and bottom ( $\perp$ ) type. In addition  $F_{\sqcap}^e$  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  $F_{\sqcap}^e$  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.*  $F_{\sqcap}^e$  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 predicative instantiations (i.e., monotype instantiations), we have to restrict the subtyping relation on monotypes 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. Otherwise, we can trivially have  $\tau := \perp \sqcap (\perp \rightarrow \perp)$  and  $\sigma := \perp$ , breaking such property. So the monotypes in  $F_{\sqcap}^e$  still consist of 3 cases: unit type, type variables (i.e.,  $a \in \Psi$ ), and function types of monotypes. On the other hand, all impredicative instantiations, i.e., instantiations containing universal types, top and bottom, and intersection and union types, have to be explicitly annotated.

Even if we could have taken a non-greedy approach, introducing intersection, union, top and bottom types in the solution domain is problematic. This would lead us to the *non-structural subtyping satisfiability/entailment problem*<sup>3</sup>, where types with different shapes can be related, bringing fundamental obstacles for bounding the depth of substitutions via any kind of standard occurs-check. That is, in most algorithms with unification [Dunfield and Krishnaswami 2013],  $\hat{a} \leq \tau$  ( $\tau \neq \hat{a}$ ) can be safely rejected without actually unifying  $\hat{a}$  with  $\tau$ , if  $\hat{a} \in \text{fv}(\tau)$ , because there is no solution for  $\hat{a}$  anyway. This check ensures that each successful unification always eliminates one existential variable. However, this is false when the solution domain is equipped with a non-structural subtyping relation (e.g.,  $\perp \leq \rightarrow$ ,  $\sqcap \leq \rightarrow$ , and  $\forall \leq \rightarrow$ ). Take  $\tau = \hat{a} \rightarrow \perp$  as an example,  $\perp$ ,  $\perp \sqcap (\perp \rightarrow \perp)$ ,  $\forall a. a$  all become valid solutions. What is worse, the number of solutions can even be infinite  $((\perp \rightarrow \perp) \rightarrow \perp, \dots, (\perp \sqcap (\perp \rightarrow \perp)) \sqcap ((\perp \sqcap (\perp \rightarrow \perp)) \rightarrow \perp), \dots, \forall a. a \rightarrow \perp, \dots)$ . The decidability of non-structural subtyping satisfiability for intersection types [Dudenhefner et al. 2016] and entailment for top and bottom types [Su et al. 2002] remain open problems.

*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.

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

<sup>3</sup>Satisfiability means whether a set of constraints has a solution. Entailment means whether a set of constraints entails another one, and is related to the simplification and transformation of constraints.

$\Psi \vdash A \leq B$

$A$  is a subtype of  $B$

$$\begin{array}{c}
\frac{}{\Psi \vdash \perp \leq \perp} \leq \perp \qquad \frac{}{\Psi \vdash A \leq \top} \leq \top \qquad \frac{}{\Psi \vdash \perp \leq A} \leq \perp \qquad \frac{}{\Psi \vdash a \leq a} \leq \text{TVar} \\
\frac{\Psi \vdash B_1 \leq A_1 \quad \Psi \vdash A_2 \leq B_2}{\Psi \vdash A_1 \rightarrow A_2 \leq B_1 \rightarrow B_2} \leq \rightarrow \qquad \frac{\Psi \vdash \tau \quad \Psi \vdash [\tau/a]A \leq B \quad B \neq \forall}{\Psi \vdash \forall a. A \leq B} \leq \forall \text{L} \qquad \frac{\Psi, \tilde{a} \vdash A \leq B}{\Psi \vdash \forall a. A \leq \forall a. B} \leq \forall \\
\frac{\Psi \vdash A \leq B_1 \quad \Psi \vdash A \leq B_2}{\Psi \vdash A \leq B_1 \sqcap B_2} \leq \sqcap \text{R} \qquad \frac{\Psi \vdash A_1 \leq B}{\Psi \vdash A_1 \sqcap A_2 \leq B} \leq \sqcap \text{L}_1 \qquad \frac{\Psi \vdash A_2 \leq B}{\Psi \vdash A_1 \sqcap A_2 \leq B} \leq \sqcap \text{L}_2 \\
\frac{\Psi \vdash A_1 \leq B \quad \Psi \vdash A_2 \leq B}{\Psi \vdash A_1 \sqcup A_2 \leq B} \leq \sqcup \text{L} \qquad \frac{\Psi \vdash A \leq B_1}{\Psi \vdash A \leq B_1 \sqcup B_2} \leq \sqcup \text{R}_1 \qquad \frac{\Psi \vdash A \leq B_2}{\Psi \vdash A \leq B_1 \sqcup B_2} \leq \sqcup \text{R}_2
\end{array}$$

Fig. 2. Subtyping

Otherwise, with the original free variable check in [Zhao and Oliveira \[2022\]](#) 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  $\forall a. a \rightarrow a$  twice, we will first get  $\forall b. (\forall a. a \rightarrow a \sqcap b) \leq \forall a. a \rightarrow a$ , and then  $(\forall a. a \rightarrow a) \sqcap (\forall a. a \rightarrow a) \leq (\forall a. a \rightarrow a) \rightarrow (\forall a. a \rightarrow a)$ . 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  $\leq \forall \text{L}$ , union types can use the usual free variable check.

### 3.2 Subtyping

Figure 2 shows the rules of subtyping relation. The basic rules are standard, including rules  $\leq \rightarrow$ ,  $\leq \top$ , and  $\leq \perp$ . Rules  $\leq \sqcap \text{R}$ ,  $\leq \sqcap \text{L}_1$ ,  $\leq \sqcap \text{L}_2$ ,  $\leq \sqcup \text{L}$ ,  $\leq \sqcup \text{R}_1$ ,  $\leq \sqcup \text{R}_2$  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  $\leq \forall \text{L}$  and  $\leq \forall$  deal with HRP. Rule  $\leq \forall \text{L}$  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  $\tau$ . 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  $\leq \forall \text{L}$ . In previous systems with these two rules [\[Cui et al. 2023; Zhao and Oliveira 2022\]](#), a simple syntactic check “ $B \neq \forall. *$ ” in rule  $\leq \forall \text{L}$  ensures that rule  $\leq \forall$  always takes priority when both sides are universal types (this prioritization 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:

- $A \leq B : \forall a. \forall b. b \rightarrow a \leq (\forall a. a \rightarrow \perp) \sqcap (\forall a. a \rightarrow \perp)$
- $B \leq C : (\forall a. a \rightarrow \perp) \sqcap (\forall a. a \rightarrow \perp) \leq (\forall a. a \rightarrow \perp)$
- $A \not\leq C : \forall a. \forall b. b \rightarrow a \not\leq \forall a. a \rightarrow \perp$

This counterexample is because  $A \leq B$  can use the  $\leq \forall \text{L}$  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  $(\forall a. a \rightarrow \perp) \sqcup (\forall a. a \rightarrow \perp)$ ) and when

$B$  is not idempotent to  $C$  (e.g., change  $B$  to  $(\forall a. a \rightarrow \mathbb{1}) \sqcup \perp$ ). The direct solution to avoid such a bypass is to always prioritize the rules for intersection and union types (rules  $\leq \sqcap R$ ,  $\leq \sqcup R_1$  and  $\leq \sqcup R_2$ ) 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.  $\forall a. (a \rightarrow \mathbb{1}) \sqcup (a \rightarrow (\mathbb{1} \rightarrow \mathbb{1})) \leq (\mathbb{1} \rightarrow \mathbb{1}) \sqcup (\mathbb{1} \rightarrow (\mathbb{1} \rightarrow \mathbb{1}))$ . 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:

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

This syntactic check enjoys a good semantic interpretation that if  $A^{\neq \forall}$  then forall  $\tilde{a} \in \Psi, B, C$ ,  $\Psi \vdash [C/a]A \not\leq \forall a. B$ . Thus,  $\leq \forall L$  with this syntactic check is strictly more expressive than  $F_{\leq}^e$  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:  $\forall a. (a \rightarrow \mathbb{1}) \sqcap (a \rightarrow (\mathbb{1} \rightarrow \mathbb{1})) \leq (\mathbb{1} \rightarrow \mathbb{1}) \sqcap ((\mathbb{1} \rightarrow \mathbb{1}) \rightarrow (\mathbb{1} \rightarrow \mathbb{1}))$ .

*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  $\Psi \vdash A$ , then  $\Psi \vdash A \leq A$ .*

**THEOREM 3.2 (SUBTYPING TRANSITIVITY).** *Given  $\vdash \Psi$ ,  $\Psi \vdash A$ ,  $\Psi \vdash B$  and  $\Psi \vdash C$ , if  $\Psi \vdash A \leq B$ , and  $\Psi \vdash B \leq C$  then  $\Psi \vdash A \leq C$ .*

### 3.3 Typing

Figure 3 shows the bidirectional type system with the checking and inference judgments. Variables, annotations, unit value and type abstractions are directly inferable, and they are dealt with by rules  $\Rightarrow \text{Var}$ ,  $\Rightarrow \text{Anno}$ ,  $\Rightarrow \text{Unit}$  and  $\Rightarrow \Lambda$  respectively. We also allow the inference of abstractions as long as their types are monotypes, as in rule  $\Rightarrow \rightarrow \text{Mono}$ . 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 ( $\Leftarrow \rightarrow$ ) or  $\top$  ( $\Leftarrow \rightarrow \top$ ). An expression can also be checked against a supertype, if it infers a subtype ( $\Leftarrow \text{Sub}$ ).  $\Leftarrow \sqcap$ ,  $\Leftarrow \sqcup_1$ , 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  $\lambda x.x : (\mathbb{1} \rightarrow \mathbb{1}) \sqcap ((\mathbb{1} \rightarrow \mathbb{1}) \rightarrow (\mathbb{1} \rightarrow \mathbb{1}))$ . Rules for union introduction are needed since the subsumption rule  $\Leftarrow \text{Sub}$  does not cover the case when  $e$  cannot infer either branch.

*Matching.* The matching judgment  $\Psi \vdash A \triangleright B \rightarrow C$  in Figure 3 states that a type  $A$  can be regarded as a subtype of function type  $B \rightarrow C$ , 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  $F_{\leq}^b$  [Cui et al. 2023], which uses a similar judgment to deal with HRP with bounded quantification. Rules  $\triangleright \rightarrow$ ,  $\triangleright \perp$  are the base cases,

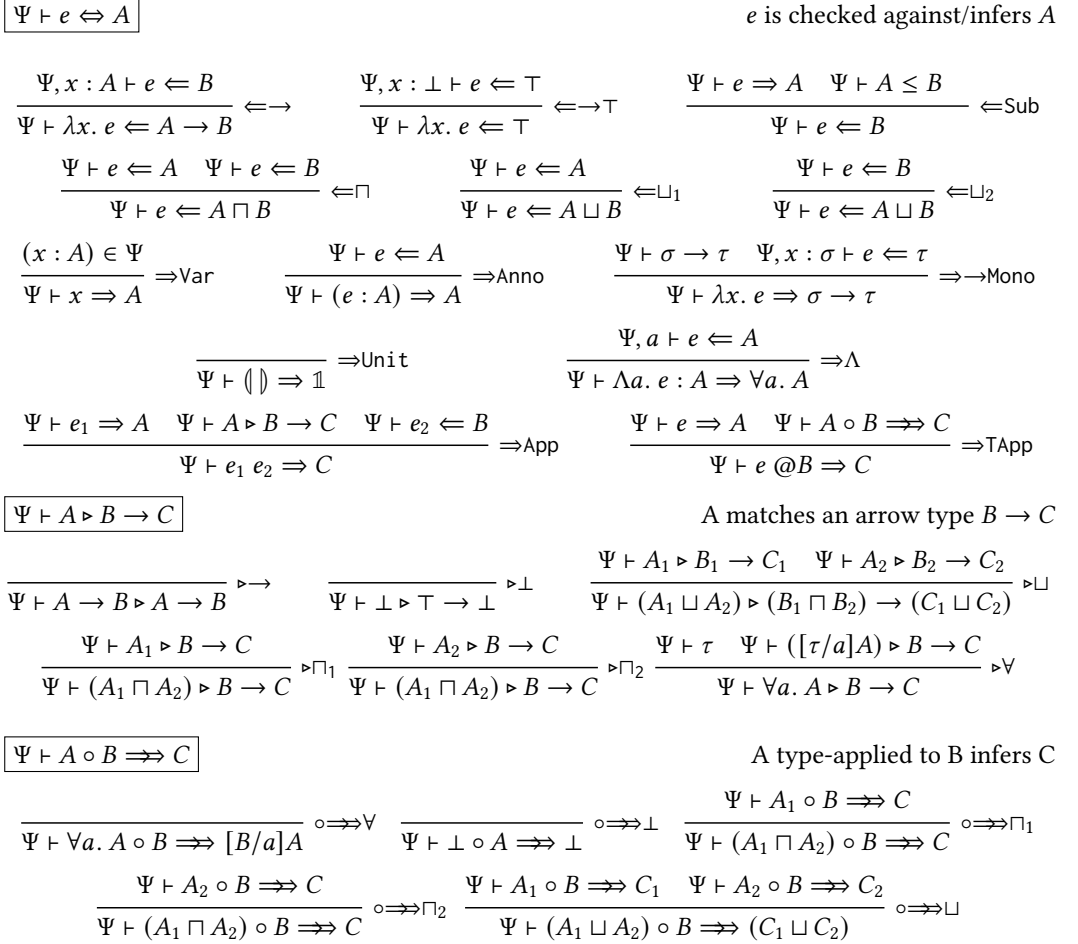


Fig. 3. Checking, Inference, Matching and Type Application

since a function type or  $\perp$  can be regarded as a function type directly. Rule  $\triangleright \forall$  converts a universal type to a function type by guessing a monotype  $\tau$  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  $F_{\sqcap, \sqcup}^e$ :  $(A_1 \rightarrow B_1) \sqcap (A_2 \rightarrow B_2) \leq (A_1 \sqcup A_2) \rightarrow (B_1 \sqcap B_2)$ .

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  $\vdash \Psi$  and  $\Psi \vdash A$ ,*

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

*Type application.* The type application judgment  $\Psi \vdash A \circ B \Rightarrow C$  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  $\circ \Rightarrow \forall$  and  $\circ \Rightarrow \perp$  are the base cases where the type-application result is

known:  $[B/a]A$  and  $\perp$ . Similarly to matching, we also need to extend the type application to deal with intersection and union types. Rule  $\circ \Rightarrow \sqcap_1$  and  $\circ \Rightarrow \sqcap_2$  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  $F_{\sqcap}^e$  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  $\circ \Rightarrow \sqcup$  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  $\perp$ ), as formulated by the following lemma. A more general version, where  $A$  containing  $\perp$  is considered, is shown in Appendix C.

LEMMA 3.4 (SPECIFICATION OF TYPE-APPLICATION). *Given  $\vdash \Psi, \Psi \vdash A$  and  $\Psi \vdash B$ ,*

- (1) *if  $A \perp$  and  $\Psi \vdash A \circ B \Rightarrow C$ , then exists  $A', \Psi \vdash \forall a. A', \Psi \vdash A \leq \forall a. A', C = [B/a]A'$ ;*
- (2) *if  $\Psi \vdash \forall a. A'$  and  $\Psi \vdash A \leq \forall a. A'$ , then exists  $C$  s.t.  $\Psi \vdash A \circ B \Rightarrow C$  and  $\Psi \vdash C \leq [B/a]A'$ .*

## 4 Algorithmic System

This section introduces an algorithmic type system that implements the specification of  $F_{\sqcap}^e$  presented in Section 3 using the worklist approach [Zhao et al. 2019]. A worklist  $\Gamma$  is an ordered list of both (type) variable declarations (with bindings) and works, whose syntax is introduced in Sec. 4.1. The algorithm can be viewed as a non-deterministic rewriting system over the worklist. In particular, at each step, one or more rules (introduced in Sec. 4.2) could be applied according to the last entry in the worklist to reduce (or, rewrite) the worklist. If a worklist can be reduced to an empty worklist, it is accepted. Otherwise, it is rejected. 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,  $a, \tilde{a}, \hat{a}$ , respectively. Due to this uniform representation, algorithmic types have the same syntax as declarative types<sup>4</sup>. The expression syntax remains unchanged as well. Works  $w$  are judgments to process. Two new works are added to  $F_{\sqcap}^e$  compared with  $F_{\leq}^e$ : 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.

Work	$w$	$::=$	$A \leq B \mid e \leftarrow A \mid e \Rightarrow_{\alpha} \omega \mid A \circ B \Rightarrow_{\alpha} \omega \mid A \triangleright_{\alpha, \beta} \omega \mid$ $A \rightarrow B \bullet e \Rightarrow_{\alpha} \omega \mid (A_1 \rightarrow B_1) \cup_{\triangleright} (A_2 \rightarrow B_2) \blacktriangleright_{\alpha} \omega \mid$ $A_1 \cup_{\circ} A_2 \blacktriangleright_{\alpha} \omega$
Algorithmic worklist	$\Gamma$	$::=$	$\cdot \mid \Gamma, a \mid \Gamma, \tilde{a} \mid \Gamma, \hat{a} \mid \Gamma, x : A \mid \Gamma \Vdash \omega$

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  $\omega$ , which is syntactic sugar for  $A \triangleright w$ , i.e., a meta function from type(s) to work. The argument for a continuation is represented with Greek letters  $\alpha, \beta, \gamma$ . The continuation application is now presented as  $\omega \diamond A$ . This representation is more aligned with our formalization.  $\Gamma \vdash A, \Gamma \vdash e, \Gamma \vdash w$ , and  $\vdash \Gamma$  denote the well-formedness of each syntactic category, whose detailed definitions are covered in Appendix B.1.

<sup>4</sup>We use declarative types mean types in the bidirectional system for brevity, similar for monotypes, expressions, etc.

$\Gamma \longrightarrow \Gamma'$

 $\Gamma$  worklist-reduces to  $\Gamma'$ 

$$\begin{array}{l}
\Gamma \Vdash \mathbb{1} \leq \mathbb{1} \longrightarrow_5 \Gamma \\
\Gamma \Vdash a \leq a \longrightarrow_6 \Gamma \\
\Gamma \Vdash A \leq \top \longrightarrow_7 \Gamma \\
\Gamma \Vdash \perp \leq A \longrightarrow_8 \Gamma \\
\Gamma \Vdash A_1 \rightarrow A_2 \leq B_1 \rightarrow B_2 \longrightarrow_9 \Gamma \Vdash B_1 \leq A_1 \Vdash A_2 \leq B_2 \\
\Gamma \Vdash \forall a. A \leq B \longrightarrow_{10} \Gamma, \widehat{a} \Vdash A \leq B \quad \text{when } B^{\neq \forall} \\
\Gamma \Vdash \forall a. A \leq \forall a. B \longrightarrow_{11} \Gamma, \widetilde{a} \Vdash A \leq B \\
\Gamma \Vdash A \leq B_2 \sqcap B_2 \longrightarrow_{12} \Gamma \Vdash A \leq B_1 \Vdash A \leq B_2 \\
\Gamma \Vdash A_1 \sqcap A_2 \leq B \longrightarrow_{13} \Gamma \Vdash A_1 \leq B \\
\Gamma \Vdash A_1 \sqcap A_2 \leq B \longrightarrow_{14} \Gamma \Vdash A_2 \leq B \\
\Gamma \Vdash A_1 \sqcup A_2 \leq B \longrightarrow_{15} \Gamma \Vdash A_1 \leq B \Vdash A_2 \leq B \\
\Gamma \Vdash A \leq B_1 \sqcup B_2 \longrightarrow_{16} \Gamma \Vdash A \leq B_1 \\
\Gamma \Vdash A \leq B_1 \sqcup B_2 \longrightarrow_{17} \Gamma \Vdash A \leq B_2 \\
\Gamma \Vdash a \leq \tau \longrightarrow_{18} \{\tau/a\}\Gamma \quad \text{when } \widehat{a} \in \Gamma \wedge a \notin \text{fv}(\tau) \\
\Gamma \Vdash \tau \leq a \longrightarrow_{19} \{\tau/a\}\Gamma \quad \text{when } \widehat{a} \in \Gamma \wedge a \notin \text{fv}(\tau) \\
\Gamma \Vdash a \leq A \rightarrow B \longrightarrow_{20} \{a_1 \rightarrow a_2/a\}(\Gamma, \widehat{a}_1, \widehat{a}_2, a \leq A \rightarrow B) \\
\quad \text{when } \widehat{a} \in \Gamma \wedge \Gamma \not\vdash^m A \rightarrow B \\
\Gamma \Vdash A \rightarrow B \leq a \longrightarrow_{21} \{a_1 \rightarrow a_2/a\}(\Gamma, \widehat{a}_1, \widehat{a}_2, A \rightarrow B \leq a) \\
\quad \text{when } \widehat{a} \in \Gamma \wedge \Gamma \not\vdash^m A \rightarrow B
\end{array}$$

Fig. 4. Algorithmic Worklist Reduction (Subtyping)

$\Gamma \longrightarrow \Gamma'$  denotes the one-step worklist reduction and  $\Gamma \longrightarrow^* \Gamma'$  denotes the zero-or-more-step 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.

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

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 17 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 specification 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  $\tau$  instantiation in its specification 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



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

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

Fig. 5. Worklist Substitution

their specification counterparts. In rules 9, 12, and 15, multiple new entries are pushed back to the worklist, while their specification 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 ( $a \notin \text{fv}(A \rightarrow B)$ ) would cause the loss of completeness in  $F_{\sqcap}^e$ . 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  $a$ .

The substitution operations in these four rules are encapsulated in the worklist substitution  $\{\tau/\widehat{a}\}\Gamma$ . The  $\{\tau/\widehat{a}\}\Gamma$  operation not only substitutes every occurrence of  $\widehat{a}$  to  $\tau$  in  $\Gamma$  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.

Rules 22-24 are the algorithmic counterparts of  $\Leftarrow\text{Sub}$ ,  $\Leftarrow\rightarrow$  and  $\Leftarrow\rightarrow\top$ . Rules 25-27 are new, corresponding to rules  $\Leftarrow\sqcap$ ,  $\Leftarrow\sqcup_1$  and  $\Leftarrow\sqcup_2$  in the specification. The premise of the inference judgment in  $\Leftarrow\text{Sub}$  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  $a_1 \rightarrow a_2$  using the worklist substitution.

Rules 29-35 are the algorithmic counterparts of  $\Rightarrow\text{Var}$ ,  $\Rightarrow\text{Anno}$ ,  $\Rightarrow\Lambda$ ,  $\Rightarrow\text{Unit}$ ,  $\Rightarrow\text{App}$ ,  $\Rightarrow\text{TApp}$  and  $\Rightarrow\rightarrow\text{Mono}$ . 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

$\Gamma \longrightarrow \Gamma'$

 $\Gamma$  worklist-reduces to  $\Gamma'$ 

$$\begin{aligned}
& \Gamma \Vdash e \Leftarrow B \longrightarrow_{22} \Gamma \Vdash e \Rightarrow_{\alpha} \alpha \leq B \\
& \Gamma \Vdash \lambda x. e \Leftarrow A \rightarrow B \longrightarrow_{23} \Gamma, x : A \Vdash e \Leftarrow B \\
& \Gamma \Vdash \lambda x. e \Leftarrow \top \longrightarrow_{24} \Gamma, x : \perp \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 \rightarrow 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 \quad \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_2} (\gamma_1 \rightarrow \gamma_2 \bullet e_2 \Rightarrow_{\alpha} \omega)) \\
& \Gamma \Vdash e @A \Rightarrow_{\alpha} \omega \longrightarrow_{34} \Gamma \Vdash e \Rightarrow_{\beta} (\beta \circ A \Rightarrow_{\alpha} \omega) \\
& \Gamma \Vdash \lambda x. e \Rightarrow_{\alpha} \omega \longrightarrow_{35} \Gamma, \widehat{a}_1, \widehat{a}_2 \Vdash \omega \diamond (a_1 \rightarrow a_2), x : a_1 \Vdash e \Leftarrow a_2 \\
& \Gamma \Vdash A \rightarrow B \triangleright_{\alpha, \beta} \omega \longrightarrow_{36} \Gamma \Vdash \omega \diamond A \diamond B \\
& \Gamma \Vdash \perp \triangleright_{\alpha, \beta} \omega \longrightarrow_{37} \Gamma \Vdash \top \rightarrow \perp \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 \rightarrow \beta_2 \cup_{\gamma_1 \rightarrow \gamma_2} \triangleright_{\alpha_1, \alpha_2} \omega)) \\
& \Gamma \Vdash a \triangleright_{\alpha, \beta} \omega \longrightarrow_{42} \{a_1 \rightarrow 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 \rightarrow B_1) \cup_{\triangleright} (A_2 \rightarrow B_2) \triangleright_{\alpha, \beta} \omega \longrightarrow_{43} \Gamma \Vdash \omega \diamond (A_1 \sqcap A_2) \diamond (B_1 \sqcup B_2) \\
& \Gamma \Vdash A \rightarrow B \bullet e \Rightarrow_{\alpha} \omega \longrightarrow_{44} \Gamma \Vdash \omega \diamond B \Vdash e \Leftarrow A \\
& \Gamma \Vdash \forall a. A \circ B \Rightarrow_{\alpha} \omega \longrightarrow_{45} \Gamma \Vdash \omega \diamond ([B/a]A) \\
& \Gamma \Vdash \perp \circ A \Rightarrow_{\alpha} \omega \longrightarrow_{46} \Gamma \Vdash \omega \diamond \perp \\
& \Gamma \Vdash A_1 \sqcap A_2 \circ B \Rightarrow_{\alpha} \omega \longrightarrow_{47} \Gamma \Vdash A_1 \circ B \Rightarrow_{\alpha} \omega \\
& \Gamma \Vdash A_1 \sqcap A_2 \circ B \Rightarrow_{\alpha} \omega \longrightarrow_{48} \Gamma \Vdash A_2 \circ B \Rightarrow_{\alpha} \omega \\
& \Gamma \Vdash A_1 \sqcup A_2 \circ B \Rightarrow_{\alpha} \omega \longrightarrow_{49} \Gamma \Vdash A_1 \circ B \Rightarrow_{\beta_1} (A_2 \circ B \Rightarrow_{\beta_2} (\beta_1 \cup_{\circ} \beta_2 \triangleright \omega)) \\
& \Gamma \Vdash A_1 \cup_{\circ} A_2 \triangleright \omega \longrightarrow_{50} \Gamma \Vdash \omega \diamond (A_1 \sqcup A_2)
\end{aligned}$$

Fig. 6. Algorithmic Worklist Reduction (Typing)

the second result (i.e., the codomain type) to the original continuation  $\omega$ . 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  $\omega$  to  $a_1 \rightarrow a_2$  and checking the body  $e$  against  $a_2$ .

Rules 36-38 are the algorithmic counterparts of  $\triangleright$ ,  $\triangleright \perp$  and  $\triangleright \forall$ . Rules 36 and 37 are two base cases where a function type is known (by lifting  $\perp$  to  $\top \rightarrow \perp$ ). 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 specification 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 ( $\beta_1$  and  $\beta_2$ ) of  $\beta_1 \rightarrow \beta_2 \Downarrow \gamma_1 \rightarrow \gamma_2 \blacktriangleright \omega$ . Then  $A_2$  is matched and passes the result to the latter two arguments ( $\gamma_1$  and  $\gamma_2$ ) 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  $A_2 \blacktriangleright \dots$  is a bit concerning since it sees an extended context due to the process of  $A_1 \blacktriangleright \dots$ . 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  $A_2 \blacktriangleright \dots$  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  $\circ \Rightarrow \forall$ ,  $\circ \Rightarrow \perp$ ,  $\circ \Rightarrow \sqcap_1$ ,  $\circ \Rightarrow \sqcap_2$ , and  $\circ \Rightarrow \sqcup$ . 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 specification. 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  $\cdot \vDash e \Rightarrow \alpha \leq \top \longrightarrow^* \cdot$ .

**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, checking subsumption, and type safety. Then we discuss the soundness and completeness of the algorithmic type system. For soundness 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  $B^\circ$  [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.

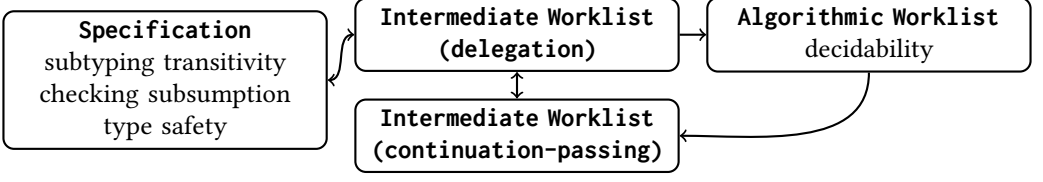


Fig. 7. Overall Proof Structure. Arrows indicate the relative soundness and completeness between two systems. Properties proved in each system are listed in each box.

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  $\Psi_1, [C/a]\Psi_2 \vdash [C/a]A \leq [C/a]B$ .*

*Checking subsumption.* To prove the checking subsumption theorem, 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  $\Psi, A, A'$  well-formed,*

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

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

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  $\Psi, \Psi', e, A$  well-formed and  $\Psi' <: \Psi$ ,*

- (1) *if  $\Psi \vdash e \Leftarrow A$ ,  $\Psi \vdash A' \leq A$ , 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 product and sum types ( $F_{\times+}$ ). 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  $\Psi, e, A$  well-formed, if  $\Psi \vdash e \Leftarrow A \hookrightarrow e'$  then  $|\Psi| \vdash_{F_{\times+}} e' : |A|$*

## 5.2 Soundness and Completeness

To help formalize the correspondence between the specification and algorithmic systems, we build an intermediate system: intermediate worklist [Zhao et al. 2019]. The intermediate worklist has a similar syntax to  $\Gamma$  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, reduction by delegation  $\Omega \rightarrow \Omega'$  and reduction by continuation-passing  $\Omega \rightarrow_{\omega} \Omega'$ , for soundness and completeness proofs, respectively.  $\Omega \rightarrow \Omega'$  is a rephrasing of the judgments in the bidirectional system using the worklist syntax, while  $\Omega \rightarrow_{\omega} \Omega'$  mimics the algorithmic reduction using the continuation-passing style but still guesses the monotype  $\tau$  instead of introducing existential variables. The detailed rules of well-formedness and reduction of the intermediate worklist are listed in Appendix B.1.  $\Omega \rightarrow^* \cdot$  and  $\Omega \rightarrow_{\omega}^* \cdot$  are proved equivalent:

**THEOREM 5.5 (EQUIVALENCE OF INTERMEDIATE WORKLIST REDUCTION).** *If  $\vdash \Omega$ , then  $\Omega \longrightarrow^* \cdot$  iff  $\Omega \longrightarrow_{\omega}^* \cdot$ .*

For soundness and completeness of the algorithm, the form presented in Thm. 4.1 cannot be proved by induction directly. This form is too specific as it only includes worklists with: (1) only one inference or checking work and; (2) all expressions and types well-formed under the empty context. Proving by induction requires generalizing this theorem to discuss the properties of a wider range of worklists with: (1) arbitrary number and type of works; and (2) free variables. The latter generalization is tricky since the algorithmic worklist has one more type of free variables, existential variables. Thus, we need to define a relation to relate algorithmic worklists and intermediate worklists. Previous works [Cui et al. 2023; Zhao and Oliveira 2022; Zhao et al. 2019] use the following transfer relation, which relates a worklist  $\Omega$  with all algorithmic worklists  $\Gamma$  if  $\Gamma$  is equal to  $\Omega$  under a substitution of all existential variables in  $\Gamma$  to a well-formed declarative monotype.

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

This relation is analogous to logical relations used in the proof of contextual equivalence [Reynolds 1983]: they are both developed to interpret free variables. Here, this relation interprets free existential variables as arbitrary well-formed declarative monotypes. This transfer relation is conceptually enough for the proof, but it presents three practical problems. Firstly, it analyzes the algorithmic worklist from the beginning to the end, contrary to how it is inductively defined. Secondly, it has pervasive use of substitution, especially the substitution over the whole worklist. Thirdly, it erases the information of the positions and instantiations of existential variables. These problems complicate the reasoning. The reversed definition starting from the beginning of the worklist and the erasure of the structure complicate the inductive reasoning of certain properties. The substitution operation requires a large number of inversion lemmas to reason about the correspondence between the shape of type, expression, and work of two related worklists  $\Gamma$  and  $\Omega$  (e.g. if the last work in  $\Omega$  is a subtyping judgment, so should be for  $\Gamma$ ). Not to mention the intrinsic difficulty of reasoning about substitutions in proof assistants like Coq without built-in support for binders.

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 ( $\theta \vDash \omega^s \rightsquigarrow \omega^{s'}$  and  $\theta \vDash \omega^d \rightsquigarrow \omega^{d'}$ )<sup>5</sup> works ( $\theta \vDash w \rightsquigarrow w'$ ), and worklists ( $\theta \vDash \Gamma \rightsquigarrow \Omega \vDash \theta'$ ).  $\theta$  is a substitution set  $\theta := \cdot \mid \theta, a \mid \theta, \widehat{a} \mid \theta, \widehat{a} : \tau$  that keeps track of each existential variable  $a$  and its instantiation  $\tau$ , which is a well-formed declarative monotype.  $\theta$  also keeps track of type variables so that the well-formedness of each instantiation  $\tau$  can be checked just by inspecting  $\theta$ .

The transfer relation of types, expressions, continuations, and works should be read as: under the current substitution set  $\theta$ , 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  $\tau$  bound in  $\theta$ , respectively. The transfer relation of types, expressions, and worklists is shown in Figure 8. The transfer relation of continuations and works follows the same routine, as they always get decomposed to transfer types and expressions. The concrete rules are shown in Figure 15 in Appendix B.3. The transfer of worklists ( $\theta \vDash \Gamma \rightsquigarrow \Omega \vDash \theta'$ ) should be read as, the worklist  $\Gamma$  is transferred to  $\Omega$  under substitution set  $\theta$  and  $\theta$  is extended to  $\theta'$  with more type variables and existential variables in  $\Gamma$ . When the last entry of the algorithmic worklist is a type variable, subtype variable, or existential variable,  $\theta$  gets updated. Type variables

<sup>5</sup>Instead of using HOAS, we adopt a defunctionalized style for continuation, with details shown in Appendix B.4. So the continuation has its own syntax categories.

$\theta \vDash A \rightsquigarrow A'$	$A$ is transferred to $A'$ under $\theta$
$\frac{}{\theta \vDash \mathbb{1} \rightsquigarrow \mathbb{1}} \quad \frac{}{\theta \vDash \top \rightsquigarrow \top} \quad \frac{}{\theta \vDash \perp \rightsquigarrow \perp}$	$\frac{a \in \theta \vee \widehat{a} \in \theta \quad \widehat{a} : \tau \in \theta}{\theta \vDash a \rightsquigarrow a \quad \theta \vDash a \rightsquigarrow \tau} \quad \frac{\theta, a \vDash A \rightsquigarrow A'}{\theta \vDash \forall a. A \rightsquigarrow \forall a. A'}$
$\frac{\theta \vDash A_1 \rightsquigarrow A'_1 \quad \theta \vDash A_2 \rightsquigarrow A'_2}{\theta \vDash A_1 \rightarrow A_2 \rightsquigarrow A'_1 \rightarrow A'_2} \quad \frac{\theta \vDash A_1 \rightsquigarrow A'_1 \quad \theta \vDash A_2 \rightsquigarrow A'_2}{\theta \vDash A_1 \sqcap A_2 \rightsquigarrow A'_1 \sqcap A'_2} \quad \frac{\theta \vDash A_1 \rightsquigarrow A'_1 \quad \theta \vDash A_2 \rightsquigarrow A'_2}{\theta \vDash A_1 \sqcup A_2 \rightsquigarrow A'_1 \sqcup A'_2}$	
$\theta \vDash e \rightsquigarrow e'$	$e$ is transferred to $e'$ under $\theta$
$\frac{}{\theta \vDash x \rightsquigarrow x} \quad \frac{}{\theta \vDash () \rightsquigarrow ()} \quad \frac{\theta \vDash e \rightsquigarrow e'}{\theta \vDash \lambda x. e \rightsquigarrow \lambda x. e'}$	$\frac{\theta \vDash e'_1 \rightsquigarrow e'_1 \quad \theta \vDash e'_2 \rightsquigarrow e'_2}{\theta \vDash e_1 e_2 \rightsquigarrow e'_1 e'_2}$
$\frac{\theta \vDash e \rightsquigarrow e' \quad \theta \vDash A \rightsquigarrow A'}{\theta \vDash e : A \rightsquigarrow e' : A'} \quad \frac{\theta \vDash e \rightsquigarrow e' \quad \theta \vDash A \rightsquigarrow A'}{\theta \vDash e @ A \rightsquigarrow e' @ A'} \quad \frac{\theta, a \vDash e \rightsquigarrow e' \quad \theta, a \vDash A \rightsquigarrow A'}{\theta \vDash \Lambda a. e : A \rightsquigarrow \Lambda a. e' : A'}$	
$\theta \vDash \Gamma \rightsquigarrow \Omega \vDash \theta'$	$\Gamma$ is transferred to $\Omega$ with $\theta$ updated to $\theta'$
$\frac{}{\theta \vDash \cdot \rightsquigarrow \cdot \vDash \theta} \quad \frac{\theta \vDash \Gamma \rightsquigarrow \Omega \vDash \theta'}{\theta \vDash \Gamma, a \rightsquigarrow \Omega, a \vDash \theta', a} \quad \frac{\theta \vDash \Gamma \rightsquigarrow \Omega \vDash \theta'}{\theta \vDash \Gamma, \widehat{a} \rightsquigarrow \Omega, \widehat{a} \vDash \theta', \widehat{a}} \quad \frac{\theta \vDash \Gamma \rightsquigarrow \Omega \vDash \theta' \quad [\theta'] \vdash \tau}{\theta \vDash \Gamma, \widehat{a} \rightsquigarrow \Omega \vDash \theta', \widehat{a} : \tau}$	
$\frac{\theta \vDash \Gamma \rightsquigarrow \Omega \vDash \theta' \quad \theta' \vDash A \rightsquigarrow A'}{\theta \vDash \Gamma, x : A \rightsquigarrow \Omega, x : A' \vDash \theta'} \quad \frac{\theta \vDash \Gamma \rightsquigarrow \Omega \vDash \theta' \quad \theta' \vDash w \rightsquigarrow w'}{\theta \vDash \Gamma \Vdash w \rightsquigarrow \Omega \Vdash w' \vDash \theta'}$	

Fig. 8. Syntax-directed transfer for types, expressions and worklists.

and subtype variables are kept in the transferred intermediate worklist and existential variables get erased. The last two cases (variables and works) of worklist transfer are straightforward. It is not hard to informally verify that, when the input substitution set is empty, this new syntax-directed transfer relation is equivalent to the one used by Zhao et al. [2019].

Soundness and completeness are built upon this new syntax-directed transfer to relate a intermediate 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  $\theta, \Omega$ , s.t.  $\cdot \vDash \Gamma \rightsquigarrow \Omega \vDash \theta$ , and  $\Omega \longrightarrow^* \cdot$ .*

**THEOREM 5.7 (COMPLETENESS).** *If  $\Omega \longrightarrow_{\omega}^* \cdot, \vdash \Gamma$ , and  $\cdot \vDash \Gamma \rightsquigarrow \Omega \vDash \theta$ , 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.* 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  $\theta$  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  $\widehat{a} : \tau$  ( $\theta \stackrel{\widehat{a}}{\equiv} \theta'$ ). 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  $\cdot \vDash \{\tau/a\}\Gamma \rightsquigarrow \Omega \vDash \theta$ , then exists  $\theta', \tau'$  s.t.  $\cdot \vDash \Gamma \rightsquigarrow \Omega \vDash \theta'$ ,  $\theta \vDash \tau \rightsquigarrow \tau'$ ,  $\theta' \vDash \tau \rightsquigarrow \tau'$ , and  $\widehat{a} : \tau' \in \theta', \theta \stackrel{\widehat{a}}{\equiv} \theta'$ , and  $\vdash \theta'$ ;*



splitting proposed in  $F_{\leq}^b$  [Cui et al. 2023]: when every split destructs a polytype function type  $A \rightarrow B$ , every non-monotype component in it will have a smaller rank. This generalizes the split measure used by Cui et al. to provide a uniform treatment for all polytypes and avoids the unintuitive post-computation needed by the original measure. This measure always decreases in rule 20 and 21 whether  $a \in \text{fv}(A \rightarrow B)$  or not, so the occurs-check is not needed for them.

*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 new judgments becomes structurally smaller. For example, in rule 9,  $A_1, A_2, B_1, B_2$  are all structurally smaller than  $A_1 \rightarrow A_2$  and  $B_1 \rightarrow B_2$ . However, in  $F_{\sqcup\cap}^e$ , certain types and expressions may just get duplicated twice without any changes. 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  $|B|_{\sqcup\cap}$  times and  $B$  may be duplicated up to  $|A|_{\sqcup\cap}$  times ( $|\cdot|_{\sqcup\cap}$  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  $|A|_{\leq} \cdot (|B|_{\sqcup\cap} + 1) + |B|_{\leq} \cdot (|A|_{\sqcup\cap} + 1)$ . Expression duplication happens in checking (rule 25,  $e \leftarrow A_1 \sqcap A_2$ ). 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. This estimation is possible because intersection and union types that trigger duplications must originate from the annotations in the expressions.

*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:  $|\Gamma|_{\Rightarrow}$  and  $|\Gamma|_{\Rightarrow, \omega}$  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,  $|\Gamma|_{\Rightarrow}$  counts the size of the type being type-applied, and  $|\Gamma|_{\Rightarrow, \omega}$  counts the number of the type-application judgment. So the termination of each type-application: for any type  $A, B, A \circ B \Rightarrow \omega$  will reduce to  $\omega \diamond C$  or not after a certain number of steps, 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  $|\Gamma|_{\Rightarrow}, |\Gamma|_{\Rightarrow, \omega}$  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. These new measures isolate the reasoning of matching and type application and simplify the overall logic.

## 6 Extensions

In this section, we present two extensions of  $F_{\sqcup\cap}^e$ , which are useful in practice. The first extension adds a form of labels to  $F_{\sqcup\cap}^e$ , and enables an encoding of records, discussed in Sec. 6.1. The second extension widens monotypes to include intersections and union types, discussed in Sec. 6.2. 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. Sec. 6.3 further discusses the possibility of adding more features to  $F_{\sqcup\cap}^e$ , for which we have no proofs or formalization.



## 6.1 Encoding Records as Intersection Types

The type system of  $F_{\sqcap}^e$  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  $F_{\sqcap}^e$ , 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 `Label  $l$`  syntax. Expressions are extended with three new forms: singleton record  $\langle l \mapsto e \rangle$ , 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. We further enforce that  $e_2$  must be another record in record extension expressions as an extra condition in well-formedness.

...    Label name ::=  $l$

Types	$A, B, C$	::=	$\dots \mid \text{Label } l$
Monotypes	$\tau, \sigma$	::=	$\dots \mid \text{Label } l$
Expressions	$e, t$	::=	$\dots \mid \langle l \mapsto e \rangle \mid \langle l_1 \mapsto e_1, e_2 \rangle \mid e.l$

Four rules are added to the type system: one in subtyping ( $\leq \text{Label}$ ); and three in typing ( $\Rightarrow \langle \rangle$ ,  $\Rightarrow \langle \rangle \text{Cons}$  and  $\Rightarrow \langle \rangle \text{Proj}$ ). The subtyping rule is simply a reflexive rule for the new type constructor. The typing rules for records are all inference rules. A singleton record infers a function type from the label type to the types of its carried expression. A record extension infers an intersection type of its head and tail types. For example,  $\langle l_1 \mapsto \langle \rangle, \langle l_2 \mapsto \langle \rangle \rangle$  infers  $(\text{Label } l_1 \rightarrow \mathbb{1}) \sqcap (\text{Label } l_2 \rightarrow \mathbb{1})$ . With such an encoding, the record projection can be completely dealt with by the existing matching relation, as shown in the rule  $\Rightarrow \langle \rangle \text{Proj}$ . Compared with rule  $\Rightarrow \text{App}$ , the only difference is that the last premise is changed from  $\Psi \vdash e_2 \Leftarrow B$  to  $\Psi \vdash \text{Label } l \leq B$ .

$$\frac{\Psi \vdash e \Rightarrow A}{\Psi \vdash \langle l \mapsto e \rangle \Rightarrow \text{Label } l \rightarrow A} \Rightarrow \langle \rangle \qquad \frac{\Psi \vdash e_1 \Rightarrow A_1 \quad \Psi \vdash e_2 \Rightarrow A_2}{\Psi \vdash \langle l_1 \mapsto e_1, e_2 \rangle \Rightarrow (\text{Label } l_1 \rightarrow A_1) \sqcap A_2} \Rightarrow \langle \rangle \text{Cons}$$

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

This extension still has the good properties of subtyping transitivity and checking subsumption as the base system does, and it is also equipped with a sound and complete algorithmic system. The algorithm is extended with three new works and seven 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. Since the record expressions always become structurally smaller for all the new rules, and the label type is nothing different from the unit type, it should be easy to extend the current measure definition to the new syntax and follow the same reasoning. 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  $F_{\sqcap}^e$  has good properties, completeness is achieved at the cost of 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 these, while still employing our algorithm. The extended monotype definition is  $\tau, \sigma ::= \mathbb{1} \mid a \mid \tau \rightarrow \sigma \mid \tau \sqcap \sigma \mid \tau \sqcup \sigma$  ( $a$  is still a type variable). With this new definition,  $\mathbb{1} \sqcap (\mathbb{1} \rightarrow \mathbb{1})$  and

(Label  $l_1 \rightarrow \mathbb{1}$ )  $\sqcap$  (Label  $l_2 \rightarrow \mathbb{1}$ ) 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 as the previous one. 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  $\Gamma \Vdash \mathbb{1} \sqcap (\mathbb{1} \rightarrow \mathbb{1}) \leq a$  and  $\widehat{a} \in \Gamma$ , our algorithm will first try to solve  $a$  to  $\mathbb{1} \sqcap (\mathbb{1} \rightarrow \mathbb{1})$ . If this solution fails, the algorithm will continue to try  $\mathbb{1}$  and  $\mathbb{1} \rightarrow \mathbb{1}$ . 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  $\lambda x.x \Leftarrow (\text{Int} \rightarrow \text{Int}) \sqcap (\text{Bool} \rightarrow \text{Bool})$  if  $a$  is solved to  $(\text{Int} \rightarrow \text{Int}) \sqcap (\text{Bool} \rightarrow \text{Bool})$ . 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 since occurs-check is still employed in this system. However, proving this, if possible, would require a new proof strategy.

### 6.3 Prospect for More Features

In this section, we will briefly discuss the possibility of adding two more features into  $F_{\sqcup \sqcap}^e$ : gradual typing and distributive subtyping.

*Gradual typing.* Since we use TypeScript as the major motivating programming language, it is natural to first consider another important feature of it, gradual typing. The combination between bidirectional HRP and gradual typing has been previously studied by Xie et al. [2019]. They argued that “gradual typing and polymorphism are orthogonal and can be combined in a principled way”. Their rule extension to the declarative system was surprisingly simple, with two new rules in the subtyping ( $\star$  means an unknown type):

$$\frac{}{\Psi \vdash \star \leq A} \qquad \frac{}{\Psi \vdash A \leq \star}$$

As this system was also based on DK’s implicit predicative HRP system, we think these ideas can be combined with our work. On the other hand, the combination of intersection and union types with gradual typing has also been studied by Castagna and Lanvin [2017] through the semantic subtyping approach. Semantic subtyping allows complex transformation and simplification of types, and usually comes with other set-theoretic operators like negation ( $\neg$ ), greatly different from our syntactic approach. Another challenge is that, after adding the gradual typing, we would possibly want a new set of desirable properties (e.g., gradual guarantee) for the bidirectional specification.

*Distributive subtyping.* Another feature closely related to intersection and union types is distributive subtyping rules.

$$\frac{}{(A_1 \sqcup A_2) \sqcap B \leq (A_1 \sqcap B) \sqcup (A_2 \sqcap B)} \leq \sqcup \text{Dist} \sqcap$$

$$\frac{}{(\forall a. A) \sqcap (\forall a. B) \leq \forall a. (A \sqcap B)} \leq \sqcap \text{Dist} \forall \qquad \frac{}{(A \rightarrow B_1) \sqcap (A \rightarrow B_2) \leq A \rightarrow (B_1 \sqcap B_2)} \leq \sqcap \text{Dist} \rightarrow$$

$$\frac{}{\forall a. (A \sqcup B) \leq (\forall a. A) \sqcup (\forall a. B)} \leq \sqcup \text{Dist} \forall \qquad \frac{}{(A_1 \rightarrow B) \sqcap (A_2 \rightarrow B) \leq (A_1 \sqcup A_2) \rightarrow B} \leq \sqcup \text{Dist} \rightarrow$$

Various forms of distributivity have been implemented by different languages, including Scala, TypeScript, Ceylon, CDuce [Castagna and Frisch 2005] and Julia [Nardelli et al. 2018]. Rioux et al. [2023] provides a systematic study of the distributivity between intersection and union types, and function types and universal types.

Distributivity has non-trivial interactions with existing features. For HRP, the distributivity with function types allows intersection and union types nested inside function types to be decomposed. In  $F_{\sqcup\cap}^e$ , the only instantiation to make  $\forall a. a \rightarrow \text{Int} \leq (\text{Int} \sqcup \text{Bool}) \rightarrow \text{Int}$  hold is  $\text{Int} \sqcup \text{Bool}$ . With distributivity, if the subtyping is still transitive,  $\forall a. a \rightarrow \text{Int} \leq (\text{Int} \rightarrow \text{Int}) \sqcap (\text{Bool} \rightarrow \text{Int}) \leq (\text{Int} \sqcup \text{Bool}) \rightarrow \text{Int}$  only requires  $a$  to be instantiated to  $\text{Int}$  and  $\text{Bool}$  separately. For explicit type applications, the distributivity with universal types needs the type application judgment to consider both branches for the intersection type case. Both seem to complicate reasoning, but we believe that an extension of  $F_{\sqcup\cap}^e$  with distributivity rules is possible.

## 7 Related Work

*Polymorphic type inference with intersections and union types.* Most works on type inference with parametric 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  $\mathbf{P}$ , which comes with a decidable type inference algorithm. Intersection types and parametric polymorphism are restricted. In  $\mathbf{P}$ , 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 an algebraic subtyping lattice. The types in the system are also polarized: intersection types can only appear in negative positions; while union types can only 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  $F_{\sqcup\cap}^e$ .

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 as MLSub on the allowed positions of intersection and union types. Dunfield [2009] presents a bidirectional type system with higher-rank polymorphism, as well as intersection and union types. Like our work, Dunfield employs a greedy-instantiation strategy. The corresponding type inference algorithm is sound. However, as observed by Dunfield and Krishnaswami [2013, 2021] later, 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 considers 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  $F_{\leq}^e$  to extend a predicative HRP type

system with *impredicative* explicit type applications. Cui et al. [2023] extends  $F_{\leq}^e$  with bounded quantification, resulting a variant of kernel  $F_{\leq}$  [Cardelli et al. 1994; Cardelli and Wegner 1985]. Our work incorporates the idea of  $F_{\leq}^e$  for explicit type applications. The main additions over  $F_{\leq}^e$  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 [Chrzęszcz 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  $r_1$  is a subtype of the type of  $r_2$ . With uncurried functions, both  $ex_1$  and  $ex_2$  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  $ex_3$  is accepted, and  $ex_4$  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  $F_{\sqcup\cap}^e$ . In particular,  $F_{\sqcup\cap}^e$  supports unrestricted forms of intersection and union types.  $F_{\sqcup\cap}^e$  is equipped with a sound, complete, and decidable algorithm that infers monotype instantiations and unannotated functions. We also discuss two variants of  $F_{\sqcup\cap}^e$ . One incorporates more practical features, such as handling records, by encoding them to existing features of  $F_{\sqcup\cap}^e$ . The other provides more inference at the cost of completeness. 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.

## Acknowledgments

We are grateful to the anonymous reviewers for their valuable comments. We also want to thank Xuejing Huang for sharing her experience on proofs related to intersection and union types, and Mingqi Xue for his suggestion of the toolchain and early experiments on the Coq formalization. The research is supported by the Practical Type Inference with Bounded Quantification and Union and Intersection Types collaboration project (TC20230508031) between Huawei and The University of Hong Kong and project number 17209821 of Hong Kong Research Grants Council.

## References

- Brian Aydemir and Stephanie Weirich. 2010. LNgén: Tool support for locally nameless representations. *Draft available at <http://www.cis.upenn.edu/~sweirich/papers/Lngen>* 61 (2010), 108.
- Jan Bessai, Boris Döder, Andrej Dudenhefner, Tzu-Chun Chen, and Ugo de'Liguoro. 2014. Typing Classes and Mixins with Intersection Types. In *Proceedings Seventh Workshop on Intersection Types and Related Systems, ITRS 2014, Vienna, Austria, 18 July 2014 (EPTCS, Vol. 177)*, Jakob Rehof (Ed.), 79–93. <https://doi.org/10.4204/EPTCS.177.7>
- 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>
- Giuseppe Castagna and Alain Frisch. 2005. A Gentle Introduction to Semantic Subtyping. In *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3580)*, Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung (Eds.), Springer, 30–34. [https://doi.org/10.1007/11523468\\_3](https://doi.org/10.1007/11523468_3)
- 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 and Victor Lanvin. 2017. Gradual typing with union and intersection types. *Proc. ACM Program. Lang.* 1, ICFP (2017), 41:1–41:28. <https://doi.org/10.1145/3110285>
- 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](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 Satellite 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](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](https://doi.org/10.1016/0890-5401(88)90009-0)
- Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. 2018. Julia subtyping: a rational reconstruction. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 113:1–113:27. <https://doi.org/10.1145/3276483>
- Martin Odersky and Konstantin Läuffer. 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>
- John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2* (Boston, Massachusetts, USA) (ACM '72). Association for Computing Machinery, New York, NY, USA, 717–740. <https://doi.org/10.1145/800194.805852>
- John C Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. *Information Processing* (1983), 513–523.
- Nick Rioux, Xuejing Huang, Bruno C. d. S. Oliveira, and Steve Zdancewic. 2023. A Bowtie for a Beast: Overloading, Eta Expansion, and Extensible Data Types in  $F_{\rightarrow\leftarrow}$ . *Proc. ACM Program. Lang.* 7, POPL, Article 18 (jan 2023), 29 pages. <https://doi.org/10.1145/3571211>

- 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>
- Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. 2007. Ott: effective tool support for the working semanticist. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (Freiburg, Germany) (ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 1–12. <https://doi.org/10.1145/1291151.1291155>
- 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*.
- Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of typed scheme. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 395–406. <https://doi.org/10.1145/1328438.1328486>
- 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>
- Ningning Xie, Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. Consistent Subtyping for All. *ACM Transactions on Programming Languages and Systems* 42, 1, Article 2 (2019). <https://doi.org/10.1145/3310339>
- Jinxu Zhao and Bruno C. d. S. Oliveira. 2022. Elementary Type Inference. In *36th European Conference on Object-Oriented Programming*. ECOOP, 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>

## A Examples in $F_{\sqcup}^e$

Table 1. Examples used in the comparison of TypeScript with  $F_{\sqcup}^e$ , where **V** represents the variant with intersection and union type inference. Record extension is enabled by default in both systems.

$F_{\sqcup}^e$ program	$F_{\sqcup}^e$	<b>V</b>
<b>let</b> f1 : (Label m → Int) → Int = $\lambda o. o.m$ <b>in</b> <b>let</b> o1 = ⟨m ↦ 1, ⟨n ↦ True, ⟨⟩⟩⟩ <b>in</b> f1 o1 -- ex1_1	✓	✓
<b>let</b> g1 : (Label n → Bool) → Bool = $\lambda o. o.n$ <b>in</b> <b>let</b> o1 = ⟨m ↦ 1, ⟨n ↦ True, ⟨⟩⟩⟩ <b>in</b> g1 o1 -- ex1_2	✓	✓
<b>let</b> h1 : ((Label m → Int) $\sqcap$ (Label n → Int)) $\sqcup$ ((Label k → Int → Int) $\sqcap$ (Label m → Int)) = $\lambda o. o.m$ <b>in</b> ...	✓	✓
<b>let</b> f2 : ((Int → Int) $\sqcap$ (Bool → Bool)) → Bool = $\lambda g. g$ True <b>in</b> ...	✓	✓
( $\lambda f. f$ 1 2) : ((Int → Int → Int) $\sqcap$ (Int → Bool → Int)) → Int -- f3_1	✓	✓
( $\lambda f. f$ 1 True) : ((Int → Int → Int) $\sqcap$ (Int → Bool → Int)) → Int -- f3_2	✓	✓
<b>let</b> f4 = $\Lambda a. (\lambda x. x) : a \rightarrow a$ <b>in</b> f4 1 -- ex4_1	✓	✓
<b>let</b> f4 = $\Lambda a. (\lambda x. x) : a \rightarrow a$ <b>in</b> f4 @Int 1 -- ex4_2	✓	✓
<b>let</b> f5 = $\Lambda a. (\lambda x. \lambda y. x) : a \rightarrow a \rightarrow a$ <b>in</b> f5 1 True -- ex5_1	✗	✗
<b>let</b> f5 = $\Lambda a. (\lambda x. \lambda y. x) : a \rightarrow a \rightarrow a$ <b>in</b> f5 @(Bool $\sqcup$ Int) 1 True -- ex5_2	✓	✓
<b>let</b> g5 = $\Lambda a. (\lambda x. \lambda y. \lambda z. 1) : (a \rightarrow \text{Int}) \rightarrow (a \rightarrow \text{Int}) \rightarrow a \rightarrow \text{Int}$ <b>in</b> g5 (( $\lambda x. 1$ ) : Int → Int) (( $\lambda y. 2$ ) : Bool → Int) -- ex5_3	✗	✗
g5 @(Int $\sqcap$ Bool) (( $\lambda x. 1$ ) : Int → Int) (( $\lambda y. 2$ ) : Bool → Int) -- ex5_4	✓	✓
<b>let</b> f6 = $\Lambda a. (\lambda x. 2) : (a \rightarrow \text{Int}) \rightarrow \text{Int}$ <b>in</b> <b>let</b> g6 = ( $\lambda f. 1$ ) : (Int → Int) → Int <b>in</b> f6 g6 -- ex6	✓	✓
<b>let</b> f7 = $\Lambda a. (\lambda x. x) : a \rightarrow a$ <b>in</b> f7 @(∀a. a → a) f7 -- ex7_1	✓	✓
<b>let</b> f7 = $\Lambda a. (\lambda x. x) : a \rightarrow a$ <b>in</b> f7 @(∀a. (a $\sqcap$ a) → (a $\sqcup$ a)) f7 -- ex7_2	✓	✓
<b>let</b> f8 : (∀a. a → a → Int) → Int) → Int = $\lambda x. 1$ <b>in</b> <b>let</b> g8_1 : (Int → (Int $\sqcap$ Bool) → Int) → Int = $\lambda f. 1$ <b>in</b> f8 g8_1 -- ex8_1	✓	✓
<b>let</b> g8_2 : ((Int $\sqcup$ Bool) → Int → Int) → Int = $\lambda f. 1$ <b>in</b> f8 g8_2 -- ex8_2	✗	✗
<b>let</b> g8_3 : ((Int $\sqcap$ Bool) → Int → Int) → Int = $\lambda f. 1$ <b>in</b> f8 g8_3 -- ex8_3	✓	✓
<b>let</b> g8_4 : (Int → (Int $\sqcup$ Bool) → Int) → Int = $\lambda f. 1$ <b>in</b> f8 g8_4 -- ex8_4	✗	✓
<b>let</b> f9 : (∀a. Int → a → Int) → Bool → Int = $\lambda k. k$ 3 <b>in</b> <b>let</b> h9 : (∀a. ∀b. b → a → b) → Bool → Int = $\lambda k. k$ f9 k <b>in</b> ...	✓	✓
<b>let</b> g9 : (∀a. Int → a → Int) → Bool → Int = $\lambda k. k$ @Bool 3 <b>in</b> ( $\lambda k. k$ @Bool 3) : (∀a. ∀b. b → a → b) → Bool → Int -- ex9_1	✓	✓
( $\lambda k. k$ @Bool 3) : (∀a. ∀b. b → a → b) → Bool → Int -- ex9_2	✓	✓
<b>let</b> f10 : (Int → Int) → Int = $\lambda x. 1$ <b>in</b> <b>let</b> h10 = ( $\lambda x. \lambda y. y$ ) 1 <b>in</b> f10 h10 -- ex10	✓	✓
<b>let</b> f11 : ((Int → Int) $\sqcap$ (Bool → Bool)) → Int = $\lambda g. g$ 1 <b>in</b> f11 ( $\lambda x. x$ ) -- ex11	✓	✓
( $\lambda x. 1$ ) : (∀a. a) → Int -- ex12_1	✓	✓
( $\lambda x. 1$ ) : (∀a. (a → Bool) $\sqcap$ (a → Int)) → Int -- ex12_2	✓	✓
( $\lambda x. x.m$ ) ⟨m ↦ 1, ⟨⟩⟩ -- ex13	✓	✓
<b>let</b> g14 : (Int $\sqcup$ Bool) → (Int $\sqcup$ Bool) → Int = $\Lambda a. \lambda x. \lambda y. 1$ <b>in</b> <b>let</b> h14_1 : Int → Bool → Int = g14 <b>in</b> ...	✗	✓
<b>let</b> h14_2 : Int → Bool → Int = $\Lambda a. (\lambda x. \lambda y. 1) : a \rightarrow a \rightarrow \text{Int}$ <b>in</b> ...	✗	✗
<b>let</b> f15 : ((∀a. a → a) → Int) → Int = $\lambda x. 1$ <b>in</b> <b>let</b> h15 : (∀a. (a → a) → (a → a)) → Int = $\lambda x. 1$ <b>in</b> f15 h15 -- ex15	✗	✗



We include a Haskell prototype implementation in the supplementary materials, which implements all the algorithmic rules in the paper. Additionally, to make the examples more interesting, we have implemented a few extensions. These include polymorphic lists ( $[a]$ ), a case analysis expression for pattern matching on lists, recursion via a fixpoint operator, and recursive let expressions.

## B Full Set of Rules

### B.1 Well-formedness Definitions

$\Psi \vdash A$	$A$ is well-formed in $\Psi$
$\frac{}{\Psi \vdash \perp} \text{wf}_{\text{dunit}}$	$\frac{}{\Psi \vdash \top} \text{wf}_{\text{d}\top}$
$\frac{}{\Psi \vdash \perp} \text{wf}_{\text{d}\perp}$	$\frac{a \in \Psi \vee \tilde{a} \in \Psi}{\Psi \vdash a} \text{wf}_{\text{d}\text{var}}$
$\frac{\Psi \vdash A \quad \Psi \vdash B}{\Psi \vdash A \rightarrow B} \text{wf}_{\text{d}\rightarrow}$	$\frac{\Psi, a \vdash A \quad a \in^s A}{\Psi \vdash \forall a. A} \text{wf}_{\text{d}\forall}$
$\frac{\Psi \vdash A \quad \Psi \vdash B}{\Psi \vdash A \sqcap B} \text{wf}_{\text{d}\sqcap}$	$\frac{\Psi \vdash A \quad \Psi \vdash B}{\Psi \vdash A \sqcup B} \text{wf}_{\text{d}\sqcup}$
$\Psi \vdash e$	$e$ is well-formed in $\Psi$
$\frac{x : A \in \Psi}{\Psi \vdash x} \text{wf}_{\text{d}\text{tmvar}}$	$\frac{}{\Psi \vdash ()} \text{wf}_{\text{d}\text{tmunit}}$
$\frac{\Psi \vdash A \quad \Psi \vdash e}{\Psi \vdash (e : A)} \text{wf}_{\text{d}\text{anno}}$	$\frac{\Psi, x : A \vdash e}{\Psi \vdash \lambda x. e} \text{wf}_{\text{d}\text{abs}}$
$\frac{\Psi \vdash A \quad \Psi \vdash e}{\Psi \vdash e @ A} \text{wf}_{\text{d}\text{tApp}}$	$\frac{\Psi \vdash e_1 \quad \Psi \vdash e_2}{\Psi \vdash e_1 e_2} \text{wf}_{\text{d}\text{app}}$
$\frac{\Psi, a \vdash A \quad \Psi, a \vdash e \quad a \in^s A}{\Psi \vdash \Lambda a. e : A} \text{wf}_{\text{d}\text{tLam}}$	
$\vdash_{\text{t}} \Psi$	$\Psi$ is a well-formed typing context
$\frac{}{\vdash_{\text{t}} \cdot} \text{wfe}_{\cdot}$	$\frac{\vdash_{\text{t}} \Psi}{\vdash_{\text{t}} \Psi, a} \text{wfe}_a$
	$\frac{\vdash_{\text{t}} \Psi \quad \Psi \vdash A}{\vdash_{\text{t}} \Psi, x : A} \text{wfe}_{\text{of}}$
$\vdash \Psi$	$\Psi$ is a well-formed subtyping context
$\frac{\vdash_{\text{t}} \Psi}{\vdash \Psi} \text{wfe}_{\text{t}}$	$\frac{\vdash \Psi}{\vdash \Psi, \tilde{a}} \text{wfe}_{\tilde{a}}$

Fig. 9. Well-Formedness for the Bidirectional Type System

The well-formedness of the bidirectional type system is presented in Figure 9. The rules of type and expression well-formedness are mostly standard, except the free variable restriction in  $\text{wf}_{\text{d}\forall}$  and  $\text{wf}_{\text{d}\text{tLam}}$ . For context well-formedness, we adopt the stronger invariants proposed by Cui et al. [2023] to distinguish the subtyping contexts from the typing context: no subtype variables should appear in the typing context, and no variable and type variable should appear after any subtype variable. However, we do not build a new syntax category for the typing context. We simply choose to employ the set of well-formedness rules:  $\vdash_{\text{t}} \Psi$  and  $\vdash \Psi$  for the bidirectional context. However, we find this distinction not very helpful and it even complicates the reasoning and weakens the conclusion for certain properties (as the properties are now proved for a more restricted context).

The well-formedness of the algorithmic systems is presented in Figure 11. We also have a syntactic category for algorithmic context. It can be obtained from the work erasure function  $[\Gamma]$  in Figure 10 removes all works in the worklist. The rules for type and expression well-formedness are defined on the algorithmic context. They are similar to the bidirectional counterparts. The algorithmic system has additional rules for the well-formedness of continuations, works, and worklists.

$$\boxed{[\Gamma] = \Sigma}$$

erase all works in  $\Gamma$  to get  $\Sigma$

$$\begin{aligned} [\cdot] &= \cdot \\ [\Gamma, a] &= [\Gamma] \\ [\Gamma, \tilde{a}] &= [\Gamma] \\ [\Gamma, \hat{a}] &= [\Gamma] \\ [\Gamma, x : A] &= [\Gamma], x : A \\ [\Gamma, w] &= [\Gamma] \end{aligned}$$

Fig. 10. Work erasement Function

$$\boxed{\vdash \Sigma}$$

$\Sigma$  is a well-formed algorithmic context

$$\frac{}{\vdash \cdot} \text{wfe}_\cdot \quad \frac{\vdash \Sigma}{\vdash \Sigma, a} \text{wfe}_a \quad \frac{\vdash \Sigma}{\vdash \Sigma, \tilde{a}} \text{wfe}_{\tilde{a}} \quad \frac{\vdash \Sigma}{\vdash \Sigma, \hat{a}} \text{wfe}_{\hat{a}} \quad \frac{\vdash \Sigma \quad \Sigma \vdash A}{\vdash \Sigma, x : A} \text{wfe}_{\text{of}}$$

Fig. 11. Well-Formedness for the Algorithmic Context

$$\boxed{\vdash_t \Gamma}$$

$\Gamma$  is a well-formed typing worklist

$$\frac{}{\vdash_t \cdot} \text{wft}_\cdot \quad \frac{\vdash_t \Gamma}{\vdash_t \Gamma, a} \text{wft}_a \quad \frac{\vdash_t \Gamma}{\vdash_t \Gamma, \hat{a}} \text{wft}_{\hat{a}} \quad \frac{\vdash_t \Gamma \quad [\Gamma] \vdash_t A}{\vdash_t \Gamma, x : A} \text{wft}_{\text{of}}$$

$$\frac{\vdash_t \Gamma \quad [\Gamma] \vdash_t w \quad w \neq * \leq *}{\vdash_t \Gamma \Vdash w} \text{wft}_w$$

$$\boxed{\vdash \Gamma}$$

$\Gamma$  is a well-formed subtyping worklist

$$\frac{\vdash_t \Gamma}{\vdash \Gamma} \text{wft}_t \quad \frac{\vdash \Gamma}{\vdash \Gamma, \tilde{a}} \text{wf}_{\tilde{a}} \quad \frac{\vdash \Gamma}{\vdash \Gamma, \hat{a}} \text{wf}_{\hat{a}} \quad \frac{\vdash \Gamma \quad [\Gamma] \vdash A \quad [\Gamma] \vdash B}{\vdash \Gamma \Vdash A \leq B} \text{wf}_{\leq}$$

Fig. 12. Well-Formedness of Algorithmic Worklist

Due to the invariance imposed for the bidirectional type system to distinguish the typing and subtyping context, the algorithmic system must have that as well. This restriction is defined on algorithmic worklists directly, as shown in Figure 12: a typing worklist ( $\vdash_t \Gamma$ ) is dedicated to typing judgments, while the latter part ( $\vdash \Gamma$ ) is for subtyping judgments. Variables and type variables are restricted to appearing only in the typing part, whereas subtyping judgments and subtype variables are confined to the subtyping part.

$\Sigma \vdash A$	A is well-formed in $\Sigma$
$\frac{}{\Sigma \vdash \mathbb{1}} \text{wf\_unit}$	$\frac{}{\Sigma \vdash \top} \text{wf\_}\top$
$\frac{}{\Sigma \vdash \perp} \text{wf\_}\perp$	$\frac{a \in \Sigma \vee \tilde{a} \in \Sigma \vee \hat{a} \in \Sigma}{\Sigma \vdash a} \text{wf\_var}$
$\frac{\Sigma \vdash A \quad \Sigma \vdash B}{\Sigma \vdash A \rightarrow B} \text{wf\_}\rightarrow$	$\frac{\Sigma, a \vdash A \quad a \in {}^s A}{\Sigma \vdash \forall a. A} \text{wf\_}\forall$
$\frac{\Sigma \vdash A \quad \Sigma \vdash B}{\Sigma \vdash A \sqcap B} \text{wf\_}\sqcap$	$\frac{\Sigma \vdash A \quad \Sigma \vdash B}{\Sigma \vdash A \sqcup B} \text{wf\_}\sqcup$
$\Sigma \vdash e$	e is well-formed in $\Sigma$
$\frac{x : A \in \Sigma}{\Sigma \vdash x} \text{wf\_tmvar}$	$\frac{}{\Sigma \vdash ()} \text{wf\_tmunit}$
$\frac{\Sigma, x : A \vdash e}{\Sigma \vdash \lambda x. e} \text{wf\_abs}$	$\frac{\Sigma \vdash e_1 \quad \Sigma \vdash e_2}{\Sigma \vdash e_1 e_2} \text{wf\_app}$
$\frac{\Sigma \vdash A \quad \Sigma \vdash e}{\Sigma \vdash (e : A)} \text{wf\_anno}$	$\frac{\Sigma \vdash A \quad \Sigma \vdash e}{\Sigma \vdash e @A} \text{wf\_tApp}$
$\frac{\Sigma, a \vdash A \quad \Sigma, a \vdash e \quad a \in {}^s A}{\Sigma \vdash \Lambda a. e : A} \text{wf\_tLam}$	
$\Sigma \vdash \omega^s, \Sigma \vdash \omega^d$	$\omega^s, \omega^d$ is well-formed in $\Sigma$
$\frac{\Sigma \vdash A}{\Sigma \vdash \_ \leq A} \text{wf}^s \leq$	$\frac{\Sigma \vdash A \quad \Sigma \vdash \omega^s}{\Sigma \vdash \_ \circ A \Rightarrow \omega^s} \text{wf}^s \circ$
$\frac{\Sigma \vdash \omega^d}{\Sigma \vdash \_ \triangleright \omega^d} \text{wf}^s \triangleright$	$\frac{\Sigma \vdash A \quad \Sigma \vdash \omega^s}{\Sigma \vdash \_ \cup A \triangleright \omega^s} \text{wf}^s \cup \circ$
$\frac{\Sigma \vdash A \quad \Sigma \vdash B \quad \Sigma \vdash \omega^s}{\Sigma \vdash \_ \cup A \circ B \Rightarrow \omega^s} \text{wf}^s \cup \circ$	$\frac{\Sigma \vdash A \quad \Sigma \vdash \omega^s}{\Sigma \vdash \_ \cup \circ A \triangleright \omega^s} \text{wf}^s \cup \circ$
$\frac{\Sigma \vdash e \quad \Sigma \vdash \omega^s}{\Sigma \vdash \_ \rightarrow \_ \bullet e \Rightarrow_{\alpha} \omega^s} \text{wf}^d \bullet$	$\frac{\Sigma \vdash A \quad \Sigma \vdash \omega^d}{\Sigma \vdash \_ \rightarrow \_ \cup A \triangleright \omega^d} \text{wf}^d \cup \triangleright$
$\frac{\Sigma \vdash A \quad \Sigma \vdash B \quad \Sigma \vdash \omega^d}{\Sigma \vdash \_ \rightarrow \_ \cup_{\triangleright} (A \rightarrow B) \triangleright \omega^d} \text{wf}^d \cup_{\triangleright}$	
$\Sigma \vdash w$	w is well-formed in $\Sigma$
$\frac{\Sigma \vdash A \quad \Sigma \vdash B}{\Sigma \vdash A \leq B} \text{wf} \leq$	$\frac{\Sigma \vdash e \quad \Sigma \vdash A}{\Sigma \vdash e \leftarrow A} \text{wf} \leftarrow$
$\frac{\Sigma \vdash e \quad \Sigma \vdash \omega^s}{\Sigma \vdash e \Rightarrow \omega^s} \text{wf} \Rightarrow$	$\frac{\Sigma \vdash A \quad \Sigma \vdash \omega^d}{\Sigma \vdash A \triangleright \omega^d} \text{wf} \triangleright$
$\frac{\Sigma \vdash A \quad \Sigma \vdash B \quad \Sigma \vdash \omega^s}{\Sigma \vdash A \circ B \Rightarrow \omega^s} \text{wf} \circ$	$\frac{\Sigma \vdash C \quad \Sigma \vdash A \quad \Sigma \vdash B \quad \Sigma \vdash \omega^s}{\Sigma \vdash C \cup A \circ B \Rightarrow \omega^s} \text{wf} \cup \circ$
$\frac{\Sigma \vdash B \quad \Sigma \vdash C \quad \Sigma \vdash A \quad \Sigma \vdash \omega^d}{\Sigma \vdash (B \rightarrow C) \cup A \triangleright \omega^d} \text{wf} \cup \triangleright$	$\frac{\Sigma \vdash A_1 \rightarrow B_1 \quad \Sigma \vdash A_2 \rightarrow B_2 \quad \Sigma \vdash \omega^d}{\Sigma \vdash (A_1 \rightarrow B_1) \cup_{\triangleright} (A_2 \rightarrow B_2) \triangleright \omega^d} \text{wf} \cup_{\triangleright}$
$\frac{\Sigma \vdash A_1 \quad \Sigma \vdash A_2 \quad \Sigma \vdash \omega^s}{\Sigma \vdash A_1 \cup_{\circ} A_2 \triangleright \omega^s} \text{wf} \cup_{\circ}$	

## B.2 Reduction Rules of Bidirectional Worklist

For soundness and completeness proofs, we employ two sets of reduction rules for the bidirectional worklist system: worklist reduction by delegation  $\Omega \longrightarrow \Omega'$  and worklist reduction by continuation-passing  $\Omega \longrightarrow_{\omega} \Omega'$ . The reduction by delegation delegates the current work to the bidirectional system directly, and passes the result (if any) to the following continuation. Their rules are presented in Figure 13 and Figure 14 (we omit the  $\omega$  in  $\longrightarrow_{\omega}$  in Figure 14). The continuation is represented by the defunctionalized style in Figure 13, and HOAS in Figure 14. It should be easy to recover the defunctionalized version of the worklist-reduction by continuation-passing (which is aligned with the actual formalization) by referencing the algorithmic ones shown in Figure 16 and Figure 17.

$[\Omega] = \Psi$	erase all works in $\Omega$ to get $\Psi$
$[\cdot] = \cdot$	
$[\Omega, a] = [\Omega], a$	
$[\Omega, \tilde{a}] = [\Omega], \tilde{a}$	
$[\Omega, x : A] = [\Omega], x : A$	
$[\Omega \Vdash w] = [\Omega]$	
$\Omega \longrightarrow \Omega'$	$\Omega$ reduces to $\Omega'$
$\Omega, a \longrightarrow \Omega$	
$\Omega, \tilde{a} \longrightarrow \Omega$	
$\Omega, x : A \longrightarrow \Omega$	
$\Omega \Vdash A \leq B \longrightarrow \Omega$	when $[\Omega] \vdash A \leq B$
$\Omega \Vdash e \leftarrow A \longrightarrow \Omega$	when $[\Omega] \vdash e \leftarrow A$
$\Omega \Vdash e \Rightarrow_{\alpha} \omega^s \longrightarrow \Omega \Vdash \omega^s \diamond A$	when $[\Omega] \vdash e \Rightarrow A$
$\Omega \Vdash A \triangleright_{\alpha, \beta} \omega^d \longrightarrow \Omega \Vdash \omega^d \diamond B \diamond C$	when $[\Omega] \vdash A \triangleright B \rightarrow C$
$\Omega \Vdash A \circ B \Rightarrow_{\alpha} \omega^s \longrightarrow \Omega \Vdash \omega^s \diamond C$	when $[\Omega] \vdash A \circ B \Rightarrow C$
$\Omega \Vdash A \rightarrow B \bullet e \Rightarrow_{\alpha} \omega^s \longrightarrow \Omega \Vdash \omega^s \diamond B$	when $[\Omega] \vdash e \leftarrow A$
$\Omega \Vdash C_1 \uplus A_2 \circ B_2 \Rightarrow \omega^s \longrightarrow \Omega \Vdash \omega^s \diamond (C_1 \sqcup C_2)$	when $[\Omega] \vdash A_2 \circ B_2 \Rightarrow C_2$
$\Omega \Vdash (B_1 \rightarrow C_1) \uplus A_2 \triangleright \omega^d \longrightarrow \Omega \Vdash \omega^d \diamond (B_1 \sqcap B_2) \diamond (C_1 \sqcup C_2)$	when $[\Omega] \vdash A_2 \triangleright B_2 \rightarrow C_2$
$\Omega \Vdash A_1 \uplus_{\circ} A_2 \blacktriangleright \omega^s \longrightarrow \Omega \Vdash \omega^s \diamond (A_1 \sqcup A_2)$	
$\Omega \Vdash (A_1 \rightarrow B_1) \uplus_{\bullet} (A_2 \rightarrow B_2) \longrightarrow \Omega \Vdash \omega^d \diamond (A_1 \sqcap A_2) \diamond (B_1 \sqcup C_2)$	

Fig. 13. Bidirectional Worklist Reduction by Delegation

The connection between the worklist reduction by delegation and the bidirectional system under empty context is straightforward by its definition (stronger equivalence can be established as well):

**THEOREM B.1 (SOUNDNESS AND COMPLETENESS OF BIDIRECTIONAL WORKLIST REDUCTION BY DELEGATION).** *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  $\cdot \Vdash e \Rightarrow \_ \leq \top \longrightarrow^* \cdot$ .

$\Omega \longrightarrow \Omega'$

 $\Omega$  reduces to  $\Omega'$ 

$$\begin{array}{l}
\Omega, a : \square \longrightarrow_1 \Omega \quad \Omega, a : \tilde{\square} \longrightarrow_2 \Omega \quad \Omega, x : A \longrightarrow_3 \Omega \\
\Omega \Vdash \mathbb{1} \leq \mathbb{1} \longrightarrow_4 \Omega \quad \Omega \Vdash a \leq a \longrightarrow_5 \Omega \quad \Omega \Vdash A \leq \top \longrightarrow_6 \Omega \quad \Omega \Vdash \perp \leq A \longrightarrow_7 \Omega \\
\Omega \Vdash A_1 \rightarrow A_2 \leq B_1 \rightarrow B_2 \longrightarrow_8 \Omega B_1 \leq A_1 \Vdash A_2 \leq B_2 \Vdash \\
\quad \Omega \Vdash \forall a. A \leq B \longrightarrow_9 \Omega \Vdash [\tau/a]A \leq B \quad \text{when } B^{\neq \forall} \\
\Omega \Vdash \forall a. A_1 \leq \forall a. A_2 \longrightarrow_{10} \Omega, \tilde{a} \Vdash A_1 \leq A_2 \\
\Omega \Vdash A \leq B_2 \sqcap B_2 \longrightarrow_{11} \Omega \Vdash A \leq B_1 \Vdash A \leq B_2 \\
\Omega \Vdash A_1 \sqcap A_2 \leq B \longrightarrow_{12} \Omega \Vdash A_1 \leq B \\
\Omega \Vdash A_1 \sqcap A_2 \leq B \longrightarrow_{13} \Omega \Vdash A_2 \leq B \\
\Omega \Vdash A_1 \sqcup A_2 \leq B \longrightarrow_{14} \Omega \Vdash A_1 \leq B \Vdash A_2 \leq B \\
\Omega \Vdash A \leq B_1 \sqcup B_2 \longrightarrow_{15} \Omega \Vdash A \leq B_1 \\
\Omega \Vdash A \leq B_1 \sqcup B_2 \longrightarrow_{16} \Omega \Vdash A \leq B_2 \\
\Omega \Vdash e \Leftarrow B \longrightarrow_{17} \Omega \Vdash e \Rightarrow_{\alpha} \alpha \leq B \\
\Omega \Vdash \lambda x. e \Leftarrow A \rightarrow B \longrightarrow_{18} \Omega, x : A \Vdash e \Leftarrow B \\
\Omega \Vdash \lambda x. e \Leftarrow \top \longrightarrow_{19} \Omega, x : \perp \Vdash e \Leftarrow \top \\
\Omega \Vdash e \Leftarrow A_1 \sqcap A_2 \longrightarrow_{20} \Omega \Vdash e \Leftarrow A_1 \Vdash e \Leftarrow A_2 \\
\Omega \Vdash e \Leftarrow A_1 \sqcup A_2 \longrightarrow_{21} \Omega \Vdash e \Leftarrow A_1 \\
\Omega \Vdash e \Leftarrow A_1 \sqcup A_2 \longrightarrow_{22} \Omega \Vdash e \Leftarrow A_2 \\
\Omega \Vdash x \Rightarrow_{\alpha} \omega \longrightarrow_{23} \Omega \Vdash [A/a]\omega \quad \text{when } x : A \in \Omega \\
\Omega \Vdash e : A \Rightarrow_{\alpha} \omega \longrightarrow_{24} \Omega \Vdash \omega \diamond A \Vdash e \Leftarrow A \\
\Omega \Vdash (\Lambda a. e : A) \Rightarrow_{\alpha} \omega \longrightarrow_{25} \Omega \Vdash \omega \diamond (\forall a. A), a : \square \Vdash e \Leftarrow A \\
\Omega \Vdash () \Rightarrow_{\alpha} \omega \longrightarrow_{26} \Omega \Vdash \omega \diamond \mathbb{1} \\
\Omega \Vdash e_1 e_2 \Rightarrow_{\alpha} \omega \longrightarrow_{27} \Omega \Vdash e_1 \Rightarrow_{\beta} (\beta \triangleright_{\gamma_1, \gamma_2} (\gamma_1 \rightarrow \gamma_2 \bullet e_2 \Rightarrow_{\alpha} \omega)) \\
\Omega \Vdash e @A \Rightarrow_{\alpha} \omega \longrightarrow_{28} \Omega \Vdash e \Rightarrow_{\beta} (\beta \circ A \Rightarrow_{\alpha} \omega) \\
\Omega \Vdash \lambda x. e \Rightarrow_{\alpha} \omega \longrightarrow_{29} \Omega \Vdash \omega \diamond (\tau_1 \rightarrow \tau_2), x : \tau_1 \Vdash e \Leftarrow \tau_2 \\
\Omega \Vdash \perp \triangleright_{\alpha, \beta} \omega \longrightarrow_{30} \Omega \Vdash \top \rightarrow \perp \triangleright_{\alpha, \beta} \omega \\
\Omega \Vdash A \rightarrow B \triangleright_{\alpha, \beta} \omega \longrightarrow_{31} \Omega \Vdash \omega \diamond A \diamond B \\
\Omega \Vdash \forall a. A \triangleright_{\alpha, \beta} \omega \longrightarrow_{32} \Omega \Vdash [\tau/a]A \triangleright_{\alpha, \beta} \omega \\
\Omega \Vdash A_1 \sqcap A_2 \triangleright_{\alpha, \beta} \omega \longrightarrow_{33} \Omega \Vdash A_1 \triangleright_{\alpha, \beta} \omega \\
\Omega \Vdash A_1 \sqcap A_2 \triangleright_{\alpha, \beta} \omega \longrightarrow_{34} \Omega \Vdash A_2 \triangleright_{\alpha, \beta} \omega \\
\Omega \Vdash A_1 \sqcup A_2 \triangleright_{\alpha_1, \alpha_2} \omega \longrightarrow_{35} \Omega \Vdash A_1 \triangleright_{\beta_1, \beta_2} (A_2 \triangleright_{\gamma_1, \gamma_2} (\beta_1 \rightarrow \beta_2 \cup_{\triangleright} \gamma_1 \rightarrow \gamma_2 \triangleright_{\alpha_1, \alpha_2} \omega)) \\
\Omega \Vdash (A_1 \rightarrow B_1) \cup_{\triangleright} (A_2 \rightarrow B_2) \triangleright \omega \longrightarrow_{36} \Omega \Vdash \omega \diamond (A_1 \sqcap A_2) \diamond (B_1 \sqcup B_2) \\
\Omega \Vdash A_1 \rightarrow A_2 \bullet e \Rightarrow_{\alpha} \omega \longrightarrow_{37} \Omega \Vdash \omega \diamond A_2 \Vdash e \Leftarrow A_1 \\
\Omega \Vdash \forall a. A \circ B \Rightarrow_{\alpha} \omega \longrightarrow_{38} \Omega \Vdash \omega \diamond ([B/a]A) \\
\Omega \Vdash \perp \circ A \Rightarrow_{\alpha} \omega \longrightarrow_{39} \Omega \Vdash \omega \diamond \perp \\
\Omega \Vdash A_1 \sqcap A_2 \circ B \Rightarrow_{\alpha} \omega \longrightarrow_{40} \Omega \Vdash A_1 \circ B \Rightarrow_{\alpha} \omega \\
\Omega \Vdash A_1 \sqcap A_2 \circ B \Rightarrow_{\alpha} \omega \longrightarrow_{41} \Omega \Vdash A_2 \circ B \Rightarrow_{\alpha} \omega \\
\Omega \Vdash A_1 \sqcup A_2 \circ B \Rightarrow_{\alpha} \omega \longrightarrow_{42} \Omega \Vdash A_1 \circ B \Rightarrow_{\beta} (A_2 \circ B \Rightarrow_{\gamma} \beta \cup_{\circ} \gamma \triangleright_{\alpha} \omega) \\
\Omega \Vdash A_1 \cup_{\circ} A_2 \triangleright_{\alpha} \omega \longrightarrow_{43} \Omega \Vdash \omega \diamond (A_1 \sqcup A_2)
\end{array}$$

Fig. 14. Bidirectional Worklist Reduction by Continuation-passing

### B.3 Syntax-directed Transfer of Continuation and Work

$\theta \vDash \omega^s \rightsquigarrow \omega^{s'}$

$\omega^s$  is transferred to  $\omega^{s'}$  under  $\theta$

$$\frac{\theta \vDash A \rightsquigarrow A'}{\theta \vDash \_ \leq A \rightsquigarrow \_ \leq A'} \quad \frac{\theta \vDash A \rightsquigarrow A' \quad \theta \vDash \omega^s \rightsquigarrow \omega^{s'}}{\theta \vDash \_ \circ A \Rightarrow_{\alpha} \omega^s \rightsquigarrow \_ \circ A' \Rightarrow_{\alpha} \omega^{s'}} \quad \frac{\theta \vDash \omega^s \rightsquigarrow \omega^{s'}}{\theta \vDash \_ \triangleright_{\alpha, \beta} \omega^d \rightsquigarrow \_ \triangleright_{\alpha, \beta} \omega^{d'}}$$

$$\frac{\theta \vDash A \rightsquigarrow A' \quad \theta \vDash B \rightsquigarrow B' \quad \theta \vDash \omega^s \rightsquigarrow \omega^{s'}}{\theta \vDash \_ \cup A \circ B \Rightarrow \omega^s \rightsquigarrow \_ \cup A' \circ B' \Rightarrow \omega^{s'}} \quad \frac{\theta \vDash A \rightsquigarrow A' \quad \theta \vDash \omega^s \rightsquigarrow \omega^{s'}}{\theta \vDash \_ \cup_{\circ} A \triangleright \omega^s \rightsquigarrow \_ \cup_{\circ} A' \triangleright \omega^{s'}}$$

$\theta \vDash \omega^d \rightsquigarrow \omega^{d'}$

$\omega^d$  is transferred to  $\omega^{d'}$  under  $\theta$

$$\frac{\theta \vDash e \rightsquigarrow e' \quad \theta \vDash \omega^s \rightsquigarrow \omega^{s'}}{\theta \vDash \_ \rightarrow \_ \bullet e \Rightarrow \omega^s} \quad \frac{\theta \vDash A \rightsquigarrow A' \quad \theta \vDash \omega^d \rightsquigarrow \omega^{d'}}{\theta \vDash \_ \rightarrow \_ \cup A \triangleright \omega^d \rightsquigarrow \_ \rightarrow \_ \cup A' \triangleright \omega^{d'}}$$

$$\frac{\theta \vDash A \rightsquigarrow A' \quad \theta \vDash B \rightsquigarrow B' \quad \theta \vDash \omega^d \rightsquigarrow \omega^{d'}}{\theta \vDash \_ \rightarrow \_ \cup_{\triangleright} (A \rightarrow B) \triangleright \omega^d \rightsquigarrow \_ \rightarrow \_ \cup_{\triangleright} (A' \rightarrow B') \triangleright \omega^{d'}}$$

$\theta \vDash w \rightsquigarrow w'$

$w^d$  is transferred to  $w'$  under  $\theta$

$$\frac{\theta \vDash A \rightsquigarrow A' \quad \theta \vDash B \rightsquigarrow B'}{\theta \vDash A \leq B \rightsquigarrow A' \leq B'} \quad \frac{\theta \vDash e \rightsquigarrow e' \quad \theta \vDash A \rightsquigarrow A'}{\theta \vDash e \leftarrow A \rightsquigarrow e \leftarrow A'} \quad \frac{\theta \vDash e \rightsquigarrow e' \quad \theta \vDash \omega^s \rightsquigarrow \omega^{s'}}{\theta \vDash e \Rightarrow \omega^s \rightsquigarrow e' \Rightarrow \omega^{s'}}$$

$$\frac{\theta \vDash A \rightsquigarrow A' \quad \theta \vDash B \rightsquigarrow B' \quad \theta \vDash \omega^s \rightsquigarrow \omega^{s'}}{\theta \vDash A \circ B \Rightarrow \omega^s \rightsquigarrow A \circ B \Rightarrow \omega^{s'}} \quad \frac{\theta \vDash A \rightsquigarrow A' \quad \theta \vDash \omega^d \rightsquigarrow \omega^{d'}}{\theta \vDash A \triangleright \omega^d \rightsquigarrow A' \triangleright \omega^{d'}}$$

$$\frac{\theta \vDash A \rightsquigarrow A' \quad \theta \vDash B \rightsquigarrow B' \quad \theta \vDash e \rightsquigarrow e' \quad \theta \vDash \omega^s \rightsquigarrow \omega^{s'}}{\theta \vDash A \rightarrow B \bullet e \Rightarrow \omega^s \rightsquigarrow A' \rightarrow B' \bullet e' \Rightarrow \omega^{s'}}$$

$$\frac{\theta \vDash A \rightsquigarrow A' \quad \theta \vDash B \rightsquigarrow B' \quad \theta \vDash C \rightsquigarrow C' \quad \theta \vDash \omega^s \rightsquigarrow \omega^{s'}}{\theta \vDash C \cup A \circ B \Rightarrow \omega^s \rightsquigarrow C' \cup A' \circ B' \Rightarrow \omega^{s'}}$$

$$\frac{\theta \vDash A \rightsquigarrow A' \quad \theta \vDash B \rightsquigarrow B' \quad \theta \vDash C \rightsquigarrow C' \quad \theta \vDash \omega^d \rightsquigarrow \omega^{d'}}{\theta \vDash (B \rightarrow C) \cup A \triangleright \omega^d \rightsquigarrow (B' \rightarrow C') \cup A' \triangleright \omega^{d'}}$$

$$\frac{\theta \vDash A_1 \rightsquigarrow A'_1 \quad \theta \vDash A_2 \rightsquigarrow A'_2 \quad \theta \vDash \omega^s \rightsquigarrow \omega^{s'}}{\theta \vDash A_1 \cup_{\circ} A_2 \triangleright \omega^s \rightsquigarrow A'_1 \cup_{\circ} A'_2 \triangleright \omega^{s'}}$$

$$\frac{\theta \vDash e \rightsquigarrow e' \quad \theta \vDash \omega^s \rightsquigarrow \omega^{s'}}{\theta \vDash (A_1 \rightarrow B_1) \cup_{\triangleright} (A_2 \rightarrow B_2) \triangleright \omega^d \rightsquigarrow (A'_1 \rightarrow B'_1) \cup_{\triangleright} (A'_2 \rightarrow B'_2) \triangleright \omega^{d'}}$$

Fig. 15. Syntax-directed Transfer of Continuation and Work

The syntax-directed transfer rules of continuation and work are shown in Figure 15. There is nothing special about these rules, as every rule is just inductively defined and based on the transfer of expressions and types. The transfer of two types of continuation ( $\omega^s$  and  $\omega^d$ ) needs a mutually inductive definition.

## B.4 Defunctionalizing the Continuation

In Section 4, the continuation is simply described as a function (in the meta-language, which is Coq in our setting) that takes types as arguments and returns a work. However, it is usually unpleasant to mix HOAS and locally nameless in the binding structure. In addition, reasoning about functions in Coq would probably require the use of some extra axioms. It also ignores the fact that  $F_{\sqcup\sqcap}^e$  has a fixed structure for continuation. For example, a continuation that returns a subtyping work can only be  $\alpha \leq B$ , not  $A \leq \beta$  or  $\alpha \leq \beta$ , according to rules that create such continuations. To capture such properties and simplify the proof, we use the following defunctionalized [Reynolds 1972] style to inductively define the continuation (i.e. turning functions to data types).

There are two main differences compared to the syntax used in the previous sections: (1) the continuation is made into an independent syntax category using inductive definitions; (2) continuations that expect one type and two types are distinguished; (3)  $\_$  is used at places that expects an unknown result and meta-argument  $(\alpha, \beta)$  is not used any more. The connection between continuation and work is clear: when holes in a continuation are fully applied, the continuation will become a work. The reduction rules guarantee that the continuation is always fully applied.

$$\begin{array}{lcl}
 \text{Work} & w & ::= A \leq B \mid e \leftarrow A \mid e \Rightarrow \omega^s \mid A \circ B \Rightarrow \omega^s \mid A \triangleright \omega^d \\
 & & A \rightarrow B \bullet e \Rightarrow \omega^s \mid C \sqcup A \circ B \Rightarrow \omega^s \mid (B \rightarrow C) \sqcup A \triangleright \omega^d \\
 & & (A_1 \rightarrow B_1) \sqcup_\triangleright (A_2 \rightarrow B_2) \blacktriangleright \omega^d \mid A_1 \sqcup_\circ A_2 \blacktriangleright \omega^s \\
 \text{Continuation} & \omega^s & ::= \_ \leq A \mid \_ \circ A \Rightarrow \omega^s \mid \_ \triangleright \omega^d \mid \_ \sqcup A \circ B \Rightarrow \omega^s \mid \_ \sqcup_\circ A \blacktriangleright \omega^s \\
 & \omega^d & ::= \_ \rightarrow \_ \bullet e \Rightarrow \omega^s \mid \_ \rightarrow \_ \sqcup A \triangleright \omega^d \mid \_ \rightarrow \_ \sqcup_\triangleright (A \rightarrow B) \blacktriangleright \omega^d
 \end{array}$$

One limitation of the defunctionalized continuations is that the scoping is more rigid: results can only be passed to the next continuation, instead of arbitrary subsequent continuations. Additional effort has to be taken to represent the argument passed to the subsequent continuation used in rule 41 ( $\gamma_1$  and  $\gamma_2$ ) and 49 ( $\beta_2$ ) in Figure 4. This marks the third difference in syntax: (3) new intermediate works and continuations are introduced to cache the intermediate results (those marked gray). New reduction rules are introduced to process these works and continuations. Take the later three type-application rules as an example: rule 4 processes the left branch and relays to the second branch; when the result of the first branch is ready, rule 5 switches the focus to the second branch and relays the result of the first branch; when the results of both branches are ready, rule 6 combines them.

$$\begin{array}{l}
 \Gamma \Vdash A_1 \sqcup A_2 \triangleright \omega \longrightarrow_1 \Gamma \Vdash A_1 \triangleright (\_ \rightarrow \_ \sqcup A_2 \triangleright \omega) \\
 \Gamma \Vdash (B_2 \rightarrow C_2) \sqcup A_1 \triangleright \omega \longrightarrow_2 \Gamma \Vdash A_1 \triangleright (\_ \rightarrow \_ \sqcup_\triangleright (B_2 \rightarrow C_2) \blacktriangleright \omega) \\
 \Gamma \Vdash (A_1 \rightarrow B_1) \sqcup_\triangleright (A_2 \rightarrow B_2) \blacktriangleright \omega \longrightarrow_3 \Gamma \Vdash \omega \diamond (A_1 \sqcap A_2) \diamond (B_1 \sqcup B_2) \\
 \Gamma \Vdash A_1 \sqcup A_2 \circ B \Rightarrow \omega \longrightarrow_4 \Gamma \Vdash A_1 \circ B \Rightarrow (\_ \sqcup A_2 \circ B \Rightarrow \omega) \\
 \Gamma \Vdash C_2 \sqcup A_1 \circ B_1 \Rightarrow \omega \longrightarrow_5 \Gamma \Vdash A_1 \circ B \Rightarrow (\_ \sqcup_\circ C_2 \blacktriangleright \omega) \\
 \Gamma \Vdash A_1 \sqcup_\circ A_2 \blacktriangleright \omega \longrightarrow_6 \Gamma \Vdash \omega \diamond (A_1 \sqcup A_2)
 \end{array}$$

An example of reduction with such a defunctionalized style for type-application judgment with union types is shown below.

$$\begin{array}{l}
 \Gamma \Vdash ((\forall a.a) \sqcup \perp) \circ \mathbb{1} \Rightarrow \omega \\
 \longrightarrow^1 \Gamma \Vdash \forall a.a \circ \mathbb{1} \Rightarrow (\_ \sqcup (\perp \circ \mathbb{1}) \Rightarrow \omega) \longrightarrow^4 \Gamma \Vdash \perp \sqcup_\circ \mathbb{1} \blacktriangleright \omega \\
 \longrightarrow^2 \Gamma \Vdash \mathbb{1} \sqcup (\perp \circ \mathbb{1}) \Rightarrow \omega \longrightarrow^5 \Gamma \Vdash \omega \diamond (\mathbb{1} \sqcup \perp) \\
 \longrightarrow^3 \Gamma \Vdash \perp \circ \mathbb{1} \Rightarrow \_ \sqcup_\circ \mathbb{1} \blacktriangleright \omega
 \end{array}$$

The full set of algorithmic reduction rules using the defunctionalized continuation is shown in Figure 16 and Figure 17. We additionally make continuation applications  $\omega^s \diamond A$  and  $\omega^d \diamond A \diamond B$  as works, and design specific rules (rules 53 and 54) to handle their reduction. The reduction rules themselves are trivial: turning the applied continuation to the corresponding work, as described by the side condition  $w^s \diamond A = w$ . This relation is total and deterministic and could have been modeled as a function directly instead.

$\Gamma \longrightarrow \Gamma'$		$\Gamma$ worklist-reduces to $\Gamma'$
	$\Gamma, a \longrightarrow_1 \Gamma$	
	$\Gamma, \tilde{a} \longrightarrow_2 \Gamma$	
	$\Gamma, \widehat{a} \longrightarrow_3 \Gamma$	
	$\Gamma, x : A \longrightarrow_4 \Gamma$	
	$\Gamma \Vdash \perp \leq \perp \longrightarrow_5 \Gamma$	
	$\Gamma \Vdash a \leq a \longrightarrow_6 \Gamma$	
	$\Gamma \Vdash A \leq \top \longrightarrow_7 \Gamma$	
	$\Gamma \Vdash \perp \leq A \longrightarrow_8 \Gamma$	
$\Gamma \Vdash A_1 \rightarrow A_2 \leq B_1 \rightarrow B_2$	$\longrightarrow_9 \Gamma \Vdash B_1 \leq A_1 \Vdash A_2 \leq B_2$	
	$\Gamma \Vdash \forall a. A \leq B \longrightarrow_{10} \Gamma, \widehat{a} \Vdash A \leq B$	when $B^{\neq \forall}$
$\Gamma \Vdash \forall a. A_1 \leq \forall a. A_2$	$\longrightarrow_{11} \Gamma, \tilde{a} \Vdash A_1 \leq A_2$	
$\Gamma \Vdash A \leq B_2 \sqcap B_2$	$\longrightarrow_{12} \Gamma \Vdash A \leq B_1 \Vdash A \leq B_2$	
$\Gamma \Vdash A_1 \sqcap A_2 \leq B$	$\longrightarrow_{13} \Gamma \Vdash A_1 \leq B$	
$\Gamma \Vdash A_1 \sqcap A_2 \leq B$	$\longrightarrow_{14} \Gamma \Vdash A_2 \leq B$	
$\Gamma \Vdash A_1 \sqcup A_2 \leq B$	$\longrightarrow_{15} \Gamma \Vdash A_1 \leq B \Vdash A_2 \leq B$	
$\Gamma \Vdash A \leq B_1 \sqcup B_2$	$\longrightarrow_{16} \Gamma \Vdash A \leq B_1$	
$\Gamma \Vdash A \leq B_1 \sqcup B_2$	$\longrightarrow_{17} \Gamma \Vdash A \leq B_2$	
$\Gamma \Vdash a \leq \tau$	$\longrightarrow_{18} \{\tau/a\}\Gamma$	when $\widehat{a} \in \Gamma \wedge a \notin \text{fv}(\tau)$
$\Gamma \Vdash \tau \leq a$	$\longrightarrow_{19} \{\tau/a\}\Gamma$	when $\widehat{a} \in \Gamma \wedge a \notin \text{fv}(\tau)$
$\Gamma \Vdash a \leq A \rightarrow B$	$\longrightarrow_{20} \{a_1 \rightarrow a_2/a\}(\Gamma, \widehat{a}_1, \widehat{a}_2, a \leq A \rightarrow B)$	when $\widehat{a} \in \Gamma \wedge \Gamma \mu^m (A \rightarrow B)$
$\Gamma \Vdash A \rightarrow B \leq a$	$\longrightarrow_{21} \{a_1 \rightarrow a_2/a\}(\Gamma, \widehat{a}_1, \widehat{a}_2, A \rightarrow B \leq a)$	when $\widehat{a} \in \Gamma \wedge \Gamma \mu^m (A \rightarrow B)$
$\Gamma \Vdash e \Leftarrow B$	$\longrightarrow_{22} \Gamma \Vdash e \Rightarrow \_ \leq B$	
$\Gamma \Vdash \lambda x. e \Leftarrow A \rightarrow B$	$\longrightarrow_{23} \Gamma, x : A \Vdash e \Leftarrow B$	
$\Gamma \Vdash \lambda x. e \Leftarrow \top$	$\longrightarrow_{24} \Gamma, x : \perp \Vdash e \Leftarrow \top$	
$\Gamma \Vdash e \Leftarrow A_1 \sqcap A_2$	$\longrightarrow_{25} \Gamma \Vdash e \Leftarrow A_1 \Vdash e \Leftarrow A_2$	
$\Gamma \Vdash e \Leftarrow A_1 \sqcup A_2$	$\longrightarrow_{26} \Gamma \Vdash e \Leftarrow A_1$	
$\Gamma \Vdash e \Leftarrow A_1 \sqcup A_2$	$\longrightarrow_{27} \Gamma \Vdash e \Leftarrow A_2$	
$\Gamma \Vdash \lambda x. e \Leftarrow a$	$\longrightarrow_{28} \{a_1 \rightarrow a_2/a\}(\Gamma, \widehat{a}_1, \widehat{a}_2 \Vdash \lambda x. e \Leftarrow a)$	when $\widehat{a} \in \Gamma$
$\Gamma \Vdash x \Rightarrow \omega^s$	$\longrightarrow_{29} \Gamma \Vdash \omega^s \diamond A$	when $x : A \in \Gamma$
$\Gamma \Vdash e : A \Rightarrow \omega^s$	$\longrightarrow_{30} \Gamma \Vdash \omega \diamond A \Vdash e \Leftarrow A$	
$\Gamma \Vdash (\Lambda a. e : A) \Rightarrow \omega^s$	$\longrightarrow_{31} \Gamma \Vdash \omega \diamond (\forall a. A), a \Vdash e \Leftarrow A$	
$\Gamma \Vdash (\_) \Rightarrow \omega^s$	$\longrightarrow_{32} \Gamma \Vdash \omega \diamond \perp$	

Fig. 16. Algorithmic Worklist Reduction using Defunctionalized Continuation (Part I)



$$\begin{array}{l}
\Gamma \Vdash e_1 e_2 \Rightarrow \omega^S \longrightarrow_{33} \Gamma \Vdash e_1 \Rightarrow (\_ \triangleright (\_ \rightarrow \_ \bullet e_2 \Rightarrow \omega^S)) \\
\Gamma \Vdash e @A \Rightarrow \omega^S \longrightarrow_{34} \Gamma \Vdash e \Rightarrow (\_ \circ A \Rightarrow \omega^S) \\
\Gamma \Vdash \lambda x. e \Rightarrow \omega^S \longrightarrow_{35} \Gamma, \widehat{a}_1, \widehat{a}_2 \Vdash \omega \diamond (a_1 \rightarrow a_2), x : a_1 \Vdash e \Leftarrow a_2 \\
\Gamma \Vdash A \rightarrow B \triangleright \omega^d \longrightarrow_{36} \Gamma \Vdash \omega^d \diamond A \diamond B \\
\Gamma \Vdash \perp \triangleright \omega^d \longrightarrow_{37} \Gamma \Vdash \top \rightarrow \perp \triangleright \omega^d \\
\Gamma \Vdash \forall a. A \triangleright \omega^d \longrightarrow_{38} \Gamma, \widehat{a} \Vdash A \triangleright \omega^d \\
\Gamma \Vdash A_1 \sqcap A_2 \triangleright \omega^d \longrightarrow_{39} \Gamma \Vdash A_1 \triangleright \omega^d \\
\Gamma \Vdash A_1 \sqcap A_2 \triangleright \omega^d \longrightarrow_{40} \Gamma \Vdash A_2 \triangleright \omega^d \\
\Gamma \Vdash A_1 \sqcup A_2 \triangleright \omega^d \longrightarrow_{41} \Gamma \Vdash A_1 \triangleright (\_ \rightarrow \_ \cup A_2 \triangleright \omega) \\
\Gamma \Vdash a \triangleright \omega^d \longrightarrow_{42} \{a_1 \rightarrow a_2/a\}(\Gamma, \widehat{a}_1, \widehat{a}_2 \Vdash a \triangleright \omega^d) \quad \text{when } \widehat{a} \in \Gamma \\
\Gamma \Vdash (B_2 \rightarrow C_2) \cup A_1 \triangleright \omega^d \longrightarrow_{43} \Gamma \Vdash A_1 \triangleright (\_ \rightarrow \_ \cup B_2 \rightarrow C_2 \triangleright \omega) \\
\Gamma \Vdash (A_1 \rightarrow B_1) \cup (A_2 \rightarrow B_2) \triangleright \omega^d \longrightarrow_{44} \Gamma \Vdash \omega^d \diamond (A_1 \sqcap A_2) \diamond (B_1 \sqcup B_2) \\
\Gamma \Vdash A \rightarrow B \bullet e \Rightarrow \omega^S \longrightarrow_{45} \Gamma \Vdash \omega^S \diamond B \Vdash e \Leftarrow A \\
\Gamma \Vdash \forall a. A \circ B \Rightarrow \omega^S \longrightarrow_{46} \Gamma \Vdash \omega^S \diamond ([B/a]A) \\
\Gamma \Vdash \perp \circ A \Rightarrow \omega^S \longrightarrow_{47} \Gamma \Vdash \omega \diamond \perp \\
\Gamma \Vdash A_1 \sqcap A_2 \circ B \Rightarrow \omega^S \longrightarrow_{48} \Gamma \Vdash A_1 \circ B \Rightarrow \omega^S \\
\Gamma \Vdash A_1 \sqcap A_2 \circ B \Rightarrow \omega^S \longrightarrow_{49} \Gamma \Vdash A_2 \circ B \Rightarrow \omega^S \\
\Gamma \Vdash A_1 \sqcup A_2 \circ B \Rightarrow \omega^S \longrightarrow_{50} \Gamma \Vdash A_1 \circ B \Rightarrow (\_ \cup A_2 \circ B \Rightarrow \omega^S) \\
\Gamma \Vdash C_2 \cup A_1 \circ B_1 \Rightarrow \omega^S \longrightarrow_{51} \Gamma \Vdash A_1 \circ B \Rightarrow (\_ \cup \circ C_2 \triangleright \omega^S) \\
\Gamma \Vdash A_1 \cup \circ A_2 \triangleright \omega \longrightarrow_{52} \Gamma \Vdash \omega \diamond (A_1 \sqcup A_2) \\
\Gamma \Vdash \omega^S \diamond A \longrightarrow_{53} \Gamma \Vdash w \quad \text{when } \omega^S \diamond A = w \\
\Gamma \Vdash \omega^d \diamond A \diamond B \longrightarrow_{54} \Gamma \Vdash w \quad \text{when } \omega^d \diamond A \diamond B = w
\end{array}$$

Fig. 17. Algorithmic Worklist Reduction using Defunctionalized Continuation (Part II)

### B.5 Rules of $F_{\sqcup\sqcap}^e$ with Records

The additional well-formedness check for record expressions is straightforward to implement, as demonstrated below. Singleton records are records and record extensions are well-formed if their tails are records. Some other restrictions like the uniqueness of labels are not enforced, which influences the determinism of the projection. Since the focus of this record system is mainly to demonstrate the expressive power of the type system, we pay less attention to the semantics of the program. However, these restrictions, if wanted, can be added to the language easily.

$$\frac{\Psi \vdash e_1 \quad \Psi \vdash e_2 \quad e_2^{=(*)}}{\Psi \vdash \langle l_1 \mapsto e_1, e_2 \rangle} \quad \frac{}{\langle l \mapsto e \rangle^{=(*)}} \quad \frac{e_2^{=(*)}}{\langle l_1 \mapsto e_1, e_2 \rangle^{=(*)}}$$

The algorithmic system of  $F_{\sqcup\sqcap}^e$  with records still shares the same type and expression as the bidirectional system. The syntax of work extended with three new works: label inference ( $A \blacktriangleright_\alpha \omega$ ), label inference intersection ( $A \blacklozenge B \blacktriangleright_\alpha \omega$ ) and projection inference ( $A \rightarrow B \blacksquare C \Rightarrow_\alpha \omega$ ). Seven new rules are added to the algorithmic reduction.

$$\begin{aligned}
\text{Work } w ::= & \dots \mid A \blacktriangleright_{\alpha} \omega \mid A \pitchfork B \blacktriangleright_{\alpha} \omega \mid A \rightarrow B \blacksquare C \Rightarrow_{\alpha} \omega \\
\Gamma \Vdash \text{Label } l \leq \text{Label } l & \longrightarrow_1 \Gamma \\
\Gamma \Vdash \langle l \mapsto e \rangle \Rightarrow_{\alpha} \omega & \longrightarrow_2 \Gamma \Vdash e \Rightarrow_{\beta} (\text{Label } l \rightarrow \beta \blacktriangleright_{\alpha} \omega) \\
\Gamma \Vdash \langle l_1 \mapsto e_1, e_2 \rangle \Rightarrow_{\alpha} \omega & \longrightarrow_3 \Gamma \Vdash e_1 \Rightarrow_{\beta} (e_2 \Rightarrow_{\gamma} (\text{Label } l_1 \rightarrow \beta \pitchfork \gamma \blacktriangleright_{\alpha} \omega)) \\
\Gamma \Vdash e.l \Rightarrow_{\alpha} \omega & \longrightarrow_4 \Gamma \Vdash e \Rightarrow_{\alpha} (\alpha \triangleright_{\beta, \gamma} (\beta \rightarrow \gamma \blacksquare \text{Label } l \Rightarrow_{\alpha} \omega)) \\
\Gamma \Vdash A \blacktriangleright_{\alpha} \omega & \longrightarrow_5 \Gamma \Vdash \omega \diamond A \\
\Gamma \Vdash A_1 \pitchfork A_2 \blacktriangleright_{\alpha} \omega & \longrightarrow_6 \Gamma \Vdash \omega \diamond (A_1 \sqcap A_2) \\
\Gamma \Vdash A \rightarrow B \blacksquare C \Rightarrow_{\alpha} \omega & \longrightarrow_7 \Gamma \Vdash \omega \diamond B \Vdash C \leq A
\end{aligned}$$

Rules 5 and 6 seem to do nothing because the task of label inference and label inference intersection works is just post-processing the result, e.g., combining Label  $l$  and  $A$  into a single function type Label  $l \rightarrow A$ . Since rule 3 also uses nested continuation. Its defunctionalization requires creating a new type of intermediate work to relay the inference result of  $e_1$ , similar to the defunctionalization of the union type case of matching and type application. In the defunctionalized version, four new works, four new continuations, and eight new rules are added compared to base  $F_{\sqcup}^e$ .

### C Additional Discussion of $F_{\sqcup}^e$

*Specification of type application with  $\perp$ .* The specification of type application where  $\perp$  could appear in  $A$  is shown below. There are two differences in the  $\circ \Rightarrow$  to  $\leq$  direction: (1). there is an additional  $A \leq \perp$  case; (2). the syntactic equality  $[B/a]A' = C$  is replaced by the equivalent relation ( $\Psi \vdash C \leq [B/a]A'$  and  $\Psi \vdash [B/a]A' \leq C$ ). Though this specification looks complicated, its meaning is intuitive: if  $A$  can be type applied, we can find a bot-like or universal-type-like supertype  $A'$  of  $A$ , whose type-application result is equivalent to  $A$ . The  $\leq$  to  $\circ \Rightarrow$  remains unchanged.

LEMMA C.1 (SPECIFICATION OF TYPE APPLICATION). *Given all contexts and types well-formed,*

- (1) if  $\Psi \vdash A \circ B \Rightarrow C$ , then either
  - (a) exists  $A'$ ,  $\Psi \vdash A \leq \forall a.A'$ ,  $\Psi \vdash C \leq [B/a]A'$  and  $\Psi \vdash [B/a]A' \leq C$ ; or
  - (b)  $\Psi \vdash A \leq \perp$  and  $\Psi \vdash C \leq \perp$ ;
- (2) if  $\Psi \vdash A \leq \forall a.A'$ , then exists  $C$  s.t.  $\Psi \vdash A \circ B \Rightarrow C$  and  $\Psi' \vdash C \leq [B/a]A'$ .

*Greediness.* The greediness of the algorithm means it always uses the first monotype seen in the subtyping judgment  $a \leq \tau$  or  $\tau \leq a$  to solve  $a$ . However, the “first” subtyping judgment is an implementation-dependent idea, which may differ from the original position  $a$  appears in the universal type. In our current algorithmic rule,  $\Gamma \Vdash a \rightarrow a \leq A \rightarrow B \longrightarrow \Gamma \Vdash A \leq a \Vdash a \leq B$  where the codomain type is solved first. In the base  $F_{\sqcup}^e$  system, this order makes no difference since solving whichever first leads to the same behavior. However, in the  $F_{\sqcup}^e$  with instantiable intersection and union types, different reduction rules would lead to different examples being accepted/rejected.

The different behavior between our implementation and TypeScript on `ex8_2` ( $\forall a.a \rightarrow a \rightarrow \text{Int} \leq \text{Int} \rightarrow (\text{Int} \sqcup \text{Bool}) \rightarrow \text{Int}$ ) and `ex8_4` ( $\forall a.a \rightarrow a \rightarrow \text{Int} \leq (\text{Int} \sqcup \text{Bool}) \rightarrow \text{Int} \rightarrow \text{Int}$ ) is due to this: our solving prioritizes the codomain type to solve the instantiation while TypeScript prioritizes the domain type. We believe that changing  $\Gamma \Vdash A_1 \rightarrow B_1 \leq A_2 \rightarrow B_2 \longrightarrow \Gamma \Vdash A_2 \leq A_1 \Vdash B_1 \leq B_2$  to  $\Gamma \Vdash A_1 \rightarrow B_1 \leq A_2 \rightarrow B_2 \longrightarrow \Gamma \Vdash B_1 \leq B_2 \Vdash A_2 \leq A_1$  would fix this inconsistency, and it fits people’s intuition on greediness better since the domain type also appears first in the original type.

In our algorithmic rules,  $\Gamma \Vdash a \leq \tau \longrightarrow \{\tau/a\}\Gamma$  overlaps with  $\Gamma \Vdash A \leq B_1 \sqcup B_2 \longrightarrow \Gamma \Vdash A \leq B_1$  and  $\Gamma \Vdash A \leq B_1 \sqcup B_2 \longrightarrow \Gamma \Vdash A \leq B_2$ . Similarly,  $\Gamma \Vdash \tau \leq a \longrightarrow \{\tau/a\}\Gamma$  overlaps with

$\Gamma \Vdash A_1 \sqcap A_2 \leq B \longrightarrow \Gamma \Vdash A_1 \leq B$  and  $\Gamma \Vdash A_1 \sqcap A_2 \leq B \longrightarrow \Gamma \Vdash A_2 \leq B$ . These overlappings (or backtracking) are essential to guarantee this variant strictly accepts more programs. Otherwise, as the rejection of `ex8_3` by TypeScript demonstrates, a fully greedy implementation would reject programs where an instantiation without intersection and union types exists. This algorithm is still greedy in that it still fully relies on the first instantiation found.  $\cdot, \hat{a} \Vdash a \leq \mathbb{1} \Vdash a \leq \mathbb{1} \sqcap (\mathbb{1} \rightarrow \mathbb{1})$  reduces, but  $\cdot, \hat{a} \Vdash a \leq \mathbb{1} \sqcap (\mathbb{1} \rightarrow \mathbb{1}) \Vdash a \leq \mathbb{1}$  does not, just because the order of instantiation changes. TypeScript does opt for less backtracking even if it rejects some more programs.

## D Proof Details

### D.1 Proof Details of Type Safety

$A \hookrightarrow A^F$

$A$  is translated to  $A^F$

$$\begin{array}{c}
\frac{}{\mathbb{1} \hookrightarrow \mathbb{1}} \hookrightarrow_{\text{Unit}} \quad \frac{}{a \hookrightarrow a} \hookrightarrow_{\text{Var}} \quad \frac{}{\top \hookrightarrow \mathbb{1}} \hookrightarrow_{\top} \quad \frac{}{\perp \hookrightarrow \forall a. a} \hookrightarrow_{\perp} \quad \frac{A \hookrightarrow A^F \quad B \hookrightarrow B^F}{A \rightarrow B \hookrightarrow A^F \rightarrow B^F} \hookrightarrow_{\rightarrow} \\
\frac{A \hookrightarrow A^F}{\forall a. A \hookrightarrow \forall a. A^F} \hookrightarrow_{\forall} \quad \frac{A \hookrightarrow A^F \quad B \hookrightarrow B^F}{A \sqcap B \hookrightarrow A^F \times B^F} \hookrightarrow_{\sqcap} \quad \frac{A \hookrightarrow A^F \quad B \hookrightarrow B^F}{A \sqcup B \hookrightarrow A^F + B^F} \hookrightarrow_{\sqcup}
\end{array}$$

$\Psi \hookrightarrow \Delta$

$\Psi$  is translated to  $\Delta$

$$\begin{array}{c}
\frac{}{\cdot \hookrightarrow \cdot} \hookrightarrow_{\cdot} \quad \frac{\Psi \hookrightarrow \Psi^F \quad A \hookrightarrow A^F}{\Psi, x : A \hookrightarrow \Delta, x : A^F} \hookrightarrow_x \quad \frac{\Psi \hookrightarrow \Psi^F}{\Psi, a \hookrightarrow \Delta, a} \hookrightarrow_a \quad \frac{\Psi \hookrightarrow \Psi^F}{\Psi, \tilde{a} \hookrightarrow \Delta, a} \hookrightarrow_{\tilde{a}}
\end{array}$$

Fig. 18. Type and Context Translation

$\Psi \vdash A \leq B \hookrightarrow co$

Subtyping  $\Psi \vdash A \leq B$  is elaborated to  $co$

$$\begin{array}{c}
\frac{}{\Psi \vdash \mathbb{1} \leq \mathbb{1} \hookrightarrow \lambda(x : \mathbb{1}). x} \leq_{\text{Unit}} \quad \frac{}{\Psi \vdash A \leq \top \hookrightarrow \lambda(x : |A|). ()} \leq_{\top} \quad \frac{}{\Psi \vdash \perp \leq A \hookrightarrow \lambda(x : \forall a. a). x @|A|} \leq_{\perp} \\
\frac{}{\Psi \vdash a \leq a \hookrightarrow \lambda(x : a). x} \leq_{\text{Var}} \quad \frac{\Psi \vdash B_1 \leq A_1 \hookrightarrow co_1 \quad \Psi \vdash A_2 \leq B_2 \hookrightarrow co_2}{\Psi \vdash A_1 \rightarrow A_2 \leq B_1 \rightarrow B_2 \hookrightarrow \lambda(f : |A_1 \rightarrow A_2|). \lambda(x : |B_1|). co_2 (f (co_1 x))} \leq_{\rightarrow} \\
\frac{\Psi \vdash [\tau/a]A \leq B \hookrightarrow co_1 \quad B^{\neq \forall}}{\Psi \vdash \forall a. A \leq B \hookrightarrow \lambda(x : |\forall a. A|). co_1 (x @|\tau|)} \leq_{\forall} \quad \frac{\Psi, \tilde{a} \vdash A_1 \leq A_2 \hookrightarrow co_1}{\Psi \vdash \forall a. A_1 \leq \forall a. A_2 \hookrightarrow \lambda(x : |\forall a. A_1|). \Lambda a. co_1 (x @a)} \leq_{\forall} \\
\frac{\Psi \vdash A \leq B_1 \hookrightarrow co_1 \quad \Psi \vdash A \leq B_2 \hookrightarrow co_2}{\Psi \vdash A \leq B_1 \sqcap B_2 \hookrightarrow \lambda(x : |A|). \langle co_1 x, co_2 x \rangle} \leq_{\sqcap} \\
\frac{\Psi \vdash A_1 \leq B \hookrightarrow co_1}{\Psi \vdash A_1 \sqcap A_2 \leq B \hookrightarrow \lambda(x : |A_1 \sqcap A_2|). co_1 (\pi_1 x)} \leq_{\sqcap L_1} \quad \frac{\Psi \vdash A_2 \leq B \hookrightarrow co_1}{\Psi \vdash A_1 \sqcap A_2 \leq B \hookrightarrow \lambda(x : |A_1 \sqcap A_2|). co_1 (\pi_2 x)} \leq_{\sqcap L_2} \\
\frac{\Psi \vdash A_1 \leq B \hookrightarrow co_1 \quad \Psi \vdash A_2 \leq B \hookrightarrow co_2}{\Psi \vdash A_1 \sqcup A_2 \leq B \hookrightarrow \lambda(x : |A_1 \sqcup A_2|). \text{case } x \text{ of } y \mapsto co_1 y; z \mapsto co_2 z} \leq_{\sqcup} \\
\frac{\Psi \vdash A \leq B_1 \hookrightarrow co_1}{\Psi \vdash A \leq B_1 \sqcup B_2 \hookrightarrow \lambda(x : |A|). \text{inl} (co_1 x)} \leq_{\sqcup R_1} \quad \frac{\Psi \vdash A \leq B_2 \hookrightarrow co_1}{\Psi \vdash A \leq B_1 \sqcup B_2 \hookrightarrow \lambda(x : |A|). \text{inr} (co_2 x)} \leq_{\sqcup R_2}
\end{array}$$

Fig. 19. Elaboration of Bidirectional Subtyping

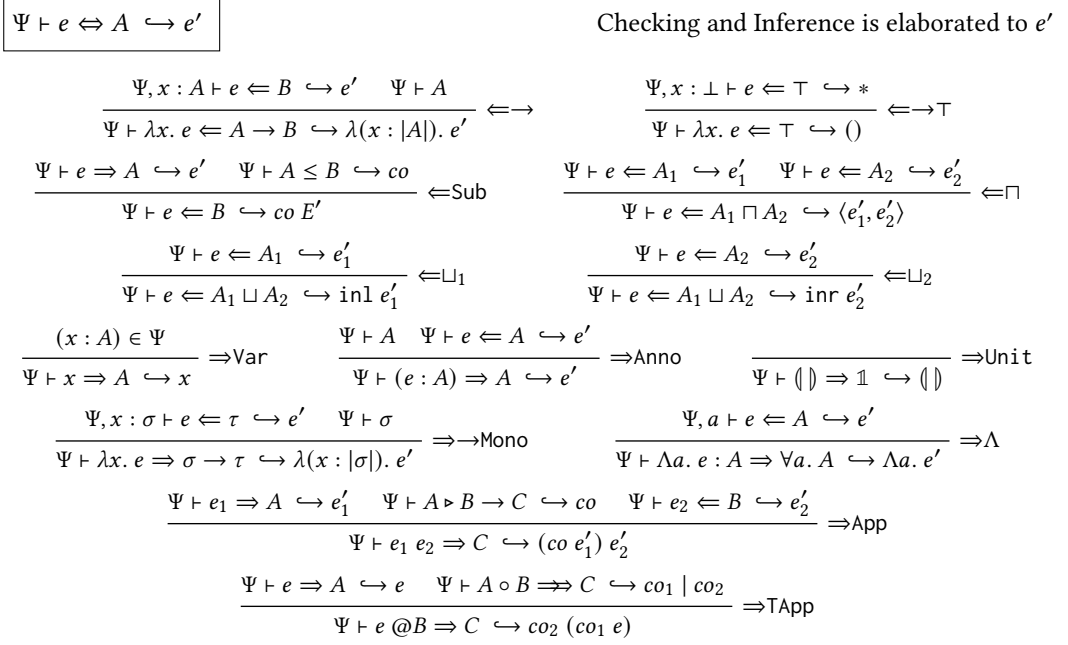


Fig. 20. Elaboration of Checking and Inference

Type variables	$a, b$	Variables	$x, y$
Types	$A, B, C ::=$	$\mathbb{1} \mid a \mid \forall a. A \mid A \rightarrow B \mid A + B \mid A \times B$	
Expressions	$e, t ::=$	$x \mid () \mid \lambda x : A. e \mid e_1 e_2 \mid e : A \mid e @A \mid \Lambda a. e \mid \text{inl } e \mid \text{inr } e \mid \text{case } e \text{ of } \text{inl } x \rightarrow e_1; \text{inr } y \rightarrow e_2 \mid \Pi_1 e \mid \Pi_2 e \mid \langle e_1, e_2 \rangle$	

The type safety is proved by elaborating expressions in  $F_{\sqcup\cap}^e$  to expressions in System  $F$  with a terminal type  $\mathbb{1}$ , and product ( $A \times B$ ) and sum ( $A + B$ ) type. For brevity, we use  $F_{\times+}$  to denote this target type system. The elaboration of intersection and union is similar to the rules by [Dunfield \[2014\]](#) except we do not have an expression-level merge operator. This target system is known to be type-safe.

The type translation is shown in 18. Intersection type  $A \sqcap B$  is translated to product type  $A \times B$ . Union type  $A \sqcup B$  is translated to sum type  $A + B$ . Both  $\top$  and  $\mathbb{1}$  is translated to  $\mathbb{1}$ .  $\perp$  is translated to  $\forall a. a$ . For context translation, the subtype variable is translated to an ordinary type variable as  $\tilde{a}, \Psi \hookrightarrow a, \Delta$ .

The main goal of the elaboration on type-level relations (subtyping  $A \leq B$ , matching  $A \triangleright B \rightarrow C$ , and type application  $A \circ B \Rightarrow C$ ) is to produce a coercion function. For subtyping, this function should be of type  $|A| \rightarrow |B|$ . For matching, this function should be of type  $|A| \rightarrow |B| \rightarrow |C|$ . For type application, the situation is a bit complicated as  $B$  is included in the relation. It should produce two coercions.  $co_1$  converts  $|A|$  to another type  $|D|$  that can be type applied (i.e.  $|\perp|$  and  $|\forall a. A'|$ ), and  $co_2$  coerce type  $|D|$  to the desired output type  $|C|$  by type applying  $|D|$  to  $|B|$ . There are two tasks for such coercion: (1) inserting explicit type application to places where implicit instantiation is invoked; (2) inserting projections ( $\pi_1, \pi_2$ ) and injections ( $\text{inl}, \text{inr}$ ) where a branch is chosen implicitly by rules related to intersection and union types.

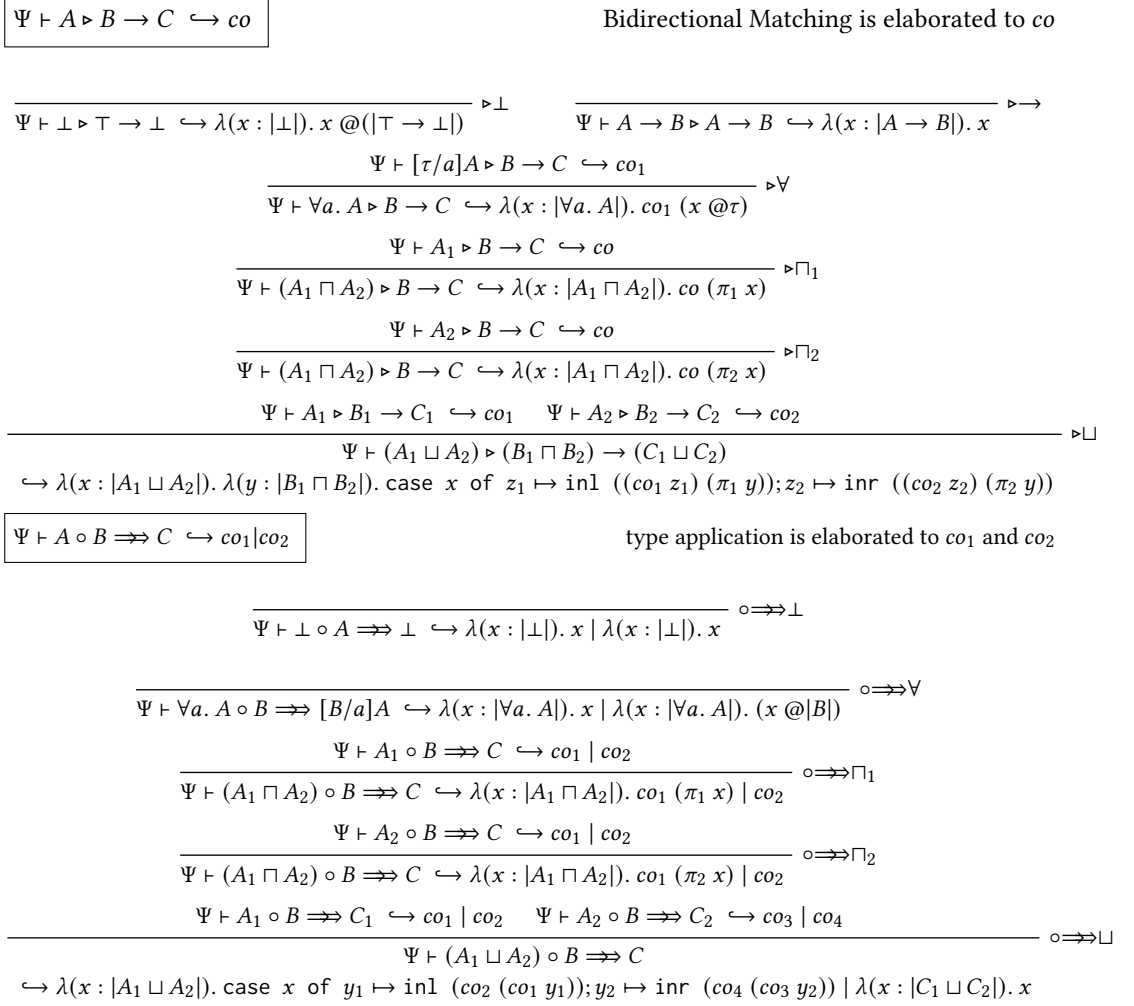


Fig. 21. Elaboration of Matching and Type Application

LEMMA D.1 (SOUNDNESS OF SUBTYPING, MATCHING AND TYPE-APPLICATION BY ELABORATING TO SYSTEM  $F_{\times+}$ ). *Given everything well-formed and  $\Psi \hookrightarrow \Delta$ ,*

- (1) *If  $\Psi \vdash A \leq B$  then  $\Delta \vdash_{F_{\times+}} co : |A| \rightarrow |B|$ .*
- (2) *If  $\Psi \vdash A \triangleright B \rightarrow C \hookrightarrow co$  then  $\Delta \vdash_{F_{\times+}} co : |A| \rightarrow |B| \rightarrow |C|$ .*
- (3) *If  $\Psi \vdash A \circ B \Rightarrow C \hookrightarrow co_1 | co_2$  then,  $\exists D$ , s.t.  $\Delta \vdash_{F_{\times+}} co_1 : |A| \rightarrow |D|$  and  $\Delta \vdash_{F_{\times+}} co_2 : |D| \rightarrow |C|$ .*

The soundness of elaboration for inference ( $\Psi \vdash e \Rightarrow A$ ) and checking ( $\Psi \vdash e \Leftarrow A$ ) states that the elaborated term  $e'$  has type  $|A|$ .

THEOREM D.2 (TYPE SOUNDNESS BY ELABORATING TO SYSTEM  $F_{\times+}$ ). *Given everything well-formed and  $\Psi \hookrightarrow \Delta$ ,*

- (1) *If  $\Psi \vdash e \Rightarrow A \hookrightarrow e'$  then  $\Delta \vdash_{F_{\times+}} e' : |A|$ .*
- (2) *If  $\Psi \vdash e \Leftarrow A \hookrightarrow e'$  then  $\Delta \vdash_{F_{\times+}} e' : |A|$ .*

## D.2 Proof Details of Decidability

In this section, we provide an overview of the decidability. The decidability proof is based on a lexicographic group of 9 measures on the worklist  $\Gamma$ : ( $|\Gamma|_e$ ,  $|\Gamma|_\omega$ ,  $|\Gamma|_{\Rightarrow}$ ,  $|\Gamma|_{\Rightarrow\omega}$ ,  $|\Gamma|_>$ ,  $|\Gamma|_{>\omega}$ ,  $|\Gamma|_{\rightarrow}$ ,  $|\Gamma|_{\hat{a}}$ ,  $|\Gamma|_{\leq}$ ). We will mostly demonstrate the high-level ideas behind the design of these measures since their correctness has been formally checked. The measures presented here are an overestimation, so possibly some of them can be modified to have a more precise overestimation. All the rule indexing in this section is based on the defunctionalized version shown in Figure 16 and Figure 17.

$ A _{\cup\cap}$	number of intersections and unions in $A$
	$ 1 _{\cup\cap} =  \top _{\cup\cap} =  \perp _{\cup\cap} =  a _{\cup\cap} = 0$ $ A \rightarrow B _{\cup\cap} =  A _{\cup\cap} +  B _{\cup\cap}$ $ \forall a. A _{\cup\cap} =  A _{\cup\cap}$ $ A \cap B _{\cup\cap} =  A _{\cup\cap} +  B _{\cup\cap} + 2$ $ A \sqcup B _{\cup\cap} =  A _{\cup\cap} +  B _{\cup\cap} + 2$
$ A _{\cup\cap a}, \Xi e _{\cup\cap a}$	number of intersections, unions and bound type variables in $A, e$
	$\Xi () _{\cup\cap a} = 0$ $\Xi x _{\cup\cap a} = n \quad \text{when } x : n \in \Xi$ $\Xi \lambda x. e _{\cup\cap a} = \Xi_{x:0} e _{\cup\cap a}$ $\Xi e_1 e_2 _{\cup\cap a} = 2 \Xi e_1 _{\cup\cap a} + \Xi e_2 _{\cup\cap a} + 1$ $\Xi \Lambda a. e : A _{\cup\cap a} =  \forall a. A _{\cup\cap a}$ $\Xi e @A _{\cup\cap a} = (\Xi e _{\cup\cap a} + 1) \cdot ( A _{\cup\cap a} + 2)$ $\Xi e : A _{\cup\cap a} = (\Xi e _{\cup\cap a} + 1) \cdot ( A _{\cup\cap a} + 2)$
$\ \Gamma\  = \Xi$	compute $\Xi(  \cdot  _{\cup\cap a}$ context) from $\Gamma$
	$\ \cdot\  = \cdot$ $\ \Gamma, a\  = \ \Gamma\ $ $\ \Gamma, \hat{a}\  = \ \Gamma\ $ $\ \Gamma, \hat{a}\  = \ \Gamma\ $ $\ \Gamma, \hat{a}\  = \ \Gamma\ $ $\ \Gamma, \hat{a}\  = \ \Gamma\ $

Fig. 22. Definition of  $| \cdot |_{\cup\cap}$ ,  $| \cdot |_{\cup\cap a}$  and  $\|\Gamma\|$

The definition  $| \cdot |_e$  is the most complicated one, due to the intrinsic complication of the duplication of expression. The duplication of expression is caused by the checking introduction rule of intersection ( $e \Leftarrow A_1 \cap A_2$ ). It seems that we can easily estimate the duplication from  $e, A_1$ , and  $A_2$ . However, checking judgment can be created by the inference judgment in rules 30 ( $e : A \Rightarrow \omega^s$ ), 31 ( $\Lambda a. e : A \Rightarrow \omega^s$ ), and 33 ( $e_1 e_2 \Rightarrow \omega^s$ ,  $e_2$  will be checked by the domain type matched from the type of  $e_1$ ). This makes the first difficulty: for inference, the type information is unknown yet, so we must estimate it from the expressions themselves. The estimation for annotation and type abstraction is simple since the type information is fully given. The estimation for application is tricky since

$e_1$  can be any expression, especially when  $e_1$  is  $x$  and type-application. This estimation is finally done by  $|\cdot|_{\sqcup\cap a}$ , and  $\|\Gamma\|$ , whose definition is in Figure 22.  $|e|_{\sqcup\cap a}$  roughly estimates the number of duplications if another expression  $e_2$  is checked by the type inferred from  $e$ .  $|\cdot|_{\sqcup\cap a}$  considers the number of intersection and union types and bound type variables that appear in  $e$ . The number of bound type variables is considered due to the existence of  $e @A$ , so that the bound type variables could be replaced by  $A$  later so that more intersection and union types can be introduced. This measure is employed in many places of the definition. The function  $|\Gamma|$  precomputes the size of the type of each variable in the context  $\Gamma$ , which is used in  $|\cdot|_e$ .

Figure 23 shows the definition of  $|\cdot|_e$ , which is based on  $|\cdot|_{\sqcup\cap a}$ . The second complication is that the size cannot be simply computed by the pattern matching the expression. Suppose we want to estimate the size of  $(e : A) : B$ ,  $e : A$  will be duplicated by a number related to  $B$ , and for each duplication,  $e$  will be duplicated by a number related to  $A$ , which means the estimation must carry the outer duplication information in computing the size of the inner components. This leads to a tail-recursively computed measure  $\Xi|e|_e^n$ ,  $\Xi|\omega^s|_e^n$  and  $\Xi|\omega^d|_e^{(n_1, n_2)}$ , where the left subscript  $\Xi$  denotes the precomputed context, and the right superscript is the size carried by tail recursion. For  $\Xi|\omega^d|_e^{(n_1, n_2)}$  the size carried by its tail recursion, as well as its output, is a pair. The first element of the pair is the size of the negative position, and the second is the size of the positive position. The passed pair enables more precise approximation. To summarize, there are two types of recursion in the computation of  $|\cdot|_e$ : recursing over the continuation chain in  $|\omega|_e$  and recursing over the expression in  $|e|_e$ .

Figure 24 shows the definition of  $|\cdot|_\omega$ . Matching and type-application judgments count as zero in this measure as they are reasoned modularly by their own measures. Checking, inference, application inference, and continuation applications are non-zero. Since only application inference has a corresponding continuation, it is the only non-zero measure in  $|\omega^s|_\omega$  and  $|\omega^d|_\omega$ .

Figure 26 shows the definition of  $|\cdot|_b$ . Figure 25 shows the definition of  $|\cdot|_{\Rightarrow\omega}$ . Figure 27 shows the definition of  $|\cdot|_{\Rightarrow\omega}$  and  $|\cdot|_{b_\omega}$ . Since the size of the type being applied could increase as  $(\forall a. a \circ ((\forall a. a) \sqcup (\forall a. a)))$ , the information along the continuation chain must be considered in advance to get the correct estimation. Thus,  $|\cdot|_{\Rightarrow\omega}$  needs to recurse over the continuation chain. The situation is simpler for matching since the type being matched always gets smaller.  $|\cdot|_b$  is simply a summation over all continuation.  $|\cdot|_{\Rightarrow\omega}$  and  $|\cdot|_{b_\omega}$  are mainly introduced due to the defunctionalization and they help reason about the reduction of the intermediate works and continuations that relay the information.

$|\cdot|_{\rightarrow}$  is shown in Figure 26. Its meaning is well justified by its name: sum of rank of non-mono components.  $|\widehat{a}|_{\leq}$  is a simple measure that counts the existential variable declaration in  $\Gamma$ .

$|\cdot|_{\leq}$  is for the reasoning of subtyping. Duplication also happens in subtyping in rules 12 and 15. However, the information in subtyping judgment is always full, so the estimation is much simpler than in the case of expression. The key design is  $|A \leq B|_{\leq} = |A|_{\leq} \cdot |B|_{\sqcup\cap} + |B|_{\leq} \cdot |A|_{\sqcup\cap}$  where  $|A|_{\sqcup\cap}$  is a simpler version of  $|A|_{\sqcup\cap a}$  that only counts the intersection and union type, but not the bound type variables. As the last measure,  $|\cdot|_{\leq}$  also deals with some miscellaneous cases like the garbage collection of variables and type variables, so it also counts the number of such declarations.

All the first 7 measures are smaller or unchanged under worklist substitution with arbitrary monotype, as formulated by the following theorem.  $|\widehat{a}|_{\leq}$  decreases by 1 since the target  $a$  is removed from  $\Gamma$ .

**THEOREM D.3 (MEASURE STABILITY UNDER WORKLIST SUBSTITUTION).** *Given a well-formed  $\Gamma$ , and a well-formed monotype  $\tau$  under  $\Gamma$ ,*

- (1)  $|\{\tau/a\}\Gamma|_e \leq |\Gamma|_e$
- (2)  $|\{\tau/a\}\Gamma|_\omega \leq |\Gamma|_\omega$



- (3)  $|\{\tau/a\}\Gamma|_{\Rightarrow} \leq |\Gamma|_{\Rightarrow}$
- (4)  $|\{\tau/a\}\Gamma|_{\Rightarrow\omega} \leq |\Gamma|_{\Rightarrow\omega}$
- (5)  $|\{\tau/a\}\Gamma|_{\triangleright} \leq |\Gamma|_{\triangleright}$
- (6)  $|\{\tau/a\}\Gamma|_{\triangleright\omega} \leq |\Gamma|_{\triangleright\omega}$
- (7)  $|\{\tau/a\}\Gamma|_{\rightarrow} \leq |\Gamma|_{\rightarrow}$

*Case Analysis.* We summarize the decreasing measure of each rule. This set of measures must decrease after a fixed number of reduction steps, so the reasoning is simple, ignoring the complication of designing the measures.

The measure decreases after a one-step reduction for the following rules.

- For rules 23-27 and 29-35,  $|\Gamma|_e$  decreases.
- For rules 22, 45, 53, and 54,  $|\Gamma|_\omega$  decreases.
- For rules 48-50,  $|\Gamma|_{\Rightarrow}$  decreases.
- For rule 51,  $|\Gamma|_{\Rightarrow\omega}$  decreases.
- For rules 37-42,  $|\Gamma|_{\triangleright}$  decreases.
- For rule 43,  $|\Gamma|_{\triangleright\omega}$  decreases.
- For rules 10, 20 and 21,  $|\Gamma|_{\rightarrow}$  decreases.
- For rules 18 and 19,  $|\Gamma|_{\hat{a}}$  decreases.
- For rules 1-9 and 11-17,  $|\Gamma|_{\leq}$  decreases.

In the remaining cases, the measure decreases after a two-step reduction.

- For rule 28,  $|\Gamma|_e$  decreases.
- For rules 46, 47,  $|\Gamma|_{\Rightarrow}$  decreases.
- For rule 52,  $|\Gamma|_{\Rightarrow\omega}$  decreases.
- For rules 36 and 44,  $|\Gamma|_{\triangleright}$  decreases.

$$\boxed{\Xi|e|_e^n, \Xi|\omega^s|_e^n, \Xi|\omega^d|_e^{(n_1, n_2)}, \Xi|w|_e, |\Gamma|_e}$$

expression size of  $e$ ,  $\omega^s$ ,  $\omega^d$ ,  $w$ , and  $\Gamma$

$$\begin{aligned} \Xi|(|)|_e^n &= \Xi|x|_e^n = n + 1 \\ \Xi|\lambda x. e|_e^n &= \Xi_{x:n}|e|_e^n + n + 1 \\ \Xi|e_1 e_2|_e^n &= \Xi|e_1|_e^n + (\Xi|e_1|_{\sqcup a} + 1) \cdot \Xi|e_2|_e^{(\Xi|e_1|_{\sqcup a} + 1) \cdot (n+1) - 1} + 1 \\ \Xi|\Lambda a. e : A|_e^n &= (n + 1) \cdot (\Xi|e|_e^{|\Lambda a|} + 1) \\ \Xi|e @A|_e^n &= (n + 1) \cdot (\Xi|e|_e^{(|\Lambda a| + 1) \cdot (n+1) - 1} + 1) \\ \Xi|e : A|_e^n &= (n + 1) \cdot (\Xi|e|_e^{|\Lambda a|} + 1) \\ \Xi|_ \leq A|_e^n &= 0 \\ \Xi|_ \circ A &\Rightarrow \omega^s|_e^n = \Xi|\omega^s|_e^{(|\Lambda a| + 2) \cdot n} \\ \Xi|_ \triangleright \omega^d|_e^n &= \sum \Xi|\omega^d|_e^{(n, n)} \\ \Xi|_ \sqcup A \circ B &\Rightarrow \omega^s|_e^n = \Xi|\omega^s|_e^{|\Lambda a| \cdot (|\Lambda b| + 2) + n + 2} \\ \Xi|_ \sqcup \circ A \triangleright \omega^s|_e^n &= \Xi|\omega^s|_e^{|\Lambda a| + n + 2} \\ \Xi|_ \rightarrow \bullet e &\Rightarrow \omega^s|_e^{(n_1, n_2)} = (0, \Xi|e|_e^{n_1} \cdot (n_1 + 1) + \Xi|\omega^s|_e^{n_2}) \\ \Xi|_ \rightarrow \sqcup A \triangleright \omega^d|_e^{(n_1, n_2)} &= \Xi|\omega^d|_e^{(n_1 + |\Lambda a| + 2, n_2 + |\Lambda a| + 2)} \\ \Xi|_ \rightarrow \sqcup \triangleright (A \rightarrow B) \triangleright \omega^d|_e^{(n_1, n_2)} &= \Xi|\omega^d|_e^{(n_1 + |\Lambda a| + 2, n_2 + |\Lambda a| + 2)} \\ \Xi|A \leq B|_e &= 0 \\ \Xi|e \Leftarrow A|_e &= \Xi|e|_e^{|\Lambda a|} \\ \Xi|e \Rightarrow \omega^s|_e &= \Xi|e|_e^0 + \Xi|\omega^s|_e^{|\Lambda a|} \\ \Xi|A \circ B &\Rightarrow \omega^s|_e = \Xi|\omega^s|_e^{|\Lambda a| \cdot (|\Lambda b| + 2)} \\ \Xi|A \triangleright \omega^d|_e &= \sum \Xi|\omega^d|_e^{(|\Lambda a|, |\Lambda a|)} \\ \Xi|A \rightarrow B \bullet e &\Rightarrow \omega^s|_e = |\Lambda a| \cdot (\Xi|e|_e^{|\Lambda a|} + 1) + \Xi|\omega^s|_e^{|\Lambda b|} \\ \Xi|C \sqcup A \circ B &\Rightarrow \omega^s|_e = \Xi|\omega^s|_e^{|\Lambda a| \cdot (|\Lambda b| + 2) + |\Lambda c| + 2} \\ \Xi|(B \rightarrow C) \sqcup A \triangleright \omega^d|_e &= \sum \Xi|\omega^d|_e^{(|\Lambda b| + |\Lambda a| + 2, |\Lambda c| + |\Lambda a| + 2)} \\ \Xi|(A_1 \rightarrow B_1) \sqcup \triangleright (A_2 \rightarrow B_2) \triangleright \omega^d|_e &= \sum \Xi|\omega^d|_e^{(|\Lambda a_1| + |\Lambda a_2| + 2, |\Lambda b_1| + |\Lambda b_2| + 2)} \\ \Xi|(A_1 \sqcup \circ A_2 \triangleright \omega^s)|_e &= \Xi|\omega^s|_e^{(|\Lambda a_1| + |\Lambda a_2| + 2)} \\ \Xi|\omega^s \diamond A|_e &= \Xi|\omega^s|_e^{|\Lambda a|} \\ \Xi|\omega^d \diamond A, B|_e &= \sum \Xi|\omega^d|_e^{(|\Lambda a|, |\Lambda b|)} \\ |\Gamma|_e &= \sum_{w \in \Gamma} \|w\| |w|_e \end{aligned}$$

Fig. 23. Definition of  $|\cdot|_e$

$$|\omega^s|_\omega, |\omega^d|_\omega$$

number of judgments of  $\omega^s, \omega^d$ 

$$\begin{aligned} | \_ \leq A |_\omega &= 0 \\ | \_ \circ A \Rightarrow_\alpha \omega^s |_\omega &= |\omega^s|_\omega \\ | \_ \triangleright \omega^d |_\omega &= |\omega^d|_\omega \\ | \_ \cup A \circ B \Rightarrow \omega^s |_\omega &= |\omega^s|_\omega \\ | \_ \cup \circ A \blacktriangleright \omega^s |_\omega &= |\omega^s|_\omega \end{aligned}$$

$$\begin{aligned} | \_ \rightarrow \_ \bullet e \Rightarrow \omega^s |_\omega &= |\omega^s|_\omega + 5 \\ | \_ \rightarrow \_ \cup A \triangleright \omega^d |_\omega &= |\omega^d|_\omega \\ | \_ \rightarrow \_ \cup \triangleright (A \rightarrow B) \blacktriangleright \omega^d |_\omega &= |\omega^d|_\omega \end{aligned}$$

$$|w|_\omega$$

number of judgments of  $w$ 

$$\begin{aligned} |A \leq B|_\omega &= 0 \\ |e \leftarrow A|_\omega &= 3 \\ |e \Rightarrow \omega^s|_\omega &= |\omega^s|_\omega + 2 \\ |A \circ B \Rightarrow \omega^s|_\omega &= |\omega^s|_\omega \\ |A \triangleright \omega^d|_\omega &= |\omega^d|_\omega \\ |A \rightarrow B \bullet e \Rightarrow \omega^s|_\omega &= |\omega^s|_\omega + 5 \end{aligned}$$

$$\begin{aligned} |C \cup A \circ B \Rightarrow \omega^s|_\omega &= |\omega^s|_\omega \\ |(B \rightarrow C) \cup A \triangleright \omega^d|_\omega &= |\omega^d|_\omega \\ |(A_1 \rightarrow B_1) \cup \triangleright (A_2 \rightarrow B_2) \blacktriangleright \omega^d|_\omega &= |\omega^d|_\omega \\ |(A_1 \cup \circ A_2 \blacktriangleright \omega^s)|_\omega &= |\omega^s|_\omega \\ |\omega^s \diamond A|_\omega &= |\omega^s|_\omega + 1 \\ |\omega^d \diamond A \diamond B|_\omega &= |\omega^d|_\omega + 1 \end{aligned}$$

$$|\Gamma|_\omega$$

number of judgments in  $\Gamma$ 

$$|\Gamma|_\omega = \sum_{w \in \Gamma} |w|_\omega$$

Fig. 24. Definition of  $|\cdot|_\omega$

$|A|_{\Rightarrow}$ type size of type-application of  $A$ 

$$\begin{aligned}
|1|_{\Rightarrow} &= |\top|_{\Rightarrow} = |\perp|_{\Rightarrow} = |A \rightarrow B|_{\Rightarrow} = 0 \\
|\forall a. A|_{\Rightarrow} &= |A|_{\Rightarrow} + |A|_a + 1 \\
|A \sqcap B|_{\Rightarrow} &= |A|_{\Rightarrow} + |B|_{\Rightarrow} + 1 \\
|A \sqcup B|_{\Rightarrow} &= |A|_{\Rightarrow} + |B|_{\Rightarrow} + 1
\end{aligned}$$

 $|\omega^s|_{\Rightarrow}^n, |\omega^d|_{\Rightarrow}^n$ type size of type-application of  $\omega^s$  and  $\omega^d$ 

$$\begin{aligned}
|\_ \leq A|_{\Rightarrow}^n &= n & |\_ \rightarrow \_ \bullet e|_{\Rightarrow} &\Rightarrow \omega^s|_{\Rightarrow}^n = |\omega^s|_{\Rightarrow}^n \\
|\_ \circ A|_{\Rightarrow} &\Rightarrow \omega^s|_{\Rightarrow}^n = |\omega^s|_{\Rightarrow}^{(|A|_{\Rightarrow}+2) \cdot n} & |\_ \rightarrow \_ \cup A \triangleright \omega^d|_{\Rightarrow}^n &= |\omega^d|_{\Rightarrow}^n \\
|\_ \triangleright \omega^d|_{\Rightarrow}^n &= |\omega^d|_{\Rightarrow}^n & |\_ \rightarrow \_ \cup_b (A \rightarrow B) \blacktriangleright \omega^d|_{\Rightarrow}^n &= |\omega^d|_{\Rightarrow}^n \\
|\_ \cup A \circ B|_{\Rightarrow} &\Rightarrow \omega^s|_{\Rightarrow}^n = |\omega^s|_{\Rightarrow}^{(|A|_{\Rightarrow} \cdot (|B|_{\Rightarrow}+2) + n + 1)} \\
|\_ \cup_o A \blacktriangleright \omega^s|_{\Rightarrow}^n &= |\omega^s|_{\Rightarrow}^{(|A|_{\Rightarrow} + n + 1)}
\end{aligned}$$

 $|w|_{\Rightarrow}$ type size of type-application of  $w$ 

$$\begin{aligned}
|A \leq B|_{\Rightarrow} &= |e \leftarrow A|_{\Rightarrow} = 0 \\
|e \Rightarrow \omega^s|_{\Rightarrow} &= |\omega^s|_{\Rightarrow}^0 \\
|A \circ B|_{\Rightarrow} &\Rightarrow \omega^s|_{\Rightarrow} = |\_ \circ B|_{\Rightarrow} \Rightarrow \omega^s|_{\Rightarrow}^{(|A|_{\Rightarrow})} \\
|A \triangleright \omega^d|_{\Rightarrow} &= |\omega^d|_{\Rightarrow}^0 \\
|A \rightarrow B \bullet e|_{\Rightarrow} &\Rightarrow \omega^s|_{\Rightarrow} = |\omega^s|_{\Rightarrow}^0 \\
|C \cup A \circ B|_{\Rightarrow} &\Rightarrow \omega^s|_{\Rightarrow} = |\omega^s|_{\Rightarrow}^{(|C|_{\Rightarrow} + |A|_{\Rightarrow} \cdot (|B|_{\Rightarrow} + 2) + 1)} \\
|(B \rightarrow C) \cup A \triangleright \omega^d|_{\Rightarrow} &= |\omega^d|_{\Rightarrow}^0 \\
|(A_1 \rightarrow B_1) \cup_b (A_2 \rightarrow B_2) \blacktriangleright \omega^d|_{\Rightarrow} &= |\omega^d|_{\Rightarrow}^0 \\
|(A_1 \cup_o A_2 \blacktriangleright \omega^s)|_{\Rightarrow} &= |\omega^s|_{\Rightarrow}^{(|A_1|_{\Rightarrow} + |A_2|_{\Rightarrow} + 1)}
\end{aligned}$$

 $|\Gamma|_{\Rightarrow}$ type size of type-application of  $\Gamma$ 

$$|\Gamma|_{\Rightarrow} = \sum_{w \in \Gamma} |w|_{\Rightarrow}$$

Fig. 25. Definition of  $|\cdot|_{\Rightarrow}$

$$\boxed{|A|_{\triangleright}, |\omega^s|_{\triangleright}, |\omega^d|_{\triangleright}, |w|_{\triangleright}, |\Gamma|_{\triangleright}}$$

type size of matching of  $A$ ,  $\omega^s$ ,  $\omega^d$ ,  $w$ , and  $\Gamma$ 

$$|1|_{\triangleright} = |\top|_{\triangleright} = 0$$

$$|\perp|_{\triangleright} = |a|_{\triangleright} = 2$$

$$|A \rightarrow B|_{\triangleright} = 1$$

$$|\forall a. A|_{\triangleright} = |A|_{\triangleright} + 1$$

$$|A \sqcap B|_{\triangleright} = |A|_{\triangleright} + |B|_{\triangleright} + 1$$

$$|A \sqcup B|_{\triangleright} = |A|_{\triangleright} + |B|_{\triangleright} + 2$$

$$|\_ \triangleright \omega^d|_{\triangleright} = |\omega^d|_{\triangleright} + 1$$

$$|\omega^s|_{\triangleright} = 0 \quad \text{otherwise}$$

$$|\_ \rightarrow \_ \bullet e \Rightarrow \omega^s|_{\triangleright} = 0$$

$$|\_ \rightarrow \_ \cup A \triangleright \omega^d|_{\triangleright} = |\omega^d|_{\triangleright} + |A|_{\triangleright} + 1$$

$$|\_ \rightarrow \_ \cup (A \rightarrow B) \triangleright \omega^d|_{\triangleright} = |\omega^d|_{\triangleright} + 1$$

$$|A \triangleright \omega^d|_{\triangleright} = |A|_{\triangleright} + |\omega^d|_{\triangleright}$$

$$|(B \rightarrow C) \cup A \triangleright \omega^d|_{\triangleright} = |A|_{\triangleright} + |\omega^d|_{\triangleright} + 1$$

$$|(A_1 \rightarrow B_1) \cup (A_2 \rightarrow B_2) \triangleright \omega^d|_{\triangleright} = |\omega^d|_{\triangleright} + 1$$

$$|w|_{\triangleright} = 0 \quad \text{otherwise}$$

$$|\Gamma|_{\triangleright} = \sum_{w \in \Gamma} |w|_{\triangleright}$$

$$\boxed{\Gamma|A|_{\rightarrow}^n, \Gamma|w|_{\rightarrow}, |\Gamma|_{\rightarrow}}$$

sum of rank non-mono component of  $A$ ,  $w$ , and  $\Gamma$ 

$$\Gamma|1|_{\rightarrow}^n = 0$$

$$\Gamma|\top|_{\rightarrow}^n = \Gamma|\perp|_{\rightarrow}^n = n$$

$$\Gamma|a|_{\rightarrow}^n = 0 \quad \text{when } a \in \Gamma \vee \hat{a} \in \Gamma$$

$$\Gamma|a|_{\rightarrow}^n = n \quad \text{when } \tilde{a} \in \Gamma$$

$$\Gamma|A \rightarrow B|_{\rightarrow}^n = \Gamma|A|_{\rightarrow}^{n+1} + \Gamma|B|_{\rightarrow}^{n+1}$$

$$\Gamma|\forall a. A|_{\rightarrow}^n = \Gamma|A|_{\rightarrow}^n + n$$

$$\Gamma|A \sqcap B|_{\rightarrow}^n = \Gamma|A|_{\rightarrow}^n + \Gamma|B|_{\rightarrow}^n + n$$

$$\Gamma|A \sqcup B|_{\rightarrow}^n = \Gamma|A|_{\rightarrow}^n + \Gamma|B|_{\rightarrow}^n + n$$

$$\Gamma|A \leq B|_{\rightarrow} = \Gamma|A|_{\rightarrow} \cdot |B|_{\sqcup \cap a} + \Gamma|B|_{\rightarrow} \cdot |A|_{\sqcup \cap a}$$

$$\Gamma|w|_{\rightarrow} = 0 \quad \text{otherwise}$$

$$|\Gamma|_{\rightarrow} = \sum_{w \in \Gamma} \Gamma|w|_{\rightarrow}$$

$$\boxed{|A|_{\leq}, |w|_{\leq}, |\Gamma|_{\leq}}$$

type size in subtyping judgment of  $A$ ,  $w$ , and  $\Gamma$ 

$$|1|_{\leq} = |\top|_{\leq} = |\perp|_{\leq} = |a|_{\leq} = 1$$

$$|A \rightarrow B|_{\leq} = |A|_{\leq} + |B|_{\leq} + 1$$

$$|\forall a. A|_{\leq} = |A|_{\leq} + 2$$

$$|A \sqcap B|_{\leq} = |A|_{\leq} + |B|_{\leq} + 1$$

$$|A \sqcup B|_{\leq} = |A|_{\leq} + |B|_{\leq} + 1$$

$$|A \leq B|_{\leq} = |A|_{\leq} \cdot |B|_{\sqcup \cap} + |B|_{\leq} \cdot |A|_{\sqcup \cap}$$

$$|w|_{\leq} = 0 \quad \text{otherwise}$$

$$|\Gamma|_{\leq} = \sum_{w \in \Gamma} |w|_{\leq} + \sum_{x:A \in \Gamma} 1 + \sum_{a \in \Gamma} 1 + \sum_{\tilde{a} \in \Gamma} 1 + \sum_{\hat{a} \in \Gamma} 1$$

Fig. 26. Definition of  $|\cdot|_{\triangleright}$ ,  $|\cdot|_{\rightarrow}$  and  $|\cdot|_{\leq}$

$$|\omega^s|_{\Rightarrow \omega}, |\omega^d|_{\Rightarrow \omega}$$

$$\begin{aligned} | \_ \leq A |_{\Rightarrow \omega} &= 0 \\ | \_ \circ A \Rightarrow \omega^s |_{\Rightarrow \omega} &= |\omega^s|_{\Rightarrow \omega} + 1 \\ | \_ \triangleright \omega^d |_{\Rightarrow \omega} &= |\omega^d|_{\Rightarrow \omega} \\ | \_ \cup A \circ B \Rightarrow \omega^s |_{\Rightarrow \omega} &= |\omega^s|_{\Rightarrow \omega} + 3 \\ | \_ \cup \circ A \blacktriangleright \omega^s |_{\Rightarrow \omega} &= |\omega^s|_{\Rightarrow \omega} + 1 \end{aligned}$$

$$|w|_{\Rightarrow \omega}, |\Gamma|_{\Rightarrow \omega}$$

$$\begin{aligned} |A \leq B|_{\Rightarrow \omega} &= |e \leftarrow A|_{\Rightarrow \omega} = 0 \\ |e \Rightarrow \omega^s|_{\Rightarrow \omega} &= |\omega^s|_{\Rightarrow \omega} \\ |A \circ B \Rightarrow \omega^s|_{\Rightarrow \omega} &= |\omega^s|_{\Rightarrow \omega} + 1 \\ |A \triangleright \omega^d|_{\Rightarrow \omega} &= |\omega^d|_{\Rightarrow \omega} \\ |A \rightarrow B \bullet e \Rightarrow \omega^s|_{\Rightarrow \omega} &= |\omega^s|_{\Rightarrow \omega} \end{aligned}$$

$$|\omega^s|_{\triangleright \omega}, |\omega^d|_{\triangleright \omega}$$

$$\begin{aligned} | \_ \leq A |_{\triangleright \omega} &= 0 \\ | \_ \circ A \Rightarrow \omega^s |_{\triangleright \omega} &= |\omega^s|_{\triangleright \omega} \\ | \_ \triangleright \omega^d |_{\triangleright \omega} &= |\omega^d|_{\triangleright \omega} + 1 \\ | \_ \cup A \circ B \Rightarrow \omega^s |_{\triangleright \omega} &= |\omega^s|_{\triangleright \omega} \\ | \_ \cup \circ A \blacktriangleright \omega^s |_{\triangleright \omega} &= |\omega^s|_{\triangleright \omega} \end{aligned}$$

$$|w|_{\triangleright \omega}, |\Gamma|_{\triangleright \omega}$$

$$\begin{aligned} |A \leq B|_{\triangleright \omega} &= |e \leftarrow A|_{\triangleright \omega} = 0 \\ |e \Rightarrow \omega^s|_{\triangleright \omega} &= |\omega^s|_{\triangleright \omega} \\ |A \circ B \Rightarrow \omega^s|_{\triangleright \omega} &= |\omega^s|_{\triangleright \omega} \\ |A \triangleright \omega^d|_{\triangleright \omega} &= |\omega^d|_{\triangleright \omega} + 1 \\ |A \rightarrow B \bullet e \Rightarrow \omega^s|_{\triangleright \omega} &= |\omega^s|_{\triangleright \omega} \end{aligned}$$

type-application judgment size of  $\omega^s$  and  $\omega^d$

$$\begin{aligned} | \_ \rightarrow \_ \bullet e \Rightarrow \omega^s |_{\Rightarrow \omega} &= |\omega^s|_{\Rightarrow \omega} \\ | \_ \rightarrow \_ \cup A \triangleright \omega^d |_{\Rightarrow \omega} &= |\omega^d|_{\Rightarrow \omega} \\ | \_ \rightarrow \_ \cup \triangleright (A \rightarrow B) \blacktriangleright \omega^d |_{\Rightarrow \omega} &= |\omega^d|_{\Rightarrow \omega} \end{aligned}$$

type-application judgment size of  $w$  and  $\Gamma$

$$\begin{aligned} |C \cup A \circ B \Rightarrow \omega^s|_{\Rightarrow \omega} &= |\omega^s|_{\Rightarrow \omega} + 3 \\ |(B \rightarrow C) \cup A \triangleright \omega^d|_{\Rightarrow \omega} &= |\omega^d|_{\Rightarrow \omega} \\ |(A_1 \rightarrow B_1) \cup \triangleright (A_2 \rightarrow B_2) \blacktriangleright \omega^d|_{\Rightarrow \omega} &= |\omega^d|_{\Rightarrow \omega} \\ |(A_1 \cup \circ A_2 \blacktriangleright \omega^s)|_{\Rightarrow \omega} &= |\omega^s|_{\Rightarrow \omega} + 1 \\ |\Gamma|_{\Rightarrow \omega} &= \sum_{w \in \Gamma} |w|_{\Rightarrow \omega} \end{aligned}$$

matching judgment size of  $\omega^s$  and  $\omega^d$

$$\begin{aligned} | \_ \rightarrow \_ \bullet e \Rightarrow \omega^s |_{\triangleright \omega} &= |\omega^s|_{\triangleright \omega} \\ | \_ \rightarrow \_ \cup A \triangleright \omega^d |_{\triangleright \omega} &= |\omega^d|_{\triangleright \omega} + 3 \\ | \_ \rightarrow \_ \cup \triangleright (A \rightarrow B) \blacktriangleright \omega^d |_{\triangleright \omega} &= |\omega^d|_{\triangleright \omega} + 1 \end{aligned}$$

matching judgment size of  $w$  and  $\Gamma$

$$\begin{aligned} |C \cup A \circ B \Rightarrow \omega^s|_{\triangleright \omega} &= |\omega^s|_{\triangleright \omega} \\ |(B \rightarrow C) \cup A \triangleright \omega^d|_{\triangleright \omega} &= |\omega^d|_{\triangleright \omega} + 3 \\ |(A_1 \rightarrow B_1) \cup \triangleright (A_2 \rightarrow B_2) \blacktriangleright \omega^d|_{\triangleright \omega} &= |\omega^d|_{\triangleright \omega} + 1 \\ |(A_1 \cup \circ A_2 \blacktriangleright \omega^s)|_{\triangleright \omega} &= |\omega^s|_{\triangleright \omega} \\ |\Gamma|_{\triangleright \omega} &= \sum_{w \in \Gamma} |w|_{\triangleright \omega} \end{aligned}$$

Fig. 27. Definition of  $|\cdot|_{\Rightarrow}$  and  $|\cdot|_{\triangleright}$

## E Formalization in Coq

In this section, we discuss some details of our Coq formalization. One interesting point is that previous work on HRP using the worklist approach was done in Abella [Gacek 2008], which uses higher-order abstract syntax for representing variable binding. In Coq, we need to use a first-order representation, and we have chosen the locally nameless representation [Charguéraud 2012] for that. The choice of first-order variable binding also has an impact on the representation of continuations. The file structure of the proof and the corresponding code of the definitions and theorems listed in the paper are also provided.

### E.1 Locally Nameless Representation

Since Coq does not have native support for binding structure as Abella does, we pick the locally nameless representation as the technique to deal with them manually. Compared to De Bruijn indices, locally nameless representation is more flexible in modifying the context. This is crucial for our formalization since worklist substitution often reorders the worklist in a very complex way. We choose Ott [Sewell et al. 2007] and LNgen [Aydemir and Weirich 2010] as the toolchain to automatically generate definitions and lemmas related to locally nameless representations.

*Renaming lemma.* One special proof burden of locally nameless is the renaming lemma, i.e., a property holds by renaming a free variable to another fresh name. Though co-finite quantification reduces the need for renaming lemmas in various positions, we find two patterns where the renaming lemma is indispensable: the first being,  $P(x) \vee \neg P(x)$ ; the second being  $\exists x, P(x)$ , where  $x$  could be anything containing something that uses binders. For example,  $x$  can be worklists, works, types or expressions. The first pattern appears in the proof of decidability. The induction hypothesis generated for these two patterns are *forall*  $a, a \notin L \supset (P(x'(a)) \vee \neg P(x'(a)))$  and *forall*  $a, a \notin L \supset (\text{exists } x', P(x'(a)))$ . However, what we want is usually (*forall*  $a, a \notin L \supset P(x'(a)) \vee (\text{forall } a, a \notin L \supset \neg P(x))$ ) and *exists*  $x', (\text{forall } a, a \notin L \supset P(x'(a)))$ . Since such a distributivity for logical connective is not true for arbitrary  $P$ , we have to manually prove renaming lemmas. For example, the renaming lemma for worklist reduction, which is used in the decidability proof, is as follows:

LEMMA E.1 (WORKLIST REDUCTION RENAMING). *If*  $\Gamma, b \notin \text{fv}(\Gamma)$ , *and*  $\Gamma \longrightarrow^* \cdot$ , *then*  $[b/a]\Gamma \longrightarrow^* \cdot$ .

### E.2 File Structure of Proof

In this section, we present an overview of our formalization in Coq. Proofs of  $F_{\sqcap}^e$  and its variants can be found in the supplementary materials. The proof script for the base  $F_{\sqcap}^e$  consists of 40669 lines of Coq code, 18,262 of which are generated by Ott and LNgen. Two variants consist of fewer lines of code since we do not have decidability proof and type safety proof for them. The file structure of the proofs of base  $F_{\sqcap}^e$  is shown below, under the proof/uni directory. The proof of  $F_{\sqcap}^e$  with records is under the proof/uni\_rec directory and the proof of  $F_{\sqcap}^e$  with records and instantiable intersection/union types is under the proof/uni\_monoiu directory.

*Translation Table for the Proof Scripts.* Table 2 and 3 provides a mapping from symbolic names in the paper to the textual names in the proof script. However, thanks to the notations system in Coq, most definitions are already written with a similar syntax as in the paper.

Received 2024-07-11; accepted 2024-11-07

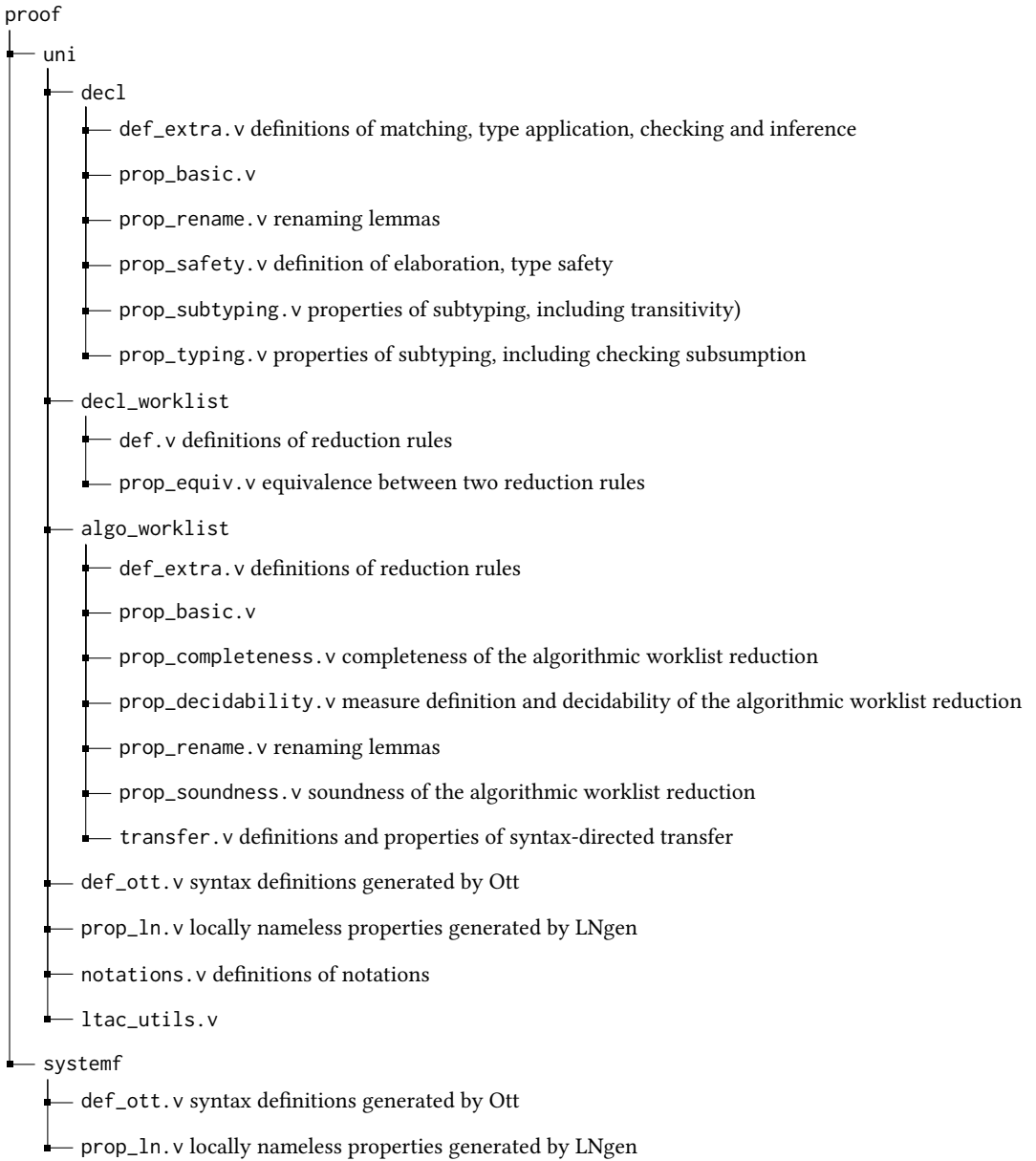




Table 2. Mapping Table from Symbolic Definitions in the Paper to the Definition Names in the Proof

Symbol	Coq name
$A$	typ
$\tau$	d_mono_typ, a_mono_typ
$A^{\neq\forall}$	d_wneq_forall, a_wneq_forall
$A^{\perp}$	bot_free
$e$	exp
$\omega^s, \omega^d$	conts, contd
$w$	work
$\Psi$	denv
$\Sigma$	aenv
$\Xi$	nenv
$\Delta$	fenv
$\Psi' \leq \Psi$	d_subenv
$\Psi \vdash A \leq B$	d_sub
$\Psi \vdash e \Leftrightarrow A$	d_chk_inf
$\Psi \vdash A \triangleright B \rightarrow C$	d_infabs
$\Psi \vdash A \circ B \Rightarrow C$	d_infapp
$\{\tau/a\}\Gamma$	aworklist_subst
$\Gamma \longrightarrow^* \cdot$	a_wl_red
$\Omega \longrightarrow^* \cdot$	d_wl_del_red
$\Omega \longrightarrow^* \cdot$	d_wl_red
$\theta$	subst_set
$\theta \vDash \Gamma \rightsquigarrow \Omega \ni \theta'$	transfer
$ \cdot _e$	exp_size, exp_size_conts/d, exp_size_work, exp_size_wl
$ \cdot _\omega$	judge_size_work, judge_size_wl
$ \cdot _{\Rightarrow}$	infapp_depth, infapp_depth_conts/d, infapp_depth_work, infapp_depth_wl
$ \cdot _{\Rightarrow\omega}$	infapp_judge_size_work, infapp_judge_size_wl
$ \cdot _{\triangleright}$	infabs_depth, infabs_depth_conts/d, infabs_depth_work, infabs_depth_wl
$ \cdot _{\triangleright\omega}$	infabs_judge_size_work, infabs_judge_size_wl
$ \cdot _{\rightarrow}$	split_depth, split_depth_work, split_depth_wl
$ \cdot _{\bar{a}}$	num_etvar_wl
$ \cdot _{\leq}$	weight, weight_work, weight_wl
$ \Psi  \vdash_{F_{X+}} e : A$	f_typing

Table 3. Mapping Table from Theorems in the Paper to the Theorem Names in the Proof

Symbol	Coq name
Theorem 2.1	d_sub_inst
Theorem 2.2	d_chk_subsumption
Theorem 3.1	d_sub_reflexivity
Theorem 3.2	d_sub_transitivity
Lemma 3.3	d_infabs_soundness, d_infabs_completeness
Lemma 3.4	d_infapp_soundness1, d_infapp_completeness
Theorem 4.1	a_wl_red_chk_soundness, a_wl_red_inf_soundness a_wl_red_chk_completeness, a_wl_red_inf_completeness
Theorem 4.2	a_wf_wl_red_decidable_thm
Theorem 5.1	d_sub_subst_stvar
Theorem 5.2	d_infabs_subsumption_same_env, d_infapp_subsumption_same_env
Theorem 5.3	d_chk_inf_subsumption
Theorem 5.4	d_chk_inf_elab_sound_f
Theorem 5.5	d_wl_red_sound, d_wl_red_complete
Theorem 5.6	a_wl_red_soundness
Theorem 5.7	a_wl_red_completeness
Lemma 5.8	aworklist_subst_transfer_same_dworklist_rev aworklist_subst_transfer_same_dworklist
Lemma 5.9	trans_typ_etvar_s_in_more_num_arrow d_sub_more_num_arrow_in_mono_typ