Disjoint Polymorphism with Intersection and Union Types

Baber Rehman*
Huawei Technologies
Hong Kong SAR, China
brehman@connect.hku.hk

Bruno C. d. S. Oliveira
The University of Hong Kong
Hong Kong SAR, China
bruno@cs.hku.hk

Abstract

Intersection and union types are advance programming features and are able to encode various classical programming constructs. The significance of intersection and union types is visible by the fact that these types are available in many modern programming languages including Scala, TypeScript and Ceylon. (Un-tagged) Union types are normally eliminated using a type-based switch construct. The branches of the switch construct may overlap thus resulting in an ambiguous semantics. Recently, a disjointness based approach so called λ_u has been proposed to deal with ambiguity in (un-tagged) union elimination. When studied with intersection types and parametric polymorphism, λ_u poses an un-intuitive ground type restriction on type variable bounds. This restriction reduces the expressiveness of the calculus. In this paper, we propose a novel disjointness algorithm based on union splittable types. The novel disjointness algorithm does not require ground type restriction on type variable bounds. Therefore, the resulting calculus is more expressive. We prove soundness and completeness of our disjointness algorithm (without parametric polymorphism) w.r.t disjointness specifications for monomorphic λ_u. All the metatheory of this paper has been formalized in Coq theorem prover.

CCS Concepts

• **Theory of computation → Type theory**.

Keywords

Intersection types, Union types, Disjointness, Polymorphism

ACM Reference Format:
Baber Rehman and Bruno C. d. S. Oliveira. 2024. Disjoint Polymorphism with Intersection and Union Types. In *Proceedings of the 26th ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '24), September 20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3678721.3686230

1 Background

This section briefly introduces λ_u [14] and the limitations in disjointness algorithm for polymorphic λ_u. Essence of λ_u lies in disjoint switches meaning that branches with overlapping types are not allowed in a type-based switch construct ensuring deterministic semantics. For example, λ_u rejects the following program[1]:

```
Bool isInteger (x : Int | Bool) = switch (x)
                                    (x:Int)      → true
                                    (y:Int|Bool) → false
```

1.1 λ_u Calculus

Syntax and subtyping. Figure 1 presents the syntax and subtyping for λ_u [14]. Types include top, bottom, integer, function, union, intersection and null types. Expressions consist of variable, integer,

$$
\begin{array}{rcl}
A, B, C & ::= & \top \mid \bot \mid \mathsf{Int} \mid A \to B \mid A \vee B \mid A \wedge B \mid \mathsf{Null} \\
e & ::= & x \mid i \mid \lambda x.e \mid e_1\ e_2 \mid \mathsf{null} \\
 & & \mid \mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} \\
v & ::= & i \mid \lambda x.e \mid \mathsf{null} \\
\Gamma & ::= & \cdot \mid \Gamma, x : A \\
A^\circ, B^\circ, C^\circ & ::= & \mathsf{Int} \mid \mathsf{Null} \mid A \to B
\end{array}
$$

$\boxed{A <: B}$ *(Subtyping)*

$$\textsc{S-TOP}\ \frac{}{A <: \top} \qquad \textsc{S-INT}\ \frac{}{\mathsf{Int} <: \mathsf{Int}} \qquad \textsc{S-BOT}\ \frac{}{\bot <: A} \qquad \textsc{S-NULL}\ \frac{}{\mathsf{Null} <: \mathsf{Null}}$$

$$\textsc{S-ARROW}\ \frac{B_1 <: A_1 \quad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2} \qquad \textsc{S-ORA}\ \frac{A <: C \quad B <: C}{A \vee B <: C} \qquad \textsc{S-ORB}\ \frac{A <: B}{A <: B \vee C}$$

$$\textsc{S-ORC}\ \frac{A <: C}{A <: B \vee C} \qquad \textsc{S-ANDA}\ \frac{A <: B \quad A <: C}{A <: B \wedge C} \qquad \textsc{S-ANDB}\ \frac{A <: C}{A \wedge B <: C} \qquad \textsc{S-ANDC}\ \frac{B <: C}{A \wedge B <: C}$$

$\boxed{\Gamma \vdash e : A}$ *(Typing)*

$$\textsc{TYP-INT}\ \frac{}{\Gamma \vdash i : \mathsf{Int}} \qquad \textsc{TYP-NULL}\ \frac{}{\Gamma \vdash \mathsf{null} : \mathsf{Null}} \qquad \textsc{TYP-VAR}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\textsc{TYP-APP}\ \frac{\Gamma \vdash e_1 : A \to B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B} \qquad \textsc{TYP-SUB}\ \frac{\Gamma \vdash e : A \quad A <: B}{\Gamma \vdash e : B}$$

$$\textsc{TYP-ABS}\ \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B} \qquad \textsc{TYP-AND}\ \frac{\Gamma \vdash e : A \quad \Gamma \vdash e : B}{\Gamma \vdash e : A \wedge B}$$

$$\textsc{TYP-SWITCH}\ \frac{\Gamma \vdash e : A \vee B \quad \Gamma, x : A \vdash e_1 : C \quad \Gamma, y : B \vdash e_2 : C \quad A * B}{\Gamma \vdash \mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} : C}$$

Figure 1: Syntax and subtyping for λ_u with intersection types.

*Also with The University of Hong Kong.

[1]Note that we use | in code and ∨ in metatheory for union types. Similarly, we use & in code and ∧ in metatheory for intersection types.

FTfJP '24, September 20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-1111-4/24/09
https://doi.org/10.1145/3678721.3686230

lambda, application, null and a type-based switch expression. Integer, null, and lambda expressions are also values. The context Γ is standard and stores variable bindings. Subtyping is also standard for a calculus with intersection and union types.

Disjointness. Interesting constituent of λ_u is the type-based disjointness ($A * B$) in a switch expression. Disjointness restricts branches with overlapping types. Therefore, the expression under scrutinee falls in a maximum of one branch. Disjointness specifications are shown in Definition 1.1. Essentially, two types are disjoint if they do not share an ordinary common subtype. Ordinary types are shown in Figure 1. Integer, function, and null types are ordinary types. Interested readers may refer to [14] for details about algorithmic disjointness. Nevertheless, disjointness algorithm is sound and complete w.r.t Definition 1.1.

Definition 1.1 ($\wedge$-Disjointness). $A * B ::= \nexists\, C^\circ,\ C^\circ <: A$ and $C^\circ <: B$.

Type system and operational semantics. The type system for λ_u is shown at the bottom of Figure 1. Rule TYP-SWITCH is an interesting rule. This rule type-checks a switch expression for eliminating union types. The last premise $A * B$ employs disjointness. Rule TYP-SWITCH type-checks a switch expression only if e has a union type $A \vee B$, first branch has type A, second branch has type B, and types A and B are disjoint. Operational semantics for λ_u is shown in Figure 2. $\lfloor v \rfloor$ is the approximate type relation and shown next:

Approximate Type $\lfloor v \rfloor$

$\lfloor i \rfloor$	=	Int
$\lfloor \lambda x.e \rfloor$	=	$\top \to \bot$
$\lfloor \text{null} \rfloor$	=	Null

1.2 λ_u with Disjoint Polymorphism

Rehman et al. [14] study λ_u with a variant of parametric polymorphism called disjoint polymorphism [1]. Polymorphic λ_u [14] poses a ground type restriction on type variable bounds[2]. Ground types constitute of all the types except type variables. This means that a type variable cannot be declared as a bound to another type variable. While this is a common approach in many polymorphic calculi [5], this approach limits the expressiveness of the calculus. For example, two type variables cannot be declared disjoint in the presence of ground type restriction. This restrains us from writing the following program:

```
Bool isFirstMatch [X * Y] (x : X | Y) = switch (x)
                                          (x:X) → true
                                          (y:Y) → false
```

Since the bound of type variable X is another type variable Y, therefore, this program will not type-check with ground type restriction on type variable bound. Any type except the type variable can be a bound of a type variable. The following program will type-check:

```
Bool isInteger [X * Int] (x : X | Int) = switch (x)
                                           (x:Int)   → true
                                           (y:X)     → false
```

[2]Note that *bound* in this context corresponds to the disjointness constraint on a type variable. The disjointness constraint restricts the possible instantiation of a type variable. For example, $[\Gamma,\ \alpha * \text{Int}]$, where α can be instantiated with all the types that are disjoint with Int such as Null.

Notice that the bound of type variable X in the program above is a base type i.e. Int.

Limitation of disjoint polymorphism in λ_u. Rehman et al. [14] study a disjointness algorithm based on Least Ordinary Subtypes (LOS) that accounts for intersection and union types. LOS is a function that computes a set of least ordinary subtypes of the input type. The disjointness algorithm simply states that two types are disjoint if set intersection of LOS of two types is an empty set. When extended with parametric polymorphism, the LOS based disjointness algorithm requires an un-intuitive ground type restriction on type variable bounds.

Our contributions. In this paper we study a variant of λ_u with disjoint polymorphism [1] and without a ground type restriction on type variable bounds. This makes our calculus expressive than the one discussed in [14]. For example, the program *isFirstMatch* type-checks in our calculus. All the metatheory has been formalized in Coq theorem prover and is available at: *https://github.com/baberrehman/FTfJP2024-artifact*.

2 Revised Disjointness Algorithm

We develop a novel disjointness algorithm for intersection and union types by exploiting union ordinary and union splittable types [9]. We study two variants of λ_u with newly developed disjointness, one without polymorphism and another with polymorphism. The first calculus establishes a connection with the calculi without polymorphism. We show that the disjointness in Section 2 is sound and complete with respect to the disjointness specifications (Definition 1.1). The second calculus revisits disjoint polymorphism in Section 3 and proposes a revised disjointness algorithm without ground types. Syntax and subtyping for λ_u with intersection types is shown in Figure 1.

2.1 Union Ordinary and Union Splittable Types

Union ordinary and union splittable types [9] play an essential role in the formulation of the novel disjointness algorithm. These types are shown at the top in Figure 3. $\top$, Int, $A \to B$ and Null are union ordinary types as shown by rules UO-TOP, UO-INT, UO-ARROW, and UO-UNIT respectively. An intersection type is union ordinary only if both of its parts are union ordinary types as shown in rule UO-AND. For example, $\text{Int} \wedge \top$ is a union ordinary type. Whereas, $(\text{Int} \vee A \to B) \wedge \top$ is not union ordinary because left part of the intersection is not union ordinary i.e $\text{Int} \vee A \to B$.

Union types are never union ordinary types. On the contrary, union types are union splittable types by rule USP-OR. Intersection types are union splittable if either of the component of the intersection is union splittable by rules USP-ANDL and USP-ANDR. A type is either union ordinary or union splittable as stated in lemma 2.1.

LEMMA 2.1 (EXCLUSIVITY OF UNION ORDINARY AND UNION SPLITTABLE TYPES). *$\forall\, A$, A is either union ordinary or union splittable and never both.*

2.2 Algorithmic Disjointness

The algorithmic disjointness based on union ordinary and union splittable types is shown in Figure 3. Rules AD-BOT, AD-INTARR,

$\boxed{e \longrightarrow e'}$ *(Operational Semantics)*

$$\textsc{step-appl}\ \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \textsc{step-appr}\ \frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'} \qquad \textsc{step-beta}\ \frac{}{(\lambda x.e)\ v \longrightarrow e[x \rightsquigarrow v]}$$

$$\textsc{step-switch}\ \frac{e \longrightarrow e'}{\mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow \mathsf{switch}\ e'\ \{(x : A) \to e_1, (y : B) \to e_2\}}$$

$$\textsc{step-switchl}\ \frac{\lfloor v \rfloor <: A}{\mathsf{switch}\ v\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow e_1[x \rightsquigarrow v]} \qquad \textsc{step-switchr}\ \frac{\lfloor v \rfloor <: B}{\mathsf{switch}\ v\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow e_2[y \rightsquigarrow v]}$$

Figure 2: Operational semantics for λ_u.

$\boxed{A^{\circledcirc}}$ *(Union Ordinary Types)*

$$\textsc{uo-top}\ \frac{}{\top^{\circledcirc}} \qquad \textsc{uo-int}\ \frac{}{\mathsf{Int}^{\circledcirc}} \qquad \textsc{uo-arrow}\ \frac{}{(A \to B)^{\circledcirc}} \qquad \textsc{uo-unit}\ \frac{}{\mathsf{Null}^{\circledcirc}} \qquad \textsc{uo-and}\ \frac{A^{\circledcirc} \quad B^{\circledcirc}}{(A \wedge B)^{\circledcirc}}$$

$\boxed{B \triangleleft A \triangleright C}$ *(Union Splittable Types)*

$$\textsc{usp-or}\ \frac{}{A \triangleleft A \vee B \triangleright B} \qquad \textsc{usp-andl}\ \frac{A_1 \triangleleft A \triangleright A_2}{A_1 \wedge B \triangleleft A \wedge B \triangleright A_2 \wedge B} \qquad \textsc{usp-andr}\ \frac{B_1 \triangleleft B \triangleright B_2}{A \wedge B_1 \triangleleft A \wedge B \triangleright A \wedge B_2}$$

$\boxed{A *_{ax} B}$ *(Disjointness Axioms)*

$$\textsc{ad-bot}\ \frac{}{\bot *_{ax} A} \qquad \textsc{ad-intarr}\ \frac{}{\mathsf{Int} *_{ax} A \to B} \qquad \textsc{ad-intnull}\ \frac{}{\mathsf{Int} *_{ax} \mathsf{Null}} \qquad \textsc{ad-nullarr}\ \frac{}{\mathsf{Null} *_{ax} A \to B}$$

$$\textsc{ad-sym}\ \frac{A *_{ax} B}{B *_{ax} A}$$

$\boxed{A *_a B}$ *(Disjointness)*

$$\textsc{ad-orll}\ \frac{A_1 \triangleleft A \triangleright A_2 \qquad A_1 *_a B \qquad A_2 *_a B}{A *_a B} \qquad \textsc{ad-axiom}\ \frac{A *_{ax} B}{A *_a B}$$

$$\textsc{ad-orrr}\ \frac{B_1 \triangleleft B \triangleright B_2 \qquad A *_a B_1 \qquad A *_a B_2}{A *_a B} \qquad \textsc{ad-andll}\ \frac{A_1 *_a B \qquad B^{\circledcirc}}{(A_1 \wedge A_2) *_a B}$$

$$\textsc{ad-andlss}\ \frac{A_2 *_a B \qquad B^{\circledcirc}}{(A_1 \wedge A_2) *_a B} \qquad \textsc{ad-andrr}\ \frac{A *_a B_1 \qquad A^{\circledcirc}}{A *_a (B_1 \wedge B_2)} \qquad \textsc{ad-andrss}\ \frac{A *_a B_2 \qquad A^{\circledcirc}}{A *_a (B_1 \wedge B_2)}$$

$$\textsc{ad-emptyl}\ \frac{A *_a B}{(A \wedge B) *_a C} \qquad \textsc{ad-emptyr}\ \frac{B *_a C}{A *_a (B \wedge C)}$$

Figure 3: Disjointness based on splittable types for λ_u.

AD-INTNULL, and AD-NULLARR are trivial disjointness axioms. The novelty of the disjointness algorithm lies in the disjointness rules for intersection and union types.

Rule AD-ORLL states that if A is union splittable into A_1 and A_2 then A is disjoint to B only if A_1 and A_2 are disjoint to B. Rule AD-ORRR is symmetric to rule AD-ORLL. Rules AD-ANDLL and AD-ANDLSS state that an intersection type $A_1 \wedge A_2$ is disjoint to another type B when B is union ordinary and either A_1 or A_2 is disjoint to B. Rules AD-ANDRR and AD-ANDRSS are symmetric to rules AD-ANDLL and AD-ANDLSS. Rules AD-EMPTYL and AD-EMPTYR state that an intersection of two disjoint types is disjoint with any other type. The intersection of two disjoint types forms an empty type or a bottom-like type, which is disjoint with any other type. The following example illustrates our novel disjointness algorithm:

- $(\mathsf{Int} \vee \mathsf{Bool}) *_a \mathsf{String}$: $\mathsf{Int} \vee \mathsf{Bool}$ is disjoint to String by rule AD-ORLL.

Essence of Union Ordinary Types. Note that the union ordinary premise in rules AD-ANDLL, AD-ANDLSS, AD-ANDRR, and AD-ANDRSS is optional. This premise only makes the rules less overlapping. It allows the application of rules AD-ANDLL, AD-ANDLSS, AD-ANDRR, and AD-ANDRSS only if one type is an intersection type and the other type is a union ordinary type. When the other type is not union ordinary type, the disjointness algorithm falls to the union rules. The algorithm then splits the other type until it becomes union ordinary and then applies either of the rules AD-ANDLL, AD-ANDLSS, AD-ANDRR, and AD-ANDRSS.

Essence of Union Splittable Types. A naive disjointness algorithm without union splittable types may potentially be incomplete. For example, $(\mathsf{Int} \vee \mathsf{Bool} \vee \mathsf{String}) \wedge (\mathsf{Bool} \vee \mathsf{String} \vee \mathsf{Char})$ and $(\mathsf{String} \vee \mathsf{Char} \vee \mathsf{Int}) \wedge (\mathsf{Char} \vee \mathsf{Int} \vee \mathsf{Bool})$ are clearly disjoint types but a naive algorithm that works on the principal of strict smaller reductions may fails to classify them as disjoint types without union splittable types. It does not matter whether we break the left intersection or the right intersection first, we cannot make these two types disjoint. Importantly the two types as a whole are disjoint. But if we drop any component from either of the intersection, the smaller types are no longer disjoint.

Union splittable types come to the rescue in such cases and solve the incompleteness problem of the disjointness algorithm. Note that union ordinary types are optional because union ordinary types just make the rules less overlapping. The disjointness algorithm stays sound and complete without union ordinary types. But union splittable types are essential. The disjointness algorithm will not be complete without union splittable types.

Soundness and completeness of disjointness. We prove that the novel disjointness algorithm is sound and complete with respect to the disjointness specifications (definition 1.1).

Lemma 2.2 (Soundness of disjointness algorithm). $\forall\ A\ B$, $A *_a B \longrightarrow A * B$.

Lemma 2.3 (Completeness of disjointness algorithm). $\forall\ A\ B$, $A * B \longrightarrow A *_a B$.

$$
\begin{array}{rcl}
A, B, C & ::= & \ldots \mid \mathsf{P} \mid \alpha \mid \forall(\alpha * A).B \\
e & ::= & \ldots \mid \mathsf{new}\ \mathsf{P} \mid e\,A \mid \Lambda(\alpha * A).e \\
v & ::= & \ldots \mid \mathsf{new}\ \mathsf{P} \mid \Lambda(\alpha * A).e \\
\Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, \alpha * A \\
\Delta & ::= & \cdot \mid \Delta, \mathsf{P} <: A \\
A^\circ, B^\circ, C^\circ & ::= & \ldots \mid \forall(\alpha * A).B \mid \mathsf{P}
\end{array}
$$

$\boxed{A^{\circledcirc}}$ *(Union Ordinary Types)*

$$\text{UO-TVAR}\ \frac{}{\alpha^{\circledcirc}} \qquad \text{UO-ALL}\ \frac{}{\forall(\alpha * A).B^{\circledcirc}} \qquad \text{UO-NOM}\ \frac{}{\mathsf{P}^{\circledcirc}}$$

Figure 4: Extended Syntax and union ordinary types for polymorphic λ_u.

2.3 Metatheory

Typing and operational semantics. Subtyping, typing and operational semantics essentially stay the same and are shown in Figure 2. This calculus preserves the standard properties of subtyping, type-safety and determinism as shown below:

Lemma 2.4 (Subtyping Reflexivity). *$A <: A$*

Lemma 2.5 (Subtyping Transitivity). *If $A <: B$ and $B <: C$ then $A <: C$*

Theorem 2.6 (Type Preservation). *If $\Gamma \vdash e : A$ and $e \longrightarrow e'$ then $\Gamma \vdash e' : A$.*

Theorem 2.7 (Progress). *If $\Gamma \vdash e : A$ then either e is a value; or e can take a step to e'.*

Theorem 2.8 (Determinism). *If $\Gamma \vdash e : A$ and $e \longrightarrow e_1$ and $e \longrightarrow e_2$ then $e_1 = e_2$.*

3 Polymorphic Disjointness

Section 2 presents a novel disjointness algorithm by exploiting union ordinary and union splittable types. We show that the disjointness algorithm is sound and complete with respect to the disjointness specifications. In this section we extend the calculus from Section 2 with disjoint polymorphism. Importantly, we show that the ground type restriction on type variable bounds is no longer needed with the novel disjointness algorithm.

Syntax, union ordinary, and union splittable types. The syntax for polymorphic λ_u is shown at the top in Figure 4. Types are extended with the nominal types P, type variables α, and disjoint quantifiers $\forall(\alpha * A).B$. The syntactic category of expressions now include a new expression (new P) to construct instances of type P. It also includes type applications $e\,A$ and type abstractions $\Lambda(\alpha * A).e$. Expressions new P and $\Lambda(\alpha * A).e$ are also values. Typing context Γ also has entries for type variables $\Gamma, \alpha * A$. A new context Δ keeps a list and subtyping bounds of nominal types.

Union ordinary types are shown in Figure 4. Union ordinary types are extended with type variables (rule UO-TVAR), disjoint quantifiers (rule UO-ALL) and nominal types (rule UO-NOM). Union splittable types stay the same as in Section 2.

3.1 Disjointness

Note that we use $*$ for algorithmic disjointness in this section for simplicity. Disjointness specifications for disjoint polymorphism is an open problem. The disjointness algorithm with polymorphism is shown in Figure 5. In addition to the disjointness rules from Figure 3, we add axioms for universal types and the nominal types. Universal types are disjoint to all the base types and so are the nominal types. Type variables are disjoint to all the subtypes of its bound as shown in rules ADPP-VARR and ADPP-VARL. For example in a context $[\Gamma, \alpha * \top]$, α is essentially disjoint to all the types because all the types are subtype of $\top$. In another context $[\Gamma, \alpha * \mathsf{Int} \vee \mathsf{Bool}]$, α is disjoint with all the subtypes of $\mathsf{Int} \vee \mathsf{Bool}$ including Int and Bool but is not disjoint to String.

$\boxed{\Delta; \Gamma \vdash A *_{ax} B}$ *(Disjointness Axioms)*

$$\text{ADPA-INTALL}\ \frac{}{\Delta; \Gamma \vdash \mathsf{Int} *_{ax} \forall(\alpha * A).B} \qquad \text{ADPA-NULLALL}\ \frac{}{\Delta; \Gamma \vdash \mathsf{Null} *_{ax} \forall(\alpha * A).B}$$

$$\text{ADPA-ARRALL}\ \frac{}{\Delta; \Gamma \vdash C \to D *_{ax} \forall(\alpha * A).B} \qquad \text{ADPA-PINT}\ \frac{}{\Delta; \Gamma \vdash \mathsf{P} *_{ax} \mathsf{Int}}$$

$$\text{ADPA-PARR}\ \frac{}{\Delta; \Gamma \vdash \mathsf{P} *_{ax} A \to B} \qquad \text{ADPA-PNULL}\ \frac{}{\Delta; \Gamma \vdash \mathsf{P} *_{ax} \mathsf{Null}} \qquad \text{ADPA-PALL}\ \frac{}{\Delta; \Gamma \vdash \mathsf{P} *_{ax} \forall(\alpha * A).B}$$

$\boxed{\Delta; \Gamma \vdash A * B}$ *(Disjointness)*

$$\text{ADPP-VARR}\ \frac{\alpha * A \in \Gamma \qquad \Delta; \Gamma \vdash B <: A}{\Delta; \Gamma \vdash B * \alpha} \qquad \text{ADPP-VARL}\ \frac{\alpha * A \in \Gamma \qquad \Delta; \Gamma \vdash B <: A}{\Delta; \Gamma \vdash \alpha * B}$$

$$\text{ADPP-NOM}\ \frac{\mathsf{P}_1 :: \Delta(\mathsf{P}_1) \cap \mathsf{P}_2 :: \Delta(\mathsf{P}_2) = \{\}}{\Delta; \Gamma \vdash \mathsf{P}_1 * \mathsf{P}_2}$$

Figure 5: Extended disjointness rules with union splittable types for polymorphic λ_u.

Note that we scrap the optional union ordinary premise from rules AD-ANDLL, AD-ANDLSS, AD-ANDRR, and AD-ANDRSS. This simplifies the metatheory with type variables. We also drop rules AD-EMPTYL and AD-EMPTYR from disjointness algorithm in Figure 5. Dropping rules AD-EMPTYL and AD-EMPTYR restricts writing some programs but all the practical programs still type-check. Generally, it does not allow writing empty intersection types in branches. For example, the following program will no longer type-check because of the empty type in the first branch i.e Int ∧ Bool.

```
Bool isInt (x : Int | Bool) = switch (x)
                                (x:Int&Bool) → true
                                (y:Int)      → true
                                (z:Bool)     → false
```

Since we cannot construct a value of type Int∧Bool in contemporary system, the first branch in the above program has no practical significance. Therefore not allowing such empty intersections does not affect the programs in practice.

Disjointness for nominal types. The disjointness rule for nominal types (rule ADPP-NOM) is interesting. It states that two nominal

types P_1 and P_2 are disjoint if the intersection of their subtypes is an empty set. Nominal subtypes ($\Delta(A)$) is a function that finds the subtypes of type A in Δ and returns a list. Note that nominal subtypes is a transitive closure and shown next:

Nominal Subtypes $\boxed{\Delta(A)}$

$$\begin{array}{lcl} \cdot(A) & = & \{\} \\ (\Delta', \mathsf{P} <: B)(A) & = & \begin{cases} \{\mathsf{P}\} \cup \Delta'(A) \textbf{ if } \mathsf{P} <: A \in \Delta \\ \Delta'(A) \textbf{ otherwise} \end{cases} \end{array}$$

Is Nominal Subtype $\boxed{A <: B \in \Delta}$

$$\begin{array}{lcl} A <: B \in \cdot & = & \text{FALSE} \\ A <: B \in (\Delta', \mathsf{P} <: C) & = & \begin{cases} \text{TRUE } \textbf{if } A == P \textit{ and } B == C \\ C <: B \in \Delta' \textbf{ if } A == P \textit{ and } B \neq C \\ A <: B \in \Delta' \textbf{ otherwise} \end{cases} \end{array}$$

For example, in a context Δ = {Person <: ⊤, Student <: Person, GradStudent <: Student, Robot <: ⊤, OptimumPrime <: Robot}:

- Person is disjoint to Robot as per rule ADPP-NOM, because the set intersection of the subtypes of Person and Robot is empty i.e {Person, Student, GradStudent} ∩ {Robot, OptimumPrime} = {}.
- Whereas, Person is not disjoint to GradStudent, because the set intersection of the subtypes of Person and GradStudent is not empty i.e {Person, Student, GradStudent} ∩ {GradStudent} = {GradStudent}.

Contravariance of disjointness. Contravariance[3] of disjointness (lemma 3.1) states that if two types A and B are disjoint, then the subtypes of A are also disjoint with B. In general subtypes of disjoint types are disjoint as well. For example if $A \to B$ is disjoint to Int ∨ Null, then $A \to B$ is disjoint to both Int and Null among other subtypes of Int ∨ Null. Similarly if a type A is disjoint to ⊤, then A is disjoint with all the types.

Lemma 3.1 (Contravariance of disjointness). *If* $\Delta; \Gamma \vdash A * B$ *and* $\Delta; \Gamma \vdash C <: A$ *then* $\Delta; \Gamma \vdash C * B$.

Expressiveness of disjointness. Our novel disjointness algorithm allows writing the programs that are not allowed in the polymorphic λ_u proposed in [14] due to the ground type restriction. In particular, our calculus allows writing programs by declaring type variables as bounds of other type variables:

```
Bool isFirstMatch [X * Y] (x : X | Y) = switch (x)
                                          (x:X) → true
                                          (y:Y) → false
```

3.2 Metatheory

Subtyping, typing, and operational semantics. Subtyping, typing and operational semantics are altered to lift the ground type restriction on type variable bounds and are shown in Figure 6. Note that subtyping, typing, and operational semantics extend all the rules from Section 2. We also show the well-formedness relation at the top in Figure 6.

Modifications in metatheory. The subtyping changes are reflected by rule POLYS-ALLDISJ in Figure 6. Note that first premise does not have ground type restriction. A_1 and A_2 can be any types. Typing changes are shown in rules PTYP-TAPDISJ and PTYP-TABSDISJ in

$\boxed{ok\ \Delta}$ *(Well-formed Nominal Context)*

$$\textsc{okp-empty}\ \frac{}{ok\ \cdot} \qquad \textsc{okp-sub}\ \frac{ok\ \Delta \quad \Delta; \Gamma \vdash \mathsf{P}_2 \quad \mathsf{P}_1 \notin dom\ \Delta}{ok\ \Delta, \mathsf{P}_1 <: \mathsf{P}_2}$$

$\boxed{\Delta; \Gamma \vdash A}$ *(Well-formed Types)*

$$\textsc{wftp-int}\ \frac{}{\Delta; \Gamma \vdash \mathsf{Int}} \qquad \textsc{wftp-tvar}\ \frac{\alpha * A \in \Gamma}{\Delta; \Gamma \vdash \alpha} \qquad \textsc{wftp-top}\ \frac{}{\Delta; \Gamma \vdash \top} \qquad \textsc{wftp-bot}\ \frac{}{\Delta; \Gamma \vdash \bot}$$

$$\textsc{wftp-arrow}\ \frac{\Delta; \Gamma \vdash A \quad \Delta; \Gamma \vdash B}{\Delta; \Gamma \vdash A \to B} \qquad \textsc{wftp-all}\ \frac{\Delta; \Gamma \vdash A \quad \Delta; \Gamma, \alpha * A \vdash B}{\Delta; \Gamma \vdash \forall(\alpha * A).B}$$

$$\textsc{wftp-or}\ \frac{\Delta; \Gamma \vdash A \quad \Delta; \Gamma \vdash B}{\Delta; \Gamma \vdash A \vee B} \qquad \textsc{wftp-and}\ \frac{\Delta; \Gamma \vdash A \quad \Delta; \Gamma \vdash B}{\Delta; \Gamma \vdash A \wedge B}$$

$$\textsc{wftp-prim}\ \frac{\mathsf{P} \in dom\ \Delta}{\Delta; \Gamma \vdash \mathsf{P}} \qquad \textsc{wftp-null}\ \frac{}{\Delta; \Gamma \vdash \mathsf{Null}}$$

$\boxed{\Delta; \Gamma \vdash A <: B}$ *(Subtyping)*

$$\textsc{polys-tvar}\ \frac{ok\ \Delta \quad \Delta; \Gamma \vdash \alpha}{\Delta; \Gamma \vdash \alpha <: \alpha} \qquad \textsc{polys-alldisj}\ \frac{\Delta; \Gamma \vdash A_1 <: A_2 \quad \Delta; \Gamma, \alpha * A_2 \vdash B_1 <: B_2}{\Delta; \Gamma \vdash \forall(\alpha * A_1).B_1 <: \forall(\alpha * A_2).B_2}$$

$$\textsc{polys-prefl}\ \frac{ok\ \Delta \quad \Delta; \Gamma \vdash \mathsf{P}}{\Delta; \Gamma \vdash \mathsf{P} <: \mathsf{P}} \qquad \textsc{polys-pin}\ \frac{ok\ \Delta \quad \Delta; \Gamma \vdash \mathsf{P}_1 \quad \mathsf{P}_2 \in \Delta(\mathsf{P}_1)}{\Delta; \Gamma \vdash \mathsf{P}_2 <: \mathsf{P}_1}$$

$\boxed{\Delta; \Gamma \vdash e : A}$ *(Typing)*

$$\textsc{ptyp-prim}\ \frac{ok\ \Delta \quad \Delta; \Gamma \vdash \mathsf{P}}{\Delta; \Gamma \vdash \mathsf{new}\ \mathsf{P} : \mathsf{P}} \qquad \textsc{ptyp-tapdisj}\ \frac{\Delta; \Gamma \vdash e : \forall(\alpha * A).C \quad \Delta; \Gamma \vdash B * A}{\Delta; \Gamma \vdash e\,B : C[\alpha \rightsquigarrow B]}$$

$$\textsc{ptyp-tabsdisj}\ \frac{\Delta; \Gamma, \alpha * A \vdash e : B}{\Delta; \Gamma \vdash \Lambda(\alpha * A).e : \forall(\alpha * A).B}$$

$\boxed{\Delta; \Gamma \vdash e \longrightarrow e'}$ *(Operational Semantics)*

$$\textsc{polystep-tappl}\ \frac{\Delta; \Gamma \vdash e \longrightarrow e'}{\Delta; \Gamma \vdash e\,B \longrightarrow e'\,B} \qquad \textsc{polystep-tappdisj}\ \frac{}{\Delta; \Gamma \vdash (\Lambda(\alpha * A).e)\,B \longrightarrow e[\alpha \rightsquigarrow B]}$$

Approximate Type $\lfloor v \rfloor_{\Delta;\Gamma}$

$$\begin{array}{lcl} \lfloor \Lambda(\alpha * A).e \rfloor_{\Delta;\Gamma} & = & \forall(\alpha * \bot).\bot \\ \lfloor \mathsf{new}\ \mathsf{P} \rfloor_{\Delta;\Gamma} & = & \mathsf{P} \end{array}$$

Figure 6: Extended subtyping, typing, and operational semantics for polymorphic λ_u.

[3] Alpuim et al. [1] proved covariance for supertypes in the context of intersection types. We call this property contravariance due to subtypes in our context of union types.

Figure 6. Similarly, changes for operational semantics are shown in Figure 6. Importantly, we no longer use syntactic category of ground types and the bound of a type variable can be any other type.

Type safety and determinism. Polymorphic λ_u with updated disjointness preserves standard properties of subtyping, type-safety and determinism.

Lemma 3.2 (Subtyping Reflexivity). $\Delta; \Gamma \vdash A <: A$

Lemma 3.3 (Subtyping Transitivity). *If* $\Delta; \Gamma \vdash A <: B$ *and* $\Delta; \Gamma \vdash B <: C$ *then* $\Delta; \Gamma \vdash A <: C$

Theorem 3.4 (Type Preservation). *If* $\Delta; \Gamma \vdash e : A$ *and* $\Delta; \Gamma \vdash e \longrightarrow e'$ *then* $\Delta; \Gamma \vdash e' : A$.

Theorem 3.5 (Progress). *If* $\Delta; \cdot \vdash e : A$ *then either e is a value; or e can take a step to* e'.

Theorem 3.6 (Determinism). *If* $\Delta; \Gamma \vdash e : A$ *and* $\Delta; \Gamma \vdash e \longrightarrow e_1$ *and* $\Delta; \Gamma \vdash e \longrightarrow e_2$ *then* $e_1 = e_2$.

Substitution, narrowing, and weakening lemmas. Type-safety proofs need type substitution (lemma 3.7), type narrowing (lemma 3.8), and type weakening (lemma 3.9) lemmas. Narrowing lemma essentially explains the relation between disjointness and subtyping. It states that it is safe to replace the bound of a type variable with a supertype of its existing bound. Weakening lemma states that if a relation is valid in a smaller context, then it stays valid in an enlarged context given that the enlarged context is well-formed. Note that we do not show corresponding subtyping and disjointness lemmas, such as subtyping substitution and disjointness substitution due to space constraints. All of these lemmas are available in the Coq formalization of this paper.

Lemma 3.7 (Typing Substitution). *If* $\Delta; \Gamma, \alpha * A_1 \vdash e : B$ *and* $\Delta; \Gamma \vdash A_2 * A_1$ *then* $\Delta; \Gamma[\alpha \rightsquigarrow A_2] \vdash e[\alpha \rightsquigarrow A_2] : B[\alpha \rightsquigarrow A_2]$

Lemma 3.8 (Typing narrowing). *If* $\Delta; \Gamma, \alpha * A_1 \vdash e : B$ *and* $\Delta; \Gamma \vdash A_1 <: A_2$ *then* $\Delta; \Gamma, \alpha * A_2 \vdash e : B$

Lemma 3.9 (Typing weakening). *If* $\Delta; \Gamma_1, \Gamma_2 \vdash e : B$ *and ok* $\Gamma_1, \Gamma_3, \Gamma_2$ *then* $\Delta; \Gamma_1, \Gamma_3, \Gamma_2 \vdash e : B$

More auxiliary lemmas. We discuss a few more interesting auxiliary lemmas in this paragraph. Lemma 3.10, lemma 3.11 and lemma 3.12 are essential in proving the metatheory. Lemma 3.10 states that if a union ordinary type $A^{\circledcirc}$ is a subtype of a union splittable type B ($B_1 \triangleleft B \triangleright B_2$), then A is subtype of either B_1 or B_2. Lemma 3.11 states disjointness symmetry. Finally, lemma 3.12 states that if A and B are disjoint types, then this is not the case that a value v checks against both A and B.

Lemma 3.10 (Subtyping inversion of union ordinary and union splittable types). *If* $\Delta; \Gamma \vdash A <: B$ *and* $B_1 \triangleleft B \triangleright B_2$ *and* $A^{\circledcirc}$ *then* $\Delta; \Gamma \vdash A <: B_1 \vee \Delta; \Gamma \vdash A <: B_2$.

Lemma 3.11 (Disjointness symmetry). *If* $\Delta; \Gamma \vdash A * B$ *then* $\Delta; \Gamma \vdash B * A$.

Lemma 3.12 (Exclusivity of Disjoint Types). *If* $\Delta; \Gamma \vdash A * B$ *then* $\nexists\, v$ *such that both* $\Delta; \Gamma \vdash v : A$ *and* $\Delta; \Gamma \vdash v : B$ *holds.*

4 Related Work

Intersection and union types have extensively been studied in the literature. We discuss the work closest to ours in this section.

Intersection types. Coppo et al. [4] and Pottinger [13] initially studied intersection types in programming languages to assign meaningful types to terms. Compagnoni and Pierce [3] studied multiple interface inheritance by exploiting intersection types. Pierce [12] studied a calculus with intersection types, union types and polymorphism. Intersection types have also been studied in the context of refinement types [7]. Refinement types increase the expressiveness of types but not the terms. The merge operator, an introduction form for the intersection types, was first introduced in Forsythe programming language by Reynolds [15]. Dunfield [6] studied merge operator in a calculus with union types.

Disjoint intersection types. Oliveira et al. [11] studied disjoint intersection types to overcome the non-deterministic behaviour of the merge operator. Alpuim et al. [1] studied disjoint intersection types with disjoint polymorphism. Recently, Huang and Oliveira [8] proposed a direct operational semantics for the merge operator. However, this line of work does not count for union types and a type-based switch construct.

Union types. Union types were introduced in programming languages by MacQueen et al. [10]. They proposed an implicit elimination rule for union types. Barbanera et al. [2] solved the type preservation problem of implicit union elimination rule by parallel reduction. Single-branch case construct for union types is proposed by Pierce [12]. Rioux et al. [16] studied merge operator together with intersection and union types. However, their merge operator is restricted to functions. Recently, Rehman et al. [14] studied disjoint switches as a deterministic elimination form for union types. The order of branches of a switch construct does not matter in their calculus due to the disjointness constraint. However, their disjointness algorithm poses an ad-hoc restriction on type variable bounds when studied with disjoint polymorphism.

5 Conclusion and Future Work

We present a type-safe and deterministic calculus with intersection types, union types and disjoint polymorphism. The determinism of the calculus is ensured by employing a notion of disjointness. Disjointness restricts overlapping branches of a switch construct. Thus scrutinee can fall in a maximum of one branch. We present a novel disjointness algorithm which naturally extends for disjoint polymorphism without ad-hoc restrictions on type variable bounds. All the metatheory has been formalized in Coq theorem prover.

There are a few future explorations of the proposed calculus. The first future direction is to study the proposed calculus with the merge operator. Disjointness for union types and the type-based switch expression is essentially dual to the disjointness for intersection types with the merge operator. Another interesting and practical future direction is to study disjoint switches with gradual typing. The challenge of studying disjointness with gradual typing is because of the unknown type.

Acknowledgments

We thank the reviewers for the insightful comments. We also thank Xuejing Huang for technical discussions.

References

[1] João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. 2017. Disjoint Polymorphism. In *European Symposium on Programming (ESOP)*.

[2] Franco Barbanera, Mariangiola Dezaniciancaglini, and Ugo Deliguoro. 1995. Intersection and union types: syntax and semantics. *Information and Computation* 119, 2 (1995), 202–230.

[3] Adriana B Compagnoni and Benjamin C Pierce. 1996. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science* 6, 5 (1996), 469–501.

[4] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 1981. Functional characters of solvable terms. *Mathematical Logic Quarterly* 27, 2-6 (1981), 45–58.

[5] 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*. 60–72.

[6] Joshua Dunfield. 2014. Elaborating intersection and union types. *Journal of Functional Programming* 24, 2-3 (2014), 133–165.

[7] Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 268–277.

[8] Xuejing Huang and Bruno C. d. S. Oliveira. 2020. A Type-Directed Operational Semantics For a Calculus with a Merge Operator. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 26:1–26:32. https://doi.org/10.4230/LIPIcs.ECOOP.2020.26

[9] Xuejing Huang and Bruno C d S Oliveira. 2021. Distributing intersection and union types with splits and duality (functional pearl). *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–24.

[10] David MacQueen, Gordon Plotkin, and Ravi Sethi. 1984. An ideal model for recursive polymorphic types. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 165–174.

[11] Bruno C. d. S. Oliveira, Zhiyuan Shi, and Joao Alpuim. 2016. Disjoint intersection types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 364–377.

[12] Benjamin C Pierce. 1991. *Programming with intersection types, union types*. Technical Report. and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University.

[13] Garrel Pottinger. 1980. A type assignment for the strongly normalizable λ-terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism* (1980), 561–577.

[14] Baber Rehman, Xuejing Huang, Ningning Xie, and Bruno C d S Oliveira. 2022. Union Types with Disjoint Switches. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[15] John C Reynolds. 1997. Design of the Programming Language F orsythe. In *ALGOL-like languages*. Springer, 173–233.

[16] 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. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 515–543.

Received 2024-06-26; accepted 2024-07-24