Determinism with Intersection and Union Types

Baber Rehman
The University of Hong Kong
brehman@cs.hku.hk

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

ABSTRACT

Calculi with disjointness received significant research interest recently. The disjointness restricts potential problematic programs. An example is the disjoint intersection types proposed for deterministic merge operator. A variant of disjointness is also implemented in the type-based switch construct in Ceylon programming language. Such a type-based switch construct provides deterministic elimination of the union types. Another variant of disjointness has recently been implemented in Scala match types. The disjointness plays an integral role in maintaining the determinism in these calculi. We propose a novel disjointness algorithm for a calculus with intersection and union types. Our disjointness algorithm naturally extends for disjoint polymorphism without any ad-hoc restrictions. Importantly, we explore the integration of merge operator and type-based switch expression in a deterministic manner based on disjointness and unambiguous upcasts.

1 INTRODUCTION

Intersection and union types are naturally able to encode various advance programming features. The merge operator ($e_1,, e_2$) [Dunfield 2014; Reynolds 1997] is an introduction form for the intersection types. It allows constructing terms of multiple (non-overlapping) types. However, the unrestricted merge operator is not deterministic. A program with unrestricted merge operator can evaluate to multiple different programs. This makes the programming with unrestricted merge operator unrealistic. Disjoint intersection types [Oliveira et al. 2016] study a restricted and deterministic form of the merge operator. The determinism is achieved with the assistance of a notion of disjointness which allows certain merges and restricts certain merges. In particular, a merge of 1,,true is allowed because the type of 1 is disjoint with the type of true i.e. Int is disjoint with Bool. Whereas, a merge of 1,,2 is not allowed because Int is not disjoint with Int.

Surprisingly, a similar problem occurs in the elimination of (untagged) union types where multiple branches of a type-based switch construct overlap. The Ceylon programming language [King 2013] implements a switch expression based on disjointness. Multiple branches of a switch expression cannot overlap in such an implementation. The disjointness in the presence of union types and type-based switch construct is formally studied by Rehman et al. [2022] in a calculus called λ_u. The calculus λ_u is further enriched with the intersection types, subtyping distributivity, and disjoint polymorphism [Alpuim et al. 2017]. Unfortunately, their extension with disjoint polymorphism results in a disjointness algorithm with ad-hoc restrictions on type variables. For example, the following program is not allowed in their calculus:

```
firstMatch [X*Int, Y*X] (x:X, y:Y){}
```

Due to the ad-hoc ground type restriction on type variables, `X * Y` is rejected by Rehman et al. [2022]. We propose a novel disjointness algorithm for intersection and union types which naturally extends for disjoint polymorphism without ad-hoc restrictions. In particular, our disjointness algorithm accepts `firstMatch` function. Additionally, we outline a research plan for the deterministic composition of intersection types, merge operator, union types, and a type-based switch construct. This study combines two independent lines of research i.e. disjoint intersection types [Oliveira et al. 2016] and disjoint switches [Rehman et al. 2022]. Such an integration turns out to be non-trivial and raises novel challenges in determinism.

2 BACKGROUND

Intersection types. Coppo et al. [1981] and Pottinger [1980] initially studied intersection types in programming languages to assign meaningful types to terms. Compagnoni and Pierce [1996] studied multiple interface inheritance by exploiting intersection types. Pierce [1991] studied a calculus with intersection types, union types and polymorphism. Intersection types have also been studied in the context of refinement types [Freeman and Pfenning 1991]. Refinement types increase the expressiveness of types. They pose a syntactic restriction on intersection types where the intersection of two types is only allowed if two types are refinements of one another. None of these works study merge operator.

Merge operator. The merge operator was first introduced in Forsythe programming language by Reynolds [1997]. Dunfield [2014] studied merge operator in a calculus with union types. Dunfield followed an elaboration semantics where merges elaborate to pairs. However, the source semantics of Dunfield's calculus is non-deterministic.

Disjoint intersection types. Oliveira et al. [2016] studied disjoint intersection types to overcome the non-deterministic behaviour of the merge operator. They proposed a disjointness constraint in the formation of the merge operator. A merge of two expressions is allowed only if their types are disjoint. In summary, non-overlapping types are disjoint types. For example, a merge of 1,,true is allowed but a merge of 1,,2 is not allowed in the presence of such a disjointness constraint. The calculi studied by Dunfield [2014] and Oliveira et al. [2016] adopt an elaboration semantics. Recently, Huang and Oliveira [2020] 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. [1984]. They proposed an implicit elimination rule for union types. Barbanera et al. [1995] solved the type preservation problem of implicit union elimination rule by parallel reduction. Single-branch case construct for union types is proposed by Pierce [1991]. Rioux et al. [2023] studied merge operator together with intersection and union types. However, their merge operator is restricted to functions. Recently, Rehman et al. [2022] studied disjoint switches as a deterministic elimination form for union types. The order of branches of a switch construct

$$\text{Type} \quad A, B, C \quad ::= \quad \top \mid \bot \mid \mathsf{Int} \mid A \to B \mid A \vee B \mid A \wedge B$$

$\boxed{A *_a B}$ *(Disjointness with intersection and union types)*

$$\frac{}{\bot *_a A}\ \textsc{ad-btml} \qquad \frac{}{A *_a \bot}\ \textsc{ad-btmr} \qquad \frac{}{\mathsf{Int} *_a A \to B}\ \textsc{ad-intl} \qquad \frac{}{A \to B *_a \mathsf{Int}}\ \textsc{ad-intr}$$

$$\frac{A_1 \lhd A \rhd A_2 \qquad A_1 *_a B \qquad A_2 *_a B}{A *_a B}\ \textsc{ad-orll} \qquad \frac{A_2 *_a B \qquad B^{\circledcirc}}{(A_1 \wedge A_2) *_a B}\ \textsc{ad-andlss}$$

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

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

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

Figure 1: Disjointness based on splittable types.

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. Also, their study does not count for the merge operator.

3 APPROACH

The novel disjointness algorithm for intersection and union types is shown in Figure 1. The algorithm is based on union ordinary and union splittable types [Huang and Oliveira 2021]. We skip the definition of union ordinary and union splittable types due to space constraints. Interested readers may refer to Huang and Oliveira [2021] for more details. In summary, any type having a union operator either at top-level or nested is called union splittable type. For example, Int $\vee$ Bool is a union splittable type (Int $\lhd$ Int $\vee$ Bool $\rhd$ Bool). On the contrary, a type is union ordinary ($A^{\circledcirc}$) if it is not union splittable.

Interesting disjointness rules. The rules AD-ORLL and AD-ORRR are of significant interest. These rules state that a union splittable type $A_2 \lhd A \rhd A_1$ is disjoint to another type B if A_1 and A_2 are disjoint to B. Union splittable types play an integral role in the completeness of disjointness algorithm. The algorithm will not be complete without union splittable types. We also prove the soundness and completeness of our disjointness algorithm with respect to the disjointness specifications proposed by Rehman et al. [2022]. Unlike λ_u [Rehman et al. 2022], our novel disjointness algorithm naturally extends for disjoint polymorphism. In particular, the extension with disjoint polymorphism does not require a so called ground type restriction on type variable bounds. The extended rules with disjoint polymorphism are shown in Figure 2. These rules trivially state that a type variable is disjoint to all the subtypes of its bound.

$\boxed{\Gamma \vdash A *_a B}$ *(Disjointness extension with polymorphism)*

$$\frac{\alpha * A \in \Gamma \qquad \Gamma \vdash B <: A}{\Gamma \vdash B *_a \alpha}\ \textsc{adp-varr} \qquad \frac{\alpha * A \in \Gamma \qquad \Gamma \vdash B <: A}{\Gamma \vdash \alpha *_a B}\ \textsc{adp-varl}$$

Figure 2: Disjointness with polymorphism.

Continuous investigation. The ongoing study further investigates the design of a deterministic calculus by integrating disjoint intersection types and disjoint switches (λ_u). So far none of the calculi study disjoint intersection types and disjoint switches together. Putting together merge operator and the disjoint switches results in novel challenges for determinism. In particular, multiple upcast paths from certain intersection types to union types contribute towards non-determinism. For example, a value of type Int $\wedge$ Bool such as 1,,true follows two paths to upcast to Int $\vee$ Bool. It results in 1 via Int and true via Bool. We plan to investigate non-determinism further as a part of ongoing work.

Restricting ambiguous upcasts. A possible solution under investigation is to restrict the ambiguous upcasts in subsumption rule. This adds an extra premise in subsumption rule, which restricts certain upcasts. Such a premise restricts the upcast of 1,,true from Int $\wedge$ Bool to Int $\vee$ Bool. Whereas, it allows upcasting from Int $\wedge$ Bool to Int or Int $\vee$ String. The updated subsumption rule is (where $A <_u B$ is the unambiguous relation):

$$\frac{e : A \qquad A <: B \qquad A <_u B}{\Gamma \vdash e : B}\ \textsc{t-sub}$$

4 CONTRIBUTIONS

The integration of intersection and union types is known to be non-trivial. Oliveira et al. [2016] study disjointness for intersection types and the merge operator. Rehman et al. [2022] study a dual notion of disjointness for union types and the type-based switch expression. They also extend the disjointness algorithm with intersection types (without merge operator). The disjointness algorithm for intersection and union types proposed by Rehman et al. [2022] depends on Least Ordinary Subtypes (LOS). Unfortunately, such an algorithm fails to naturally scale for disjoint polymorphism and results in ad-hoc restrictions. We propose a novel disjointness algorithm that naturally scales for disjoint polymorphism. Importantly, our ongoing study investigates the design of a deterministic calculus with all the aforementioned features.

In summary the contributions of this work are:

- A novel disjointness algorithm for intersection and union types
- An extension of disjointness algorithm with disjoint polymorphism
- Mechanical formalization in Coq theorem prover
- Ongoing work contributes to the the development of a novel calculus with intersection types, merge operator, union types, and a type-based switch expression. Determinism is an important challenge in this line of work

REFERENCES

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

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

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.

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

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

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.

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

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.

Gavin King. 2013. The Ceylon language specification, version 1.0.

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.

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.

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

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.

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) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 25:1–25:31. https://doi.org/10.4230/LIPIcs.ECOOP.2022.25

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

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.