Foundations and Applications of Intersection and Union Types

Xin Yu

1 Intruduction

Intersection and union types are common in many programming languages and are used to model a wide range of features. In systems that incorporate both of these dual type constructs, several fundamental questions arise:

1. How can the duality between intersection and union types be exploited to simplify the type system?
2. How do these types interact with complex features such as subtyping, especially in the presence of distributivity?
3. How should the rules for introducing and eliminating these types be restricted to ensure type safety?

This report presents recent research that addresses these questions—particularly through three simple yet powerful concepts: duotyping, splittable types, and disjointness. We then demonstrate how intersection and union types can be applied to resolve challenges related to functions with named and optional arguments.

2 Foundation

2.1 Duotyping for Simplicity Through Duality

Subtyping and supertyping, intersection and union types, top and bottom types are common dual features. Duality is worth considering when designing the language, developing the metatheory, and discovering new features, which is *Duotyping* [4] do in the context of subtyping. Compared to the traditional subtyping relation that takes two types as inputs, duotyping introduces a third parameter of the mode which can be subtyping (<:) or supertyping (:>).

Considering a system including the top type $\top$, the bottom type $\bot$, integer type Int, function types $A \to B$, intersection types $A \wedge B$ and union types $A \vee B$:

$$Types\ A, B ::= \top \mid \bot \mid Int \mid A \to B \mid A \wedge B \mid A \vee B$$

Mode and functions related to duality is defined as follows:

$$\text{Mode } \Diamond ::= \;<: \;\mid\; >:$$

$$\rceil <: \lceil \;=\; \top \qquad \overline{<:} \;=\; >: \qquad A <:_? B \;=\; A \wedge B$$

$$\rceil >: \lceil \;=\; \bot \qquad \overline{>:} \;=\; <: \qquad A >:_? B \;=\; A \vee B$$

Shorter Specifications. Definitions above connect the dual features together and allow dual rules in traditional subtyping to be compressed, hence shorter specifications. For example, rule TS-ANDB, TSP-ANDB, TS-ORB, TSP-ORB in declarative traditional subtyping can be zipped to rule GDS-LEFT and GDS-DUAL:

$$\frac{A_1 <: A}{A_1 \wedge A_2 <: A}\ \textsc{ts-andb} \qquad \frac{A <: A_1}{A <: A_1 \vee A_2}\ \textsc{ts-orb} \qquad \frac{A >: A_1}{A >: A_1 \wedge A_2}\ \textsc{tsp-andb}$$

$$\frac{A_1 >: A}{A_1 \vee A_2 >: A}\ \textsc{tsp-orb}$$

Figure 1: traditional subtyping rules for intersection and union types

$$\frac{A \Diamond C}{(A \Diamond_? B) \Diamond C}\ \textsc{gds-Left} \qquad \frac{B\ \overline{\Diamond}\ A}{A \Diamond B}\ \textsc{gds-dual}$$

Figure 2: Duotyping rules for intersection and union types

In addition to simple type systems, duotyping can scale up to support more complex and expressive systems featuring various distributivity rules among intersection, union, and arrow types [2], like BCD subtyping. For example, rule S-DISTOR and S-DISTAND can be represented by a single rule D-DISTOR:

$$\frac{}{(A_1 \vee B) \wedge (A_2 \vee B) \leq (A_1 \wedge A_2) \vee B}\ \textsc{S-distOr}$$

$$\frac{}{(A_1 \vee A_2) \wedge B \leq (A_1 \wedge B) \vee (A_2 \wedge B)}\ \textsc{S-distAnd}$$

$$\frac{}{((A_1\ \overline{\Diamond}_?\ B) \Diamond_? (A_2\ \overline{\Diamond}_?\ B)) \Diamond ((A_1 \Diamond_? A_2)\ \overline{\Diamond}_?\ B)}\ \textsc{D-distOr}$$

Figure 3: traditional subtyping and duotyping rules for distributivity between intersection and union types

Easier Proofs. The abstraction over modes in duotyping introduces a new formulation of the subtyping relation, which brings flexibility to metatheoretical proofs such as transitivity. Since $A \between B$ is equivalent to $B \overline{\between} A$, we gain an additional way to transform propositions into equivalent forms, which results in simpler proofs in some cases. Moreover, a proof over the generalized relation $A \between B$ simultaneously covers both $A <: B$ and $A :> B$, enabling unified reasoning and reducing duplication in proofs.

Sum-up: Trade-off and Benefits. Introducing advanced features into type systems often involves trade-offs. In the case of duotyping, its conciseness comes at the cost of being less intuitive to understand. However, by abstracting the subtyping relation into a parameterized mode (subtyping or supertyping), duotyping provides a symmetric and unified treatment of dual type constructs. This approach not only results in more concise specifications and implementations, but also simplifies metatheoretical proofs. As such, it lays a solid foundation for languages that support both intersection and union types, and even other dual pairs of type constructs.

2.2 Splittable Types for Decomposition of Compound Types

Huang et al. [2] proposed an algorithm for deciding subtyping (and logical entailment) in the presence of intersection types, union types and distributivity rules. The ideas of duotyping and splittable types are the key to the algorithm: duotyping reflects the duality inherent in intersection and union types, as discussed earlier, while splittable types capture the structural patterns that arise in the components of distributivity rules, enabling the algorithm to handle subtyping/duotyping relation by decomposing the compound types.

The Intuition of Split. The algorithm targets BCD subtyping [1]. From the rules of BCD subtyping, we can observe a "decompose–compose" relationship between the two types involved in the subtyping judgment. For instance, in the rule S-DistArrR, the type $(A \to B_1) \wedge (A \to B_2)$ can be seen as the result of decomposing $A \to (B_1 \wedge B_2)$ into two components—$A \to B_1$ and $A \to B_2$—and then composing them back together using $\wedge$. Since the essence of splitting lies in decomposing compound type constructs, the approach may naturally scale to type systems extended with additional compound types besides intersection and union types.

$$\frac{}{(A \to B_1) \wedge (A \to B_2) \leq A \to (B_1 \wedge B_2)}\ \textsc{S-distArrR}$$

2.3 Disjointness for Balance Between Unambiguity and Flexibility

Disjointness supports the principle of "correct by construction." For intersection types, we restrict merge operation to be disjoint [3]: the values being merged do not share a common supertype, so the resulting value can be safely upcast to a supertype. For untagged union types, we restrict `switch` to be disjoint [5]: the two branches of the `switch` do not share a common subtype, allowing the scrutinee to unambiguously select one branch and safely downcast to the corresponding type. Compared to labeled records or tagged unions, unlabeled intersection and untagged union values constrained by disjointness offer greater flexibility, as a single field is not rigidly associated with a specific label or tag while still enabling type-safe operations. And this kind of flexibility naturally leads to disjoint polymorphism [6] where type variables are constraint by disjointess.

Disjoint Intersection. Programs revolve around the principles of introduction and elimination, with type safety ensuring coherence between the two. In this context, introduction refers to merging values, while elimination refers to extracting components from a merged value. On the introduction side, the merge operation (`,,`) is restricted by disjointness. For example, the expression `1,,3.14` : `Int` $\wedge$ `Float` is disallowed if both `Int` and `Float` are subtypes of `Num`. The intuition is that a merged value may be upcast to a supertype during evaluation—such as when passed to a function—and ambiguity in such an upcast is undesirable, as it undermines type safety. So disjointness avoids conflicts at the construction of the expression, thereby ensuring unambiguous behavior during elimination.

Disjoint Union. Another application of disjointness is in type-directed switch constructs. When introducing a construct like switch $e\ \{(x : A) \to e_1, (y : B) \to e_2\}$, both exhaustiveness and unambiguity are ensured by requiring that the scrutinee e has type $A \vee B$ where A and B are disjoint. This constraint guarantees that the value of e can be safely and unambiguously downcast to exactly one of the branches during elimination, avoiding conflicts or overlap between cases. As a result, the control flow remains predictable and type-safe, even in the absence of runtime tags or explicit type annotations.

Note on the Definition of Disjointness How disjointness is defined depends heavily on the language setting. When a type system supports only intersection types or only union types, a simpler definition of disjointness suffices. However, as more advanced type constructs—such as both intersection and union types, merge operations, and type-directed control flow (e.g., switch) —are introduced, the notion of disjointness becomes more nuanced. The table 1 summarizes three definitions of disjointness used in different settings, each tailored to the capabilities of the corresponding language. As the expressiveness of the language increases, the design of disjointness must account for greater interactions between type forms and their operational behavior.

Definition	**Language Setting**			
	intersection	union	merge op	switch
($\top$-Disjointness) $A * B ::= \forall C,\ A <: C$ and $B <: C$ then C is top-like [8, 3]	✓		✓	
($\bot$-Disjointness) $A * B ::= \forall C,\ C <: A$ and $C <: B$ then C is bottom-like [5]		✓		✓
($\wedge$-Disjointness) $A * B ::= \nexists C^\circ,\ C^\circ <: A$ and $C^\circ <: B$ [6]	✓	✓	✓	

Table 1: Disjointness Definitions and Corresponding Language Settings

3 Application

3.1 Type Safety for Functions with Named and Optional Arguments

As Sun et al. [7] proposed, intersection and union types can jointly contribute to the formalization of named and optional arguments. Their work is motivated by the observation that static type analysis for functions with optional named arguments may fail to prevent runtime errors in the presence of subtyping—particularly due to subtyping rules for records. For example, a value declared to have the type `{"host": String, "port": String}` may actually have the more specific type `{"host": String, "port": String, "debug": String}`. Such a value would be statically allowed to be passed to a function expecting an argument of type `{"host": String, "port": String, "debug"?: Bool}`, but would result in a runtime type error when the field `"debug"` is interpreted inconsistently.

Since the core issue lies in the presence of "absent" optional arguments that are obscured by the declared type, the solution must handle these cases with care. Sun et al. [7] address this by designing a type-safe core language and elaborating the surface language into this core. During this elaboration, the argument list is rewritten to align with the function's parameter list. Specifically, (a) each present argument is checked to ensure its type matches the expected parameter type up to subtyping, and (b) each absent optional argument is explicitly filled with the value `null`. This guarantees that the function always receives a complete and well-typed argument list, eliminating runtime errors due to fields of unexpected types.

Beyond addressing the type safety issue, the surface language—together with its elaboration scheme—also supports several desirable features:

1. **First-class named arguments**: Named arguments can be passed to and returned from functions just like ordinary values.

2. **Commutativity**: Argument order does not matter at the call site, as the elaboration phase rewrites the argument list to align with the function's parameter list.

3. **Optionality**: Parameters marked with `?` in the source language are elaborated into union types with $Null$ in the target language, enabling safe omission of optional arguments.

4. **Dependent default values**: Default values that depend on earlier arguments are supported through nested let-bindings, allowing one binding to reference variables introduced by previous ones.

4 Conclusion

In this report, we explore the core ideas of duotyping, splittable types, and disjointness, which together form a solid foundation for languages that feature both intersection and union types. We examine how these types can be used to maintain type safety while preserving a variety of desirable language features. This line of work shows intuitive and thoughtful design of type system with dual constructs can achieve elegance, safety and expressive power.

References

[1] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *The Journal of Symbolic Logic*, 48(4):931–940, 1983.

[2] Xuejing Huang and Bruno C. d. S. Oliveira. Distributing intersection and union types with splits and duality (functional pearl). *Proc. ACM Program. Lang.*, 5(ICFP), August 2021.

[3] XUEJING HUANG, JINXU ZHAO, and BRUNO C. D. S. OLIVEIRA. Taming the merge operator. *Journal of Functional Programming*, 31:e28, 2021.

[4] Bruno C. d. S. Oliveira, Cui Shaobo, and Baber Rehman. The Duality of Subtyping. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:29, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[5] Baber Rehman, Xuejing Huang, Ningning Xie, and Bruno C. d. S. Oliveira. Union Types with Disjoint Switches. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:31, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[6] Baber Rehman and Bruno C. d. S. Oliveira. Disjoint polymorphism with intersection and union types. In *Proceedings of the 26th ACM International Workshop on Formal Techniques for Java-like Programs*, FTfJP 2024, page 23–29, New York, NY, USA, 2024. Association for Computing Machinery.

[7] Yaozhu Sun and Bruno C. d. S. Oliveira. Named arguments as intersections, optional arguments as unions. In Viktor Vafeiadis, editor, *Programming Languages and Systems*, pages 347–373, Cham, 2025. Springer Nature Switzerland.

[8] Xu Xue, Bruno C. d. S. Oliveira, and Ningning Xie. Applicative intersection types. In *Programming Languages and Systems: 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings*, page 155–174, Berlin, Heidelberg, 2022. Springer-Verlag.

A Rule Tree: A Representation of Rule Set

When reading rule sets of formalizations, I found it somewhat difficult to reason about the system on a rule-by-rule basis, as the rules are neither ordered nor categorized into distinct parts. This motivated me to explore a more structural way to represent rule sets. One simple observation is that while some premises are shared across multiple rules, no two rules share exactly the same set of premises. This suggests a tree-like representation, where internal nodes correspond to shared premises and the leaves represent individual rules.

Example: Rule Tree for Algorithmic Duotyping. The rules for algorithmic duotyping [2] are shown in Figure 4, with the corresponding rule tree illustrated in Figure 5. Nodes with only one child are abbreviated using ellipses (...), except in the case of AD-Int. In this tree, each path from the root to a leaf represents the sequence of premises leading to the conclusion of a particular rule. Note that the shape of the judgment itself—e.g., a judgment of the form A is $A_1 \to A_2$—is also treated as a kind of premise in the rule tree.

$\boxed{A \diamond_a B}$ *(Algorithmic Duotyping)*

$$\textsc{AD-int}\ \frac{}{Int \diamond_a Int} \qquad \textsc{AD-bound}\ \frac{}{A \diamond_a \rceil\diamond\lceil}$$

$$\textsc{AD-arrow}\ \frac{A_1 \to A_2{}^{<} \quad B_1 \to B_2{}^{<} \quad A_1 \overline{\diamond}_a B_1 \quad A_2 \diamond_a B_2}{A_1 \to A_2 \diamond_a B_1 \to B_2}$$

$$\textsc{AD-dual}\ \frac{A^{\diamond} \qquad B^{\diamond} \qquad B \overline{\diamond}_a A}{A \diamond_a B}$$

$$\textsc{AD-and}\ \frac{B_1 \lhd B_{\diamond} \rhd B_2 \quad A \diamond_a B_1 \quad A \diamond_a B_2}{A \diamond_a B}$$

$$\textsc{AD-andL}\ \frac{B^{\diamond} \quad A_1 \lhd A_{\diamond} \rhd A_2 \quad A_1 \diamond_a B}{A \diamond_a B} \qquad \textsc{AD-andR}\ \frac{B^{\diamond} \quad A_1 \lhd A_{\diamond} \rhd A_2 \quad A_2 \diamond_a B}{A \diamond_a B}$$

Figure 4: Algorithmic duotyping

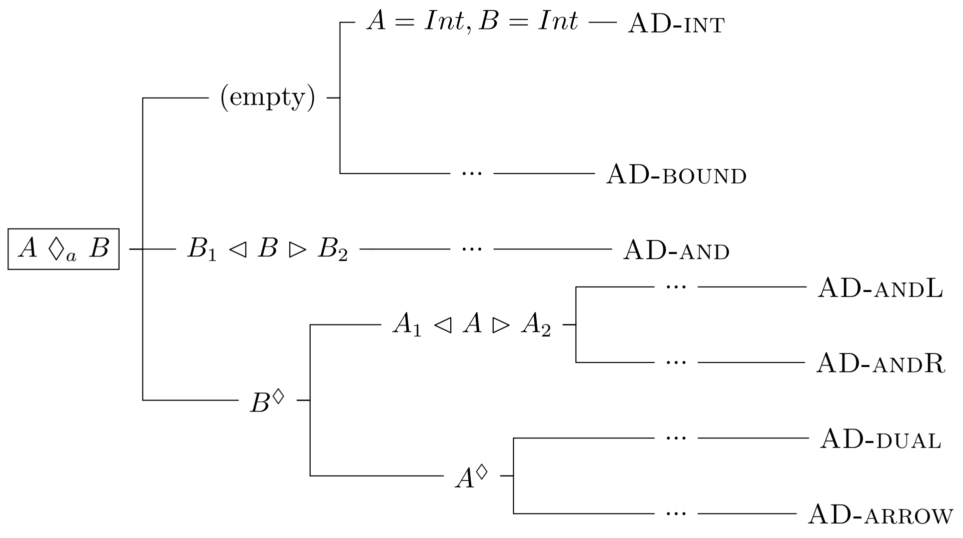

Figure 5: Rule tree for algorithmic duotyping

Advantages. This representation offers several benefits:

1. **Better Readability**: Rule trees help readers better understand a rule set by presenting the rules in a more organized and hierarchical format. Common premises are factored out as shared nodes at higher levels.

2. **Structural Insight**: The tree suggests a possible structural partitioning of the rules based on shared premises.

3. **Design Guidance**: It may serve as a useful auxiliary tool in the design and analysis of type system rules, revealing patterns or redundancies that may not be evident in a flat presentation.

4. **Implementation Alignment**: Rule trees align more closely with implementation strategies, making them potentially more practical for mechanization or coding.