Liberating Merges via Apartness and Guarded Subtyping

HAN XU, Princeton University, United States
XUEJING HUANG, The University of Hong Kong, China and IRIF, Université Paris Cité, France
BRUNO C. D. S. OLIVEIRA, The University of Hong Kong, China

The merge operator is a powerful construct in programming languages, enabling flexible composition of various components such as functions, records, or classes. Unfortunately, its application often leads to ambiguity and non-determinism, especially when dealing with overlapping types. To prevent ambiguity, approaches such as disjoint intersection types have been proposed. However, disjointness imposes strict constraints to ensure determinism, at the cost of limiting expressiveness, particularly for function overloading. This paper introduces a novel concept called *type apartness*, which relaxes the strict disjointness constraints, while maintaining type safety and determinism. Type apartness allows some overlap for overloaded functions as long as the calling contexts of those functions can be used to disambiguate upcasts in function calls. By incorporating the notion of guarded subtyping to prevent ambiguity when upcasting, our approach is the first to support *function overloading*, *return type overloading*, *extensible records*, and *nested composition* in a single calculus while preserving determinism. We formalize our calculi and proofs using Coq and prove their type soundness and determinism. Additionally, we demonstrate how type normalization and type difference provide more convenience and help resolve conflicts, enhancing the flexibility and expressiveness of the merge operator.

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

Additional Key Words and Phrases: functional languages, object oriented languages, type systems

ACM Reference Format:
Han Xu, Xuejing Huang, and Bruno C. d. S. Oliveira. 2025. Liberating Merges via Apartness and Guarded Subtyping. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 279 (October 2025), 27 pages. https://doi.org/10.1145/3763057

1 Introduction

The merge operator [Reynolds 1997] is a powerful construct that enables flexible composition of functions, records, and classes. Dunfield [2014] argued that the merge operator is valuable for language design because it allows encoding various language features within a general-purpose calculus, rather than crafting specialized calculi for each feature. The merge operator composes any two values, with extraction determined by their types. For instance, the merge $1 \,,, \mathsf{True}$ combines an integer and a boolean, giving it the intersection type $\mathsf{Int} \,\&\, \mathsf{Bool}$. Many applications of the merge operator exist, including *function overloading* [Castagna et al. 1995], *return type overloading* [Xue et al. 2022], *record concatenation* [Reynolds 1988], *nested composition* and *dynamic inheritance* [Bi and Oliveira 2018], and *composition of runtime environments* [Tan and Oliveira 2023].

Due to its power, the merge operator is hard to control. In particular, merging values with overlapping types can introduce ambiguity. For example, in the merge $1 \,,, 2$, extracting an integer is ambiguous since either 1 or 2 could be chosen. *Disjoint intersection types* [Oliveira et al. 2016] were introduced to address this issue by restricting the types of values that can be merged based on their

Authors' Contact Information: Han Xu, hx3501@princeton.edu, Princeton University, Princeton, United States; Xuejing Huang, Xuejing.Huang@irif.fr, The University of Hong Kong, China and IRIF, Université Paris Cité, France; Bruno C. d. S. Oliveira, The University of Hong Kong, China, bruno@cs.hku.hk.

ACM 2475-1421/2025/10-ART279
https://doi.org/10.1145/3763057

shared supertypes, ensuring a deterministic and well-behaved semantics [Huang et al. 2021]. The disjoint merge operator paves the way for a *compositional* style of statically typed programming where solutions to *the expression problem* [Wadler 1998] emerge naturally. A prototype programming language, called CP, illustrating this compositional programming style, has been implemented based on a core calculus with the merge operator [Zhang et al. 2021].

While disjoint intersection types prevent ambiguity, they are often too restrictive. The main issue is that disjointness requires *all possible future upcasts* to be unambiguous, limiting expressiveness, particularly for function overloading. For example, consider two functions f and g of types $\mathsf{Int} \to \mathsf{String}$ and $\mathsf{Bool} \to \mathsf{String}$. Since they share a common supertype $(\mathsf{Int} \,\&\, \mathsf{Bool}) \to \mathsf{String}$, their merge $\mathsf{f} \,,,\, \mathsf{g}$ is rejected under disjointness, as applying it to a value of type $\mathsf{Int} \,\&\, \mathsf{Bool}$ would be ambiguous. However, such strictness *unnecessarily prevents valid cases*, where a function call can be safely disambiguated by the calling context.

To overcome these limitations, we introduce *type apartness*, a relaxed alternative to disjointness. Apartness allows *overlapping* types as long as the overlapping parts remain distinguishable at the point of use. This is enough to prevent ambiguity while increasing flexibility. For example, under apartness, the merge $\mathsf{f} \,,,\, \mathsf{g}$ is accepted, and it can be applied unambiguously to an integer or a boolean. However, applying it to a merged argument, such as $1 \,,,\, \mathsf{True}$, yields an ambiguous merge of 2 strings. *Guarded subtyping* complements apartness to detect ambiguity, by ensuring that all upcasts are safe. If a function application could produce an ambiguous result, guarded subtyping rejects the application. For example, guarded subtyping would reject $(\mathsf{f} \,,,\, \mathsf{g})\ (1 \,,,\, \mathsf{True})$ while allowing $(\mathsf{f} \,,,\, \mathsf{g})\ 1$.

Apartness also improves *type difference* [Xu et al. 2023], a conflict resolution method based on type subtraction. Under apartness, type difference becomes a *total* operation, in contrast to its *partial* nature under disjointness. This new formulation of type difference ensures that conflicts between types can always be resolved, enhancing the flexibility and expressiveness of the merge operator. Additionally, we introduce the concept of *type normalization*, which systematically handles unrestricted intersection types by transforming them into a canonical form where all intersections are apart. This process eliminates redundant and conflicting type components, ensuring that the resulting types are well-formed and unambiguous. Consequently, this provides greater flexibility and convenience for programmers, allowing them to write programs with unrestricted intersection types, which is particularly useful in scenarios involving multiple inheritance.

By combining type apartness with guarded subtyping and the new formulation of type difference, our approach provides the first calculus that supports *function overloading* [Castagna et al. 1995], *return type overloading* [Xue et al. 2022], *extensible records* [Reynolds 1988], and *nested composition* [Bi and Oliveira 2018], while preserving determinism. Unlike prior work [Xue et al. 2022], which supports these features but is non-deterministic, our approach ensures a fully deterministic calculus, making it more practical and robust. Our main contributions are:

- **Type apartness**, which enables function overloading, return type overloading, extensible records, and nested composition while maintaining determinism.
- **A formalization of source and target calculi**, proving their soundness and correspondence.
- **A Coq formalization**, verifying our results.
- **A new formulation of total type difference**, improving upon previous work [Xu et al. 2023].

2 Overview

We aim to develop an *expressive* and *deterministic* calculus that supports four key features: *function overloading*, *return type overloading*, *extensible records*, and *nested composition*. This section provides background information and demonstrates how apartness and guarded subtyping enable a solution

```
class Deck { public:
    virtual void draw() { std::cout << "Draw a card." << std::endl; }
};
class Drawable { public:
    virtual void draw() { std::cout << "Create a blank canvas." << std::endl; }
};
class DrawableDeck : public Drawable, public Deck {};  // This class is accepted!
int main() {
    DrawableDeck dd;
    static_cast<Deck&>(dd).draw();  // Output: Draw a card.
    dd.draw(); //error: request for member 'draw' is ambiguous
    return 0;
}
```

Fig. 1. C++'s relaxed approach to conflict resolution.

that supports all these features while ensuring determinism. We also highlight the role of type well-formedness and type normalization in constructing sound calculi with apart merges.

2.1 C++ Method Resolution: A Source of Inspiration

Name, method, and class conflicts frequently arise in programming languages. Various approaches exist to resolve such conflicts, including forbidding redefinitions or overriding previous definitions. For instance, in many object-oriented languages, defining two methods with the same name (and signature) in the same class is (rightfully) prohibited. A related situation arises in multiple inheritance, when two parent classes define a method with the same name. A common approach is to reject such programs entirely, which avoids ambiguity but can be overly restrictive. Conservative strategies for conflict resolution limit the expressiveness of a language by disallowing many useful programs. To mitigate this, some languages take a more flexible approach by *delaying* conflict resolution until a method or value is used.

The relaxed conflict resolution approach of C++. A well-known example of delayed conflict resolution is found in C++'s multiple inheritance model. To illustrate, we adapt an example from Wang et al. [2018]. In Fig. 1, we develop two components `Deck` and `Drawable` in one system. `Deck` implements a deck of cards and defines a method `draw` for drawing a card from the deck. `Drawable` is an interface for graphics and also implements a method called `draw` for visual display. We happen to encounter a name conflict when we implement the derived class `DrawableDeck`. Here C++ adopts a delayed approach to conflict resolution. Instead of rejecting `DrawableDeck` at the point of definition, it defers ambiguity detection to the method's use site. If a method call is unambiguous, it proceeds as usual. However, calling `dd.draw()` is ambiguous because both `Deck::draw` and `Drawable::draw` are valid candidates, causing the compiler to reject the call. Nonetheless, ambiguity can be resolved by explicitly upcasting `dd` to a parent class, as shown in `static_cast<Deck&>(dd).draw()`. Such a conflict resolution strategy not only accepts a broader range of safe programs, but also allows users to use classes and methods polymorphically. Inspired by this approach, we adapt similar ideas to calculi based on intersection types and the merge operator [Dunfield 2014; Oliveira et al. 2016].

2.2 Background

The merge operator, coercive subtyping and intersection types. The merge operator [Dunfield 2014; Reynolds 1988] generalizes record concatenation to concatenation of arbitrary terms. It composes terms and allows implicit elimination based on types. For example, the following merge $1 ,, \mathsf{True} ,, \mathsf{not}$ (not is a function representing the negation on booleans) is well-typed. The type of this merge is the intersection type $\mathsf{Int} \,\&\, \mathsf{Bool} \,\&\, (\mathsf{Bool} \to \mathsf{Bool})$.

Table 1. Four applications of the merge operator with examples.

Features	Examples	Previous Works
Function overloading	showInt ,, showBool	Castagna et al. [1995]; Dunfield [2014]; Xue et al. [2022]
Return type overloading	readInt ,, readBool	Marntirosian et al. [2020]; Xue et al. [2022]
Nested composition	$(\{l = \text{"hello"}\} \,,,\, \{l = 1\}).l$	Reynolds [1988]; Xue et al. [2022]; Zhang et al. [2021]
Extensible records	$\{l_1 = \text{"hello"}\},,\{l_2 = \text{"world"}\}$	Bi et al. [2018]; Huang et al. [2021]; Xue et al. [2022]

Calculi with a merge operator use a *coercive* interpretation [Luo 1999; Reynolds 1991] of subtyping, instead of the traditional subsumptive interpretation of subtyping. In the coercive view, each subtyping judgment denotes an *implicit conversion* function. When a value is used in a context that expects a supertype, it is coerced by the corresponding function. Typically (and in our setting) the conversion function is *injective*, which means that information can be lost when converting a value to another value of a supertype. For example, in our setting the maximal type $\top$ (Top) has only one canonical value; every type is a subtype of $\top$, and every expression is converted into the top value when $\top$ is required. This is unlike subsumptive subtyping, which reflects the intuition that types correspond to sets of values, and the values of a subtype and its supertype are in a subset relation. This implies that subsumptive subtyping does not have runtime effects, and leads to the safe substitution principle [Liskov and Wing 1994]. Reynolds referred to the two interpretations as intrinsic semantics and extrinsic semantics of subtyping [Reynolds 1998]. Coercive subtyping is considered to match the intrinsic view of types because the meaning of an expression depends on how it is typed. With subsumptive subtyping, a value can be assigned multiple types.

Coherence is a key property in coercive subtyping: while multiple derivations exist for one subtyping conclusion, they are supposed to produce equivalent coercions. This is important for the language semantics to be independent from how a type derivation for an expression is constructed. Proving coherence is tricky, and intersection types make it harder. Consider types A and B that share a supertype C: for any subtype of $A \,\&\, B$, there are at least two paths that lead to C, one through A and the other through B, which may lead to possible ambiguity. Intersection types are also studied under the subsumptive interpretation with a set-theoretic model [Frisch et al. 2008]. In this model, there is no term of the type Int & Bool because Int and Bool do not share any values. But with the merge operator, we can construct terms such as 1 ,, True explicitly. Coercions arising from upcasts may have a runtime effect. In addition, any coercion to the $\top$ type is safe as $\top$ represents a singleton value. For example, the merge operator with disjoint intersection types [Huang et al. 2021; Oliveira et al. 2016] adopts the following semantics:

$$(1\,,,\,\mathsf{True}) : \mathsf{Int} \hookrightarrow 1 \qquad (1\,,,\,\mathsf{True}) : \top \hookrightarrow \top \qquad (\lambda^{\mathsf{Int}\to\mathsf{Int}} x.\, x)\,(1\,,,\,\mathsf{True}) \hookrightarrow 1$$

Upcasts have a runtime effect as they drop components from the merge. To be clear in the following discussions, we say that two coercions are equivalent when they reach the same value in the end, even if the sequence of evaluation steps dropping components may be different. We say that there is a unique coercion from one type to another when all possible coercions are equivalent.

Applications of merges. The merge operator has several applications. We summarize four applications, identified by Xue et al. [2022], in Table 1. The last entry of the table identifies works in the literature that build on such applications. We briefly explain these applications next.

- *Function overloading.* Function overloading is one kind of polymorphism, which allows choosing functions or methods according to the type or argument being applied to. For example, function overloading can choose `showInt` or `showBool` according to the argument that is applied.

$$(\mathsf{showInt} \,,,\, \mathsf{showBool})\ 1 \hookrightarrow \text{"1"}$$

$$(\mathsf{showInt},,\mathsf{showBool})\ \mathsf{True}\ \hookrightarrow \text{"True"}$$

- *Return type overloading.* Return type overloading is a feature that enables choosing an implementation from a merge depending on the context involved. A classic example is the read function in Haskell. Like show, we can define a version of read with the merge operator as

$$\mathsf{read} : (\mathsf{String} \to \mathsf{Int})\ \&\ (\mathsf{String} \to \mathsf{Bool}) = \mathsf{readInt}\,,,\,\mathsf{readBool}$$

  It is possible to disambiguate calls to read from the use context. For example, $(\mathsf{read}\,\text{"1"}) + 1$ should select readInt, since an integer is expected from the use of read.

- *Nested composition.* Nested composition reflects the distributivity behaviour of intersection types at the term level. Merges with nested composition were first proposed by Bi et al. [2018], allowing distributive extraction of nested terms such as

$$(\{l = \text{"hello"}\},,\{l = 1\}).l \hookrightarrow \text{"hello"},,1$$

  Nested composition serves as a key feature in *Compositional Programming* [Zhang et al. 2021]. Techniques based on nested composition help solving modularity problem such as the *Expression Problem* [Wadler 1998] and also enable forms of *family polymorphism* [Ernst 2001].

- *Extensible records.* Extensible records were among the first applications of merges [Reynolds 1988]. The key observation is that multi-field records can be simply encoded as merges of single-field records, such as:

$$\{l_1 = \text{"hello"}, l_2 = \text{"world"}\} \triangleq \{l_1 = \text{"hello"}\}\,,,\,\{l_2 = \text{"world"}\}$$

  The merge of single-field records should have the same behaviour as a multi-field record. That is to say, the merge should project the associated content of the single field that is designated.

$$(\{l_1 = \text{"hello"}\}\,,,\,\{l_2 = \text{"world"}\}).l_1 \hookrightarrow \text{"hello"}$$

Problem: Ambiguity of merges. Though the merge operator is powerful and expressive, without restrictions it easily leads to ambiguity. In Dunfield [2014]'s calculus, we can merge two arbitrary terms e_1 and e_2 together. Then a non-deterministic semantics of merge operator $e_1\,,,\,e_2$ is adopted where $e_1\,,,\,e_2$ can be evaluated to either e_1 and e_2. Thus, for example, we can have the following non-deterministic evaluation for the same expression:

$$1 + (1\,,,\,2)\ \hookrightarrow\ 2 \qquad 1 + (1\,,,\,2)\ \hookrightarrow\ 3$$

Note that both $(1\,,,\,2)\ \hookrightarrow\ 1$ and $(1\,,,\,2)\ \hookrightarrow\ 2$ are type-safe here. However, such cases mean that reduction is ambiguous, as multiple results are possible.

Disjoint intersection types. To address the problem of ambiguous merges Oliveira et al. [2016] proposed disjoint intersection types. The idea is that only merges of expressions with disjoint types are allowed. The design philosophy of disjoint intersection types is that a merge should be able to be used in *any context* that expects a supertype of its type. A typical example of a context is being wrapped with a type annotation. Consider f and g as two functions. For $(\mathsf{f}\,,,\,\mathsf{g}) : \top \to \mathsf{Bool}$ to exhibit unambiguous behavior, it needs to select either f or g. If their types are *disjoint*, they cannot share a common supertype like $\top \to \mathsf{Bool}$, meaning that only one of them can be used to determine the behavior. Therefore casting a merge to any supertype is acceptable. This idea leads to the definition of disjointness in the line of work on disjoint intersection types [Oliveira et al. 2016]:

Definition 2.1 (Disjointness Specification). $A *_d B \triangleq \forall C, \text{if } A \leq C \wedge B \leq C, \text{then } \top \leq C.$

If the only common supertype between two types is the top type (and types that are equivalent to top), then the only value that can have both types is the canonical top value (i.e. in the coercive interpretation the top type is interpreted as a unit type). Other values can either have type A or B but not both types at once. With disjointness, we can unambiguously extract each (non-$\top$) component under different type contexts.

$$(1\,,,\,\mathsf{True}\,,,\,\mathsf{not}) : \mathsf{Int} \hookrightarrow 1 \qquad (1\,,,\,\mathsf{True}\,,,\,\mathsf{not}) : \mathsf{Bool} \hookrightarrow \mathsf{True}$$

$$(1\,,,\,\mathsf{True}\,,,\,\mathsf{not}) : \mathsf{Bool} \to \mathsf{Bool} \hookrightarrow \mathsf{not}$$

Next we show a few more examples to illustrate disjointness on different types.

$A = \mathsf{Int}$	$B = \mathsf{Bool}$	Disjoint (no common non-top supertype)
$A = \mathsf{Int} \,\&\, \mathsf{String}$	$B = \mathsf{Bool} \,\&\, \mathsf{String}$	Common supertype: String
$A = \mathsf{Int} \to \mathsf{String}$	$B = \mathsf{Bool} \to \mathsf{String}$	Common supertype: $\mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{String}$

Only in the first example the types Int and Bool are disjoint with each other, as their only common supertype is $\top$. But the subsequent two examples are not as we can find a common supertype for them. To guarantee that upcasting is *always* valid and type-safe, semantically different components in merges cannot share any supertype, except for types that are equivalent to $\top$.

Definition 2.2 (Type Equivalence). $A =_s B \triangleq A \leq B \wedge B \leq A.$

Limitations of disjointness. While disjointness is a valuable approach, it comes with limitations. For instance, disjointness prevents conventional function overloading, such as:

$$\mathsf{show} = (\mathsf{showInt} : \mathsf{Int} \to \mathsf{String})\,,,\,(\mathsf{showBool} : \mathsf{Bool} \to \mathsf{String})$$

Here, this merge is not disjoint because the return type is the same. Thus, we can find a common supertype, such as $\mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{String}$, which would make extracting one of the functions ambiguous (both functions could be extracted). A limitation of disjointness is that it tries to ensure that *all* future casts on the merge are unambiguous. While this is a nice property to have, in some cases, like the above, it is too limiting. For example, we can use the merge above without ambiguity as:

$$\mathsf{show}\ 1 \hookrightarrow \text{"1"} \qquad \mathsf{show}\ \mathsf{True} \hookrightarrow \text{"True"}$$

The argument 1 and True can uniquely determine the function we want to apply is showInt or showBool. Ambiguity should be triggered in cases such as:

$$\mathsf{show}\ (1\,,,\,\mathsf{True}) \hookrightarrow \text{"1"} \qquad \mathsf{show}\ (1\,,,\,\mathsf{True}) \hookrightarrow \text{"True"} \qquad \mathsf{show}\ (1\,,,\,\mathsf{True}) \hookrightarrow \text{"1"}\,,,\,\text{"True"}$$

Here $1\,,,\,\mathsf{True}$ can be an argument to both showInt and showBool. Disjointness rejects many forms of overloaded functions because of this issue, and calculi with disjoint intersection types cannot fully support overloading. More generally, there is currently no deterministic calculus with merges that supports all four features in Table 1.

2.3 Apartness: Relaxing Disjointness

To admit a wider range of useful programs, and support all four features in Table 1, we introduce apartness. Apartness replaces disjointness and delays the ambiguity check in some cases. Fig. 2 graphically contrasts apartness and disjointness. Disjointness accepts no ambiguity at all, while apartness allows some levels of ambiguity like $\mathsf{showInt}\,,,\,\mathsf{showBool}$, even if there is a common supertype $\mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{String}$. Though apartness is more relaxed, it does not allow ambiguity where one type completely shadows (i.e., it is a subtype of) the other. The term shadowing denotes that a value of subtype A can always be used in place of a value of supertype B under the coercive subtyping view described in Section 2.2. When type A shadows B it is impossible to have a context that extracts the term of type B without matching the term of type A.

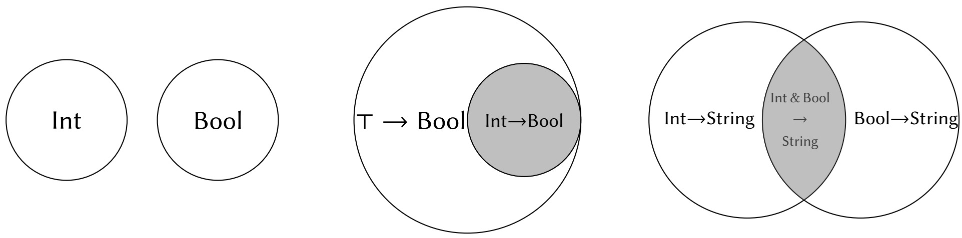

Fig. 2. Cases: A disjoint with B, A (completely) shadows B, and A apart (or partly overlaps) with B.

For instance, the merge $1\,,,\,2\,,,\,\mathsf{True}$ is ambiguous when it is used as a value of type Int and there is no supertype that can be used to remove ambiguity for extracting an integer. Thus, in this case, apartness (as well as disjointness) rejects the merge.

Guarded subtyping. By shifting from disjointness to apartness, we gain the ability to support function overloading. In such cases, the ambiguous casts can still be rejected later by our type system, through the *guarded subtyping* relation. From a coercive point of view, the subtyping relation denotes the existence of a coercion, while guarded subtyping denotes the existence of a *unique* coercion. Thus, both $(\mathsf{showInt}\,,,\,\mathsf{showBool}) : \mathsf{Int}\,\&\,\mathsf{Bool} \to \mathsf{String}$ and $(\mathsf{showInt}\,,,\,\mathsf{showBool})(1\,,,\,\mathsf{True})$ are ill-typed. Apartness accepts merges if there are casts that allow us to extract *every piece of information* in the merge. For the show example, we can find casts that can extract both functions:

$$\mathsf{show} : \mathsf{Int} \to \mathsf{String}\ (\text{extracts } \mathsf{showInt})$$
$$\mathsf{show} : \mathsf{Bool} \to \mathsf{String}\ (\text{extracts } \mathsf{showBool})$$

So the show merge should be accepted. However, if we have:

$$\mathsf{badShow1} = (\mathsf{showInt1} : \mathsf{Int} \to \mathsf{String})\,,,\,(\mathsf{showInt2} : \mathsf{Int} \to \mathsf{String})$$
$$\mathsf{badShow2} = (\mathsf{showInt1} : \mathsf{Int} \to \mathsf{String})\,,,\,(\mathsf{showInt2} : (\mathsf{Int}\,\&\,\mathsf{String}) \to \mathsf{String})$$

we should reject those merges. In the first case we cannot extract showInt1 or showInt2 without ambiguity. In the second case, we can extract showInt1 without ambiguity using $\mathsf{badShow2} : \mathsf{Int} \to \mathsf{String}$. However, we cannot extract showInt2 without ambiguity, since $\mathsf{badShow2} : (\mathsf{Int}\,\&\,\mathsf{String}) \to \mathsf{String}$ is ambiguous and can match both showInt1 and showInt2. Apartness only guarantees that we can choose components from a merge depending on the context. In contrast, disjointness guarantees that we can choose a component regardless of the context. Nevertheless, apartness (with guarded subtyping) is still able to achieve determinism, as we shall see later in Section 2.4.

Encoding records via overloading. An advantage of apartness is support for a more lightweight encoding of records via *first-class labels* [Leijen 2004] and function overloading. This encoding of records was first proposed by Castagna et al. [1995]. In this encoding a record type $\{l : A\}$ is viewed as a function type $\mathsf{Sig}\ l \to A$ from a singleton type to the value. Such feature is helpful to simplify a calculus with the merge operator. The notable point is that such simplification is not possible with disjoint intersection types because merging $\mathsf{Sig}\ l_1 \to \mathsf{Int}$ with $\mathsf{Sig}\ l_2 \to \mathsf{Int}$ will be illegal since the functions have the same return type, like in the *show* example. Thus calculi with disjointness have to use the primitive record encoding as $\{l_1 : \mathsf{Int}\}$ and $\{l_2 : \mathsf{Int}\}$ to model multi-field records. With the weaker notion of apartness, we can simplify the record encoding inside the whole system, and model records as functions where the input type is the label to be projected.

2.4 Technical Overview

In this section, we will give a more technical overview towards the key design and mechanisms for apartness, guarded subtyping as well as the features in source and target calculi.

Apartness specification. For apartness, the specification becomes harder, compared to disjointness. We need to specify that if type A is apart with type B, then every *type component* inside A and B should not shadow or be a subtype of the other. We define a (minimal) type component through the notion of a *minimal ordinary type* MinOrd B A, which states that type B is a type component (i.e., supertype and being non-intersection) of A. Here minimal is in terms of subtyping relations. We provide an intuition of MinOrd B A with some simple examples first:

$A =$	$B =$	MinOrd B A
Bool	$\top$	✗ as $\top$ is not a minimal component (Bool component subsumes it)
Bool & String	String	✓
Bool & String	Bool & String	✗ as B is an intersection type

For type Bool, the only type component is Bool itself, while for type Bool & String, a type component can be either Bool or String. Note that an intersection type can never be a minimal type component. Next we show some more complex examples, where A and B themselves contain overlapping or shadowing.

$A =$	$B =$	MinOrd B A
(Int → String) & (Bool → String)	Int → String	✓
(Int → String) & (Bool → String)	(Int & Bool) → String	✗
(Int → String) & ((Int & Bool) → String)	Int → String	✓

For (Int → String) & (Bool → String), the type component can be either Int → String or Bool → String. But the supertype Int & Bool → String cannot be a type component as it completely shadows (is the supertype) of Int → String. The type (Int → String) & ((Int & Bool) → String), is equivalent to the type Int → String in terms of the subtyping relation. Thus the type component can only be Int → String. We can now give a formal definition of MinOrd:

Definition 2.3 (Minord). MinOrd $B\ A \ \triangleq B^{\circ} \wedge A \leq B \wedge (\forall\ C, \text{if } C^{\circ} \wedge A \leq C \wedge C \leq B, \text{then } B \leq C)$.

Here B° denotes an *ordinary* type (i.e. B cannot be an intersection type), and $A \leq B$ restricts B to be a supertype of A. The condition stating that if$A \leq C \wedge C^{\circ} \wedge C \leq B$, then $B \leq C$, suggests that B is a minimal type component of A as any other type that satisfies the same requirement should be equivalent to type B. With *MinOrd* we can give the specification of apartness:

Definition 2.4 (Apartness Specification). $A *_s B \ \triangleq \forall C_1\ C_2$, if MinOrd C_1 A $\wedge$ MinOrd C_2 B, then either $C_1 = \top$ or $C_2 = \top$ or $(\neg(C_1 \leq C_2) \wedge \neg(C_2 \leq C_1))$.

Such specification checks that all the type components C_1 in A and C_2 in B do not contain any shadowing $(\neg(C_1 \leq C_2) \wedge \neg(C_2 \leq C_1))$, unless one of them is $\top$.

Guarded subtyping and lack of transitivity. Since apartness is guaranteed through merges, and it delays the check of ambiguity in some cases, we need another relation that performs a final safety check to ensure no ambiguity. This check is done via the guarded subtyping relation $A \lesssim B$. The guarded subtyping relation $A \lesssim B$ is a restricted version of subtyping that ensures that a coercion from type A to type B is *unique*.

Unlike the usual subtyping relation, the guarded subtyping relation $A \lesssim B$ is not transitive. This means that even if $A \lesssim B$ and $B \lesssim C$, it does not necessarily follow that $A \lesssim C$. For example:

$$(\mathsf{Int} \to \mathsf{String}) \,\&\, (\mathsf{Bool} \to \mathsf{String}) \lesssim (\mathsf{Int} \to \mathsf{String}) \qquad (\mathsf{Int} \to \mathsf{String}) \lesssim (\mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{String})$$

$$\neg((\mathsf{Int} \to \mathsf{String}) \,\&\, (\mathsf{Bool} \to \mathsf{String}) \lesssim \mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{String})$$

$(\mathsf{Int} \to \mathsf{String}) \,\&\, (\mathsf{Bool} \to \mathsf{String})$ is a guarded subtype of $\mathsf{Int} \to \mathsf{String}$, and $\mathsf{Int} \to \mathsf{String}$ is a guarded subtype of $\mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{String}$. However, $(\mathsf{Int} \to \mathsf{String}) \,\&\, (\mathsf{Bool} \to \mathsf{String})$ is not a guarded subtype of $\mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{String}$ because both components in the intersection are subtypes of $\mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{String}$. This demonstrates that directly replacing a type annotation with its guarded supertype can lead to ill-typed programs.

Consequently, more type information needs to be preserved at runtime to ensure type correctness. We address this by allowing type annotations to be accumulated within function-like values. By keeping track of intermediate types, we ensure that a unique path exists for future well-typed function application. To make this concrete, suppose we have a function annotated as $\lambda^{A \to A'} x.\, e : (B \to B') : (C \to C')$, even though the function itself is uniquely determined (there is no ambiguity that $\lambda^{A \to A'} x.\, e$ implements $C \to C'$ and we do not need to choose from merges of functions), we cannot drop $B \to B'$ because B maintains the path for converting the function argument from C to A, and B' is essential for converting the function application result as well.

Transitivity, determinism, and flexibility. In sufficiently rich type systems with merges, achieving transitivity, determinism, and flexibility simultaneously often proves impossible. Prior work illustrates this tension through different design trade-offs: Dunfield [2014]; Xue et al. [2022]'s calculi preserve flexibility and transitivity but sacrifice determinism; calculi based on disjoint intersection types [Oliveira et al. 2016] maintain transitivity and determinism by sacrificing flexibility. Our work takes a different approach by sacrificing transitivity to preserve determinism and flexibility. There is an analogous issue in gradual type systems, in calculi like the blame calculus [Wadler and Findler 2009]. Gradual typing systems use a consistency relation ($\sim$) that checks the compatibility of types by relaxing equality, accepting an unknown type ($\star$) in type components. However, this relation is inherently non-transitive. For example:

$$Int \sim \star \text{ is true, } \star \sim Bool \text{ is true, but } Int \sim Bool \text{ is false.}$$

Gradual typing systems also employ a consistent subtyping relation ($\lesssim$) that combines subtyping with type consistency. While $\lesssim$ supports flexibility by allowing some loss of information, it too is non-transitive. For instance:

$$Int \lesssim \star \text{ is true, } \star \lesssim Bool \text{ is true, but } Int \lesssim Bool \text{ is false.}$$

This trade-off between flexibility and transitivity is discussed by Siek and Taha [2007]. By dropping transitivity, we align with these established principles, ensuring determinism and flexibility without sacrificing type safety.

Type well-formedness. To achieve a calculus with the desirable properties of determinism and flexibility, we impose well-formedness constraints on types. These constraints ensure that types are constructed in a manner that avoids ambiguities, since all types in intersections must be apart. Well-formedness disallows unrestricted intersections such as $\mathsf{Int} \,\&\, \mathsf{Int}$ because they can lead to non-deterministic behavior when merged values are projected in different contexts. Moreover the use of guarded subtyping, instead of subtyping, prevents any remaining ambiguity arising from types that are apart, but overlapping. The calculus presented in Sec. 4 adopts this design with type well-formedness and is proven to be both type safe and deterministic.

Type normalization and unrestricted intersections. While well-formedness constraints are a simple way to ensure determinism, there are compelling reasons to allow unrestricted intersection types in a language with merges. Unrestricted intersection types provide greater convenience for programmers by enabling more expressive and flexible type constructions. For instance, in a language with unrestricted intersections we can write $1 : \mathsf{Int} \,\&\, \mathsf{Int}$, but there is still an apartness check on merges, so the terms $(1\,,,\,2) : \mathsf{Int} \,\&\, \mathsf{Int}$ or $(1\,,,\,1) : \mathsf{Int} \,\&\, \mathsf{Int}$ would be forbidden. Unrestricted intersection

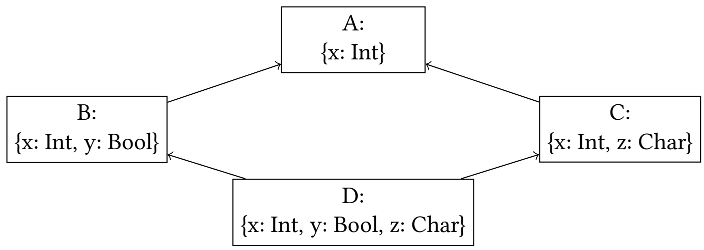

Fig. 3. Interface inheritance hierarchy with unrestricted intersections.

types are useful, but we need to be careful to make them safe during coercions. For example, $1 : \mathsf{Int} \,\&\, \mathsf{Int}$ can be safely uniquely coerced into the term $1 \,,,\, 1$, but $1 \,,,\, 1$ is not well-formed as it represents multiple choice of 1 on both sides of the merge. Instead of forbidding such terms in the first calculus λ_*, in second calculus $\lambda_{|*|}$ we normalize types into a well-formed canonical form, and then coerce the terms with normalized types. This calculus $\lambda_{|*|}$, which is presented in Sec. 5, does not have type well-formedness restrictions and allows unrestricted intersections.

The unrestricted type system allows types like $\mathsf{Int} \,\&\, \mathsf{Int}$. Such non-apart types are especially useful in modeling object-oriented programming style multiple (interface) inheritance. For example, consider the following class hierarchy in Fig. 3. In this scenario, it would be useful to use $B \,\&\, C$ to denote the type of class D. However, the intersection $B \,\&\, C$ is not apart, so it is not possible to use $B \,\&\, C$ to denote the type D in the calculus with type well-formedness. It could be cumbersome to manually write the type D directly or using type difference/annotations to resolve the conflict [Xu et al. 2023]. Instead, the calculus with normalization allows programmers to more naturally and intuitively express the combination of multiple inherited types directly as type $B \,\&\, C$.

To enable unrestricted intersections, we need to normalize types into a well-formed canonical form. For instance, a type like $\mathsf{Int} \,\&\, (\mathsf{Bool} \,\&\, \mathsf{Int})$ can be normalized to $\mathsf{Int} \,\&\, \mathsf{Bool}$, which is a well-formed type under the type system in Sec. 4. Similarly, we could normalize $B \,\&\, C$ to end up with the type D. Before type checking and execution, we translate programs from the source calculus $\lambda_{|*|}$ in Sec. 5 to the target calculus λ_* in Sec. 4. The target calculus λ_* enforces stricter well-formedness constraints and ensures that all types are normalized. Notably this translation process shows that, while there is a restriction on types in the target calculus λ_*, no expressive power is lost by this restriction on type well-formedness. The unrestricted syntax of intersection types has been given a constrained meaning by translation to well-formed intersection types, established by type normalization and a value-to-value mapping between $\lambda_{|*|}$ and λ_* in Sec. 5.

Type difference as type normalization. Type difference is an approach to conflict resolution proposed by Xu et al. [2023] to eliminate conflicts between terms of type A and type B. This mechanism for conflict resolution is based on coercive subtyping. In our calculus, type annotations can be used to manually resolve conflicts arising from overlapping types. For example, consider the variables x of type $\mathsf{Int} \,\&\, \mathsf{Bool}$ and y of type $\mathsf{Int} \,\&\, \mathsf{String}$. Attempting to merge x and y (i.e., $x \,,,\, y$) results in a conflict because they share the type Int. To resolve this conflict, we can use type annotations to explicitly exclude the conflicting type from either x or y:

$$(x : \mathsf{Bool}) \,,,\, y \quad \text{or} \quad x \,,,\, (y : \mathsf{String})$$

While type annotations effectively resolve conflicts, they can become impractical with large or complex types, requiring extensive manual annotations. For instance, with multiple inheritance, we may need to resolve conflicts in C_1 and C_2 when merging them:

```
e1 of type:
C1 = { l1:Int,
       l2:Bool,
       ...
       ln1:String→Int
};
```

```
e2 of type:
C2 = { l1:Bool,
       l2:Bool,
       ...
       ln2:Bool→Int,
}
```

Manually resolving such conflicts becomes laborious and sometimes extremely hard. To address such problems, *type difference* was designed to automate the annotations needed, enabling us to programmatically exclude conflicting parts without verbose annotations. Using type difference, we can resolve conflicts succinctly:

$$\begin{array}{ccc} e_1 ,, (e_2 \backslash C_1) & & e_1 ,, (e_2 \backslash e_1) \\ & \text{or} & \\ (e_1 \backslash C_2) ,, \; e_2 & & (e_1 \backslash e_2) ,, \; e_2 \end{array}$$

where $e \backslash A$ suggests removing all the type A components from e, and $e_1 \backslash e_2$ suggests removing all conflicting parts of e_1 and e_2 from e_1.

Though Xu et al.'s type difference is expressive, it is not total. Type difference under disjointness cannot resolve all the merge conflicts. For example, if we have type $\mathsf{Int} \to \mathsf{String}$ and $\mathsf{Bool} \to \mathsf{String}$, Xu et al.'s type difference cannot find a type $C = (\mathsf{Int} \to \mathsf{String}) \backslash (\mathsf{Bool} \to \mathsf{String})$ such that C is mergeable (disjoint) with $\mathsf{Bool} \to \mathsf{String}$, while keeping all the other information in type $\mathsf{Int} \to \mathsf{String}$. In their work, Xu et al. proposed a specification for type difference, denoted as $A \backslash_s B = C$, along with a partial algorithm that is sound and complete with respect to this specification. In contrast, our work introduces a relaxation of disjointness using a one-sided apartness relation $A <\!\!* B$. This relation ensures that the type components of B are not shadowed by those of A, but not necessarily vice versa (formally defined in Definition 6.1). Under this relaxed notion, we obtain a total specification for type difference. Besides, we present *type normalization* as a *different* algorithm that is both sound and complete with respect to the new specification. In this framework, a type A is normalized to $|A|$ through an auxiliary process that normalizes one type with respect to another, denoted $|A|_B$. Full normalization is a special case: $|A|_\top$, which resolves all conflicts in A relative to the top type.

Type normalization $|A|_B$ is sound and complete with respect to the new type difference specification: $A \backslash_s B = |A|_B$, and for all C, if $A \backslash_s B = C$, then $C \leq |A|_B$ and $|A|_B \leq C$. Importantly, under our new formulation, $|A|_B$ is algorithmic and total, and provides an algorithm for type difference. All conflicts in two non-apart types can be resolved by applying type normalization twice: first compute $|A|_B$, then compute $|B|_{|A|_B}$.

3 Apartness and Guarded Subtyping for Intersection Types

This section shows how to derive algorithmic formulations of apartness and how to formulate guarded subtyping with intersection types and a merge operator.

3.1 Syntax and Subtyping

Types. Our type syntax follows the syntax of λ_i^+ by Huang et al. [2021], extended with a bottom type and a singleton type. The top and bottom types are crucial here to support type difference and normalization, as suggested by Xu et al. [2023]. Singleton types $\mathsf{Sig}\ l$ are included for encoding record types. A labelled type (or single-field record type) $\{l : A\}$ is represented by the function type $\mathsf{Sig}\ l \to A$. A multi-field record type is then represented as an intersection of labelled types. A record type is an overloaded function, where the input type is the label to be projected.

Types	$A, B, C ::= \mathsf{Int} \mid \top \mid \bot \mid A \,\&\, B \mid A \to B \mid \mathsf{Sig}\ l$
Ordinary Types	$A^\circ, B^\circ, C^\circ ::= \mathsf{Int} \mid \top \mid \bot \mid A \to B^\circ \mid \mathsf{Sig}\ l$
Labelled type encoding	$\{l : A\} \triangleq \mathsf{Sig}\ l \to A$
Record encoding	$\{l_1 : A_1,\ \ldots,\ l_n : A_n\} \triangleq \{l_1 : A_1\} \,\&\, \ldots \,\&\, \{l_n : A_n\}$

$\boxed{A \leq B}$ *(Declarative BCD Subtyping)*

$$\frac{}{A \leq A} \qquad \frac{}{\bot \leq A} \qquad \frac{}{A \leq \top}$$

$$\frac{B \leq A \qquad C \leq D}{A \to C \leq B \to D} \qquad \frac{A \leq B \qquad B \leq C}{A \leq C}$$

$$\frac{}{A_1 \,\&\, A_2 \leq A_i} \qquad \frac{A \leq B_1 \qquad A \leq B_2}{A \leq B_1 \,\&\, B_2}$$

$$\frac{}{(A \to B_1) \,\&\, (A \to B_2) \leq A \to (B_1 \,\&\, B_2)}$$

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

$$\frac{}{A <: A} \qquad \frac{}{\bot <: A} \qquad \frac{}{A <: \top}$$

$$\frac{C^\circ \qquad D^\circ \qquad B <: A \qquad C <: D}{A \to C <: B \to D}$$

$$\frac{A \simeq A_1 \,\&\, A_2 \qquad A_i <: B}{A <: B}$$

$$\frac{B \simeq B_1 \,\&\, B_2 \qquad A <: B_1 \qquad A <: B_2}{A <: B}$$

$\boxed{\vdash_s A}$ *(Type Well-Formedness (Strict))*

$$\frac{}{\vdash_s \mathsf{Int}} \qquad \frac{}{\vdash_s \mathsf{Sig}\ l} \qquad \frac{}{\vdash_s \bot} \qquad \frac{}{\vdash_s \top} \qquad \frac{\vdash_s A \qquad \vdash_s B \qquad B^\circ}{\vdash_s A \to B}$$

$$\frac{\vdash_s A \qquad \vdash_s B \qquad A \neq \top \qquad B \neq \top \qquad A * B}{\vdash_s A \,\&\, B}$$

$\boxed{A \simeq B \,\&\, C}$ *(Type Splitting)*

$$\frac{}{A \,\&\, B \simeq A \,\&\, B}$$

$$\frac{B \simeq B_1 \,\&\, B_2}{A \to B \simeq (A \to B_1) \,\&\, (A \to B_2)}$$

Fig. 4. BCD subtyping, ordinary types, splittable types and the strict type well-formedness.

Subtyping, ordinary and splittable types. BCD subtyping [Barendregt et al. 1983] is a widely used subtyping relation for intersection types. BCD subtyping includes common intersection type rules that ensure the commutativity ($A \,\&\, B \leq B \,\&\, A$) and associativity ($A \,\&\, (B \,\&\, C) \leq (A \,\&\, B) \,\&\, C$) of subtyping, which allows us to handle multi-field record types as intersection types without any loss of precision. The most important feature of BCD-style subtyping is that it allows function types to distribute over intersection types. We have all the types in the original BCD type system. The declarative subtyping rules, presented at Fig. 4, extend BCD subtyping with rules for singleton types and the bottom type, but exclude the top arrow rule $\top \leq \top \to \top$, which does not appear to be particularly practical—though such a rule can be added with a valid coercion. The extension of singleton and bottom types is easy as they do not interact with the intersection subtyping rules.

On the right of the middle of Fig. 4 shows an algorithmic formulation of subtyping, that eliminates transitivity and is equivalent to the declarative formulation [Huang et al. 2021]. This algorithmic formulation relies on two notions, *ordinary types* and *splittable types*, which are shown in Fig. 4. *Ordinary types* [Davies and Pfenning 2000], are types that do not contain top level intersections. In BCD subtyping, arrow types such as $A \to B \,\&\, C$ may behave like intersection types due to distributivity. In the algorithmic formulation ($A <: B$) of subtyping ($A \leq B$) we restrict the notion of ordinary types to exclude such types. *Splittable types* are the complement of ordinary types. Type splitting separates the intersection components inside intersection-like types. Though a splittable type may not always have intersection at its top-level, the type is still equivalent to an intersection. To fully analyze the subtyping behaviour, we define splittable types as shown in Fig. 4.

3.2 Type Well-Formedness

Type well-formedness ensures that a type always has at most one unique coercion to another well-formed type. These conditions, along with the guarded subtyping introduced later in Sec. 3.4, prevent the casting result from being ill-typed. Type well-formedness is algorithmic, and therefore uses the algorithmic apartness in Sec. 3.3 instead of apartness specification $A *_s B$ in its definition.

We will motivate the design of type well-formedness through an intersection type example. First, we want to avoid overlap within types, specifically avoiding situations where $A \,\&\, B$ exists without $A * B$. For example, $\mathsf{Int} \,\&\, \mathsf{Int}$ leads to ambiguity inside types, since there is more than one coercion to type Int. Second, to ensure that coercions are unique, we require that the splitting results of the type are always apart. In particular, for a type like $A \to B \,\&\, C$, we need $(A \to B) * (A \to C)$ given that $A \to B \,\&\, C \simeq (A \to B) \,\&\, (A \to C)$. This requires a more rigorous approach than simply checking the apartness of intersections structurally, as $B * C$ does not imply $(A \to B) * (A \to C)$. An example occurs when B equals $\top$. Function types like $\mathsf{Int} \to \top \,\&\, \mathsf{Bool}$ may seem harmless if we only verify the apartness of the intersection in the function's return type. However, it splits into overlapping types $\mathsf{Int} \to \top$ and $\mathsf{Int} \to \mathsf{Bool}$. To achieve an algorithmic version of apartness (shown in Figure 5) satisfying the requirement of uniqueness that every type component being apart, we introduce a type well-formedness definition that possesses the following properties:

Theorem 3.1 (Well-Formed Types have Apart Components). *If $\vdash_s A$, and $A \simeq B \,\&\, C$, then $A = B \,\&\, C$, $B * C$, $\vdash_s B$ and $\vdash_s C$.*

With this property, we can later show completeness of the apartness algorithm in Sec. 3.3. Type well-formedness may seem restrictive here, as it requires arrow types (like $A \to B \,\&\, C$) to be written in its equivalent form $(A \to B) \,\&\, (A \to C)$. However, this restriction will be partially resolved in Sec. 4 by relaxing well-formedness and fully resolved in Sec. 5 by transforming every type into a well-formed type by type normalization.

3.3 Apartness

Our goal is to define an algorithmic formulation of apartness that is equivalent to the specification in Sec. 2.4. Our algorithmic apartness relation $(A * B)$ is defined for BCD subtyping [Barendregt et al. 1983], and it essentially expresses that for types A and B, it is always possible to extract all the information from values of both types without ambiguity. To formally define apartness and ambiguity, we use an auxiliary relation $A \between B$ that expresses whether two types A and B contain shadowing (i.e. if one's component is a subtype of the other's). We want to define our apartness $A * B$ based on the negation of the shadowing relation $A \between B$ (i.e., $A * B = \neg(A \between B)$).

Shadowing. The shadowing relation determines whether the type components of two types A and B shadow each other, and whether there is ambiguity between A and B. Some examples of types A and B and their shadowing relations are:

$$\begin{array}{rlr}
A = \mathsf{Int} & B = \mathsf{Bool} & \neg(A \between B) \\
A = \mathsf{Int} \,\&\, \mathsf{Bool} & B = \mathsf{Bool} & A \between B \\
A = \mathsf{Int} & B = \top & \neg(A \between B) \\
A = \mathsf{Int} \to \mathsf{String} & B = \mathsf{Bool} \to \mathsf{String} & \neg(A \between B) \\
A = \mathsf{Int} \to \mathsf{String} & B = (\mathsf{Int} \,\&\, \mathsf{Bool}) \to \mathsf{String} & A \between B
\end{array}$$

Most of these examples are straightforward. Notably, in the third example where B is $\top$, we consider $\top$ to have no type component. Therefore, shadowing of type A over type $\top$ cannot occur as the two types do not share any type component. Besides, $\top$ cannot shadow any type. The negation of this judgement then always allows the $\top$ type to be apart with any type.

$\boxed{A \between B}$ *(Type Shadowing)*

$$\frac{A \simeq A_1 \,\&\, A_2 \qquad A_i \between B}{A \between B} \qquad \frac{A^\circ \qquad B \simeq B_1 \,\&\, B_2 \qquad A \between B_i}{A \between B} \qquad \frac{A^\circ \qquad B^\circ \qquad A <: B \qquad B \neq \top}{A \between B} \qquad \frac{A^\circ \qquad B^\circ \qquad B <: A \qquad A \neq \top}{A \between B}$$

Fig. 5. Shadowing.

We provide an inductive definition of $A \between B$ in Fig. 5. Ignoring the gray conditions, we can see that shadowing is defined when one type is a subtype of the other, or when any of their splitting components have a shadowing relationship. These gray-colored side conditions reduce overlap between the rules and help make the implementation more efficient. We have proved that adding these conditions leads to an equivalent definition. With these conditions, intersection-like types are always split, and only the ordinary components are compared in subtyping checks to decide whether there is a conflict. Now we can write our algorithmic apartness $A * B$ as $\neg(A \between B)$.

Soundness and completeness of algorithmic apartness. The formulation of apartness $A * B$ via the negation of $A \between B$ is sound with respect to the specification $A *_s B$ given in Sec. 2.4. Completeness requires more work. We have $(\mathsf{Int} \to \mathsf{String}) *_s (\mathsf{Bool} \to \mathsf{String}) \,\&\, ((\mathsf{Int} \,\&\, \mathsf{Bool}) \to \mathsf{String})$ but not $(\mathsf{Int} \to \mathsf{String}) * (\mathsf{Bool} \to \mathsf{String}) \,\&\, ((\mathsf{Int} \,\&\, \mathsf{Bool}) \to \mathsf{String})$. As suggested in Section 2.4, $\mathsf{Int} \to \mathsf{String}$ has one minimal ordinary type component, namely itself, while $(\mathsf{Bool} \to \mathsf{String}) \,\&\, ((\mathsf{Int} \,\&\, \mathsf{Bool}) \to \mathsf{String})$ also has only one minimal ordinary type component $\mathsf{Bool} \to \mathsf{String}$. Then, since the only type components are unrelated by subtyping, the specification apartness holds. However, for algorithmic apartness, we recursively decompose the types and try to find the subtyping relation between them. Therefore we do not have $(\mathsf{Int} \to \mathsf{String}) * (\mathsf{Bool} \to \mathsf{String}) \,\&\, ((\mathsf{Int} \,\&\, \mathsf{Bool}) \to \mathsf{String})$ for $(\mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{String}) \leq \mathsf{Int} \to \mathsf{String}$. To address such incompleteness, we introduce a notion of well-formed types $\vdash_s A$ which restricts intersections to apart types. Under this restriction, we are now able to formulate a form of completeness.

Theorem 3.2 (Soundness of Apartness). *If $A * B$, then $A *_s B$.*

Theorem 3.3 (Completeness of Apartness). *If $\vdash_s A$, $\vdash_s B$, and $A *_s B$, then $A * B$.*

3.4 Guarded Subtyping

Next we introduce the guarded subtyping relation. With disjointness, a term of type A can always be cast to a term of type B when $A \leq B$, without introducing ambiguity. However, with apartness, this is no longer true. Since apartness allows more ambiguity, we need to have a notion to validate casting. The guarded subtyping relation is essentially a *restricted* version of subtyping. Unlike subtyping, guarded subtyping *is not transitive.*

Definition. Guarded subtyping, denoted as $A \lesssim B$, ensures that a term of type A can be unambiguously cast to a term of type B. This unambiguity arises not merely because A is a subtype of B, but importantly because there is a unique way to coerce a value of type A into a value of type B. In other words, any value of type A can be unambiguously mapped to a value of type B.

To provide more intuition, we can illustrate the guarded subtyping with some concrete examples:

$$\begin{array}{lll}
A = (\mathsf{Int} \to \mathsf{String}) \,\&\, (\mathsf{Bool} \to \mathsf{String}) & B = \mathsf{Bool} \to \mathsf{String} & A \lesssim B \\
A = (\mathsf{Int} \to \mathsf{String}) \,\&\, (\mathsf{Bool} \to \mathsf{String}) & B = (\mathsf{Int} \,\&\, \mathsf{Bool}) \to \mathsf{String} & \neg(A \lesssim B) \\
A = (\mathsf{Int} \to \mathsf{String}) \,\&\, (\mathsf{Bool} \to \mathsf{String}) & B = (\mathsf{Bool} \to \mathsf{String}) \,\&\, (\mathsf{Int} \to \mathsf{String}) & A \lesssim B \\
A = (\mathsf{Int} \to \mathsf{String}) \,\&\, (\mathsf{Bool} \to \mathsf{String}) & B = \mathsf{String} & \neg(A \lesssim B) \\
A = (\mathsf{Int} \to \mathsf{String}) \,\&\, (\mathsf{Bool} \to \mathsf{String}) & B = \top & A \lesssim B
\end{array}$$

$\boxed{A \lesssim B}$ *(Guarded Subtyping)*

$$\text{C-ANDL}\quad \frac{B^\circ \qquad A \simeq A_1 \,\&\, A_2 \qquad A_1 \lesssim B \qquad \neg(A_2 <: B)}{A \lesssim B}$$

$$\text{C-ANDR}\quad \frac{B^\circ \qquad A \simeq A_1 \,\&\, A_2 \qquad A_2 \lesssim B \qquad \neg(A_1 <: B)}{A \lesssim B}$$

$$\text{C-AND}\quad \frac{B_1 \simeq B_2 \,\&\, B_3 \qquad A \lesssim B_2 \qquad A \lesssim B_3}{A \lesssim B_1}$$

$$\text{C-ORD}\quad \frac{(A_1 \to B_1)^\circ \qquad (A_2 \to B_2)^\circ \qquad A_2 \lesssim A_1 \qquad B_1 \lesssim B_2}{A_1 \to B_1 \lesssim A_2 \to B_2}$$

$$\text{C-INT}\quad \frac{}{\mathsf{Int} \lesssim \mathsf{Int}} \qquad \text{C-SIG}\quad \frac{}{\mathsf{Sig}\, l \lesssim \mathsf{Sig}\, l} \qquad \text{C-TOP}\quad \frac{}{A \lesssim \top} \qquad \text{C-BOT}\quad \frac{}{\bot \lesssim \bot}$$

Fig. 6. Guarded subtyping.

In the first example, we can find a unique coercion from $(\mathsf{Int} \to \mathsf{String}) \,\&\, (\mathsf{Bool} \to \mathsf{String})$ to $\mathsf{Bool} \to \mathsf{String}$, since only the term of type $\mathsf{Bool} \to \mathsf{String}$ can be coerced to a term of type $\mathsf{Bool} \to \mathsf{String}$. However, when casting a term of $(\mathsf{Int} \to \mathsf{String}) \,\&\, (\mathsf{Bool} \to \mathsf{String})$ to $(\mathsf{Int} \,\&\, \mathsf{Bool}) \to \mathsf{String}$, we are unable to tell whether the resulting $(\mathsf{Int} \,\&\, \mathsf{Bool}) \to \mathsf{String}$ comes from the term that implements $\mathsf{Int} \to \mathsf{String}$ or the one that implements $\mathsf{Bool} \to \mathsf{String}$. Therefore, it is rejected by the guarded subtyping relation. In the last three examples we consider cases where B is an intersection type, a type that is apart from A, and the type $\top$. For intersection type cases, we only need to check whether every type component in type B is a guarded supertype of A. For types that do not have a subtyping relation, such coercion is not permissible as terms of B cannot provide sufficient information to build a value of A. For the $\top$ type, as there is only one value $\top$ associated with it, mapping from any set of values to it is always unique and never leads to ambiguity.

Guarded subtyping is defined in Fig. 6. Rules C-ANDL and C-ANDR suggest that only one of the two parts of an intersection-like type could contribute to the supertype, otherwise ambiguity is introduced. For example, we are not able to choose from a merge of $(\mathsf{Int} \to \mathsf{String}) \,\&\, (\mathsf{Bool} \to \mathsf{String})$ for the type $(\mathsf{Int} \,\&\, \mathsf{Bool}) \to \mathsf{String}$. Note that these two rules use the negation of subtyping rather than guarded subtyping, since they are checking if there exists a unique path from the subtype to the supertype. If the condition is relaxed to be $\neg(A_1 \lesssim B)$ or $\neg(A_2 \lesssim B)$, we will be able to derive $\mathsf{Int} \,\&\, \mathsf{Int} \,\&\, \mathsf{Int} \lesssim \mathsf{Int}$ due to $\neg(\mathsf{Int} \,\&\, \mathsf{Int} \lesssim \mathsf{Int})$. Rule C-AND is unsurprising: when the supertype is an intersection-like type, the unique coercion exists when there exists unique coercions to both of its intersection components. Finally rules C-INT, C-SIG, C-TOP, and C-ORD define the behaviour of casting for non-intersection types. The notable case is the function rule C-ORD, where we also enforce that the relation is contravariant on the input types of functions.

Soundness. The guarded subtyping relation always leads to safe casting as the casting is an upcast, and there is no ambiguity. No casting from both A and B is available unless they are cast to type $\top$. Thus, no ambiguity is caused during casting.

THEOREM 3.4 (SOUNDNESS OF GUARDED SUBTYPING). *If $A \lesssim B$, then $A \leq B$.*

THEOREM 3.5 (SAFE CASTING). *If $A * B$ and $A \leq C$ and $B \leq C$, then $\neg(A \,\&\, B \lesssim C)$ or $\top \leq C$.*

4 The λ_* Calculus

In this section, we introduce a lambda calculus with a merge operator, referred to as λ_*. Similar to previous calculi featuring disjoint intersection types (e.g. F_i^+ [Fan et al. 2022]), λ_* uses type annotations to trigger dynamic casts that coerce values. These values are often aggregated through merges that are statically checked for conflicts. We will first explain how the type system employs the concept of *apartness* to validate merges and support overloaded functions. Following this, we will explore dynamic dispatching as demonstrated in the reduction rules.

(Syntax of Expressions)

Expressions	$e ::= x \mid i \mid \top \mid \{l\} \mid \lambda^{A\to B}x.\, e \mid \mathsf{fix}^A\ x.\, e \mid e_1\, e_2 \mid e : A \mid e_1 \,,,\, e_2$
Function-Like Values	$u ::= \lambda^{A\to B}x.\, e \mid u : A \to B$
Values	$v ::= u \mid i \mid \top \mid \{l\} \mid v_1 \,,,\, v_2$

$\boxed{\vdash_r A}$ *(Relaxed Type Well-Formedness)*

$$\frac{\vdash_s A}{\vdash_r A} \qquad \frac{\vdash_r A \qquad \vdash_s B}{\vdash_r A \to B} \qquad \frac{\vdash_r A \qquad \vdash_r B \qquad A * B}{\vdash_r A \,\&\, B}$$

$\boxed{\vdash \Gamma}$ *(Well-Formedness of Typing Environments)*

$$\frac{}{\vdash \cdot} \qquad \frac{\vdash \Gamma \qquad \vdash_r A \qquad x \notin \mathrm{dom}\,\Gamma}{\vdash \Gamma, x : A}$$

$\boxed{\Gamma \vdash_t e \Leftrightarrow A}$ *(Typing Rules of λ_*)*

$$\text{TTyp-top}\ \frac{\vdash \Gamma}{\Gamma \vdash_t \top \Rightarrow \top} \qquad \text{TTyp-lit}\ \frac{\vdash \Gamma}{\Gamma \vdash_t i \Rightarrow \mathsf{Int}} \qquad \text{TTyp-sig}\ \frac{\vdash \Gamma}{\Gamma \vdash_t \{l\} \Rightarrow \mathsf{Sig}\, l}$$

$$\text{TTyp-fix}\ \frac{\Gamma, x : A \vdash_t e \Leftarrow A}{\Gamma \vdash_t \mathsf{fix}^A\ x.\, e \Rightarrow A} \qquad \text{TTyp-anno}\ \frac{\Gamma \vdash_t e \Leftarrow A}{\Gamma \vdash_t e : A \Rightarrow A}$$

$$\text{TTyp-var}\ \frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash_t x \Rightarrow A} \qquad \text{TTyp-abs}\ \frac{\vdash_r A \to B \qquad \Gamma, x : A \vdash_t e \Leftarrow B}{\Gamma \vdash_t \lambda^{A\to B}x.\, e \Rightarrow A \to B}$$

$$\text{TTyp-app}\ \frac{B \vdash A \rhd C \qquad \Gamma \vdash_t e_1 \Rightarrow A \qquad \Gamma \vdash_t e_2 \Rightarrow B}{\Gamma \vdash_t e_1\, e_2 \Rightarrow C} \qquad \text{TTyp-merge}\ \frac{A * B \qquad \Gamma \vdash_t e_1 \Rightarrow A \qquad \Gamma \vdash_t e_2 \Rightarrow B}{\Gamma \vdash_t e_1 \,,,\, e_2 \Rightarrow A \,\&\, B}$$

$$\text{TTyp-sub}\ \frac{\vdash_r B \qquad \Gamma \vdash_t e \Rightarrow A \qquad A \lesssim B}{\Gamma \vdash_t e \Leftarrow B}$$

Fig. 7. The syntax, type well-formedness and typing rules for λ_*.

4.1 Syntax and the Type System of λ_*

Expressions and values. We define the syntax of expressions and values at the top of Fig. 7. Every function in λ_* must be accompanied by annotations, including the parameter and return type. These annotations can be nested layer by layer. We refer to lambdas with multiple annotation layers as *function-like values* and use the meta-variable u to represent them. In addition to functions, λ_* supports singleton values ($\{l\}$), fixpoints ($\mathsf{fix}^A\ x.\ e$) and merges ($e_1 \,,,\, e_2$). The top value $\top$ can be viewed as a merge with zero components.

Relaxed type well-formedness. We use a specially crafted type well-formedness definition in λ_*, defined in the middle of Fig. 7. In the strict type well-formedness $\vdash_s A$ (see Fig. 4), intersections cannot contain $\top$, and arrow types $A \to B$ must enforce that the type B is a non-intersection type. The relaxed well-formedness $\vdash_r A$ loosens the strict criteria, allowing top types $\top$ to be intersected, and arrow types to have intersection types as the return types. Though the relaxed type well-formedness offers greater flexibility, its primary purpose is to enable a value-to-value mapping between $\lambda_{|*|}$ and λ_*. If a value in $\lambda_{|*|}$ were mapped to a non-value in λ_*, then the correspondence would also need to extend substitution in $\lambda_{|*|}$ (with values) to substitution in λ_* (with non-values), making it nearly impossible to track equivalence between the source and target calculi.

To ensure this value-to-value mapping, *function-like values* are allowed to have only arrow type annotations, such as $\lambda^{A\to B\,\&\,C}x.\ e$. In contrast, annotations like $\lambda x.\ e : (A \to B) \,\&\, (A \to C)$ are disallowed, as they could evaluate to a merge of functions, e.g., $\lambda^{A\to B}x.\ e \,,,\, \lambda^{A\to C}x.\ e$, which is not a value in the target calculus. Therefore, it is crucial to permit arrow types that return intersections—such as $A \to B \,\&\, C$—to be well-formed, while avoiding intersection types at the top level of function annotations. Although relaxed type well-formedness admits more components, it remains safe due to completeness with respect to apartness. Moreover, it ensures that equivalent types can always be uniquely and safely coerced.

Theorem 4.1 (Soundness of Relaxed Well-Formedness). *If $\vdash_s A$ then $\vdash_r A$.*

$\boxed{B \vdash A \not\triangleright}$ *(Inapplicability)*

$$\frac{\neg(B <: A)}{B \vdash A \to C \not\triangleright} \qquad \frac{}{B \vdash \mathsf{Int} \not\triangleright} \qquad \frac{}{B \vdash \mathsf{Sig}\ l \not\triangleright} \qquad \frac{}{B \vdash \top \not\triangleright} \qquad \frac{D \vdash A \not\triangleright \qquad D \vdash B \not\triangleright}{D \vdash A \,\&\, B \not\triangleright}$$

$\boxed{B \vdash A \triangleright C}$ *(Type Dispatching)*

$$\frac{B \lesssim A}{B \vdash A \to C \triangleright C} \qquad \frac{A_1 * B_1 \quad D \vdash A \triangleright A_1 \qquad D \vdash B \triangleright B_1}{D \vdash A \,\&\, B \triangleright A_1 \,\&\, B_1} \qquad \frac{D \vdash A \triangleright A_1 \quad D \vdash B \not\triangleright}{D \vdash A \,\&\, B \triangleright A_1} \qquad \frac{D \vdash A \not\triangleright \quad D \vdash B \triangleright B_1}{D \vdash A \,\&\, B \triangleright B_1}$$

Fig. 8. The type dispatching relation for λ_*.

Theorem 4.2 (Well-formed Types Respect Apartness). *If $\vdash_r A$ and $A \simeq B \,\&\, C$, then $B * C$.*

Theorem 4.3 (Completeness for Relaxed Well-Formedness). *For all A B that $\vdash_r A$ and $\vdash_r B$, if $A *_s B$ then $A * B$.*

Theorem 4.4 (Round-Trip Coercion). *For all A B, if $\vdash_r A$, $\vdash_r B$, $A \leq B$, and $B \leq A$, then $A \lesssim B$.*

Typing rules. As shown at the bottom of Fig. 7, a bidirectional type system [Dunfield and Krishnaswami 2021; Pierce and Turner 2000] is designed to characterize the static aspects of λ_*. Its typing judgments operate in two modes: under inference mode $\Rightarrow$, a unique type is deduced from the typing context for the expression; while in checking mode $\Leftarrow$, the type is given (along with the typing context and the expression), and the expression is verified against the type. In our calculus, every typed expression has an inferred type (e.g. A). Thus being checked (e.g., by B) is equivalent to examining the subtyping relationship between A and B. All checked and inferred types are all guaranteed to be well-formed.

Theorem 4.5 (Typing Respects Type Well-Formedness). *If $\Gamma \vdash_t e \Leftrightarrow A$, then $\vdash_r A$.*

This formulation is also algorithmic, ensuring that no information is hidden via subtyping. In other words, we are fully aware of what e can contain from $\Gamma \vdash_t e \Rightarrow A$. Therefore, we can determine that two expressions are free of conflicts simply by comparing their inferred types using *apartness* (defined in 3.3) in rule TTyp-merge. This approach is typical for type systems featuring disjoint intersection types with an unbiased merge operator. In such type systems, expressions can be composed via merging directly, as long as they do not create ambiguity, such as in $1 \,,,\, \mathsf{True} \,,,\, \mathsf{f}$ (f represents any legal function). Since apart types are not distinguishable in every possible context, our subsumption rule (rule TTyp-sub) is equipped with the *guarded subtyping* relation (defined in 3.4) to reject ambiguous casts. Merges can also be used directly as functions. For example, $(\mathsf{f} : \mathsf{Int} \to \mathsf{Bool}) \,,,\, (\mathsf{g} : \mathsf{Bool} \to \mathsf{Bool})$ type-checks when applied to an integer or boolean argument. In rule TTyp-app, the two inference types of the applied term and the argument are passed to the *type dispatching relation*, which computes the return type.

Type dispatching. Type dispatching is defined in Fig. 8. In $\lambda_{|*|}$ and λ_*, type dispatching is only used for well-formed types. So, in the following text, we can assume the input type A and B are well-formed, and we try to get a well-formed type as return type.

First we define $B \vdash A \not\triangleright$ to denote that A cannot serve as a function type that takes a value of type B as an argument: there is no type C such that $A <: B \to C$. For conciseness, We use $B \vdash A \triangleright$ to represent the negation of $B \vdash A \not\triangleright$. We use the subtyping judgment in inapplicability instead of $\neg(B \lesssim A)$ because we want to forbid all *possible* coercions instead of all *unique* coercions from B to A. Then we define the type dispatching relation $B \vdash A \triangleright C$ which does two more things: It computes the minimal return type, and also ensures that there is no ambiguity in the application process. $B \vdash A \triangleright C$ indicates that applying a term of type A to a term of type B should return a term

$\boxed{v_1 \hookrightarrow_A v_2}$ *(Casting)*

$$\frac{A \simeq B \,\&\, C \qquad v \hookrightarrow_B v_1 \qquad v \hookrightarrow_C v_2}{v \hookrightarrow_A v_1 \,,, v_2} \qquad \frac{u : A \to B \qquad (C \to D)^\circ \qquad C <: A \qquad B <: D}{u \hookrightarrow_{C \to D} u : C \to D}$$

$$\frac{A^\circ \qquad v_1 \hookrightarrow_A v_1'}{v_1 \,,, v_2 \hookrightarrow_A v_1'} \qquad \frac{A^\circ \qquad v_2 \hookrightarrow_A v_2'}{v_1 \,,, v_2 \hookrightarrow_A v_2'} \qquad \frac{}{i \hookrightarrow_{\mathsf{Int}} i} \qquad \frac{}{\{l\} \hookrightarrow_{(\mathsf{Sig}\, l)} \{l\}} \qquad \frac{}{v \hookrightarrow_\top \top}$$

$\boxed{v_1 \bullet v_2 \rightrightarrows e}$ *(Parallel Application)*

$$\frac{\begin{array}{c} B \vdash A_1 \rhd \qquad B \vdash A_2 \rhd \\ v_1 : A_1 \qquad v_2 : A_2 \qquad v : B \end{array}}{v_1 \,,, v_2 \bullet v \rightrightarrows (v_1\, v) \,,, (v_2\, v)} \qquad \frac{v_2 : A \qquad v : B \qquad B \vdash A \not\rhd}{v_1 \,,, v_2 \bullet v \rightrightarrows v_1\, v} \qquad \frac{v_1 : A \qquad v : B \qquad B \vdash A \not\rhd}{v_1 \,,, v_2 \bullet v \rightrightarrows v_2\, v}$$

$\boxed{e_1 \hookrightarrow e_2}$ *(Small-Step Semantics for λ_*)*

$$\textsc{StepN-beta}\ \frac{v \hookrightarrow_A v'}{(\lambda^{A \to B} x.\, e)\, v \hookrightarrow (e[x \mapsto v']) : B} \qquad \textsc{StepN-absanno}\ \frac{v \hookrightarrow_A v'}{(u : A \to B)\, v \hookrightarrow (u\, v') : B} \qquad \textsc{StepN-papp}\ \frac{(v_1 \,,, v_2) \bullet v_3 \rightrightarrows e}{(v_1 \,,, v_2)\, v_3 \hookrightarrow e}$$

$$\textsc{StepN-annov}\ \frac{\neg(\mathbf{value}\ v : A) \qquad v \hookrightarrow_A v'}{v : A \hookrightarrow v'} \qquad \textsc{StepN-anno}\ \frac{e \hookrightarrow e'}{e : A \hookrightarrow e' : A} \qquad \textsc{StepN-fix}\ \frac{}{\mathsf{fix}^A\, x.\, e \hookrightarrow e[x \mapsto \mathsf{fix}^A\, x.\, e] : A} \qquad \textsc{StepN-appl}\ \frac{e_1 \hookrightarrow e_1'}{e_1\, e_2 \hookrightarrow e_1'\, e_2}$$

$$\textsc{StepN-appr}\ \frac{e_2 \hookrightarrow e_2'}{v_1\, e_2 \hookrightarrow v_1\, e_2'} \qquad \textsc{StepN-mergel}\ \frac{e_1 \hookrightarrow e_1'}{e_1 \,,, e_2 \hookrightarrow e_1' \,,, e_2} \qquad \textsc{StepN-merger}\ \frac{e_2 \hookrightarrow e_2'}{v_1 \,,, e_2 \hookrightarrow v_1 \,,, e_2'}$$

Fig. 9. The operational semantics of λ_*.

of type C. Since the function argument is cast before beta reduction, we need to verify that the argument type is a guarded subtype of the parameter type. Thus, type dispatching formalizes the behaviour of an applicative intersection type. The soundness theorems for type dispatching, which computes the most specific return type given a function type A and an argument type B, are:

Theorem 4.6 (Soundness and Minimality of Type Dispatching). *If $B \vdash A \rhd C$, then:*

- *If $\vdash_r A$, then $\vdash_r C$.*
- $A <: B \to C$
- *$\forall\, D$, if $A <: B \to D$ then $C <: D$*

However, completeness is intractable here because guarded subtyping poses a unique coercion requirement on the argument type. This is hard to track in the subtyping relation; therefore we leave it here as an open problem. Note that for every well-typed expression in λ_*, its type is determined in a syntax-directed way. In the following text, we use $v : A$ to denote such a syntactic lookup, meaning that the inferred type of v is A.

4.2 Operational Semantics

In Fig. 9, we introduce three reduction relations to define the operational semantics of λ_*. The casting relation $v_1 \hookrightarrow_A v_2$ takes a value v_1 and a type A and generates a value v_2, upcasting v_1 to match type A. The parallel application relation $v_1 \bullet v_2 \rightrightarrows e$ distributes the argument v_2 into the merge v_1 and achieves dynamic dispatching. The small-step reduction $e_1 \hookrightarrow e_2$ indicates that e_1 evaluates to e_2. At runtime, we replace all occurrences of guarded subtyping $A \lesssim B$ by subtyping $A \leq B$ since type checking is performed prior to evaluation. Moreover, $A \leq B$ is computationally cheaper than $A \lesssim B$, since it merely checks for the *existence* of a coercion whereas guarded subtyping verifies the *uniqueness* of coercions.

Function application. The evaluation rules are presented at the bottom of Fig. 9. There are three rules for function application: rule StepN-beta, rule StepN-absanno, and rule StepN-papp, which cover lambdas, function-like values, and merges, respectively. In rule StepN-beta, we first cast the application argument by the parameter annotation, and then use the result for beta reduction. The entire term is then wrapped by the function return type annotation, which denotes a delayed cast. Rule StepN-absanno handles the type annotations of a function-like value similarly. Eventually, when all outer annotations are decomposed, a function-like value becomes a lambda function, which then follows the beta rule. In addition to function-like values, merges can also be at the function position. Dynamic dispatching in rule StepN-papp is achieved via *parallel application.*

Parallel application and casting. Defined in the middle of Fig. 9, the parallel application relation $v_1 \bullet v_2 \;\rightrightarrows\; e$ constructs an application expression e for each function contained in v_1, provided that the resulting application is well-typed. The unwanted parts of merges are filtered out. This type matching is performed dynamically because functions always have explicit type annotations. By comparing the parameter type with the type of the argument, it is straightforward to determine whether the requirements for a function are satisfied. As the type system already ensures the well-typedness of the application results, we only need to apply the function whenever the argument type matches (denoted by $B \vdash A \rhd$).

The definition of casting is shown at the top of Fig. 9. All annotations can be interpreted as a cast-to-do, unless they are part of values, as indicated by rule StepN-annov (while annotated values can only be function-like values). Type annotations may be decomposed (rules StepN-beta and StepN-absanno) or duplicated (rule StepN-fix), similar to what occurs in a cast calculus within the gradual typing setting [Wadler and Findler 2009]. Casting of *splittable* types ensures that no intersection types remain at the top level or in the function return type. Then, for each such *ordinary* type, a value is selected and refined (in the case of functions) so that the inference type of the value matches that type (recall that type splitting relation is defined in Fig. 4). Finally, all these values are merged. For example, $\text{'a'} \,,,\, 1 \,,,\, \mathsf{True} \hookrightarrow_{\mathsf{Bool}\,\&\,\mathsf{Int}} \mathsf{True} \,,,\, 1$.

This process involves treating a merge as a function. For instance, $\mathsf{Int} \to (\mathsf{Int}\,\&\,\mathsf{Bool}) \simeq (\mathsf{Int} \to \mathsf{Int})\,\&\,(\mathsf{Int} \to \mathsf{Bool})$ means that a merge of type $(\mathsf{Int} \to \mathsf{Int})\,\&\,(\mathsf{Int} \to \mathsf{Bool})$ will be generated instead of a value of type $\mathsf{Int} \to (\mathsf{Int}\,\&\,\mathsf{Bool})$ in $v_1 \hookrightarrow_{\mathsf{Int}\to(\mathsf{Int}\,\&\,\mathsf{Bool})} v_2$. To characterize the relationship between the actual type and the desired type, we use *runtime subtyping* in our type preservation theorem, which we will explain later. Without any constraints, the casting rule for ordinary types is non-deterministic: both $1 \,,,\, 2 \hookrightarrow_{\mathsf{Int}} 1$ and $1 \,,,\, 2 \hookrightarrow_{\mathsf{Int}} 2$ are derivable. However, a well-typed cast always produces a unique result.

Theorem 4.7 (Determinism of Casting in λ_*). *If $v : A$ and $A \lesssim B$, then from $v \hookrightarrow_B v_1$ and $v \hookrightarrow_B v_2$ we can derive $v_1 = v_2$.*

As one can observe, our casting always attaches new annotation to function-like values. However, the roundtrip coercion still preserves behavioral equivalence after erasing the type annotations. We define the type annotation erasing function be $[\![e]\!]$, then we have the behavioral equivalence as:

Theorem 4.8 (Behavioral Equivalence of Round-Trip Coercion). *If for all $i \in [1, n]$, $\vdash_r A_i$, and v_{i-1} can be coerced to v_i via $v_{i-1} \hookrightarrow_{A_i} v_i$, then if $A_j = A_k$, then $[\![v_j]\!] = [\![v_k]\!]$.*

4.3 Type Safety

Although each well-typed program has a unique inferred type, this type is not precisely preserved during evaluation. We use the *runtime subtyping relation $A \ll_T B$* to connect the type of the reduced term with that of the redex. The definition and full details of it can be found in the Appendix.

Theorem 4.9 (Preservation of λ_*).

- *If* $\cdot \vdash_t v_1 \Rightarrow A$, $A \lesssim B$, $\vdash_r B$, *and* $v_1 \hookrightarrow_B v_2$, *then* $\exists\, C$, *s.t.* $\cdot \vdash_t v_2 \Rightarrow C$ *and* $C \ll_T B$.
- *If* $\cdot \vdash_t e \Rightarrow A$ *and* $e \hookrightarrow e'$ *then* $\exists A'$, $A' \ll_T A$, *and* $\cdot \vdash_t e' \Rightarrow A'$.
- *If* $\cdot \vdash_t e \Leftarrow A$ *and* $e \hookrightarrow e'$ *then* $\cdot \vdash_t e \Leftarrow A$.

The operational semantics of λ_* is *deterministic* for well-typed expressions.

Theorem 4.10 (Determinism of the Small-Step Reduction in λ_*). *If* $\cdot \vdash_t e \Rightarrow A$ *and* $e \hookrightarrow e_1$ *and* $e \hookrightarrow e_2$ *then* $e_1 = e_2$.

Moreover, our calculus adheres to the principle: *Well-typed programs do not go wrong.* We can establish type soundness through a progress theorem and the above preservation theorems.

Theorem 4.11 (Progress). *If* $\cdot \vdash_t e \Rightarrow A$ *then* e *is a value or* $\exists e'$, $e \hookrightarrow e'$.

Theorem 4.12 (Type Soundness of λ_*). *If* $\cdot \vdash_t e \Leftrightarrow A$ *and* $e \hookrightarrow^* e'$ *then* e' *is a value or* $\exists e''$, $e' \hookrightarrow e''$.

5 The Source Calculus $\lambda_{|*|}$

In this section, we introduce the lambda calculus $\lambda_{|*|}$ together with a type normalization procedure. The target calculus λ_* imposes strong type well-formedness restrictions. Here, we demonstrate that $\lambda_{|*|}$ can model a source calculus without these restrictions. Through type normalization, all well-typed $\lambda_{|*|}$ programs can be translated into well-typed λ_* programs, indicating that λ_*'s type restrictions are not fundamental but merely simplify the design of a calculus with apartness.

5.1 Type Normalization

Though type apartness $A * B$ along with the well-formedness condition $\vdash_r A$ provides a sound calculus λ_*, unrestricted intersection types can be useful in certain cases, as illustrated in Fig. 3. To accommodate such cases, we impose a weaker restriction on types: instead of requiring A and B to be apart directly, we check their normalized forms for apartness. We use type normalization to compute these normalized forms so that any types satisfying apartness specification $A *_s B$ will have their normalized form algorithmically apart. The formulation of type normalization process is not unique, as long as the return type has all of its type components apart and being equivalent to the original type. However, as suggested by Theorems 4.4 and 4.8, the coercion between any two equivalent types is unique, type-safe and behaviour preserving, therefore any type normalization process exhibit no observable difference.

Normalization as a divide-and-conquer process. $|A|_B$ denotes that the type A will be normalized under the type B. A denotes the type we want to normalize, and we want to produce a result that does not overlap with B. B also interacts with the intermediate result during the divide-and-conquer process to avoid unnecessary elimination, which will be shown below. We do the normalization of each component of a type A from left to right. The intuition behind type normalization of A under B is that A can be viewed as a series (or a list) of type components concatenated together with the type operator $\&$. During normalization, we aim to eliminate duplicated components. The process follows a divide-and-conquer strategy by splitting A into two intersection components, A_1 and A_2.

In our algorithm, we first normalize the right-hand side series of components A_2 under $A_1 \& B$, producing an intermediate result C. We then normalize the left-hand side A_1 using the type context $B \& C$. This approach avoids repeated normalization. For instance, given $\mathsf{Int} \& \mathsf{Int}$, we do not normalize the right-hand Int by the left-hand Int and then repeat the process for the reverse. Repeated normalization could erroneously result in $\top \& \top$. By leveraging the intermediate result C, the normalization remains deterministic and avoids such issues. To ensure determinism and avoid reducing to $\top \& \top$, we define specific cases for handling the result C produced from normalizing A_2 under $A_1 \& B$. While the cases differ, they follow the same underlying principles.

$\boxed{\text{Rules of } |A|_B \text{, where we take the normalization result } |A| = |A|_\top}$ *(Type Normalization)*

$$
\begin{array}{rcll}
|\mathsf{Int}|_B &=& \mathsf{Int} & (\text{if not } B <: \mathsf{Int}\ )\\
|\mathsf{Sig}\ l|_B &=& \mathsf{Sig}\ l & (\text{if not } B <: \mathsf{Sig}\ l\ )\\
|\bot|_B &=& \bot & (\text{if not } B <: \bot\ )\\
|A_1 \to A_2|_B &=& |A_1| \to |A_2| & (\text{if } (A_1 \to A_2)^\circ \text{ and not } B <: A_1 \to A_2\ )\\
|A|_B &=& \top & (\text{if } A^\circ \text{ and } B <: A\ )\\
|A|_B &=& |A_1|_{(B \,\&\, C)} \,\&\, C \text{ where } C = |A_2|_{B \,\&\, A_1} & (\text{if } A \simeq A_1 \,\&\, A_2,\ |A_2|_{B \,\&\, A_1} \neq \top \text{ and } |A_1|_{B \,\&\, C} \neq \top\\
|A|_B &=& |A_2|_{B \,\&\, A_1} & (\text{if } A \simeq A_1 \,\&\, A_2,\ |A_2|_{B \,\&\, A_1} \neq \top \text{and } |A_1|_{B \,\&\, (|A_2|_{B \,\&\, A_1})} = \top\ )\\
|A|_B &=& |A_1|_B & (\text{if } A \simeq A_1 \,\&\, A_2,\ |A_2|_{B \,\&\, A_1} = \top)
\end{array}
$$

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

$$\frac{\Gamma \vdash e_1 \Rightarrow A \qquad |B| \vdash |A| \rhd C \qquad \Gamma \vdash e_2 \Rightarrow B}{\Gamma \vdash e_1\, e_2 \Rightarrow C}$$

$$\frac{\Gamma \vdash e \Rightarrow A \qquad |A| \lesssim |B|}{\Gamma \vdash e \Leftarrow B}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B \qquad |A| * |B|}{\Gamma \vdash e_1 \,,, e_2 \Rightarrow A \,\&\, B}$$

$\boxed{e \hookrightarrow_S e'}$ *(Source Reduction)*

$$\frac{v \hookrightarrow_{|A|} v'}{\lambda^{A \to B} x.\, e\, v \hookrightarrow_S (e[x \mapsto v']) : |B|}$$

$$\frac{v_2 \hookrightarrow_{|A|} v_2'}{(u_1 : A \to B)\, v_2 \hookrightarrow_S (u_1\, v_2') : |B|}$$

$$\frac{\neg(\mathit{value}\ v : A) \qquad v \hookrightarrow_{|A|} v'}{v : A \hookrightarrow_S v'}$$

Fig. 10. Type normalization, (selected) typing and reduction rules for $\lambda_{|*|}$.

For example, consider normalizing Int & (Bool & Int) under ⊤. The process proceeds as follows:

$$
\begin{array}{rcr}
|\mathsf{Int}|_{(\top \,\&\, \mathsf{Int}) \,\&\, \mathsf{Bool}} = \top & \neg(\top \,\&\, \mathsf{Bool} <: \mathsf{Int}) & (1)\\
|\mathsf{Bool}|_{(\top \,\&\, \mathsf{Int}) \,\&\, \mathsf{Int}} = \mathsf{Bool} & \neg(\top \,\&\, \mathsf{Int} <: \mathsf{Bool}) & (2)\\
|\mathsf{Bool} \,\&\, \mathsf{Int}|_{\top \,\&\, \mathsf{Int}} = \mathsf{Bool} & (1) \textit{ and } (2) & (3)\\
|\mathsf{Int}|_{\top \,\&\, \mathsf{Bool}} = \mathsf{Int} & \neg(\top \,\&\, \mathsf{Bool} <: \mathsf{Int}) & (4)\\
|\mathsf{Int} \,\&\, (\mathsf{Bool} \,\&\, \mathsf{Int})|_\top = \mathsf{Int} \,\&\, \mathsf{Bool} & (3) \textit{ and } (4) & (5)
\end{array}
$$

The final result is Int & Bool, demonstrating how the process avoids ambiguity and ensures a deterministic outcome. This divide-and-conquer approach operates bottom up. At step (5), we aim to normalize Int & (Bool & Int) under type ⊤. To do so, we split Int & (Bool & Int) and normalize both Int and Bool & Int separately. Following a similar strategy, we observe that at step (3), Bool & Int normalizes to Bool under ⊤ & Int, while Int normalizes to Int under ⊤ & Bool. As a result, after normalization, we obtain Int & Bool. This approach is essential for achieving complete and sound normalization while preserving determinism.

Normalization starts from an empty context (⊤). The operation $|A|_\top = C$ takes A as input and ⊤ as context, producing C as the normalized form of A. We use $|A|$ to denote the result of $|A|_\top$. The normalization process is idempotent, complete, deterministic, and sound.

Theorem 5.1 (Normalization is Idempotent). $||A|_B|_B = |A|_B$.

Theorem 5.2 (Totality of Normalization). *For all A B, there exists C such that $|A|_B = C$.*

Theorem 5.3 (Determinism of Normalization). *$|A|_B$ results in at most one C.*

Theorem 5.4 (Soundness of Normalization). *For all A, we have $A \leq |A|$ and $|A| \leq A$.*

Theorem 5.5 (Soundness of Guarded Subtyping). *For all A B, if $A \leq B$ and $B \leq A$, then $|A| \lesssim |B|$.*

The normalization procedure provides another way to show the completeness of apartness, since normalization eliminates shadowed components in types, as suggested in Section 3.3.

Theorem 5.6 (Completeness of Apartness). *For all A B, $|A| * |B|$ iff $A *_s B$.*

5.2 Source Typing and Operational Semantics

Source typing is not very different from the target typing of λ_*. The key difference is that we check whether the normalized result type checks. We can perform a weaker check on all the types as long as their normalized form satisfies the constraints, instead of requiring types to be well-formed and then checking the constraints. For instance, $(\mathsf{String} \to \mathsf{Bool}) \,\&\, ((\mathsf{String} \,\&\, \mathsf{Int}) \to \mathsf{Bool})$ is not mergeable with $\mathsf{Int} \to \mathsf{Bool}$ in λ_*. However, that type is mergeable in $\lambda_{|*|}$ since the normalization process erases the $(\mathsf{String} \,\&\, \mathsf{Int}) \to \mathsf{Bool}$ component during the normalization process. Likewise, we do the normalization in the reduction of $\lambda_{|*|}$. The only difference in the source reduction semantics with λ_* reduction is that, we do a type normalization whenever casting occurs.

After carefully dealing with the normalized component in typing and evaluation, we can setup the determinism and soundness properties of $\lambda_{|*|}$. Here the runtime subtyping relation $A' \ll A$, is used for preservation of the source calculus, and works just as $A' \ll_T A$ in λ_*.

Theorem 5.7 (Determinism). *If $\cdot \vdash e \Rightarrow A$ and $e \hookrightarrow_S e_1$ and $e \hookrightarrow_S e_2$ then $e_1 = e_2$.*

Theorem 5.8 (Progress). *If $\cdot \vdash e \Rightarrow A$ then e is a value or $\exists e'$, $e \hookrightarrow_S e'$.*

Theorem 5.9 (Preservation).
- *If $\cdot \vdash e \Rightarrow A$ and $e \hookrightarrow_S e'$ then $\exists A'$, $A' \ll A$, and $\cdot \vdash e' \Rightarrow A'$.*
- *If $\cdot \vdash e \Leftarrow A$ and $e \hookrightarrow_S e'$ then $\cdot \vdash e \Leftarrow A$.*

5.3 Translation to Target and Operational Correspondence

Before establishing a value-to-value mapping between the source calculus and the target calculus, we first define the relationship between well-formedness and type normalization. Specifically, well-formedness $\vdash_s A$ corresponds directly to normalization.

The source calculus permits more expressive terms—such as $1 \,,,\, \top \,,,\, \top$ or $\lambda^{\mathsf{Int} \to \mathsf{Int} \,\&\, \mathsf{Int}} x.\, e$—that would be rejected by the more restrictive type system of the target calculus. To bridge this gap, we introduce a normalization process and design the target type system of λ_* to accept all normalized source types as well-formed. This approach enables a smooth and consistent translation from source to target, freeing us from the rigid constraints of λ_*. Moreover, assuming that all type annotations are normalized during translation into λ_*, the reduction rules in the target calculus can be simplified. For example, rules like rule StepN-annov no longer require runtime type normalization (like in $\lambda_{|*|}$), simplifying evaluation in the target language.

Theorem 5.10 (Equivalence of Normalization and Well-Formedness). *$\vdash_s A$ iff $|A| = A$.*

We are now ready to establish the correspondence between $\lambda_{|*|}$ and λ_*. To incorporate type normalization into type annotations in $\lambda_{|*|}$, we introduce a term normalization process, denoted $|e|$, which translates type annotations into normalized types. Specifically, during translation, $|e|$ normalizes all type annotations: non-arrow types A become $|A|$, and arrow types $A \to B$ become $|A| \to |B|$. For instance, the term $e : (\mathsf{Int} \,\&\, \mathsf{Int})$ is normalized to $|e| : \mathsf{Int}$, while $\lambda^{A \to B \,\&\, C} x.\, e$ is normalized to $\lambda^{|A| \to |B \,\&\, C|} x.\, e$. so that a value-to-value mapping from $\lambda_{|*|}$ to λ_* is preserved, while the full details are in the Appendix. With term normalization in place, we can now formally state and prove the soundness of both typing and evaluation:

Theorem 5.11 (Type-Safety of Translation). *If $\cdot \vdash e \Rightarrow A$ then $\exists B$, $\cdot \vdash_t |e| \Rightarrow B$ and $|B| = |A|$.*

Theorem 5.12 (Operational Correspondence). *If $e \hookrightarrow_S^* e'$ then $|e| \hookrightarrow_T^* |e'|$.*

6 Type Difference

A key consequence of using apartness instead of disjointness is that type difference [Xu et al. 2023] becomes a *total* operation rather than a *partial* one. This section defines type difference in terms of apartness and proves its totality.

6.1 Partial Type Difference with Disjointness

Type difference, introduced by Xu et al. [2023], provides a way to resolve conflicts in merges with disjoint intersection types. The idea behind type difference, denoted as $A \backslash B$, is to remove conflicting type components from A, allowing the resolution of a conflicting merge of type $A \,\&\, B$ by rewriting it as $(A \backslash B) \,\&\, B$. The following examples illustrate how type difference operates.

$$\begin{array}{ll}(\mathsf{Int} \to \mathsf{Bool}) \backslash (\mathsf{Int} \to \mathsf{Bool}) \;=\; \top & (\mathsf{Int} \to \mathsf{Bool}) \backslash (\mathsf{Bool} \to \mathsf{String}) \;=\; \mathsf{Int} \to \mathsf{Bool} \\ (\mathsf{Int} \to \mathsf{Bool}) \backslash \mathsf{Int} \;=\; \mathsf{Int} \to \mathsf{Bool} & (\mathsf{Int} \to \mathsf{String}) \backslash (\mathsf{Bool} \to \mathsf{String}) \;=\; \textbf{undefined}\end{array}$$

Type difference eliminates overlapping components of A with respect to B. For instance, $\mathsf{Int} \to \mathsf{Bool}$ is completely removed when B is $\mathsf{Int} \to \mathsf{Bool}$ (or a subtype). However, when subtracting Int, the function type $\mathsf{Int} \to \mathsf{Bool}$ remains unchanged because Int does not overlap with it.

More interesting cases arise with function types. In $(\mathsf{Int} \to \mathsf{Bool}) \backslash (\mathsf{Bool} \to \mathsf{String})$, subtraction has no effect on $\mathsf{Int} \to \mathsf{Bool}$ since the two types are disjoint. However, subtracting $\mathsf{Bool} \to \mathsf{String}$ from $\mathsf{Int} \to \mathsf{String}$ presents a conflict: $\mathsf{Int} \to \mathsf{String}$ is neither fully shadowed by $\mathsf{Bool} \to \mathsf{String}$ nor completely disjoint from it. Removing $\mathsf{Int} \to \mathsf{String}$ would result in information loss, while keeping it would leave the conflict unresolved.

Despite its expressive power, type difference remains a partial operation since it cannot resolve conflicts between types like $\mathsf{Int} \to \mathsf{String}$ and $\mathsf{Bool} \to \mathsf{String}$. Our observation is that type difference is tightly coupled with the disjointness relation. The previous definition of disjointness may be too strict in identifying conflicts. As we will show, apartness allows us to define a total type difference.

6.2 Total Type Difference with Apartness

Interestingly, type difference in our setting actually corresponds to the type normalization process, and can be specified as a variant of apartness. Recall the apartness specification in Sec. 2.4.

Definition 2.4 (Apartness Specification). $A *_s B \triangleq \forall C_1\ C_2$, if $\mathsf{MinOrd}\ C_1\ A \wedge \mathsf{MinOrd}\ C_2\ B$, then either $C_1 = \top$ or $C_2 = \top$ or $(\neg(C_1 \leq C_2) \wedge \neg(C_2 \leq C_1))$.

Type normalization cannot resolve all the conflicts inside type A through the process $|A|_B$. We can glance through a few examples next:

$$\begin{array}{llll}|\mathsf{Int}|_{\mathsf{Int}} = \top & (1) & |\mathsf{Int} \to \mathsf{Bool}|_{\mathsf{String} \to \mathsf{Bool}} = \mathsf{Int} \to \mathsf{Bool} & (2) \\ |\mathsf{Int} \,\&\, \mathsf{String}|_{\mathsf{Int}} = \mathsf{String} & (3) & |\mathsf{Int} \to \mathsf{Bool}|_{(\mathsf{Int}\,\&\,\mathsf{String}) \to \mathsf{Bool}} = \mathsf{Int} \to \mathsf{Bool} & (4)\end{array}$$

As we can see, the normalization process resolves conflicts well in the first three examples, as we have the normalization result $|A|_B$ to be apart from B. We can use the merge of $|A|_B$ and B to replace the ambiguous merge A with B. But in the fourth case, the conflict is still not resolved as $\mathsf{Int} \to \mathsf{Bool}$ shadows $(\mathsf{Int} \,\&\, \mathsf{String}) \to \mathsf{Bool}$. Under the normalization process, supertype $(\mathsf{Int} \,\&\, \mathsf{String}) \to \mathsf{Bool}$ cannot normalize $\mathsf{Int} \to \mathsf{Bool}$, thus the conflict remains unsolved after the normalization process.

A closer glance at normalization shows that it removes all the conflicting components of type A that are shadowed by type B. In some sense, it is a one-sided conflict resolution that cleans things only in type A. Based on this observation, we can define the notion of *half* apartness as:

Definition 6.1 (Half Apart Specification). $A \lessdot\!\!\ast\, B \triangleq \forall C_1\ C_2$, if $\mathsf{MinOrd}\ C_1\ A \wedge \mathsf{MinOrd}\ C_2\ B$, then $C_2 = \top \vee \neg(C_1 \leq C_2)$.

which suggests that every component in the type B is not shadowed by the components in A, representing a one-side apartness. Half apartness $A \lessdot\!\!* B$ represents exactly the asymmetric variant of apartness, as we can set up the equivalence of two variants of apartness as follows:

Theorem 6.2 (Apartness in terms of Half-Apartness). *$A \lessdot\!\!* B$ and $B \lessdot\!\!* A$, iff $A *_s B$.*

During type normalization $|A|_B = C$, only type A is normalized while type B only serves as contextual information guiding through the process. Thus, we can now set up the type normalization exactly as the type difference specification [Xu et al. 2023] based on our one-sided apartness $A \lessdot\!\!* B$:

Definition 6.3 (Type Difference Specification). $A \backslash_s B = C \triangleq A \leq C \ \wedge\ B \,\&\, C \leq A \ \wedge\ B \lessdot\!\!* C$.

Furthermore, the type normalization process directly gives us an algorithmic formulation for this type difference specification:

Theorem 6.4 (Type Difference as Type Normalization).
- *$A \backslash_s B = |A|_B$.*
- *For all C, if $A \backslash_s B = C$, then $C \leq |A|_B$ and $|A|_B \leq C$.*

The two properties imply that results produced by the type difference of type A and B are equivalent to $|A|_B$. Thus we can use type normalization as an algorithm to compute type difference. Additionally, type normalization process is total: we are *always* able to resolve conflicts in two arbitrary types. This contrasts with the prior formulation disjointness-based type difference. Due to half apartness, we have to do the type difference twice (on both A and B), to resolve conflicts on two types.

Theorem 6.5 (Conflict Resolution). *if $|A|_B = C$ and $|B|_C = D$, then $C *_s D$.*

To resolve conflicts in A and B, we first do the normalization $|A|_B = C$ to get the conflict-free version of A (i.e. C). Then we do the normalization $|B|_C = D$ to get the conflict-free version of B (i.e. D). Finally, we know that the resulting conflict-free types are apart $C * D$. We do not lose information as we have $A \,\&\, B \leq C \,\&\, D$ and $C \,\&\, D \leq A \,\&\, B$ from the type difference specification.

7 Related Work

Function Overloading and the Merge Operator. Function overloading (or method overloading) is the ability to create multiple functions of the same name with different implementations. In our context, function overloading is usually achieved by the merge operator to create an overloaded function. The calculus $\lambda\&$ proposed by Castagna et al. [1995] has a restricted version of the merge operator for functions only. The merge operator is indexed by a list of types of its components. Its operational semantics uses the runtime types of values to select the "best approximate" branch of an overloaded function, but $\lambda\&$ requires runtime subtype checking to support such mechanisms.

Dunfield [2014] presented the first calculus with an unrestricted merge operator, based on Reynolds [1997]'s merge operator. Though her calculus is powerful and supports function overloading, it lacks determinism and type preservation. Xue et al. [2022] presents a calculus with a merge operator with preservation, with all 4 features discussed in Sec. 2.2. However, Xue et al. [2022]'s calculus is still non-deterministic. Other calculi with intersection types and function overloading have been proposed [Castagna et al. 2015, 2014; Castagna and Xu 2011], but these calculi do not support a merge operator, and thus avoid the ambiguity problems caused by merges.

Disjoint Intersection Types. Oliveira et al. [2016] proposed the λ_i calculus, which only allows terms of disjoint types to be merged. The goal of the disjointness restriction was to address the ambiguity in Dunfield [2014]'s calculus and to create a deterministic calculus. Various extensions [Alpuim et al. 2017; Bi and Oliveira 2018; Bi et al. 2018] were later proposed to strengthen λ_i with relaxed restrictions, nested composition and disjoint polymorphism.

Huang and Oliveira [2020] propose a type-directed operational semantics (TDOS) to specify the semantics of calculi with disjoint intersection types and a merge operator. TDOS employs parallel application to support merging multiple functions and applying them simultaneously to a single input. Additionally, TDOS does not require runtime subtype checking and allows merging and applying values of types beyond just function types. Our work also employs TDOS, leveraging apartness to allow certain ambiguous types. This relaxation enables the merging of function types such as Int → Bool and String → Bool, thereby supporting overloading—previously prohibited under strict disjointness conditions. Moreover, by combining overloading with first-class labels, we achieve a lightweight encoding of records [Castagna et al. 1995].

$F_{\bowtie}$ [Rioux et al. 2023] is a powerful calculus that supports deterministic merges. $F_{\bowtie}$ employs two dual disjointness relations, called *mergeable* and *distinguishable*, to empower the merge operator not only with intersection types but also union types. However, such expressiveness comes with trade-offs. In $F_{\bowtie}$ merges of values with primitive types, such as Int and Bool, are not allowed. As a consequence of this restriction, return type overloading is not supported either. Thus, we are not able to merge String → Int and String → Bool together, even though values of those types can always be disambiguated. In contrast, our $\lambda_{|*|}$ and λ_{*} calculi do not support union types, but support merges of primitive types, as well as return type overloading.

Conflict Resolution in OOP and the Merge Operator. In OOP, mixin classes [Ancona et al. 2003; Bracha and Cook 1990; Duggan and Sourelis 1996; Flatt et al. 1998] and traits [Fisher and Reppy 2004; Schärli et al. 2003] are two composition mechanisms for code reuse and multiple inheritance. Name conflicts are a common problem for both the merge operator and OOP. Usually, the mixin model allows overlapping fields and omits the one with lower priority. Instead, traits only allow no conflicting merges. Conflicts have to be resolved before composition by renaming or restriction.

Compositional Programming [Zhang et al. 2021] is a recently proposed modular programming paradigm based on first-class traits [Bi and Oliveira 2018], and implemented by the CP language. CP is built on top of a core calculus with disjoint intersection types and the merge operator [Fan et al. 2022]. In CP, parallel application is employed to support merging multiple functions and apply them together to one input. The use of the merge operator in CP allows it to naturally solve the Expression Problem [Wadler 1998] and offers modular pattern matching and dependency injection. Ye et al. [2024] later extend CP to imperative computational effects.

Type difference [Xu et al. 2023] was proposed to resolve conflicts in merges in settings with disjoint intersection types. Our work uses apartness instead of disjointness, and sets up type difference as type normalization. Type normalization under apartness not only retains all of the soundness and completeness properties, but also achieves totality. This enables resolving conflicts in all terms to be merged.

8 Conclusion

In this paper, we introduce *type apartness*, a weaker form of disjointness, for calculi with intersection types and a merge operator. Apartness preserves type soundness and determinism, while enabling key features: *function overloading*, *return type overloading*, *extensible records*, and *nested composition*. We develop two calculi, $\lambda_{|*|}$ and λ_{*}, which incorporate apartness and guarded subtyping. We establish connection between them via *type normalization*, which also establishes a total type difference operator [Xu et al. 2023] within the apartness framework. Besides, all the properties and theorems are verified in Coq. Future work will explore extending apartness notion to other type system constructs, such as union types [Barbanera et al. 1995] and disjoint polymorphism [Alpuim et al. 2017], broadening its applicability.

Data-Availability Statement

The companion artifact [Xu et al. 2025] includes the Coq development and the appendix.

Acknowledgments

Xuejing Huang acknowledges support from the MathInGreaterParis Fellowship Programme, funded by the HORIZON EUROPE Marie Sklodowska-Curie Actions and the Fondation Sciences Mathtématiques de Paris.

References

João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. 2017. Disjoint Polymorphism. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1145/1391289.1391293

Davide Ancona, Giovanni Lagorio, and Elena Zucca. 2003. Jam—designing a Java extension with mixins. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25, 5 (2003), 641–712. doi:10.1145/937563.937567

Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. 1995. Intersection and Union Types: Syntax and Semantics. *Information and Computation* 119, 2 (June 1995), 202–230.

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A Filter Lambda Model and the Completeness of Type Assignment. *The Journal of Symbolic Logic* 48, 4 (1983), 931–940. http://www.jstor.org/stable/2273659

Xuan Bi and Bruno C. d. S. Oliveira. 2018. Typed First-Class Traits. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. doi:10.4230/LIPIcs.ECOOP.2018.9

Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2018. The Essence of Nested Composition. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands (LIPIcs, Vol. 109)*, Todd D. Millstein (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:33. doi:10.4230/LIPIcs.ECOOP.2018.22

Gilad Bracha and William Cook. 1990. Mixin-based inheritance. *ACM Sigplan Notices* 25, 10 (1990), 303–311. doi:10.1145/97945.97982

Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. 1995. A Calculus for Overloaded Functions with Subtyping. *Information and Computation* 117, 1, 115–135. doi:10.1006/inco.1995.1033

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. 2015. Polymorphic Functions with Set-Theoretic Types: Part 2: Local Type Inference and Type Reconstruction. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 289–302. doi:10.1145/2676726.2676991

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. 2014. Polymorphic Functions with Set-Theoretic Types: Part 1: Syntax, Semantics, and Evaluation. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 5–17. doi:10.1145/2535838.2535840

Giuseppe Castagna and Zhiwu Xu. 2011. Set-Theoretic Foundation of Parametric Polymorphism and Subtyping. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 94–106. doi:10.1145/2034773.2034788

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. doi:10.1145/351240.351259

Dominic Duggan and Constantinos Sourelis. 1996. Mixin modules. *ACM SIGPLAN Notices* 31, 6 (1996), 262–273.

Jana Dunfield. 2014. Elaborating intersection and union types. *Journal of Functional Programming (JFP)* 24, 2-3 (2014), 133–165. doi:10.1006/inco.1995.1086

Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5, Article 98 (May 2021), 38 pages. doi:10.1145/3450952

Erik Ernst. 2001. Family polymorphism. In *European Conference on Object-Oriented Programming*. Springer, 303–326.

Andong Fan, Xuejing Huang, Han Xu, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2022. Direct Foundations for Compositional Programming. 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, 18:1–18:28. doi:10.4230/LIPIcs.ECOOP.2022.18

Kathleen Fisher and John Reppy. 2004. A typed calculus of traits. In *Electronic proceedings of FOOL*, Vol. 2004.

Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1998. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 171–183. doi:10.1145/268946.268961

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic Subtyping: Dealing Set-Theoretically with Function, Union, Intersection, and Negation Types. *J. ACM* 55, 4, Article 19 (Sept. 2008), 64 pages. doi:10.1145/1391289.

1391293

Xuejing Huang and Bruno C. d. S. Oliveira. 2020. A type-directed operational semantics for a calculus with a merge operator. In *The 34th European Conference on Object-Oriented Programming (ECOOP 2020*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik GmbH. The Journal's web ....

Xuejing Huang, Jinxu Zhao, and Bruno C. d. S. Oliveira. 2021. Taming the Merge Operator. 31 (2021). doi:10.1017/S0956796821000186 Publisher: Cambridge University Press.

Daan Leijen. 2004. *First-class labels for extensible rows* (technical report uu-cs-2004-51 ed.). Technical Report UU-CS-2004-51. https://www.microsoft.com/en-us/research/publication/first-class-labels-for-extensible-rows/ UTCS Technical Report.

Barbara H. Liskov and Jeannette M. Wing. 1994. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (Nov. 1994).

Zhaohui Luo. 1999. Coercive Subtyping. *J. Log. Comput.* 9, 1 (1999), 105–130. doi:10.1093/logcom/9.1.105

Koar Marntirosian, Tom Schrijvers, Bruno C. d. S. Oliveira, and Georgios Karachalias. 2020. Resolution as Intersection Subtyping via Modus Ponens. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 206 (nov 2020). doi:10.1145/3428274

Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint intersection types. In *International Conference on Functional Programming (ICFP)*. doi:10.1145/2951913.2951945

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (jan 2000), 1–44. doi:10.1145/345099.345100

John C Reynolds. 1988. *Preliminary design of the programming language Forsythe.* Technical Report. Carnegie Mellon University.

John C. Reynolds. 1991. The coherence of languages with intersection types. In *Lecture Notes in Computer Science (LNCS)*. Springer Berlin Heidelberg, 675–700.

John C Reynolds. 1997. Design of the programming language Forsythe. In *ALGOL-like languages*. 173–233.

John C Reynolds. 1998. *Theories of programming languages*. Cambridge University Press.

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 $F_{\bowtie}$. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 515–543.

Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. 2003. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming (ECOOP)*.

Jeremy Siek and Walid Taha. 2007. Gradual Typing for Objects. In *Proceedings of the 21st European Conference on ECOOP 2007: Object-Oriented Programming* (Berlin, Germany) *(ECOOP '07)*. Springer-Verlag, Berlin, Heidelberg, 2–27. doi:10.1007/978-3-540-73589-2_2

Jinhao Tan and Bruno C d S Oliveira. 2023. Dependent Merges and First-Class Environments. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Philip Wadler. 1998. The expression problem. *Java-genericity mailing list* (1998).

Philip Wadler and Robert Bruce Findler. 2009. Well-typed programs can't be blamed. In *European Symposium on Programming*. Springer, 1–16.

Yanlin Wang, Haoyuan Zhang, Bruno C. d. S. Oliveira, and Marco Servetto. 2018. FHJ: A Formal Model for Hierarchical Dispatching and Overriding. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 109)*, Todd Millstein (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 20:1–20:30. doi:10.4230/LIPIcs.ECOOP.2018.20

Han Xu, Xuejing Huang, and Bruno C. d. S. Oliveira. 2023. Making a Type Difference: Subtraction on Intersection Types as Generalized Record Operations. *Proc. ACM Program. Lang.* 7, POPL, Article 31 (jan 2023), 28 pages. doi:10.1145/3571224

Han Xu, Xuejing Huang, and Bruno C. d. S. Oliveira. 2025. Liberating Merges via Apartness and Guarded Subtyping (Artifact). doi:10.5281/zenodo.16921686

Xu Xue, Bruno C. d. S. Oliveira, and Ningning Xie. 2022. Applicative Intersection Types. In *Programming Languages and Systems*, Ilya Sergey (Ed.). Springer Nature Switzerland, Cham, 155–174.

Wenjia Ye, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2024. Imperative Compositional Programming: Type Sound Distributive Intersection Subtyping with References via Bidirectional Typing. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 342 (Oct. 2024), 30 pages. doi:10.1145/3689782

Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2021. Compositional Programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43, 3 (2021), 1–61. doi:10.1145/3460228

Received 2024-10-15; accepted 2025-08-12