Making a Type Difference

Subtraction on Intersection Types as Generalized Record Operations

HAN XU*, Peking University, China
XUEJING HUANG and BRUNO C. D. S. OLIVEIRA, The University of Hong Kong, China

In programming languages with records, objects, or traits, it is common to have operators that allow dropping, updating or renaming some components. These operators are useful for programmers to explicitly deal with conflicts and override or update some components. While such operators have been studied for record types, little work has been done to generalize and study their theory for other types.

This paper shows that, given subtyping and disjointness relations, we can specify and derive algorithmic implementations for a general type difference operator that works for other types, including function types, record types and intersection types. When defined in this way, the type difference algebra has many desired properties that are expected from a subtraction operator. Together with a generic *merge* operator, using type difference we can generalize many operations on records formalized in the literature. To illustrate the usefulness of type difference we create an intermediate calculus with a rich set of operators on expressions of arbitrary type, and demonstrate applications of these operators in *CP*, a prototype language for *Compositional Programming*. The semantics of the calculus is given by elaborating into a calculus with *disjoint intersection types* and a merge operator. We have implemented type difference and all the operators in the CP language. Moreover, all the calculi and related proofs are mechanically formalized in the Coq theorem prover.

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. 2023. Making a Type Difference: Subtraction on Intersection Types as Generalized Record Operations. *Proc. ACM Program. Lang.* 7, POPL, Article 31 (January 2023), 32 pages. https://doi.org/10.1145/3571224

1 INTRODUCTION

In programming languages with records, objects, or traits [Schärli et al. 2003], it is common to have restriction operators and other derived operators that allow dropping or updating some components. Cardelli and Mitchell [1991] did a comprehensive study of *operations on records*, identifying a variety of record operators in a calculus with records, subtyping and *bounded quantification* [Cardelli et al. 1994]. Record operators have also been extensively studied in various calculi with *row polymorphism* [Wand 1989], though typically in settings without subtyping. Several existing functional programming languages, including Haskell and OCaml, include some of these record operators. Moreover, object-oriented languages with traits or mixins, such as the Pharo language [Tesone et al. 2020] and the Jigsaw framework [Bracha 1992; Lagorio et al. 2009], also include operations for renaming and removing methods to aid with the resolution of conflicts.

*The work was conducted as part of a research internship funded by the University of Hong Kong.

Authors' addresses: Han Xu, Peking University, Beijing, China, 1800012917@pku.edu.cn; Xuejing Huang, xjhuang@cs.hku.hk; Bruno C. d. S. Oliveira, bruno@cs.hku.hk, The University of Hong Kong, Hong Kong, China.

2475-1421/2023/1-ART31
https://doi.org/10.1145/3571224

While for records and record types such operators are widely studied, there is little work on generalizing and studying the theory of such operators for other types. Nonetheless there are two lines of work that support some form of operators that deal with types other than records. In the field of *semantic subtyping* [Castagna and Frisch 2005; Frisch et al. 2008] there is a related notion that arises from set difference: we can interpret two type A and B as the sets of values that have type A and B. Thus, under such interpretation, we can obtain a new set of values by employing set difference $A \backslash B$. The resulting set denotes a type with the values which are in A but not in B. However the semantics of set difference is not what we want: set difference does not generalize record restriction and it has different applications. Shields and Meijer [2001] studied a calculus with *type-indexed rows*, where types are used instead of labels for indexing. The calculus supports a simple restriction operation, but lacks some desirable properties and does not have subtyping.

This paper studies a general notion of type difference, which enables a generalization of record restriction. Although type difference is a partial function that could fail in some cases, it poses no restriction on types that can be subtracted in general. If type difference computes a result, a number of useful properties are guaranteed, allowing values of the result type to be safely merged with other values. Type difference is specified by an intuitive definition based on subtyping and disjointness relations, resulting in a general type difference operator that works for function types, record types and intersection types. When defined in this way, type difference has a rich algebra with many useful properties that are expected from a subtraction operator.

Together with a generic disjoint merge operator [Oliveira et al. 2016], using type difference we can generalize many record operators in the literature. There are two main primitive operations that can be defined with type difference. The expression $e \backslash A$ means that the information of type A is dropped from the expression e. The related expression $e_1 \backslash e_2$ means that the information in the minimal type of e_2 is dropped from e_1. With those two primitive forms of expressions, several operators can be defined as syntactic sugar. These operators include *updates*, *biased merges*, as well as *renaming* operators (for the particular case of records). We show that all the record operators proposed by Cardelli and Mitchell are encodable with type difference and a merge operator.

To illustrate the usefulness of type difference we create an intermediate polymorphic calculus, called $\mathsf{F}_i^{\backslash}$, with a rich set operators on types. The key feature of $\mathsf{F}_i^{\backslash}$ is a type-difference operation between two types. The semantics of the calculus is given by an elaboration (a type-directed translation) into a calculus with *disjoint intersection types* [Oliveira et al. 2016] with a symmetric merge operator. The elaboration is proved to be *deterministic* and *type-safe*. The elaboration also illustrates that, although type difference is convenient and enables encodings of many operators, all the uses of type difference are encodable in terms of merges and explicit type annotations.

An important point to provide the correct intuition for type difference in our work is that the interpretation of subtyping in calculi with the merge operator and disjoint intersection types is *coercive* [Luo 1999; Reynolds 1991], rather than set-theoretic. Assume that the set of values that inhabit type A is denoted by $[\![A]\!]$. In semantic subtyping and the set-theoretic interpretation, subtyping is interpreted as set inclusion: A is a subtype of B means that $[\![A]\!]$ is a subset of $[\![B]\!]$. So it considers a subtype to be *smaller* than its supertype. Differently, in a coercive interpretation of subtyping, $A \leq B$ implies the existence of a monomorphism from $[\![A]\!]$ to $[\![B]\!]$, and leads to an injective coercion function to convert values. From any value of a subtype, a value of its supertype can be obtained. Therefore, we consider a subtype *larger* than its supertype. For instance, in the set-theoretic interpretation for the top type, $[\![\top]\!]$ is the set of all possible values. In contrast, in the coercive interpretation, $[\![\top]\!]$ is the set with a single value. In other words, $\top$ is interpreted as the unit type, instead of the type that can be assigned to all values. Since a value of the type $A \backslash B$ can be obtained from any value of A, we expect $A <: A \backslash B$. In the coercive interpretation, the type $A \backslash B$ contains less information than A, justifying its interpretation as a form of difference.

We have implemented type difference and all the operators presented in this paper in the CP language. In CP type difference is useful for implementing a variety of operations on traits, which allow for conflict resolution when inheriting from multiple traits and for renaming components of traits. We also illustrate that in CP the generalized version of the operators works even when the expression being updated has a non-record type. In particular, the generalized operators are useful to update overloaded functions. All the calculi and proofs in this paper are mechanically formalized in the Coq theorem prover. In summary, our contributions are:

- **Type difference:** We formalize the idea of information difference on types in terms of subtyping and type disjointness. Type difference enables operators that erase portions of a value that overlap with some type. Type difference works in calculi that include a general merge operator [Dunfield 2014; Oliveira et al. 2016], and generalizes record restriction [Cardelli and Mitchell 1991].
- **Algorithmic type difference:** We study two concrete formulations of type difference for languages with intersection types. The simpler formulation is for the subtyping of Barendregt, Coppo, and Dezani-Ciancaglini (BCD), extended with bottom and record types. The second formulation considers a richer subtyping relation with *disjoint polymorphism* [Alpuim et al. 2017]. Our two algorithmic formulations of type difference are sound and complete with respect to our specification of type difference.
- **Encodings for a rich set of operators:** We show that with type difference we can define two primitive operations $e\backslash A$ and $e_1\backslash e_2$. These enable encoding a variety of generic operators that include *updates*, *biased merges*, as well as *renaming* operators (for the particular case of records). All of Cardelli and Mitchell's record operators are encodable with our operators.
- **The $\mathsf{F}_i^{\setminus}$ calculus:** an intermediate calculus with disjoint polymorphism and a rich set of operators. The $\mathsf{F}_i^{\setminus}$ calculus is defined by a type-safe elaboration semantics into F_i^{+} [Bi et al. 2019; Fan et al. 2022]. The elaboration shows that the set of operators in $\mathsf{F}_i^{\setminus}$ is essentially a sophisticated form of (type-directed) syntactic sugar via type difference in terms of the primitive merge operator in F_i^{+}.
- **Mechanical formalization and implementation:** We modified the CP language (https://plground.org) to implement all the features of the $\mathsf{F}_i^{\setminus}$ calculus and we use CP to illustrate applications of type difference and various operators in this paper (online demonstration at https://plground.org/typediff/cipher). The $\mathsf{F}_i^{\setminus}$ calculus, its elaboration to F_i^{+}, and all the properties for type difference described in this paper are formalized in Coq. The Coq development and the appendix of this paper are included in the companion artifact [Xu et al. 2022].

2 OVERVIEW

This section covers background on the merge operator, disjoint intersection types and CP. We motivate type difference using CP examples, discuss why our work is beyond the scope of existing approaches and describe our key ideas. Formal definitions are deferred to Sections 3 and 4.

2.1 Background

A general type difference makes sense in calculi with a generic merge operator [Dunfield 2014] that merges two arbitrary expressions together. Thus we first review calculi with such a merge operator, and the CP language which is designed on top of such calculi.

Disjoint Intersection Types and the Merge Operator. Intersection types were introduced in the seventies in Curry-style lambda calculi [Barendregt et al. 1983; Coppo and Dezani-Ciancaglini 1978; Coppo et al. 1981; Pottinger 1980]. By composing two types with the constructor $\&$, an intersection type $A \,\&\, B$ specifies terms that have both type A and B. Although the type is not parametric, it is polymorphic, providing a way to refine the characterization of terms. A corresponding term-level constructor merge $(,,)$ was proposed by Reynolds [1988, 1997] for the language Forsythe, and later

refined by Dunfield [2014]. When two terms e_1 and e_2 of type A_1 and A_2 are merged, the whole term $e_1 \,,, e_2$ has an intersection type $A_1 \,\&\, A_2$. For example, a function of type $\mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{Bool} \,\&\, \mathsf{Int}$ is:

$$\lambda x : \mathsf{Int} \,\&\, \mathsf{Bool}.\ (\mathsf{not}\ x) \,,, (x + 1)$$

In such settings, the merge construct is similar to pairs, and intersection types play a role similar to product types. However, to extract a component from a pair, we need to use explicit projections, while with merge values components are implicitly extracted using types, such as in $(\mathsf{not}\ x)$ above.

The convenience afforded by implicit extraction using types has its costs. The language designer has to resolve conflicts when two branches of a merge overlap. We follow the design of λ_i which imposes a *disjointness* restriction on intersection types [Oliveira et al. 2016]. Two disjoint types have their least upper bound equivalent to the top type. This restriction prevents, for example, two different booleans, or functions of the same type but different implementations, to appear in one merge. Therefore merges always have deterministic behavior when guided by a type. For instance, here are one well-typed and one ill-typed expression.

✓	$\mathsf{True} \,,, (1 \,,, \mathsf{not})$	Bool is disjoint to $\mathsf{Int} \,\&\, (\mathsf{Bool} \to \mathsf{Bool})$
✗	$\mathsf{True} \,,, (1 \,,, \mathsf{False})$	Bool is not disjoint to $\mathsf{Int} \,\&\, \mathsf{Bool}$

A merge operator with a disjointness restriction is an unbiased binary term constructor. It composes expressions to form a tree structure. All the leaves on the tree are in an equal position. They can be viewed as unordered and can be rearranged when needed. The basic elements in merges include single-field records and any term of primitive types. Here we have a nested merge with three elements. The nested merge is wrapped by an annotation, which triggers an upcast. The whole expression then evaluates to a value that fits the shape of the type annotation.

$$(\{l = 1\} \,,, 2 + 3 \,,, \mathsf{not}) : \mathsf{Int} \,\&\, \{l : \mathsf{Int}\} \ \hookrightarrow^*\ 5 \,,, \{l = 1\}$$

As we can see, types direct the runtime reduction of expressions. In the F^+_i calculus [Bi et al. 2019; Fan et al. 2022], which we use in our work, merges of functions and merges of universally quantified abstractions can be applied to an argument or a type argument simultaneously.

$$(\mathsf{not} \,,, (\lambda x.\ x + 1 : \mathsf{Int} \to \mathsf{Int}))\ (1 \,,, \mathsf{False}) \ \hookrightarrow^*\ \mathsf{True} \,,, 2$$

If we replace the function not by a function of type $\mathsf{Bool} \to \mathsf{Int}$, the whole expression will be rejected during typing because $\mathsf{Bool} \to \mathsf{Int}$ is not disjoint to $\mathsf{Int} \to \mathsf{Int}$. For two function types to be disjoint, their return types must be hereditarily disjoint.

Traits and the CP Language. Disjoint intersection types with the merge operator serve as the theoretical foundation of *Compositional Programming* [Zhang et al. 2021]: a recently proposed programming paradigm. The language CP supports *first-class traits* [Bi and Oliveira 2018]. With traits as basic reusable units, CP programs can be very modular and extensible. For this paper we focus on basic uses of traits, but CP's traits are useful to solve challenging modularity problems such as the *Expression Problem* [Wadler 1998]. For further details on CP and its applications, we refer the readers to the original paper by Zhang, Sun, and Oliveira.

Below is a simple trait `defaultCipher`, which defines two methods `encrypt` and `decrypt` that both take a **`String`** and return a **`String`**. It also contains a field that indicates its name. Here we use an identity function for both encryption and decryption.

```
type Cipher = { encrypt: String→String; decrypt: String→String; name: String };
defaultCipher = trait implements Cipher ⇒ {
  encrypt = (ΛA . \(x:A) → x) @String;
  decrypt = (ΛA . \(x:A) → x) @String;
  name    = "Null Cipher";};
```

A trait is elaborated into a function that returns a record in CP's core calculus F_i^+. For instance the trait above would be elaborated to a function of type `Cipher` ⟶ `Cipher`. To model traits, CP uses the semantics of inheritance introduced by Cook and Palsberg [1989] where a class is seen as a function from a record of methods to a record of methods. The function argument denotes the self reference. The argument of the function resulting in the elaboration of a trait denotes the self-reference (i.e., **`this`** in conventional OOP languages). To create an object (typically via the use of **`new`**), we take the fixpoint of the function, fixing the self-reference. As these functions can be *merged* and act like one function, traits in CP are compositional. In the following example, we use , (the notation for merges in CP) to safely merge two traits `defaultCipher` and `defaultHelp`.

```
defaultHelp = trait [self: Cipher] ⇒ {
  showHelp = "Encrypt using " ++ name;
  test (s:String) = "After encryption and decryption the text is " ++ decrypt (encrypt s);};
simpleCipher = defaultCipher , defaultHelp;
(new simpleCipher).showHelp --> "Encrypt using Null Cipher"
```

Note that in the `defaultHelp` trait the notation `[self: Cipher]` allows us to specify the name and the type of the self-reference, similarly to Scala's self-type annotations [Scala Community 2022]. While the self-reference is always available in every trait, the self type annotation is optional. The self type annotation allows us to call the `name`, `decrypt` and `encrypt` methods, which are not implemented in the trait. Later, to instantiate the trait, we must compose it with another trait that implements the missing methods (such as `defaultCipher`). The unbiased merge (,) brings an extra safety check in compilation via its disjointness restriction, and leads to a static error when two traits being composed have conflicts. For instance, consider the following definitions:

```
caesarCipher (shift: Int) = trait inherits defaultCipher implements Cipher ⇒ {
  override encrypt = ... ;
  override decrypt = ... ;
  override name = "Caesar Cipher with a shift of " ++ toString shift;
  showHelp = "Caesar Cipher is a substitution encryption technique"};
new (caesarCipher 3 , defaultHelp) --> type error!
```

The trait `caesarCipher` inherits the methods in `defaultCipher`. However, when trying to compose `caesarCipher` with `defaultHelp` we get a type error, since both traits have a `showHelp` method.

Trait conflicts. To resolve conflicts, one option is to remove the method `showHelp` from one of the traits in advance. One way to do this in CP is to add type annotations to cast traits. Like the following example demonstrates, we can choose what to keep from the two conflicting methods.

```
dropL = new (caesarCipher 3 : Trait<Cipher>) , defaultHelp;
dropR = new caesarCipher 3 , (defaultHelp: Trait<Cipher⇒{test:String→String}>);
dropL.showHelp ++ ". " ++ dropR.showHelp --> "Encrypt using Caesar Cipher with a shift of 3.
   Caesar Cipher is a substitution encryption technique"
```

Here `dropL` drops the `showHelp` from `caesarCipher`, while `dropR` drops `showHelp` from `defaultHelp`. However, removing methods in this way is quite cumbersome. For instance, in `dropR` a programmer has to explicitly annotate all the methods that should be preserved (the annotation `{test: String→String}`). If there are several methods to be preserved this will lead to a lot of boilerplate and manual work. In essence the programmer is manually computing a type difference on types here. Therefore better mechanisms to deal with conflicts are desirable, which is where type difference will come in handy. Next we will show how type difference and other operators can enhance the flexibility of trait composition and reduce the burden of manual annotation for programmers.

2.2 Operations on Traits and Merges

The type difference operator that we propose subtracts one type from the other by removing all the common parts. It is denoted with $A \setminus B$. Take two record types for instance. Type difference does not affect the fields that only appear in the first type.

$$\{count : \mathsf{Int}; check : \mathsf{String} \to \mathsf{Bool}\} \setminus \{num : \mathsf{Int}; check : \mathsf{String} \to \mathsf{Bool}\} = \{count : \mathsf{Int}\}$$

Our type difference deals with types other than records. In particular, it can deal with function types too. Even when they are nested in other constructs, functions can be partially subtracted.

$$\{check : \mathsf{String} \to \mathsf{Int} \,\&\, \mathsf{Bool}\} \setminus \{num : \mathsf{Int}; check : \mathsf{String} \to \mathsf{Bool}\} = \{check : \mathsf{String} \to \mathsf{Int}\}$$

Therefore, type difference can be applied to trait types (which are function types returning records in the core calculus) to remove some of their methods. We make use of type difference to automatically resolve conflicts, rename components, and manually restrict traits in CP, as we shall see next.

Biased merges. In the example in Section 2.1, we use `Trait<Cipher>` to cast the trait `(caesarCipher 3)`. This type is the subtraction result of the type of two traits: `(caesarCipher 3)` and `defaultHelp`. We can simplify the conflict resolution process by defining *biased merge operators*, which use type difference to calculate the type annotation before merging two terms. There are two variants: `+`, prioritizes the left part while `,+` prioritizes the right part. The following code has the same effect as the previous example with no explicit type annotations written.

```
dropLAlt' = new caesarCipher 3 ,+ defaultHelp;    dropRAlt' = new caesarCipher 3 +, defaultHelp;
```

The biased merge operators can be used with multiple inheritance as well. In the following example, we define a trait `simpleCompression` to compress and decompress data. The trait `efficientCipher` inherits the `test` method from `simpleCipher`, but its `name` field comes from `caesarCipher`. Besides, it inherits `compress` and `decompress` from the `simpleCompression` trait and makes use of them to refine the `encrypt` and `decrypt` methods it inherits from `caesarCipher`.

```
type Compression = {compressRatio: Int; compress: String→String; decompress:String→String};
simpleCompression = trait implements Compression ⇒ { ... };
efficientCipher (shift: Int) = trait [self: Cipher&Compression] inherits simpleCipher ,+
   caesarCipher shift ,+ simpleCompression implements Cipher & Compression ⇒ {
    override encrypt (s:String) = super.encrypt (compress s);
    override decrypt (s:String) = decompress (super.decrypt s);};
(new efficientCipher 2).name --> "Caesar Cipher with a shift of 2"
```

Renaming. Now assume that we have another more advanced compression method, which deserves a `name`. We want to combine this trait with the previously defined trait `efficientCipher`. This time we choose to keep both of the conflicting fields (`name`) and rename them (as `cipherName` and `compressionName`). The rename operator is also encoded by type difference, as we will discuss later.

```
advancedCompression = trait inherits simpleCompression ⇒ {name = "A Lossless Compression";...};
betterCipher = (efficientCipher 3) [ name <- cipherName ] ,+ advancedCompression [ name <-
   compressionName ];
(new betterCipher).cipherName --> "Caesar Cipher with a shift of 3"
(new betterCipher).compressionName --> "A Lossless Compression""
```

Restriction operator. Another way to resolve conflicts is to use restriction on traits directly. We use type difference to define restriction operators on traits: `t\l` means the remainder of trait `t` after removing method `l` if it exists; `t\\A` means the remainder of trait `t` after removing every method in type `A`. Suppose that we get a `SecretCipher` from somewhere. We can use restriction on traits to add a compression feature, and cover the cipher's name with a confidentiality note:

```
secretCipher  = trait inherits defaultCipher implements Cipher ⇒ { ... };
note = trait ⇒ { cipherName = "Confidential" ; helpInfo = ... };
hiddenCipher  = ( secretCipher \ name ) , ( betterCipher \\ Cipher ) ,+ note;
```

We first remove `name` from `SecretCipher`, and merge it with `betterCipher` with its `Cipher` part removed. Then we add `note` to the whole composed trait via a biased merge. Note that this is a typical example where biased operators alone are limiting. When we need to remove methods from *both* traits we cannot use a biased operator. But, in CP, we can use explicit restriction operators instead.

2.3 Existing Work on Record Operations and Set Difference

Before introducing the technical details of our work, it is useful to review existing related notions.

Operations on records. Extensible records, or *rows*, provide a flexible and type-safe way to compose data dynamically. A record may contain multiple fields with distinct labels. Existing record calculi vary in the operations that they support [Harper and Pierce 1991a; Ohori 1995; Rémy 1990]. We are particularly interested in calculi that support subtyping. Therefore we use Cardelli and Mitchell [1991]'s (CM) proposal to demonstrate common operations in a record calculus with subtyping:

$\langle\langle l_1 = \mathsf{True}; l_2 = 1\rangle \mid l_3 = \mathsf{not}\rangle$	• **Extension**: add a field of some label to a record, knowing
$\hookrightarrow^* \langle l_1 = \mathsf{True}; l_2 = 1; l_3 = \mathsf{not}\rangle$	that the record does not contain the same label.
$\langle l_1 = \mathsf{True}; l_2 = 1\rangle \setminus l_2$	• **Restriction**: remove a field from a record.
$\hookrightarrow^* \langle l_1 = \mathsf{True}\rangle$	It does nothing if the given field label is absent.
$\langle l_1 = \mathsf{True}; l_2 = 1\rangle.l_1$	• **Extraction**: extract the field associated with the given label.
$\hookrightarrow^* \mathsf{True}$	*The following two operations are implemented by the above three.*
$\langle l_1 = \mathsf{True}; l_2 = 1\rangle[l_2 \leftarrow l_3]$	• **Renaming**: change the label name of a record field.
$\hookrightarrow^* \langle l_1 = \mathsf{True}; l_3 = 1\rangle$	The record must contain the given label.
$\langle\langle l_1 = \mathsf{True}; l_2 = 1\rangle \leftarrow l_2 = 2\rangle$	• **Overriding**: replace a field of the given label if it exists,
$\hookrightarrow^* \langle l_1 = \mathsf{True}; l_2 = 2\rangle$	otherwise extend the record.

The type system of CM reasons about both positive information and negative information, where positive information means the record labels are present in the type, and negative information means the record labels are absent in the type. A value of type $\langle l_1 : \mathsf{Int}\rangle \setminus l_2$ must have a field of Int under name l_1, and can contain any fields except for l_2. The positive information is used in extraction to guarantee the existence of the wanted field. The negative information, expressed through the special type constructor $\setminus$ similar to our type difference, guards the safety of every record extension. The subtyping of CM is complicated by the two kinds of information. Our type difference is a function on types. In $A \setminus B$ the part of A that matches with B is canceled and the whole type can be converted into a form that does not contain the difference operator. So $\{l_1 : \mathsf{Int}; l_2 : \mathsf{Int}\} \setminus l_2$ evaluates to $\{l_1 : \mathsf{Int}\}$. However, in CM's record calculus, $\langle l_1 : \mathsf{Int}; l_2 : \mathsf{Int}\rangle \setminus l_2$ is equivalent to the empty type because it excludes and contains the same label l_2. Besides, not all negative information can be eliminated. For example, $\langle\rangle \setminus l$ is the type of all records that lacks field l, which cannot be expressed without the restriction operator. The operations in CM only work for records and take fields as the basic units. They cannot deal with some of our intended applications, like the examples in Section 2.2, where traits are not modelled as records, but as *functions that return records*.

Set difference in semantic subtyping. Semantic subtyping [Frisch et al. 2008], and its later extensions to polymorphism [Castagna et al. 2015, 2014, 2016; Castagna and Xu 2011], defines subtyping via a set-theoretic model. It supports intersection and union types with distributivity. Under the interpretation that types are sets of values inhabited in it, the subtyping relation is derived from

the subset relation of values. For example, because $A \,\&\, B$ stands for the set intersection of A and B, we directly know that $A \,\&\, B$ is a subtype of A (or B).

Type difference in semantic subtyping is an abbreviation for $A \,\&\, \neg B$, defined via the type complement $\neg B$. It can be applied to any form of types, but despite the syntactic similarity, this definition in semantic subtyping has a completely different meaning to our type difference. Roughly, we want an operation that inverts intersection, so that $(A \,\&\, B) \setminus B$ gives us A. But in semantic subtyping, $(A \,\&\, B) \,\&\, (\neg B)$ is always equivalent to the empty type $\bot$: no term of type $A \,\&\, B$ satisfies $\neg B$. We cannot use set difference to cancel a field in a type, which is a typical use case for record restriction. The next two examples compare the two forms of difference on two record types:

$$\{l_1 : \mathsf{Int}; l_2 : \mathsf{Bool}\} \,\&\, \neg\{l_1 : \mathsf{Int}\} \quad = \bot \qquad \text{semantic subtyping}$$
$$\{l_1 : \mathsf{Int}; l_2 : \mathsf{Bool}\} \setminus \{l_1 : \mathsf{Int}\} \quad = \{l_2 : \mathsf{Bool}\} \qquad \text{our calculus}$$

As discussed in Section 1, the semantics of subtyping in the set-theoretical model differs from ours. The intersection of any two inhabited types is inhabited in our work thanks to the merge operator. This matches up well with a coercive interpretation since the set of values in a subtype should be *larger* than the set of values in a supertype. For example, we can actually take any Int and Bool to compose a value of type Int & Bool, like $1 \,,, \mathsf{True}$. In other words, intersection types in our work should be interpreted as a form of product types, rather than as some form of set intersection. In semantic subtyping, with no term-level constructors for composition, an intersection such as Int & Bool, becomes uninhabited, as expected from an intersection of disjoint sets. In calculi with a merge operator, from any value of a subtype, a value of its supertype can be obtained. For example, if we drop the boolean part in $1 \,,, \mathsf{True}$, the resulting value has type Int.

Type-indexed rows. Shields and Meijer [2001] proposed type-indexed rows to index each field in a record by their types (which have to be distinct) instead of labels. λ^{TIR} does not have a restriction operator, but we can implement one using a function. Consider the examples in λ^{TIR} (on the left) and in our work (on the right):

$let\ f$	
$: \forall\alpha * (\mathsf{Bool} \to \mathsf{Bool}).\ (\alpha \,\&\, (\mathsf{Bool} \to \mathsf{Bool}) \to \alpha)$	The following expressions all evaluate to 1
$= \lambda(x \,,, \_).\ x$	
✓ $f\ (1 \,,, \mathsf{not}) \hookrightarrow^* 1$	✓ $(1 \,,, \mathsf{not}) \setminus (\mathsf{Bool} \to \mathsf{Bool})$
✗ $f\ 1$	✓ $1 \setminus (\mathsf{Bool} \to \mathsf{Bool})$
✗ $f\ (1 \,,, (\lambda(x \,,, \_).\ x) : \mathsf{Bool} \,\&\, \mathsf{Int} \to \mathsf{Bool})$	✓ $(1 \,,, (\lambda x.\ x) : \mathsf{Bool} \,\&\, \mathsf{Int} \to \mathsf{Bool}) \setminus (\mathsf{Bool} \to \mathsf{Bool})$

In λ^{TIR} we use f to remove the $\mathsf{Bool} \to \mathsf{Bool}$ function from its input row. For better comparison, we write the expressions in a syntax that is close to ours. The quantification $\alpha * (\mathsf{Bool} \to \mathsf{Bool})$ here means type α is disjoint with $\mathsf{Bool} \to \mathsf{Bool}$. This function takes a record that has type $\alpha \,\&\, (\mathsf{Bool} \to \mathsf{Bool})$ and matches the pattern $(x \,,, \_)$. It then returns x, which corresponds α, the first part of the argument type, and therefore drops the $\mathsf{Bool} \to \mathsf{Bool}$ part, like $e \setminus (\mathsf{Bool} \to \mathsf{Bool})$ on the right. We use three cases to examine the function and our restriction operator. In the first one, a function not of $\mathsf{Bool} \to \mathsf{Bool}$ is present, and can be successfully removed by f. The last two expressions raise type errors as the function f requires the argument to contain a part exactly corresponding to the type to remove. In particular, λ^{TIR} does not match $(\lambda(x \,,, \_).\ x) : \mathsf{Bool} \,\&\, \mathsf{Int} \to \mathsf{Bool}$ with $\mathsf{Bool} \to \mathsf{Bool}$ because it does not support subtyping. For the same reason, the type of the remaining part of restriction may not be disjoint to $\mathsf{Bool} \to \mathsf{Bool}$ according to our definition of disjointness, which breaks desirable properties.

2.4 Key Ideas

A specification for type difference. Let $A \setminus B$ denote A minus B. To reuse standard terminology from subtractions, we say that A is the *minuend* and B is the *subtrahend* of the subtraction or type difference. We want the function to satisfy the following three properties:

(1) $A \setminus B$ is a supertype of A.
(2) $A \setminus B$ is compatible (disjoint) with B, denoted by $(A \setminus B) * B$.
(3) $(A \setminus B)\,\&\,B$ is a subtype of A.

Consider a typical case where e_1 has type A, e_2 has type B and C is the difference of A and B.

$$(e_1 : C \,,\, e_2) : A$$

Expressions of this kind typically arise during updates, such as those that we illustrated at the end of Section 2.1. The first property ensures that $e_1 : C$ is valid, because e_1 is of type A and it is cast to a supertype C. The second property guarantees that the merge is safe and compatible. The last property justifies the outermost annotation. We can imagine that A and B are two sets (here atomic types such as *Int* or *Bool* are the elements of the sets). Then the set difference of A and B is the set of elements that are in A (which corresponds to the first property), and are not in B (which corresponds to the second property). The set difference includes all elements that fulfill these two conditions, and it is the relative complement of B with respect to A. This corresponds to the third property. Formally, we use $A \setminus_s B \equiv C$ to represent that C is the type difference of A and B. The relational specification combines the three properties above:

Definition 2.1 (Type difference specification). $A \setminus_s B \equiv C \triangleq A \leq C \ \wedge\ B * C \ \wedge\ B\,\&\,C \leq A$.

Note that we use $\equiv$ in later text to mean the equivalence relation defined on subtyping, i.e., $A \equiv B$ if and only if they are mutual subtypes. An important complication, which type difference has to deal with, is that different types can be related by subtyping. Therefore we cannot simply use equality of two types to decide whether or not to remove a type component from the minuend. Take $\top \to \mathsf{Bool}$ and $\mathsf{Bool} \to \mathsf{Bool}$ as an example. The former is a subtype of the latter as it can accept any argument that the latter accepts ($\top$ accepts all legal terms). From the point of view of sets, we can say that the subtype $\top \to \mathsf{Bool}$ includes supertype $\mathsf{Bool} \to \mathsf{Bool}$.

$$(\mathsf{Int}\,\&\,(\mathsf{Bool} \to \mathsf{Bool})) \setminus (\top \to \mathsf{Bool}) \equiv \mathsf{Int}$$

We can always use a subtype to remove a supertype component in the minuend. They cannot be disjoint unless the supertype is equivalent to $\top$. So, only after $\top \to \mathsf{Bool}$ is removed, the result satisfies property (2).

Partiality of type difference. Having subtyping as a partial order also means that we cannot define a total subtraction operation that satisfies all three properties. What is the difference of $\bot$ (the infinite intersection of all types) and Int? More generally, what should the remaining part be when we subtract a supertype from a subtype that cannot be further split? Take the previous two types for example, and consider the following (incorrect) results for type difference:

$$(\top \to \mathsf{Bool}) \setminus (\mathsf{Bool} \to \mathsf{Bool}) \equiv \top \to \mathsf{Bool} \qquad \text{violates property (2)}$$
$$(\top \to \mathsf{Bool}) \setminus (\mathsf{Bool} \to \mathsf{Bool}) \equiv \top \qquad \text{violates property (3)}$$

According to property (1), the result is a supertype of $\top \to \mathsf{Bool}$. So the candidates are restricted to a function type, or a top-like type such as $\top$, or an intersection of both. For instance we could try to pick a function type, such as $\top \to \mathsf{Bool}$ itself, or $\top$ for the result. But both types will violate at least one property, and other similar types would be equally problematic. We say that the two types above are not *subtractable*. Later, in the formal development, we will discuss how to determine whether two types are subtractable in detail.

Table 1. Operations supported via type difference. Two primitive operations and syntactic sugar.

Extension/concatenation	$e_1 \,,, e_2$	Encoded directly by the merge operator
Restriction (minus a type)	$e \setminus A$	Primitive operation in $F_i^{\setminus}$
Restriction (minus a term)	$e_1 \setminus e_2$	Primitive operation in $F_i^{\setminus}$
Restriction (minus a label)	$e \setminus l$	$\triangleq e \setminus \{l : \bot\}$
Extraction	$e.l$	Encoded directly by projection
Renaming	$e[l_1 \leftarrow l_2]$	$\triangleq e \setminus l_1 \,,, \{l_2 = e.l_1\}$
Right-biased merge	$e_1 \,,^{+}\, e_2$	$\triangleq (e_1 \setminus e_2) \,,, e_2$
Left-biased merge	$e_1 \,^{+},\, e_2$	$\triangleq e_1 \,,, (e_2 \setminus e_1)$

On the other hand, given two subtractable types A and B, these three properties already limit their type difference $A \setminus B$ to a class of equivalent types. That is to say, all the types that fulfill the properties are mutual subtypes. Property (1) allows $A \setminus B$ to be as large as A. But if the type contains too many elements, it breaks property (2) because it may contain type components present in B. On the contrary, if it has too few elements, it loses property (3).

Encoding operations with type difference. Instead of demanding users to annotate terms explicitly, type difference provides a more direct way to model various desirable operations. As we have shown in CP, it facilitates updating trait methods, renaming trait components and so on. Here, we demonstrate how these operations are encoded with the help of type difference.

A first class of operations is restriction operations, which we have seen on traits. For expressions in the core calculus, we use $e \setminus A$ and $e_1 \setminus e_2$ corresponding to `\\` in CP. These operations can be applied to any terms, not only to records. Here are some simple examples.

$1 \setminus \mathsf{Int}$	$= \{\}$	$\{\}$ stands for the top value, a merge of no element
$1 \setminus 2$	$= \{\}$	To subtract a term is to subtract its type
$(1 \,,, \mathsf{True}) \setminus \mathsf{Int}$	$= \mathsf{True}$	The part of merge that does not match the subtrahend type remains
$1 \setminus \mathsf{Char}$	$= 1$	If the subtrahend does not overlap with the minuend, it cancels no term

Table 1 shows a summary and description of the operations that we support.

Type difference for overloaded functions. Besides traits, type difference of functions has other applications. For instance, we can use type difference to update implementations of an overloaded function. In the following example, inspired by Haskell's `read` and an example due to Marntirosian et al. [2020], we use a biased merge to override the conflicting part of a function.

```
readInt    : String → Int          readChar      : String → Char
readString : String → String       myNewReadChar : String → Char
let read :  (String→Int) & (String→Char) & (String→String)
        =  readInt ,, readChar ,, readString
in newRead = read ,+ myNewReadChar
```

In this example we have an overloaded `read` function that can parse a string as either an integer or a character or a string. To obtain a variant of `read` that uses a different function to read characters we can update the implementation of the corresponding type as shown in `newRead`.

Algorithms for type difference, disjointness and subtyping. We will present an algorithmic definition for type difference, which does not rely on disjointness and subtyping. As disjointness and subtyping

can be encoded by type difference, we can derive disjointness and subtyping algorithms once we have algorithmic type difference. A formal specification of disjointness is:

Definition 2.2 (Disjointness). $A * B \triangleq \forall\, C,$ if $A \leq C$ and $B \leq C$ then $C \equiv \top$.

According to the specification of type difference, we can conclude $A * B$ from $A \setminus B \equiv A$. Via the specification of disjointness we know that the other direction also holds. Therefore we have:

$$A * B \leftrightarrow A \setminus B \equiv A$$

Similarly, we have that

$$A \leq B \leftrightarrow B \setminus A \equiv \top$$

The fact that we can derive subtyping from type difference is a natural consequence of its algebraic properties. Our specification of type difference satisfies the three axioms for being a *subtraction algebra* [Jun et al. 2004; Schein 1992], which is known to determine an order relation, guaranteeing a good correspondence between the type difference and subtyping. We have the order $\leq$ representing that the left side (subtype) contains more information than the right side (supertype).

$$A \setminus (B \setminus A) \equiv A$$
$$A \setminus (A \setminus B) \equiv B \setminus (B \setminus A)$$
$$(A \setminus B) \setminus C \equiv (A \setminus C) \setminus B$$

Distributivity and disjoint polymorphism. We add universal quantified types to the subtyping relation by Barendregt et al. [1983]. Like other type constructors, universal quantifiers distribute over intersections. We make use of the idea of *splittable types* [Huang et al. 2021]: all types that contain intersections in positive positions can be split into smaller elements. Splittable types enable us to deal with distributive subtyping. Furthermore, using splittable types we can subtract a function partially, even when it is nested in other constructors, as we have shown at the beginning of Section 2.2. In addition, we support *disjoint polymorphism*, which allows a disjointness constraint to be specified for type abstractions.

$$(\Lambda(\alpha * \mathsf{Int}).\, \lambda x : \alpha.\, x \,,, 1) : \forall \alpha * \mathsf{Int}.\, \alpha \to \alpha \,\&\, \mathsf{Int}$$

This is an expression that takes a type and a term. $\alpha * \mathsf{Int}$ enforces a restriction to the type argument, and ensures that the parameter x of type α can be merged with 1. Our type difference not only can subtract a function from another, but can also handle universally quantified types and variables. Disjoint quantification also allows us to encode the type restriction in the CM calculus.

$$\text{CM}: \quad let\ f\ (R <: \langle l_1 : \mathsf{Int}\rangle \setminus l_2)\ (r : R)\ : \langle R | l_2 : \mathsf{Int}\rangle = \langle \langle r \leftarrow l_1 = r.l_1 + 1\rangle \mid l_2 = 0\rangle$$
$$\text{Us}: \quad \Lambda(\alpha * \{l_2 : \bot\}).\, \lambda x : \alpha \,\&\, \{l_1 : \mathsf{Int}\}.\, x \,,^{+}\, \{l_1 = x.l_1 + 1; l_2 = 0\}$$

The example above is from CM's paper. Function f expects a type parameter that is a subtype of $\langle l_1 : \mathsf{Int}\rangle \setminus l_2$, and a value parameter of such type, i.e., it must contain a field of l_1 and lack a field of l_2. The function increments the integer value of l_1 and extends the input with a l_2 field. The remaining fields remain unchanged. Our implementation in the second line does the same thing.

3 DERIVING A TYPE DIFFERENCE

This section shows how to derive a type difference algorithm from the specification that was given in Section 2. We present a type difference algorithm for the subtyping relation of Barendregt et al. [1983] (BCD) extended with record types, and prove its soundness, completeness and decidability with respect to the specification. Importantly, note that while the subtyping relations presented in our work are extensions of BCD subtyping, our typing relation and semantics for the calculi in our work (presented in Section 4) are very different from those in Barendregt et al.'s work.

$\boxed{A \le B}$ *(BCD subtyping)*

$$\text{S-REFL}\ \frac{}{A \le A} \qquad \text{S-BOT}\ \frac{}{\bot \le A} \qquad \text{S-TOP}\ \frac{}{A \le \top} \qquad \text{S-TOPARR}\ \frac{}{\top \le \top \to \top} \qquad \text{S-TOPRCD}\ \frac{}{\top \le \{l : \top\}}$$

$$\text{S-ARR}\ \frac{A_2 \le A_1 \qquad A_3 \le A_4}{A_1 \to A_3 \le A_2 \to A_4}$$

$$\text{S-RCD}\ \frac{A \le B}{\{l : A\} \le \{l : B\}} \qquad \text{S-DISTARR}\ \frac{}{(A \to B)\,\&\,(A \to C) \le A \to B \,\&\, C} \qquad \text{S-DISTRCD}\ \frac{}{\{l : A\}\,\&\,\{l : B\} \le \{l : A \,\&\, B\}}$$

$$\text{S-TRANS}\ \frac{A \le B \qquad B \le C}{A \le C} \qquad \text{S-ANDL}\ \frac{}{A \,\&\, B \le A} \qquad \text{S-ANDR}\ \frac{}{A \,\&\, B \le B} \qquad \text{S-AND}\ \frac{A \le B \qquad A \le C}{A \le B \,\&\, C}$$

Fig. 1. BCD subtyping.

3.1 Syntax and Subtyping

Types	$A, B, C ::= \mathsf{Int} \mid \top \mid \bot \mid A \,\&\, B \mid A \to B \mid \{l : A\}$
Ordinary types	$A^\circ, B^\circ, C^\circ ::= \mathsf{Int} \mid \top \mid \bot \mid A \to B^\circ \mid \{l : A^\circ\}$

Types. BCD subtyping [Barendregt et al. 1983] is a widely used subtyping relation for intersection types. The most important feature of BCD-style subtyping is that it allows function types to distribute over intersection types, like in rule S-DISTARR. We have all the types in the original BCD type system, extended with a bottom type and a record type. *Ordinary types* [Davies and Pfenning 2000], are types that do not contain top-level intersection types. However, in BCD subtyping, arrow types such as $A \to B \,\&\, C$ may behave like intersection types due to distributivity. Following the approach by Huang et al. [2021], we restrict the notion of ordinary types to exclude such types.

Subtyping. The subtyping rules, presented in Figure 1, extend BCD subtyping [Barendregt et al. 1983] with rules for record types and the bottom type. Terms of an intersection type must satisfy both parts of the intersection as $A \,\&\, B$ is the greatest lower bound of type A and B. The key feature of BCD subtyping is that other type constructors distribute over intersections. For example, $A \to B \,\&\, C$ is a subtype of $(A \to B)\,\&\,(A \to C)$. $\top$ is the supertype of all types, but there is a group of types equivalent to $\top$, such as $\top \,\&\, \top$ or $A \to \top$.

Notation. Note that $A \equiv B$ denotes the equivalence defined via subtyping ($A \le B$ and $B \le A$) and $=$ is used for syntactic identity, i.e., $A = B$ means that A and B are exactly the same type. We need this notation because intersection types and distributivity introduce a lot of semantically equivalent types that are not syntactically equivalent such as Int, Int & Int, Int & Int & Int... In particular, we use *top-like types* to describe types that are equivalent to $\top$.

3.2 Deterministic Type Difference

The specification ($A \setminus_s B \equiv C$) in Definition 2.1 is neither algorithmic nor deterministic. It only restricts the result of type difference to a semantically equivalent class of types. Next we will give an intermediate definition in Fig 2, which is explained in detail over this section. This definition will be helpful to derive an algorithmic version of type difference. Unlike the specification of type difference, the new definition is deterministic: there can only be a unique type for such intermediate type difference. We prove that the definition is sound and complete to the specification.

Equivalence results for type difference. The type difference specification does not distinguish two semantically equivalent types, as it is defined by subtyping and disjointness, both of which are not sensitive to the syntactic form of types. The following lemma expresses this property:

$\boxed{A \setminus_d B = C}$ *(Deterministic Type Difference)*

$$\frac{A * B}{A \setminus_d B = A}\ \text{TD-DISJOINT} \qquad \frac{B \leq A}{A \setminus_d B = A^{\top}}\ \text{TD-SUBTYPE}$$

$$\frac{A_1 \lhd A \rhd A_2 \qquad A_1 \setminus_d B = C_1 \qquad A_2 \setminus_d B = C_2 \qquad A \vdash C_1 \rhd C \lhd C_2}{A \setminus_d B = C}\ \text{TD-BCD}$$

Fig. 2. Deterministic definition of type difference, considering different three cases.

Lemma 3.1 (Type Difference Specification Safety). *If $A \setminus_s B \equiv C_0$ and $C_0 \equiv C_1$, then $A \setminus_s B \equiv C_1$.*

Take the type difference of Int and Int for example. According to the specification, all the types equivalent to $\top$ fit Int $\setminus_s$ Int. Such types include $\top$, $\top \,\&\, \top$, and Int $\to \top$. On the other hand, the type difference specification is precise enough to restrict the result to an equivalence class of types. We prove the coherence of the type difference specification using the covariance of disjointness property. With Lemmas 3.1 and 3.3, show the specification forms an equivalence relation.

Lemma 3.2 (Covariance of Disjointness). *If $A * B$ and $B \leq C$, then $A * C$.*

Lemma 3.3 (Coherence of Type Difference). *If $A \setminus_s B \equiv C_1$ and $A \setminus_s B \equiv C_2$ then $C_1 \equiv C_2$.*

Two special cases for type difference. We next show two special cases of how the specification behaves, to help us better understand the first two rules TD-DISJOINT and TD-SUBTYPE of deterministic type difference. Firstly, we have the case where $A * B$. Since the two types do not share any nontrivial supertypes other than $\top$, we should remove no part from the minuend A.

Lemma 3.4 (Disjoint Type Subtraction). *If $A * B$, then $A \setminus_s B \equiv A$.*

The second case is when $B \leq A$. In this case any part of A is a supertype of B. To find a result that is disjoint to B, we have to discard all the components of A.

Lemma 3.5 (Subtype Subtraction). *If $B \leq A$, then $A \setminus_s B \equiv \top$.*

Zero type. While the first rule TD-DISJOINT directly follows Lemma 3.4, in the second rule we use a notion of zero type, defined at the right bottom of Figure 3. The zero type function helps the second rule TD-SUBTYPE to return the same result when it overlaps with the first rule. It computes a top-like type based on any given type. Notice that $\top$ is not only the supertype of every type, but also disjoint to every type. For a minuend type A equivalent to $\top$, such as Int $\to \top$, $A \setminus B$ always matches both of the first two rules regardless of B. To make these two rules consistent, we choose to preserve the original structure of the minuend as much as possible. For the concrete example (Int $\to \top$) $\setminus B$, the result is Int $\to \top$ rather than $\top$, which is equal to (Int $\to \top$)$^{\top}$. Generally, the zero type function maintains the structure of its input and erases all the meaningful components in positive positions by using the $\top$ type instead. It does not change the input if it is already top-like.

Lemma 3.6 (Completeness of Zero Type). $A^{\top} \equiv \top$.

Lemma 3.7 (Soundness of Zero Type). *If $A \equiv \top$, then $A^{\top} = A$.*

The general case for type difference. In general, the subtrahend type is neither disjoint with nor a subtype of the minuend. Suppose that we have an intersection of two functions: (Bool $\to$ Bool) & (Int $\to$ Int), and we want to remove the function on integers, which is (Bool $\to$ Bool) & (Int $\to$ Int) $\setminus$ (Int $\to$ Int). In this case, the first two rules cannot apply. Only after the minuend is separated, (Bool $\to$ Bool) $\setminus$ (Int $\to$ Int) matches the first rule and (Int $\to$ Int) $\setminus$ (Int $\to$ Int) matches the second. By composing the two results, we obtain the type difference of the original minuend and subtrahend (Bool $\to$ Bool) & (Int $\to \top$), i.e., (Bool $\to$ Bool). From the specification, we can prove the correctness of the splitting and composition.

$\boxed{B \lhd A \rhd C}$ *(Type Splitting)*

$$\frac{}{A \lhd A \,\&\, B \rhd B} \qquad \frac{C_1 \lhd B \rhd C_2}{A \to C_1 \lhd A \to B \rhd A \to C_2} \qquad \frac{C_1 \lhd B \rhd C_2}{\{l : C_1\} \lhd \{l : B\} \rhd \{l : C_2\}}$$

$\boxed{A \vdash B_1 \rhd C \lhd B_2}$ *(Type Merging)*

$$\frac{}{A_1 \,\&\, A_2 \vdash B_1 \rhd B_1 \,\&\, B_2 \lhd B_2}$$

$$\frac{A \vdash B_1 \rhd C \lhd B_2}{D \to A \vdash D \to B_1 \rhd D \to C \lhd D \to B_2}$$

$$\frac{A \vdash B_1 \rhd C \lhd B_2}{\{l : A\} \vdash \{l : B_1\} \rhd \{l : C\} \lhd \{l : B_2\}}$$

$\boxed{A^\top \equiv B}$ *(Zero Type Function)*

$$\begin{aligned}
\top^\top &= \top \\
\bot^\top &= \top \\
\mathsf{Int}^\top &= \top \\
(A \,\&\, B)^\top &= A^\top \,\&\, B^\top \\
(A \to B)^\top &= A \to B^\top \\
\{l : B\}^\top &= \{l : B^\top\}
\end{aligned}$$

Fig. 3. The auxiliary meta functions used in the deterministic type difference.

Lemma 3.8 (Composition). *If $A_1 \setminus_s B \equiv C_1$ and $A_2 \setminus_s B \equiv C_2$, then $(A_1 \,\&\, A_2) \setminus_s B \equiv C_1 \,\&\, C_2$.*

Splittable and mergeable types. The last rule in Figure 2 deals with more than just intersection types. It is very similar to the last lemma if we replace $A_1 \lhd A \rhd A_2$ and $A \vdash C_1 \rhd C \lhd C_2$ by $A = A_1 \,\&\, A_2$ and $C = C_1 \,\&\, C_2$. Actually, these two relations generalize the *splitting* and *merging* of intersection types by taking distributivity into consideration.

For the last rule TD-BCD of type difference, we need to introduce the concept of splittable and mergeable types. Splittable types were first introduced by Huang et al. [2021] to promote a subtyping algorithm for BCD subtyping. The idea of splittable types is to specify the distributivity brought by rules S-distArr and S-distRcd explicitly so that we can eliminate the use of the transitivity rule S-trans in the subtyping deduction. We need the notion of splittable types because in the general case of type difference, the intersection may hide deeply inside certain types like $\mathsf{Int} \to \mathsf{Int} \,\&\, \mathsf{Bool}$. Subtracting components like $\mathsf{Int} \to \mathsf{Int}$ from them would be impossible if we only look at top-level intersections. Under such cases, we need the rules for splittable types defined at the top of Figure 3 to help us analyze these intersection-like types.

Mergeable types are new and they are designed to complement splittable types. Merging is defined at the left bottom part of Figure 3 as a reverse operation of splitting. $A \vdash B_1 \rhd C \lhd B_2$ means merging two types B_1 and B_2 according to the shape of the type A will result in a type C. We need the shape of A as a reference because splitting is not an injective function. Thus merging two types like $B_1 \to B_2$ and $B_1 \to B_3$ could result in either $B_1 \to B_2 \,\&\, B_3$ or $(B_1 \to B_2) \,\&\, (B_1 \to B_3)$. The shape of A ensures the determinism of such merging process. We have the following results to ensure the reversibility and determinism of the splittable and mergeable relations.

Lemma 3.9 (Split Determinism). *If $B_1 \lhd A \rhd B_2$ and $C_1 \lhd A \rhd C_2$, then $B_1 = C_1$ and $B_2 = C_2$.*

Lemma 3.10 (Determinism of Merging). *If $A \vdash B_1 \rhd C \lhd B_2$ and $A \vdash B_1 \rhd C' \lhd B_2$, then $C = C'$.*

Lemma 3.11 (Reversibility of Merging and Splitting). *If $A_1 \lhd A \rhd A_2$, then $A \vdash A_1 \rhd A \lhd A_2$; if $B \vdash A_1 \rhd A \lhd A_2$, then $A_1 \lhd A \rhd A_2$.*

3.3 Algorithmic Type Difference

Deterministic type difference depends on subtyping and disjointness. While having algorithmic formulations of disjointness and subtyping would enable us to obtain an algorithm for deterministic type difference, in this section we will show an algorithmic formulation of type difference which

$$
\begin{array}{rcll}
A \setminus_a B & = & A & (\text{if } A \equiv \top) \\
B \setminus_a A & = & B & (\text{if } A \equiv \top) \\
A \setminus_a \bot & = & A^{\top} & \\
\mathsf{Int} \setminus_a (A \to B) & = & \mathsf{Int} & \\
\mathsf{Int} \setminus_a \{l : A\} & = & \mathsf{Int} & \\
\{l : A\} \setminus_a (B \to C) & = & \{l : A\} & \\
\{l : A\} \setminus_a \mathsf{Int} & = & \{l : A\} & \\
(A \to B) \setminus_a \mathsf{Int} & = & A \to B & \\
(A \to B) \setminus_a \{l : C\} & = & A \to B & \\
\mathsf{Int} \setminus_a \mathsf{Int} & = & \top & \\
\{l_1 : A\} \setminus_a \{l_2 : B\} & = & \{l_1 : A\} & (\text{if } l_1 \neq l_2) \\
A_1 \to A_2 \setminus_a B_1 \to B_2 & = & A_1 \to (A_2 \setminus_a B_2) & (\text{if } A_2 \setminus_a B_2 = A_2 \text{ or } B_1 \setminus_a A_1 = B_1{}^{\top}) \\
\{l : A\} \setminus_a \{l : B\} & = & \{l : A \setminus_a B\} & \\
A \setminus_a B & = & C & (\text{if } A_1 \lhd A \rhd A_2 \text{ and } A \vdash (A_1 \setminus_a B) \rhd C \lhd (A_2 \setminus_a B)) \\
A \setminus_a (B_1 \,\&\, B_2) & = & (A \setminus_a B_1) \setminus_a B_2 & (\text{if } A \setminus_a B_1 \text{ is defined }) \\
A \setminus_a (B_1 \,\&\, B_2) & = & (A \setminus_a B_2) \setminus_a B_1 & (\text{if } A \setminus_a B_2 \text{ is defined })
\end{array}
$$

Fig. 4. Algorithmic type difference.

does *not* depend on subtyping and disjointness. In turn, using this algorithmic formulation we can give an algorithm for subtyping and disjointness as well, since both of these relations are definable as special cases of type difference.

Design. The rules of algorithmic type difference, presented in Figure 4, can be divided into three categories, which follow the three cases in deterministic type difference. The first set of rules is for dealing with disjointness, such as $\mathsf{Int} \setminus_a (A \to B) = \mathsf{Int}$. The second set of rules is for dealing with subtyping, such as $\mathsf{Int} \setminus_a \mathsf{Int} = \top$ and $\{l : A\} \setminus_a \{l : B\} = \{l : A \setminus_a B\}$. The last set of rules only contains the last three rules in Figure 4. They are used for dealing with mixed cases, where the subtrahend or minuend behave like intersection types.

Disjointness. Most of the rules in this category simply return the minuend if the rule operates on two disjoint types. How to judge type disjointness from the type difference is given by the following lemma. From it we directly obtain an algorithm for disjointness, that is calculating $A \setminus_a B$ to see whether it results in A or not.

Lemma 3.12 (Disjoint Type Difference). *$A * B$ if and only if $A \setminus_a B = A$.*

For most rules such as $\mathsf{Int} \setminus_a (A \to B) = \mathsf{Int}$, we can directly find the disjointness relation from the structure of the two types. There are also rules which involve substructures of types, such as $(A_1 \to A_2) \setminus_a (B_1 \to B_2) = A_1 \to A_2$ (if $A_2 \setminus_a B_2 = A_2$). Besides, if a type is equivalent to $\top$, then it is disjoint to every type, which is the case of the first and the second rule in Figure 4. We give an algorithm for judging whether a type is top-like in the Appendix. We will also introduce an algorithm for judging a top-like type extended with polymorphism in Section 4.

Subtyping Rules. Most of the rules in this category simply return the zero type of the minuend if the rule finds a subtyping relation between two types. The correctness of this behaviour is ensured by the following lemmas.

Lemma 3.13 (Zero Type Difference). *$A^\top = C$ if and only if $A \setminus_a A = C$.*

Lemma 3.14 (Subtyping Type Difference). *$B \leq A$ if and only if $A \setminus_a B = A^\top$.*

For example, the rule $\mathsf{Int} \setminus_a \mathsf{Int} = \top$ finds that Int is a subtype of Int, thus we return the zero type of Int, which is $\top$. For two record types or two function types, there is no special rules for the case that the subtrahend is a subtype of the minuend. Such rules are subsumed by the two structural rules $\{l : A\} \setminus_a \{l : B\} = \{l : A \setminus_a B\}$ and $A_1 \to A_2 \setminus_a B_1 \to B_2 = A_1 \to (A_2 \setminus_a B_2)$ (if $B_1 \setminus_a A_1 = B_1{}^\top$). Note that with the lemmas above, we can give an algorithm for $B \leq A$ by checking $A \setminus_a B = A^\top$.

Intersections. The last three rules in Figure 4 are used to deal with intersection or intersection-like types. The first rule is just the same as deterministic type difference. The rule $A \setminus_a (B_1 \,\&\, B_2) = (A \setminus_a B_1) \setminus_a B_2$ says that if a type is subtracted by an intersection type, you can first subtract its left component then subtract its right component. The rule $A \setminus_a (B_1 \,\&\, B_2) = (A \setminus_a B_2) \setminus_a B_1$ is the commutative version. We need these two rules because type difference is not total. Recall the totality counterexample in the Section 2. Similarly to the counterexample we have that $(\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$ is not subtractable by $(\mathsf{Int} \to \mathsf{Int})$. Next we consider a type difference $A \setminus_a B$ when $A = (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$ and $B = (\mathsf{Int} \to \mathsf{Int}) \,\&\, (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$. By the rule $A \setminus_a (B_1 \,\&\, B_2) = (A \setminus_a B_1) \setminus_a B_2$ alone, we will first need the calculate $(\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int} \setminus_a (\mathsf{Int} \to \mathsf{Int})$, which is not subtractable. But B is a subtype of A, and according to rule TD-SUBTYPE we expect $A \setminus_a B$ to be subtractable. Therefore, we need a commutative rule $A \setminus_a (B_1 \,\&\, B_2) = (A \setminus_a B_2) \setminus_a B_1$ to get through such cases. Note that the 3 last cases overlap. We have proved that the first rule can always be applied first in an implementation without loss of expressive power. For the commutative rules we proved that their relative order does not matter: we can apply either one in any order.

3.4 Theorems

In this section we demonstrate some properties about type difference. We will prove the determinism of type difference, the equivalence between the three type differences ($A \setminus_s B \equiv C$, $A \setminus_d B = C$, $A \setminus_a B = C$) and the decidability of type difference.

Equivalence and Determinism. Using a lemma of deterministic type difference, we can prove the equivalence of the algorithmic type difference with a direct induction.

Lemma 3.15 (Type Difference Concatenation). *If $A_1 \setminus_d B_1 = A_2$ and $A_2 \setminus_d B_2 = A_3$, then $A_1 \setminus_d (B_1 \,\&\, B_2) = A_3$.*

Lemma 3.16 (Type Difference Equivalence). *$A \setminus_d B = C$ if and only if $A \setminus_a B = C$.*

By the Lemmas 3.9 and 3.10 showing the determinism of splittable and mergeable types, we can directly show that deterministic type difference is indeed deterministic.

Theorem 3.17 (Determinism of Type Difference). *If $A \setminus_d B = C_1$ and $A \setminus_d B = C_2$, then $C_1 = C_2$.*

Completeness. Completeness means that deterministic type difference and algorithmic type difference satisfy the specification of type difference. Similarly to Lemmas 3.12 and 3.14, we can obtain the following two properties.

Lemma 3.18 (Completeness of Subtyping). *If $A \setminus_d B = C$, then $A \leq C$.*

Lemma 3.19 (Completeness of Disjointness). *If $A \setminus_d B = C$, then $B * C$.*

By direct induction we obtain:

Lemma 3.20 (Completeness of Reverse Subtyping). *If $A \setminus_d B = C$, then $B \,\&\, C \leq A$.*

Combining three properties above, we prove the completeness of type difference.

Theorem 3.21 (Type Difference Completeness). *If $A \setminus_d B = C$, then $A \setminus_s B \equiv C$.*

Soundness. Soundness means that if there exists a type C satisfying $A \setminus_s B \equiv C$, then we can find a type C' that is equivalent to C and $A \setminus_d B = C'$. We define a new notion called *subtractable* and use the decidability lemma of subtractable to prove the soundness.

Definition 3.22 (Subtractable). Subtractable $A\ B \triangleq$ For all C°, if $A \leq C^\circ$, then $B \leq C^\circ \vee B * C^\circ$.

An intuition for subtractable is that for every component of type A identified by C°, either B is a subtype of it or B is disjoint with it. Otherwise it is not subtractable, such as $\mathsf{Int} \to \mathsf{Bool}$ and $\mathsf{Bool} \to \mathsf{Bool}$. After subtractable is defined, we can prove the following lemmas directly by induction on A and B:

Lemma 3.23 (Soundness of Subtractable). *If* Subtractable $A\ B$, *then* $\exists C, A \setminus_d B = C$.

Lemma 3.24 (Decidability of Subtractable). Subtractable $A\ B$ *is decidable.*

Lemma 3.25 (Completeness of Subtractable). *If* $\exists C, A \setminus_s B \equiv C$, *then* Subtractable $A\ B$.

With the completeness Theorem 3.21, all that remains is to show that deterministic type difference is sound with respect to the type difference specification, and the equivalence between the three formulations of type differences is established. Furthermore type difference is decidable.

Theorem 3.26 (Type Difference Soundness). $A \setminus_s B \equiv C$ *if and only if* $\exists C', A \setminus_d B = C' \wedge C \equiv C'$.

Theorem 3.27 (Decidability of Type Difference). *Type difference is decidable.*

4 A CALCULUS WITH DISJOINT QUANTIFICATION AND OPERATIONS ON MERGES

In this section we extend the previously defined type difference to disjoint polymorphism [Alpuim et al. 2017] and present it in a full calculus called $\mathsf{F}_i^{\rangle}$. Disjoint polymorphism enables the calculus to support encodings of *bounded quantification* and *row polymorphism* [Xie et al. 2020]. In particular Xie et al. has shown that kernel $F_{<:}$ [Cardelli and Wegner 1985], which is the most widely used decidable calculus with bounded quantification, can be encoded in F_i^{+}. $\mathsf{F}_i^{\rangle}$ is proved to be type-safe via an elaboration into F_i^{+} [Bi et al. 2019; Fan et al. 2022] (a calculus with disjoint polymorphism but without type difference) in Section 5.

4.1 Syntax and Well-Formedness

The syntax of $\mathsf{F}_i^{\rangle}$ is:

Types	$A, B, C ::= \mathsf{Int} \mid \top \mid \bot \mid A \,\&\, B \mid A \to B \mid \{l : A\} \mid \alpha \mid \forall \alpha * A.\, B$
Ordinary types	$A^\circ, B^\circ, C^\circ ::= \alpha \mid \mathsf{Int} \mid \top \mid \bot \mid A \to B^\circ \mid \{l : A^\circ\} \mid \forall \alpha * A.\, B^\circ$
Expressions	$e ::= p \mid x \mid i \mid \{\} \mid e : A \mid e_1 \,,, e_2 \mid \mathsf{fix}\ x : A.\, e \mid \Lambda \alpha.\, e \mid \lambda x : A.\, e \mid \{l = e\}$
	$\mid \Lambda(\alpha * A).\, e \mid \lambda x.\, e \mid e_1\ e_2 \mid e\ A \mid e.l \mid e \setminus A \mid e_1 \setminus e_2$
Term contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$
Type contexts	$\Delta ::= \cdot \mid \Delta, \alpha * A$

Type and Expression Syntax. Our syntax extends the syntax of the F_i^{+} calculus [Fan et al. 2022] by the gray parts. A type variable α is bound by disjoint quantification $\forall \alpha * A.\, B$, which can only be instantiated by types that are disjoint to A. We add two new expression forms ($e_1 \setminus A$ and $e_1 \setminus e_2$) as a generalization of a record restriction operation, and allow more flexibility on annotating lambda abstractions and disjoint-quantified abstractions. The expression $e \setminus A$ removes all components

$\boxed{\Delta \vdash A \leq B}$ *(Declarative Subtyping, Additional Rules)*

$$\textsc{DS-topVar}\ \frac{\vdash \Delta \qquad \alpha * A \in \Delta \qquad \Delta \vdash A \leq \bot}{\Delta \vdash \top \leq \alpha} \qquad \textsc{DS-forall}\ \frac{\Delta \vdash A_2 \leq A_1 \qquad \Delta, \alpha * A_2 \vdash B_1 \leq B_2}{\Delta \vdash \forall \alpha * A_1.\, B_1 \leq \forall \alpha * A_2.\, B_2}$$

$$\textsc{DS-distAll}\ \frac{\vdash \Delta \qquad \Delta \vdash \forall \alpha * A.\, (B_1 \,\&\, B_2)}{\Delta \vdash (\forall \alpha * A.\, B_1) \,\&\, (\forall \alpha * A.\, B_2) \leq \forall \alpha * A.\, (B_1 \,\&\, B_2)} \qquad \textsc{DS-topAll}\ \frac{\vdash \Delta}{\Delta \vdash \top \leq \forall \alpha * \top.\, \top}$$

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

$$\textsc{D-top}\ \frac{\Delta \vdash B \qquad A \equiv_\Delta \top}{\Delta \vdash A * B} \qquad \textsc{D-rcd}\ \frac{\Delta \vdash A * B}{\Delta \vdash \{l : A\} * \{l : B\}} \qquad \textsc{D-var}\ \frac{\alpha * A \in \Delta \qquad \Delta \vdash A \leq B}{\Delta \vdash \alpha * B}$$

$$\textsc{D-ax}\ \frac{A *_{ax} B \qquad \vdash \Delta \qquad \Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A * B}$$

$$\textsc{D-and}\ \frac{\Delta \vdash A_1 * B \qquad \Delta \vdash A_2 * B}{\Delta \vdash A_1 \,\&\, A_2 * B} \qquad \textsc{D-symm}\ \frac{\Delta \vdash B * A}{\Delta \vdash A * B} \qquad \textsc{D-arr}\ \frac{\Delta \vdash A_1 \qquad \Delta \vdash B_1 \qquad \Delta \vdash A_2 * B_2}{\Delta \vdash A_1 \to A_2 * B_1 \to B_2}$$

$$\textsc{D-forall}\ \frac{\Delta \vdash A_1 \qquad \Delta \vdash A_2 \qquad \Delta, \alpha * A_1 \,\&\, A_2 \vdash B_1 * B_2}{\Delta \vdash \forall \alpha * A_1.\, B_1 * \forall \alpha * A_2.\, B_2}$$

Fig. 5. Declarative subtyping and disjointness. The remaining rules of subtyping directly extend Figure 1 with type contexts.

covered by the type A from expression e, while $e_1 \setminus e_2$ removes components from e_1 that conflict with e_1. It can be interpreted as $e_1 \setminus A_2$ assuming the minimal type of e_2 is A_2. These two expressions enable simple syntactic sugar for a wide range of operations on merges, as shown in Table 1. Finally, the most notable expression in calculi like F_i^+ is the merge operator $e_1 \,,, e_2$, which enables merging two arbitrary expressions.

Context and Well-Formedness. We have two contexts. Type contexts Δ track disjointness information of type variables. Term contexts Γ track the type information of term variables. Besides, three standard well-formedness relations $\Delta \vdash A$, $\vdash \Delta$ and $\Delta \vdash \Gamma$ are defined in the Appendix.

Notations and Specification. As subtyping and type difference now depend on the type context, all the notation needs an update. We still use $A = B$ to denote that type A and B are syntactically equivalent. But here we use $A \equiv_\Delta B$ to denote type equivalence (which is $\Delta \vdash A \leq B$ and $\Delta \vdash B \leq A$) under type context Δ. The specification of type difference is also updated.

Definition 4.1 (Type difference specification). $A \setminus_s B \equiv_\Delta C \triangleq \Delta \vdash A \leq C \ \wedge\ \Delta \vdash B \,\&\, C \leq A \ \wedge\ \Delta \vdash B * C$.

4.2 Subtyping and Disjointness

We now present the subtyping and disjointness relations, which are the same as F_i^+ [Fan et al. 2022].

Subtyping. The subtyping relation shown at the top of Figure 5 is the extension of Figure 1 with disjoint polymorphism. Here we only show the new rules. The remaining rules directly extend Figure 1 with type contexts. The rules DS-distAll and DS-topAll are natural extensions of BCD subtyping for universal types. The rule DS-topVar adds a new rule for universal quantifiers that are supertypes of $\top$. The rule DS-forall is the subtyping relation between universal types. It is very

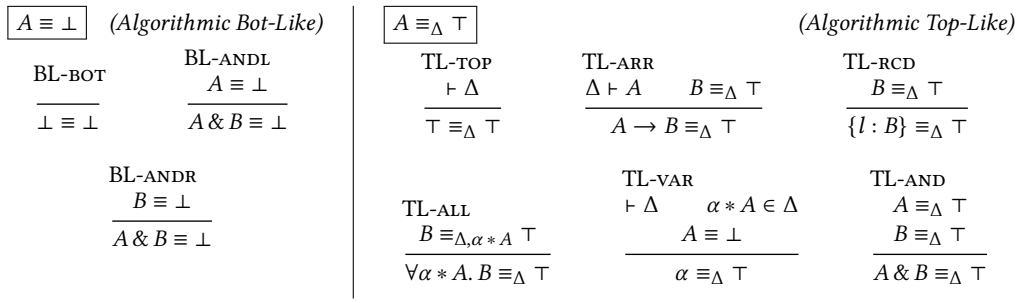

Fig. 6. Algorithmic bottom-like and top-like judgments.

similar to the function subtyping rule S-ARR, being covariant on the result type and contravariant in the argument type.

Disjointness. Disjointness is the same as in F_i^+, and it is shown at the bottom of Figure 5. Rule D-TOP shows that top-like types, which are types equivalent to top, are disjoint to every type. Rule D-VAR shows that a type variable is disjoint to the supertypes of its bound. Rule D-AX is for simple disjointness axioms for types of different shapes, with a full definition in the Appendix. For instance Int is disjoint to $A \to B$. Rule D-AND shows that an intersection type is disjoint with another type when all of its components are disjoint with that type. Rule D-SYMM shows that all the disjointness rules are symmetric. The last three rules D-RCD, D-ARR, and D-FORALL show that the disjointness of types of the same shapes is determined by the substructure of these types.

4.3 Algorithmic Top-Like and Bottom-Like Judgements

Top-like and bottom-like types. Top-like and bottom-like types are types that are equivalent to $\top$ and $\bot$, respectively. Here we only want to use algorithmic top-like types, while the bottom-like type definition serves as an auxiliary relation. We present an algorithmic definition for both relations in Figure 6. The most important addition in F_i^+'s subtyping is the rule TL-VAR for type variables, which allows a type variable to be top-like when its disjointness bound is a bottom-like type. For example, in $\forall \alpha * \bot.\, \alpha$, any type that instantiates α later must be disjoint to $\bot$. Therefore it must be disjoint to every type including itself. The correctness of the top-like judgment is given by:

Lemma 4.2 (Correctness of Algorithmic Top-like Types). *$C \equiv_\Delta \top$ if and only if $\Delta \vdash \top \le C \wedge \Delta \vdash C \le \top$.*

4.4 Zero Type and Deterministic Type Difference with Polymorphism

The zero type function and deterministic type difference are defined in Figure 7. Most extensions related to polymorphism are natural extensions that simply add the type context. Deterministic type difference is essentially the same. The zero type function changes slightly. The only changes are on type variables because we have a new top-like type using rule TL-VAR. Similarly to Section 3, the zero type function needs to have the soundness of zero type property to keep the rule TDA-DISJOINT and rule TDA-SUBTYPE consistent. So we separate two cases for type variable α. For $\alpha_\Delta^\top$, α is returned when the variable α is top-like (i.e., the bound of the variable is bottom-like). Otherwise return the type $\top$ if the variable α is not top-like.

Lemma 4.3 (Soundness of zero type). *If $A \equiv_\Delta \top$, then $A_\Delta^\top = A$.*

$$\top^{\top}_{\Delta} = \top \qquad (A \,\&\, B)^{\top}_{\Delta} = A^{\top}_{\Delta} \,\&\, B^{\top}_{\Delta} \qquad \alpha^{\top}_{\Delta} = \top \text{ (if } \alpha * A \in \Delta \text{ and } A \not\equiv \bot)$$
$$\bot^{\top}_{\Delta} = \top \qquad (A \to B)^{\top}_{\Delta} = A \to B^{\top}_{\Delta} \qquad \alpha^{\top}_{\Delta} = \alpha \text{ (if } \alpha * A \in \Delta \text{ and } A \equiv \bot)$$
$$\mathsf{Int}^{\top}_{\Delta} = \top \qquad \{l : B\}^{\top}_{\Delta} = \{l : B^{\top}_{\Delta}\} \qquad (\forall \alpha * C.\, A)^{\top}_{\Delta} = \forall \alpha * C.\, A^{\top}_{\Delta, \alpha * C}$$

$\boxed{A \setminus_d B =_\Delta C}$ *(Deterministic Type Difference)*

$$\text{TDA-SUBTYPE}\ \frac{\Delta \vdash B \le A}{A \setminus_d B =_\Delta A^{\top}_{\Delta}} \qquad \text{TDA-DISJOINT}\ \frac{\Delta \vdash A * B}{A \setminus_d B =_\Delta A} \qquad \text{TDA-BCD}\ \frac{A_1 \lhd A \rhd A_2 \qquad A \vdash C_1 \rhd C \lhd C_2 \qquad A_1 \setminus_d B =_\Delta C_1 \qquad A_2 \setminus_d B =_\Delta C_2}{A \setminus_d B =_\Delta C}$$

Fig. 7. Zero type function and deterministic type difference.

$$\begin{array}{rcll}
\mathsf{Int} \setminus_a \forall \alpha * A.\, B & =_\Delta & \mathsf{Int} & \\
\{l : A\} \setminus_a \forall \alpha * B.\, C & =_\Delta & \{l : A\} & \\
(A_1 \to A_2) \setminus_a \forall \alpha * B_1.\, B_2 & =_\Delta & A_1 \to A_2 & \\
\forall \alpha * A.\, B \setminus_a \mathsf{Int} & =_\Delta & \forall \alpha * A.\, B & \\
\forall \alpha * A.\, B \setminus_a \{l : C\} & =_\Delta & \forall \alpha * A.\, B & \\
\forall \alpha * A_1.\, A_2 \setminus_a (B_1 \to B_2) & =_\Delta & \forall \alpha * A_1.\, A_2 & \\
\alpha \setminus_a \alpha & =_\Delta & \top & (\text{if } \neg(\alpha \equiv_\Delta \top)) \\
\alpha \setminus_a A & =_\Delta & \alpha & (\text{if } \alpha * B \in \Delta \text{ and } A \setminus_a B =_\Delta A^{\top}_{\Delta}) \\
A \setminus_a \alpha & =_\Delta & A & (\text{if } \alpha * B \in \Delta \text{ and } A \setminus_a B =_\Delta A^{\top}_{\Delta}) \\
\forall \alpha * A_1.\, A_2 \setminus_a \forall \alpha * B_1.\, B_2 & =_\Delta & \forall \alpha * A_1.\, (A_2 \setminus_a B_2) & (\text{if } B_1 \setminus A_1 =_\Delta B_1^{\top}_{\Delta} \text{ and } A_2 \setminus_a B_2 =_{\Delta, \alpha * A_1} A_2^{\top}{}_{\Delta, \alpha * A_1}) \\
\forall \alpha * A_1.\, A_2 \setminus_a \forall \alpha * B_1.\, B_2 & =_\Delta & \forall \alpha * A_1.\, A_2 & (\text{if } A_2 \setminus_a B_2 =_{\Delta, \alpha * (A_1 \,\&\, B_1)} A_2)
\end{array}$$

Fig. 8. Selected algorithmic rules for type difference. Full rules can be found in the Appendix.

4.5 Algorithmic Type Difference and Theorems

Algorithmic Type Difference. Figure 8 shows selected rules for type difference, focusing on rules involving disjoint quantification. The rules assume that the type context and two types are all well-formed. We can still classify the rules into three categories: subtyping cases, disjoint cases and mixed cases. The correctness of such classification is ensured by Lemmas 3.12 to 3.14 extended by a type context. These extended lemmas also guarantee that we can still give an algorithm for subtyping by checking $A \setminus_a B =_\Delta A^{\top}_{\Delta}$, and an algorithm for disjointness by checking $A \setminus_a B =_\Delta A$.

Theorems. Most of the properties are preserved. Using the same techniques as in Section 3, we will reach the same main results below.

Lemma 4.4 (Type Difference Equivalence). *$A \setminus_d B =_\Delta C$ if and only if $A \setminus_a B =_\Delta C$.*

Theorem 4.5 (Type Difference Determinism). *If $A \setminus_d B =_\Delta C_0$ and $A \setminus_d B =_\Delta C_1$, then $C_0 = C_1$.*

Theorem 4.6 (Type Difference Completeness). *If $A \setminus_d B =_\Delta C$, then $A \setminus_s B \equiv_\Delta C$.*

Theorem 4.7 (Type Difference Soundness). *$A \setminus_s B \equiv_\Delta C$ iff $\exists C_0, A \setminus_d B =_\Delta C_0 \wedge C \equiv_\Delta C_0$.*

Theorem 4.8 (Decidability of Subtyping). *$\Delta \vdash A \le B$ is decidable.*

Theorem 4.9 (Decidability of Type Difference). *Type difference is decidable.*

4.6 Typing

We show the bidirectional typing rules in Figure 9. The gray parts are for elaborating expressions to the target language, which we will explain in the next section. We keep most of the typing rules of F_i^+ except for six rules Ela-rabs, Ela-annotabs, Ela-diff, Ela-diffe, Ela-rcdinf, and Ela-proj. The other typing rules of F_i^+ are explained in the Section 5. Rule Ela-rabs and rule Ela-annotabs are the rules that help us inferring and checking the newly-added lambda and type abstractions. Rule Ela-rcdinf offers a more convenient infer mode for records, which reduces the number of annotations with records. In F_i^+ there is only a checking rule for records. Rule Ela-diff and rule Ela-diffe are the rules that employ type difference. Rule Ela-diff means removing the type A component of an expression e, while rule Ela-diffe means removing the conflicting type components of e_1 that appear in another expression e_2. Rule Ela-proj is changed with a record projection operator $\blacktriangleright_l$ shown in Figure 9 to support multirecord selection so that an arbitrary label l_i can be retrieved from a record $\{l_1 = e_1, , ..., , l_n = e_n\}$. The latter is not directly supported by F_i^+.

5 THE TARGET LANGUAGE AND ELABORATION

This section shows the target language F_i^+ [Fan et al. 2022] and explains the elaboration of $\mathsf{F}_i^{\rangle}$.

5.1 Typing of F_i^+

If we remove the six changed rules Ela-rabs, Ela-annotabs, Ela-diff, Ela-diffe, Ela-proj, and Ela-rcdinf of $\mathsf{F}_i^{\rangle}$ as well as the elaboration component, then we have the typing rules for F_i^+ needed to show the type-safety of the elaboration.

Bidirectional Typing. The typing of F_i^+ is bidirectional [Pierce and Turner 2000]. Under the inference mode $\Rightarrow$, the minimal type of the expression is computed. Under the checking mode $\Leftarrow$, we examine whether the expression is convertible to the given type. F_i^+ chooses to use a bidirectional type system because the disjoint merge is incompatible with a general subsumption rule [Huang et al. 2021; Oliveira et al. 2016]. For example, Int is disjoint to Bool, thus we can merge terms like $1 \,,, \mathsf{True}$. But without a bidirectional type system returning the exact type, we may encounter a situation where $1 \,,, \mathsf{True}$ is merged with a boolean (since $\mathsf{Int} \,\&\, \mathsf{Bool} \leq \mathsf{Int}$), causing ambiguity. The rule Ela-lit states that the type of an integer i is Int. The rule Ela-sub is the standard subsumption rule. The rule Ela-anno shows how to check whether the expression has a given type. The rule Ela-merge allows expressions of disjoint types A and B to be merged safely. The remaining F_i^+ rules are standard bidirectional typing for lambda abstractions, applications, type abstractions, fixpoints, etc.

5.2 Elaboration to F_i^+

The elaboration generates an F_i^+ expression in the gray parts of Figure 9. Most of the rules generate an expression similar to the source expression. For instance, rule Ela-lit just keeps the same integer i from the premise. The rule Ela-anno adds an annotation to the elaborated term e' just as the term e being typed. The rules Ela-rabs, Ela-annotabs, and Ela-rcdinf add a type annotation so that the new terms can be expressed and typed in F_i^+. Only rules Ela-proj, Ela-diff, and Ela-diffe change the structure.

Record Selection. The record projection operator $\blacktriangleright_l$, shown at the bottom of Figure 9, is a function that will return $\top$ if it cannot retrieve any contents of label l, otherwise it will return a record that contains all the information. The elaboration of record projection adds an annotation $\{l : C\}$ in the elaboration so that multi-record selection can be interpreted by the target language. We also have the following lemmas to show the elaboration is safe and deterministic.

Lemma 5.1 (Determinism of Record Distribution). *If $A \blacktriangleright_l B$ and $A \blacktriangleright_l C$ then $B = C$.*

$\boxed{\Delta; \Gamma \vdash e_1 \Leftrightarrow A \rightsquigarrow e_2}$ *(Typing)*

$$\textsc{Ela-top}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash \{\} \Rightarrow \top \rightsquigarrow \{\}} \qquad \textsc{Ela-lit}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash i \Rightarrow \mathsf{Int} \rightsquigarrow i} \qquad \textsc{Ela-var}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma \qquad x : A \in \Gamma}{\Delta; \Gamma \vdash x \Rightarrow A \rightsquigarrow x}$$

$$\textsc{Ela-rabs}\ \frac{\Delta; \Gamma, x : A \vdash e \Leftarrow B \rightsquigarrow e_1}{\Delta; \Gamma \vdash \lambda x.\, e \Leftarrow A \to B \rightsquigarrow \lambda x : A.\, e_1} \qquad \textsc{Ela-abs}\ \frac{\Delta; \Gamma, x : A \vdash e \Rightarrow B \rightsquigarrow e_1}{\Delta; \Gamma \vdash \lambda x : A.\, e \Rightarrow A \to B \rightsquigarrow (\lambda x : A.\, e_1) : A \to B}$$

$$\textsc{Ela-rcdinf}\ \frac{\Delta; \Gamma \vdash e \Rightarrow A \rightsquigarrow e'}{\Delta; \Gamma \vdash \{l = e\} \Rightarrow \{l : A\} \rightsquigarrow \{l = e'\} : \{l : A\}} \qquad \textsc{Ela-tabs}\ \frac{\Delta \vdash \Gamma \qquad \Delta, \alpha * A; \Gamma \vdash e \Leftarrow B \rightsquigarrow e_1}{\Delta; \Gamma \vdash \Lambda\alpha.\, e \Leftarrow \forall \alpha * A.\, B \rightsquigarrow \Lambda\alpha.\, e_1}$$

$$\textsc{Ela-annotabs}\ \frac{\Delta \vdash \Gamma \qquad \Delta, \alpha * A; \Gamma \vdash e \Rightarrow B \rightsquigarrow e_1}{\Delta; \Gamma \vdash \Lambda(\alpha * A).\, e \Rightarrow \forall \alpha * A.\, B \rightsquigarrow (\Lambda\alpha.\, e_1) : \forall \alpha * A.\, B} \qquad \textsc{Ela-proj}\ \frac{A \blacktriangleright_l \{l : C\} \qquad \Delta; \Gamma \vdash e \Rightarrow A \rightsquigarrow e'}{\Delta; \Gamma \vdash e.l \Rightarrow C \rightsquigarrow (e' : \{l : C\}).l}$$

$$\textsc{Ela-tapp}\ \frac{\Delta \vdash A * C_1 \qquad \Delta; \Gamma \vdash e \Rightarrow B \rightsquigarrow e' \qquad B \rhd \forall \alpha * C_1.\, C_2}{\Delta; \Gamma \vdash e\, A \Rightarrow C_2[\alpha \mapsto A] \rightsquigarrow e'\, A} \qquad \textsc{Ela-diff}\ \frac{\Delta; \Gamma \vdash e \Rightarrow A \rightsquigarrow e' \qquad A \setminus_d B =_\Delta C}{\Delta; \Gamma \vdash e \setminus B \Rightarrow C \rightsquigarrow e' : C}$$

$$\textsc{Ela-app}\ \frac{A \rhd B \to C \qquad \Delta; \Gamma \vdash e_1 \Rightarrow A \rightsquigarrow e_1' \qquad \Delta; \Gamma \vdash e_2 \Leftarrow B \rightsquigarrow e_2'}{\Delta; \Gamma \vdash e_1\, e_2 \Rightarrow C \rightsquigarrow e_1'\, e_2'} \qquad \textsc{Ela-sub}\ \frac{\Delta \vdash A \leq B \qquad \Delta; \Gamma \vdash e \Rightarrow A \rightsquigarrow e'}{\Delta; \Gamma \vdash e \Leftarrow B \rightsquigarrow e'}$$

$$\textsc{Ela-merge}\ \frac{\Delta \vdash A * B \qquad \Delta; \Gamma \vdash e_1 \Rightarrow A \rightsquigarrow e_1' \qquad \Delta; \Gamma \vdash e_2 \Rightarrow B \rightsquigarrow e_2'}{\Delta; \Gamma \vdash e_1 \,,, e_2 \Rightarrow A \,\&\, B \rightsquigarrow e_1' \,,, e_2'} \qquad \textsc{Ela-fix}\ \frac{\Delta; \Gamma, x : A \vdash e \Leftarrow A \rightsquigarrow e'}{\Delta; \Gamma \vdash \mathsf{fix}\ x : A.\, e \Rightarrow A \rightsquigarrow \mathsf{fix}\ x : A.\, e'}$$

$$\textsc{Ela-anno}\ \frac{\Delta; \Gamma \vdash e \Leftarrow A \rightsquigarrow e'}{\Delta; \Gamma \vdash (e : A) \Rightarrow A \rightsquigarrow (e' : A)} \qquad \textsc{Ela-diffe}\ \frac{A \setminus_d B =_\Delta C \qquad \Delta; \Gamma \vdash e_1 \Rightarrow A \rightsquigarrow e_1' \qquad \Delta; \Gamma \vdash e_2 \Rightarrow B \rightsquigarrow e_2'}{\Delta; \Gamma \vdash e_1 \setminus e_2 \Rightarrow C \rightsquigarrow e_1' : C}$$

$\boxed{A \blacktriangleright_l B}$ *(Record Projection Distribution)*

$$\frac{}{\{l : A\} \blacktriangleright_l \{l : A\}} \qquad \frac{}{\top \blacktriangleright_l \top} \qquad \frac{}{A_1 \to A_2 \blacktriangleright_l \top} \qquad \frac{}{\forall \alpha * A_1.\, A_2 \blacktriangleright_l \top} \qquad \frac{l_1 \neq l_2}{\{l_2 : A\} \blacktriangleright_{l_1} \top}$$

$$\frac{A_1 \blacktriangleright_l \top \qquad A_2 \blacktriangleright_l \top}{A_1 \,\&\, A_2 \blacktriangleright_l \top} \qquad \frac{A_1 \blacktriangleright_l \top \qquad A_2 \blacktriangleright_l \{l : B\}}{A_1 \,\&\, A_2 \blacktriangleright_l \{l : B\}} \qquad \frac{A_1 \blacktriangleright_l \{l : B\} \qquad A_2 \blacktriangleright_l \top}{A_1 \,\&\, A_2 \blacktriangleright_l \{l : B\}}$$

$$\frac{A_1 \blacktriangleright_l \{l : B_1\} \qquad A_2 \blacktriangleright_l \{l : B_2\}}{A_1 \,\&\, A_2 \blacktriangleright_l \{l : B_1 \,\&\, B_2\}}$$

Fig. 9. Bidirectional type system for $\mathsf{F}^{\rangle}_{i}$. Note that the syntax for modes is: $\Leftrightarrow \triangleq \Leftarrow \mid \Rightarrow$.

Lemma 5.2 (Safety of Record Distribution). *If* $\Delta \vdash A, \vdash \Delta$ *and* $A \blacktriangleright_l B$, *then* $\Delta \vdash A \leq B$.

Adding Type Difference. Adding type difference is straightforward, since we just have to cast the expression with the corresponding type computed by type difference. The safety of such casts is guaranteed by the equivalence with the type difference specification. We can get the following property from the specification. Thus casting to type C is always safe.

Lemma 5.3 (Completeness of Subtyping). *If* $A \setminus_d B =_\Delta C$, *then* $\Delta \vdash A \leq C$.

5.3 Theorems

Since F_i^+ [Fan et al. 2022] is proven to be deterministic and type sound, we only need to show that the elaboration is deterministic and all elaborated terms are typeable in F_i^+. We first show that the elaboration is complete with respect to typing, that is:

Theorem 5.4 (Completeness of Elaboration with respect to Typing). $\exists e_1, \Delta; \Gamma \vdash e \Leftrightarrow A \rightsquigarrow e_1$ *iff* $\Delta; \Gamma \vdash e \Leftrightarrow A$, *where the typing here is the elaboration without the elaborated term.*

Then by the determinism of $A \blacktriangleright_l B$, we can prove the determinism of the elaboration.

Theorem 5.5 (Determinism of Inference). *If* $\Delta; \Gamma \vdash e \Rightarrow A \rightsquigarrow e_1$ *and* $\Delta; \Gamma \vdash e \Rightarrow B \rightsquigarrow e_2$, *then* $e_1 = e_2$ *and* $A = B$.

Theorem 5.6 (Determinism of Checking). *If* $\Delta; \Gamma \vdash e \Leftarrow A \rightsquigarrow e_1$ *and* $\Delta; \Gamma \vdash e \Leftarrow A \rightsquigarrow e_2$, *then* $e_1 = e_2$.

As for the typing results, since most of the elaboration rules keep the exact same terms and types, we only need non-trivial proofs for the six newly added cases. But the added cases use annotations to ensure that we get the desired type. So we have the following results, ensuring that the elaboration preserves the same type after translation.

Theorem 5.7 (Type Preservation). *If* $\Delta; \Gamma \vdash e \Leftrightarrow A \rightsquigarrow e'$, *then in* F_i^+, $\Delta; \Gamma \vdash e' \Leftrightarrow A$.

Summing up, we get the soundness of the calculus, i.e., the calculus is type-safe and deterministic.

6 RELATED WORK

In Section 2 we have already discussed in detail most closely related work, including set difference in semantic subtyping [Frisch et al. 2008], type-indexed rows [Shields and Meijer 2001] and the work by Cardelli and Mitchell. Here we discuss other related work.

Record Calculi. Wand [1989] proposed a type system that requires constraints to signal whether a field is present or absent in every record type. His calculus uses a biased operator which always overrides the first record by the second when these two records have conflicts. Harper and Pierce [1991b] enforce a compatibility check on every record concatenation and include the extension and restriction operators. Their compatibility check # means that two records are safe to concatenate without conflicts, providing a symmetric, unbiased merge. Compatibility is similar to disjointness in $\mathsf{F}_i^{\rangle}$. However, both Wand and Harper and Pierce do not consider subtyping.

Designing a record concatenation operator in a calculus with subtyping is a difficult problem, as identified by Cardelli and Mitchell [1991]. Records may contain extra labels hidden by the subsumption rule, causing label conflicts to bypass the type system. Pottier [2000], extending the work of Rémy [1995], uses a constraint-based subtyping system to resolve label conflicts. Its constraints require the presence or absence of certain labels, and supports asymmetric and symmetric concatenation. Cardelli and Mitchell's work uses a subtyping system and negative quantification to resolve conflicts, which is described in detail in Section 2. Zwanenburg [1995]'s adds record concatenation

to a variant of System $F_{<:}$ and uses a compatibility check in the quantification. For example, his functions are of the form $\lambda X \leq T ; X\#T.e$. His calculus includes intersection types and can be elaborated to System F with pairs and records. Ohori [1992]'s work first uses a notion of record kind, while Alves and Ramos [2021]'s work extends the record kind with negative information. In Alves and Ramos's work, a record kind requires explicit statements of whether a field is present or absent in the type. For example, the kind $\{\{l_1 : \tau_1, ..., l_n : \tau_n \| l'_0 : \tau'_1, ..., l'_m : \tau'_m\}\}$ denotes the set of types contain the fields before $\|$ and do not contain the fields after $\|$.

The commonality between these works is that they either use a biased operator in evaluation or use negative quantification like compatibility or label absence in the type system. Both methods are encodable in our system. The biased operator can be encoded by $(e_1 \setminus e_2) \,,, e_2$ while disjoint quantification is a negative form of quantification. As for record operators, most works include label extraction and record concatenation. Some of the works include label restriction, renaming and overriding. As we have shown in Section 2, all of these operations are encodable in our system.

Disjoint Intersection Types. Disjoint intersection types were proposed by Oliveira et al. [2016] in the λ_i calculus to address the non-determinism and subject reduction problem in Dunfield [2014]'s work. Dunfield's calculus is equipped with an unrestricted merge operator for intersection types, reducing merges in a non-deterministic process. Terms like $e_1 \,,, e_2$ can be reduced to either e_1 or e_2. λ_i imposes a disjointness restriction on the merge operator: only terms with disjoint types can be merged. Therefore, reduction for the merge operator can be deterministic once the final type of the program is decided. There are several extensions to λ_i [Alpuim et al. 2017; Bi et al. 2018, 2019] with relaxed restrictions, BCD subtyping and disjoint polymorphism. Our work is based on the recent calculus F_i^+ [Bi et al. 2019; Fan et al. 2022], which is a calculus with disjoint polymorphism.

Compositional Programming and Restriction Operators. 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. Usually, the mixin model allows overlapping fields and omits the one with lower priority, like the biased merge operator. In contrast, for two traits to compose, they must have no conflicts, like our unbiased merge operator. Conflicts have to be resolved before composition by renaming, restriction or other operators. But even in the mixin model, such explicit conflict-resolving operators are useful [Bracha 1992]. In Featherweight Jigsaw [Lagorio et al. 2009], as in $F_i^{\rangle}$, both mechanisms are supported. Users can choose between the conflict-free merge and the conflict-auto-solving merge. We share the same interpretation of overriding with Ancona and Zucca [1998, 2002]: overriding = restriction + commutative merge. Both our work and their work consider operators including renaming, restriction, and overriding. While the merge in their work has similar semantics to ours, it is limited to mixins and it always has a mixin type. Moreover, we develop a general type difference operator that works for types other than records (or record-like things like objects/modules), and we recover the operators on records/traits from that.

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 uses disjoint intersection types to naturally solve the Expression Problem [Wadler 1998] and offers modular pattern matching and dependency injection. In earlier versions of the CP language, only a weak (and ad-hoc) restriction operator was supported due to the lack of a theory of type difference. This operator enabled explicitly resolving conflicts on traits:

```
t1 = trait ⇒ { f = 1; g = "a" };
t3 = trait [self: Top] inherits t1 \ {f: Int} & t2 \ {g: String} ⇒ {
  override f = super.f + (t1 ^ self).f};
```

Since we can only remove a certain label using an exclusion operator \, this still required user-written annotations when dealing with merges and inheritance.

Similar approaches are also adopted in previously mentioned OOP designs except that they provide a more convenient way to override or rename methods to resolve conflicts. But all these works only discuss label manipulation, not dealing with function (and other forms of) type difference. In the CP language, such an ad-hoc restriction is especially problematic because it requires explicit type annotations, such as `{f : Int}` above. But with the new operators based on the type difference that we provide, we can now address the problem in CP with fewer annotations and also convenient operators such as biased merges, which were previously not available. As we have shown in Section 2.4, type difference is more generally applicable since it is not restricted to objects or records. This enables new applications, such as updating implementations of overloaded functions. Furthermore it plays well with our encoding of traits, which is based on functions that return records, rather than primitive objects or records.

Semantic Subtyping. As we have discussed in Section 2.3, semantic subtyping [Castagna and Frisch 2005; Frisch et al. 2008] adopts a set-theoretic view to define the subtraction of two types. Current systems with semantic subtyping support records and related operations with biased record concatenation [Benzaken et al. 2013; Castagna 2018]. A type-level merge operator is defined to calculate the type for concatenated or restricted records. Records have a deterministic runtime behavior, but their typing rule is quite unique. They are not typed by intersections but by label-indexed function types that may have a union type as its return type. Here we show the definition of the type-level operator as it also uses the type difference defined in semantic subtyping.

$$(R_1 \,,_A R_2)(l) \equiv \begin{cases} R_1(l) & \text{if } R_1(l)\&A \leq \bot \\ (R_1(l)\backslash A)|R_2(l) & \text{otherwise} \end{cases}$$

In the second case, $R_1(l)$ subtracts A and adds $R_2(l)$ back. In concatenation, A is a special constant that stands for an undefined field and R_1 is the rightmost record, which has the highest priority. So when R_1 is undefined on l, the corresponding field type in R_2 will be returned. We can see here that union types play a similar role to intersection types in our system as we have very different interpretations. This operator is specific to records, unlike our type difference, which is generalized to cover other types.

7 CONCLUSION

In this paper, we present a theory of type difference, and design a $\mathsf{F}_i^{\setminus}$ calculus carrying out such type difference via an elaboration to the F_i^{+} calculus. We derive a general type difference that works for all types from a specification with three simple but essential requirements to a sound, complete, deterministic and algorithmic formulation, verified by the Coq proof assistant. Besides the theoretical aspects, type difference is expressive enough to encode all the record operators in record calculi with subtyping. These operators include concatenation, restriction, overriding and renaming. Type difference is also useful for languages with traits and compositional programming, since it can deal with inheritance and merge conflicts, avoiding heavy annotations that would be otherwise necessary. Future work includes studying even more expressive subtyping relations that include additional features, such as union types [Barbanera et al. 1995].

ACKNOWLEDGMENTS

We are particularly grateful to Yaozhu Sun, who implemented type difference in CP. We thank the anonymous reviewers for their helpful comments. This work has been sponsored by Hong Kong Research Grant Council projects number 17209519, 17209520 and 17209821.

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

Sandra Alves and Miguel Ramos. 2021. An ML-style Record Calculus with Extensible Records. *Electronic Proceedings in Theoretical Computer Science* 351 (dec 2021), 1–17. https://doi.org/10.4204/eptcs.351.1

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. https://doi.org/10.1145/937563.937567

Davide Ancona and Elena Zucca. 1998. A theory of mixin modules: Basic and derived operators. *Mathematical structures in computer science* 8, 4 (1998), 401–446. https://doi.org/10.1017/S0960129598002576

Davide Ancona and Elena Zucca. 2002. A calculus of module systems. *Journal of functional programming* 12, 2 (2002), 91–132. https://doi.org/10.1017/S0956796801004257

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

Véronique Benzaken, Giuseppe Castagna, Kim Nguyen, and Jérôme Siméon. 2013. Static and dynamic semantics of NoSQL languages. *ACM SIGPLAN Notices* 48, 1 (2013), 101–114. https://doi.org/10.1145/2480359.2429083

Xuan Bi and Bruno C. d. S. Oliveira. 2018. Typed First-Class Traits. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. https://doi.org/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. https://doi.org/10.4230/LIPIcs.ECOOP.2018.22

Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. Distributive Disjoint Polymorphism for Compositional Programming. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-030-17184-1_14

Gilad Bracha. 1992. The programming language Jigsaw: mixins, modularity and multiple inheritance. The University of Utah.

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

Luca Cardelli, Simone Martini, John C Mitchell, and Andre Scedrov. 1994. An extension of system F with subtyping. *Information and Computation* 109, 1-2 (1994), 4–56.

Luca Cardelli and John Mitchell. 1991. Operations on Records. *Mathematical Structures in Computer Science* 1 (1991), 3–48.

Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.* 17, 4 (dec 1985), 471–523. https://doi.org/10.1145/6041.6042

Giuseppe Castagna. 2018. Covariance and Controvariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers). *CoRR* abs/1809.01427 (2018). arXiv:1809.01427 http://arxiv.org/abs/1809.01427

Giuseppe Castagna and Alain Frisch. 2005. A Gentle Introduction to Semantic Subtyping. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Lisbon, Portugal) *(PPDP '05)*. Association for Computing Machinery, New York, NY, USA, 198–208. https://doi.org/10.1145/1069774.1069793

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. https://doi.org/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. https://doi.org/10.1145/2535838.2535840

Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyundefinedn. 2016. Set-Theoretic Types for Polymorphic Variants. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 378–391. https://doi.org/10.1145/2951913.2951928

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. https://doi.org/10.1145/2034773.2034788

William R. Cook and Jens Palsberg. 1989. A denotational semantics of inheritance and its correctness. In *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/74878.74922

Mario Coppo and Mariangiola Dezani-Ciancaglini. 1978. A new type assignment for λ-terms. *Archiv für mathematische Logik und Grundlagenforschung* 19, 1 (1978), 139–156.

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 1981. Functional Characters of Solvable Terms. *Math. Log. Q.* 27, 2-6 (1981), 45–58. https://doi.org/10.1002/malq.19810270205

Rowan Davies and Frank Pfenning. 2000. Intersection Types and Computational Effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 198–208. https://doi.org/10.1145/351240.351259

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. https://doi.org/10.1006/inco.1995.1086

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. https://doi.org/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. https://doi.org/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. https://doi.org/10.1145/1391289.1391293

Robert Harper and Benjamin Pierce. 1991a. A record calculus based on symmetric concatenation. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 131–142. https://doi.org/10.1145/99583.99603

Robert Harper and Benjamin Pierce. 1991b. A Record Calculus Based on Symmetric Concatenation. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Orlando, Florida, USA) *(POPL '91)*. Association for Computing Machinery, New York, NY, USA, 131–142. https://doi.org/10.1145/99583.99603

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

Young Bae Jun, Hee Sik Kim, and Eun Hwan Roh. 2004. Ideal theory of subtraction algebras. *Sci. Math. Jpn. Online e-2004* (2004), 397–402.

Giovanni Lagorio, Marco Servetto, and Elena Zucca. 2009. Featherweight Jigsaw: A minimal core calculus for modular composition of classes. In *European Conference on Object-Oriented Programming*. Springer, 244–268.

Zhaohui Luo. 1999. Coercive Subtyping. *J. Log. Comput.* 9, 1 (1999), 105–130. https://doi.org/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). https://doi.org/10.1145/3428274

Atsushi Ohori. 1992. A Compilation Method for ML-Style Polymorphic Record Calculi. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico, USA) *(POPL '92)*. Association for Computing Machinery, New York, NY, USA, 154–165. https://doi.org/10.1145/143165.143200

Atsushi Ohori. 1995. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17, 6 (1995), 844–895. https://doi.org/10.1145/218570.218572

Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint intersection types. In *International Conference on Functional Programming (ICFP)*. https://doi.org/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. https://doi.org/10.1145/345099.345100

François Pottier. 2000. A 3-Part Type Inference Engine. In *Programming Languages and Systems*, Gert Smolka (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 320–335.

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

Didier Rémy. 1990. Type inference for records in a natural extension of ML. *Technical Reports (CIS)* (1990), 641.

Didier Rémy. 1995. A case study of typechecking with constrained types: Typing record concatenation. (August 1995). Presented at the workshop on Advances in types for computer science at the Newton Institute, Cambridge, UK.

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.

Scala Community. 2022. Tour of Scala Self-type. https://docs.scala-lang.org/tour/self-types.html.

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

Boris M Schein. 1992. Difference semigroups. *Communications in algebra* 20, 8 (1992), 2153–2169.

Mark Shields and Erik Meijer. 2001. Type-Indexed Rows. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (London, United Kingdom) *(POPL '01)*. Association for Computing Machinery, New York, NY, USA, 261–275. https://doi.org/10.1145/360204.360230

Pablo Tesone, Stéphane Ducasse, Guillermo Polito, Luc Fabresse, and Noury Bouraqadi. 2020. A new modular implementation for stateful traits. *Science of Computer Programming* 195 (2020), 102470.

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

M. Wand. 1989. Type inference for record concatenation and multiple inheritance. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*. 92–97. https://doi.org/10.1109/LICS.1989.39162

Ningning Xie, Bruno C. d. S. Oliveira, Xuan Bi, and Tom Schrijvers. 2020. Row and Bounded Polymorphism via Disjoint Polymorphism. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. https://doi.org/10.4230/LIPIcs.ECOOP.2020.27

Han Xu, Xuejing Huang, and Bruno C. d. S. Oliveira. 2022. Making a Type Difference: Subtraction on Intersection Types as Generalized Record Operations (Artifact). https://doi.org/10.5281/zenodo.7151418

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. https://doi.org/10.1145/3460228

Jan Zwanenburg. 1995. *Record concatenation with intersection types*. Technical Report 95-34. Eindhoven University of Technology.

A APPENDIX

The appendix mainly contains the definitions omitted in the main paper and the texts for them.

A.1 Top-like Types without Polymorphism

Here, we call a type to be top-like if it is equivalent to $\top$, i.e. $A \equiv \top$. Since the rules regarding supertype of $\top$ is only given by rules S-TopArr and S-TopRcd, and their transitive closure using the rules S-Refl, S-Trans, and S-And, we are able to give an algorithmic definition of types that is equivalent to $\top$, shown in Figure 10. The following theorem guarantees the safety of the algorithm.

Lemma A.1 (Safety of Algorithmic Top-like Types). *$A \equiv \top$ if and only if $A \leq \top$ and $\top \leq A$.*

$\boxed{A \equiv \top}$ *(Algorithmic Top-Like Types)*

$$\text{OTL-Top}\ \frac{}{\top \equiv \top} \qquad \text{OTL-And}\ \frac{A \equiv \top \qquad B \equiv \top}{A \,\&\, B \equiv \top} \qquad \text{OTL-Arr}\ \frac{B \equiv \top}{A \to B \equiv \top} \qquad \text{OTL-Rcd}\ \frac{B \equiv \top}{\{l : B\} \equiv \top}$$

Fig. 10. Algorithmic top-like types.

A.2 Well-formedness

The complete definition of well-formedness is defined in Figure 11.

$\boxed{\Delta \vdash A}$ *(Type Well-Formedness)*

$$\text{TW-Top}\ \frac{}{\Delta \vdash \top} \qquad \text{TW-Bot}\ \frac{}{\Delta \vdash \bot} \qquad \text{TW-Int}\ \frac{}{\Delta \vdash \mathsf{Int}} \qquad \text{TW-Var}\ \frac{\alpha * A \in \Delta}{\Delta \vdash \alpha} \qquad \text{TW-Rcd}\ \frac{\Delta \vdash A}{\Delta \vdash \{l : A\}} \qquad \text{TW-Arr}\ \frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \to B}$$

$$\text{TW-And}\ \frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \,\&\, B} \qquad \text{TW-All}\ \frac{\Delta \vdash A \qquad \Delta, \alpha * A \vdash B}{\Delta \vdash \forall \alpha * A.\, B}$$

$\boxed{\Delta \vdash \Gamma}$ *(Term Well-Formedness)*

$$\text{CW-Empty}\ \frac{}{\Delta \vdash \cdot} \qquad \text{CW-Cons}\ \frac{\Delta \vdash \Gamma \qquad \Delta \vdash A}{\Delta \vdash \Gamma, x : A}$$

$\boxed{\Delta \vdash A}$ *(Type Context Well-Formedness)*

$$\text{TCW-Empty}\ \frac{}{\vdash \cdot} \qquad \text{TCW-Cons}\ \frac{\vdash \Delta \qquad \Delta \vdash A}{\vdash \Delta, \alpha * A}$$

Fig. 11. Type, term and type context well-formedness.

A.3 Disjoint Axioms

Disjoint axioms shows the disjointness of types of different shapes, with the full version shown in Figure 12. Because we have the symmetric rule in the disjointness, so here we do not need to repeat rules by flipping them.

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

$$\textsc{Dax-intArr}\ \frac{}{\mathsf{Int} *_{ax} A_1 \to A_2} \qquad \textsc{Dax-intRcd}\ \frac{}{\mathsf{Int} *_{ax} \{l : A\}} \qquad \textsc{Dax-intForall}\ \frac{}{\mathsf{Int} *_{ax} \forall \alpha * A.\, B} \qquad \textsc{Dax-arrRcd}\ \frac{}{A_1 \to A_2 *_{ax} \{l : A\}}$$

$$\textsc{Dax-arrForall}\ \frac{}{A_1 \to A_2 *_{ax} \forall \alpha * A.\, B} \qquad \textsc{Dax-rcdForall}\ \frac{}{\{l : C\} *_{ax} \forall \alpha * A.\, B} \qquad \textsc{Dax-rcdNeq}\ \frac{l_1 \neq l_2}{\{l_1 : A\} *_{ax} \{l_2 : B\}}$$

Fig. 12. Disjoint axioms.

A.4 Splittable and Mergeable Types

Splittable and mergeable types also extend with universal types, shown in Figure 13.

$\boxed{B \lhd A \rhd C}$ *(Type Splitting)*

$$\textsc{Sp-and}\ \frac{}{A \lhd A \,\&\, B \rhd B} \qquad \textsc{Sp-arr}\ \frac{C_1 \lhd B \rhd C_2}{A \to C_1 \lhd A \to B \rhd A \to C_2} \qquad \textsc{Sp-rcd}\ \frac{C_1 \lhd B \rhd C_2}{\{l : C_1\} \lhd \{l : B\} \rhd \{l : C_2\}}$$

$$\textsc{Sp-forall}\ \frac{C_1 \lhd B \rhd C_2}{\forall \alpha * A.\, C_1 \lhd \forall \alpha * A.\, B \rhd \forall \alpha * A.\, C_2}$$

$\boxed{A \vdash B_1 \rhd C \lhd B_2}$ *(Type Merging)*

$$\textsc{Me-and}\ \frac{}{A_1 \,\&\, A_2 \vdash B_1 \rhd B_1 \,\&\, B_2 \lhd B_2} \qquad \textsc{Me-arr}\ \frac{A \vdash B_1 \rhd C \lhd B_2}{D \to A \vdash D \to B_1 \rhd D \to C \lhd D \to B_2}$$

$$\textsc{Me-rcd}\ \frac{A \vdash B_1 \rhd C \lhd B_2}{\{l : A\} \vdash \{l : B_1\} \rhd \{l : C\} \lhd \{l : B_2\}} \qquad \textsc{Me-forall}\ \frac{A \vdash B_1 \rhd C \lhd B_2}{\forall \alpha * A_1.\, A \vdash \forall \alpha * A_1.\, B_1 \rhd \forall \alpha * A_1.\, C \lhd \forall \alpha * A_1.\, B_2}$$

Fig. 13. The full splittable and mergeable types.

A.5 Algorithmic Type Difference

If the subtrahend and minuend are all well-formed, the full algorithmic type difference can be defined as Figure 14.

$$
\begin{array}{rcll}
A \setminus_a B & =_\Delta & A & (\text{if } A \equiv_\Delta \top) \\
B \setminus_a A & =_\Delta & B & (\text{if } A \equiv_\Delta \top) \\
A \setminus_a \bot & =_\Delta & A^\top_\Delta & \\
\mathsf{Int} \setminus_a (A \to B) & =_\Delta & \mathsf{Int} & \\
\mathsf{Int} \setminus_a \{l : A\} & =_\Delta & \mathsf{Int} & \\
\mathsf{Int} \setminus_a \forall \alpha * A.\, B & =_\Delta & \mathsf{Int} & \\
\{l : A\} \setminus_a (B \to C) & =_\Delta & \{l : A\} & \\
\{l : A\} \setminus_a \mathsf{Int} & =_\Delta & \{l : A\} & \\
\{l : A\} \setminus_a \forall \alpha * B.\, C & =_\Delta & \{l : A\} & \\
(A \to B) \setminus_a \mathsf{Int} & =_\Delta & A \to B & \\
(A \to B) \setminus_a \{l : C\} & =_\Delta & A \to B & \\
(A_1 \to A_2) \setminus_a \forall \alpha * B_1.\, B_2 & =_\Delta & A_1 \to A_2 & \\
\forall \alpha * A.\, B \setminus_a \mathsf{Int} & =_\Delta & \forall \alpha * A.\, B & \\
\forall \alpha * A.\, B \setminus_a \{l : C\} & =_\Delta & \forall \alpha * A.\, B & \\
\forall \alpha * A_1.\, A_2 \setminus_a (B_1 \to B_2) & =_\Delta & \forall \alpha * A_1.\, A_2 & \\
\mathsf{Int} \setminus_a \mathsf{Int} & =_\Delta & \top & \\
\alpha \setminus_a \alpha & =_\Delta & \top & (\text{if } \neg(\alpha \equiv_\Delta \top)) \\
\alpha \setminus_a A & =_\Delta & \alpha & (\text{if } \alpha * B \in \Delta \text{ and } A \setminus_a B =_\Delta A^\top_\Delta) \\
A \setminus_a \alpha & =_\Delta & A & (\text{if } \alpha * B \in \Delta \text{ and } A \setminus_a B =_\Delta A^\top_\Delta) \\
\{l_1 : A\} \setminus_a \{l_2 : B\} & =_\Delta & \{l_1 : A\} & (\text{if } l_1 \neq l_2) \\
\{l : A\} \setminus_a \{l : B\} & =_\Delta & \{l : A \setminus_a B\} & (\text{if } A \setminus_a B =_\Delta A^\top_\Delta) \\
\{l : A\} \setminus_a \{l : B\} & =_\Delta & \{l : A\} & (\text{if } A \setminus_a B =_\Delta A) \\
A_1 \to A_2 \setminus_a B_1 \to B_2 & =_\Delta & A_1 \to (A_2 \setminus_a B_2) & (\text{if } B_1 \setminus A_1 =_\Delta B_1{}^\top_\Delta \text{ and } A_2 \setminus_a B_2 =_\Delta A_2{}^\top_\Delta) \\
A_1 \to A_2 \setminus_a B_1 \to B_2 & =_\Delta & A_1 \to A_2 & (\text{if } A_2 \setminus_a B_2 =_\Delta A_2) \\
\forall \alpha * A_1.\, A_2 \setminus_a \forall \alpha * B_1.\, B_2 & =_\Delta & \forall \alpha * A_1.\, (A_2 \setminus_a B_2) & (\text{if } B_1 \setminus A_1 =_\Delta B_1{}^\top_\Delta \text{ and } A_2 \setminus_a B_2 =_{\Delta, \alpha * A_1} A_2{}^\top_{\Delta, \alpha * A_1}) \\
\forall \alpha * A_1.\, A_2 \setminus_a \forall \alpha * B_1.\, B_2 & =_\Delta & \forall \alpha * A_1.\, A_2 & (\text{if } A_2 \setminus_a B_2 =_{\Delta, \alpha * (A_1 \,\&\, B_1)} A_2) \\
A \setminus_a (B_1 \,\&\, B_2) & =_\Delta & (A \setminus_a B_1) \setminus_a B_2 & (\text{if } A \setminus_a B_1 \text{ proceeds}) \\
A \setminus_a (B_1 \,\&\, B_2) & =_\Delta & (A \setminus_a B_2) \setminus_a B_1 & (\text{if } A \setminus_a B_2 \text{ proceeds}) \\
A \setminus_a B & =_\Delta & C & (\text{if } A_1 \lhd A \rhd A_2 \text{ and } A \vdash (A_1 \setminus_a B) \rhd C \lhd (A_2 \setminus_a B))
\end{array}
$$

Fig. 14. Type difference

A.6 Applicative distribution

Application distribution is the same as in F_i^+. The complete definition is shown in Figure 15.

$\boxed{A \rhd B}$ *(Applicative Distribution)*

$$\text{AD-}\textsc{refl}\ \frac{}{A \rhd A} \qquad \text{AD-}\textsc{andArrow}\ \frac{A_1 \rhd B_1 \to C_1 \quad A_2 \rhd B_2 \to C_2}{A_1 \,\&\, A_2 \rhd B_1 \,\&\, B_2 \to C_1 \,\&\, C_2} \qquad \text{AD-}\textsc{andAll}\ \frac{A_1 \rhd \forall \alpha * B_1.\, C_1 \quad A_2 \rhd \forall \alpha * B_2.\, C_2}{A_1 \,\&\, A_2 \rhd \forall \alpha * B_1 \,\&\, B_2.\, (C_1 \,\&\, C_2)}$$

Fig. 15. Distributivity relations.

A.7 Record Projection Distribution

Multi-record selection is implemented using an auxiliary relation, shown in Figure 16, that will return $\top$ if it cannot retrieve contents of label l, otherwise it will return a record that contains all the information. Note that the calculus supports records with multiple labels with the same name (as long as the types associated with the labels are disjoint). Furthermore, due to the presence of distributivity rules in subtyping, projecting a label must then collect *all* the values associated with the label. The rules ADS-RCDNEQ, ADS-ALLRCD, ADS-TOPRCD, and ADS-ARRRCD show that projection fails to retrieve information for types of different shapes. The rule ADS-REFL shows the exact retrieval for a record of label l. The rules ADS-TOPTOP, ADS-TOPL, ADS-TOPR, and ADS-RCDRCD show some mixed cases for intersection types. These rules combine the information retrieved by both sides.

$\boxed{A \blacktriangleright_l B}$ *(Record Projection Distribution)*

$$\text{ADS-}\textsc{refl}\ \frac{}{\{l : A\} \blacktriangleright_l \{l : A\}} \qquad \text{ADS-}\textsc{toprcd}\ \frac{}{\top \blacktriangleright_l \top} \qquad \text{ADS-}\textsc{toptop}\ \frac{A_1 \blacktriangleright_l \top \quad A_2 \blacktriangleright_l \top}{A_1 \,\&\, A_2 \blacktriangleright_l \top} \qquad \text{ADS-}\textsc{topl}\ \frac{A_1 \blacktriangleright_l \top \quad A_2 \blacktriangleright_l \{l : B\}}{A_1 \,\&\, A_2 \blacktriangleright_l \{l : B\}}$$

$$\text{ADS-}\textsc{topr}\ \frac{A_1 \blacktriangleright_l \{l : B\} \quad A_2 \blacktriangleright_l \top}{A_1 \,\&\, A_2 \blacktriangleright_l \{l : B\}} \qquad \text{ADS-}\textsc{rcdrcd}\ \frac{A_1 \blacktriangleright_l \{l : B_1\} \quad A_2 \blacktriangleright_l \{l : B_2\}}{A_1 \,\&\, A_2 \blacktriangleright_l \{l : B_1 \,\&\, B_2\}} \qquad \text{ADS-}\textsc{arrrcd}\ \frac{}{A_1 \to A_2 \blacktriangleright_l \top}$$

$$\text{ADS-}\textsc{allrcd}\ \frac{}{\forall \alpha * A_1.\, A_2 \blacktriangleright_l \top} \qquad \text{ADS-}\textsc{rcdneq}\ \frac{l_1 \neq l_2}{\{l_2 : A\} \blacktriangleright_{l_1} \top}$$

Fig. 16. Distributivity relations.