Applicative Intersection Types

Xu Xue[1], Bruno C. d. S. Oliveira[1], and Ningning Xie[2]

[1] University of Hong Kong, China
[2] University of Cambridge, UK

Abstract. Calculi with intersection types have been used over the years to model various features, including: *overloading*, *extensible records* and, more recently, *nested composition* and *return type overloading*. Nevertheless no previous calculus supports all those features at once. In this paper we study expressive calculi with intersection types and a merge operator. Our first calculus supports an unrestricted merge operator, which is able to support all the features, and is proven to be type sound. However, the semantics is non-deterministic. In the second calculus we employ a previously proposed disjointness restriction, to make the semantics deterministic. Some forms of overloading are forbidden, but all other features are supported. The main challenge in the design is related to the semantics of applications and record projections. We propose an applicative subtyping relation that enables the inference of result types for applications and projections. Correspondingly, there is an applicative dispatching relation that is used for the dynamic semantics. The two calculi and their proofs are formalized in the Coq theorem prover.

1 Introduction

Calculi with *intersection types* [3,7,19,22] have a long history in programming languages. Reynolds [21] was the first to promote the use of intersection types in practical programming. He introduced a *merge operator* that enables building values with multiple types, where the multiple types are modelled as intersection types. Dunfield [9] refined the merge operator, to add significant additional expressive power over the original formulation by Reynolds. Over the years there have been several calculi with intersection types equipped with a merge operator, and enabling different features: *overloaded functions* [6,9], *return type overloading* [16], *extensible records* [9,21] and *nested composition* [4,14].

Nevertheless, no previous calculus supports all four features together. Some calculi enable function overloading [6], but preclude return type overloading and nested composition. On the other hand, calculi with disjoint intersection types [4,14] support return type overloading and nested composition, but disallow conventional functional overloading. Dunfield's calculus [9] supports the first three features, but not nested composition. Those features are not completely orthogonal and the interactions between them are interesting, allowing for new applications. However, the interactions also pose new technical challenges.

This paper studies expressive calculi with intersection types and a merge operator. Our goal is to design calculi that deal with all four features at once, and

study the interaction between these features. Our two main focuses are on type inference for *applications* and *record projection*, and the design of the operational semantics for such calculi. To enable all the features we introduce a specialized form of subtyping, called *applicative subtyping*, to deal with the flexible forms of applications and record projection allowed by the calculi. Correspondingly, there is an applicative dispatching relation that is used for the dynamic semantics. In addition, we explore the interactions between features. In particular, overloading and nested composition enable curried overloaded functions, while most previous work [15,6,9,5] only considers uncurried overloaded functions.

Our first calculus supports an unrestricted merge operator. This calculus is able to support all four features, and is proven to be type sound. However, the semantics is non-deterministic. In practice, in an implementation of this calculus we can employ a *biased* or *asymmetric* merge operator, which gives a (biased) preference to values on the left or right side of merges. This approach is similar to the approach taken by Dunfield in her calculus [9], and asymmetric merge (or concatenation) operators are also adopted in several calculi with extensible records [27,20]. In the second calculus we employ a previously proposed disjointness restriction [17], to make the semantics deterministic. Disjointness enables a *symmetric* merge operator, since conflicts in a merge are statically rejected rather than resolved with a biased semantics. In the second calculus some forms of overloading are forbidden, but all other features are supported. The two calculi and their proofs are formalized in the Coq theorem prover.

In summary, the contributions of this paper are:

- **Calculi supporting overloading, extensible records and nested composition.** We propose calculi with intersection types and a merge operator, which can support various features together, unlike previous calculi where only some features were supported.
- **Applicative subtyping and dispatching.** We develop a specialized applicative subtyping relation to deal with the problem of inferring result types for applications and record projections. In addition, the dynamic semantics supports a corresponding applicative dispatching relation.
- **First-class, curried overloading:** We show that the interaction between overloading and nested composition enables overloaded functions to be first-class, which allows the definition of curried overloaded functions.
- **Mechanical formalization and implementation:** All the calculi and proofs are formalized in the Coq theorem prover. The formalization is available in the artifact [1] and a prototype implementation can be found at:

`https://github.com/juniorxxue/applicative-intersection`

2 Overview

This section gives an overview of our work and introduces the key technical ideas. We then illustrate some problems, challenges and solutions when designing type systems for such calculi and features.

2.1 Background

Intersection Types and Merge Operator Intersection types describe expressions that can have multiple types. The intersection type A & B is inhabited by terms that have both type A and type B. The merge operator (denoted as ,,) has been introduced by Reynolds [21], and later refined by Dunfield [9], to create terms with intersection types at the term level. An important feature of Dunfield's calculus is that it contains a completely unrestricted merge operator, which enables most of the applications that we will discuss in this paper, except for nested composition. However, this expressive power comes at a cost. The semantics of the calculus is ambiguous. For example, `1,,2 : Int` can elaborate to both `1` and `2`. Note that intersection types in the presence of the merge operator have a different interpretation from the original meaning [7], where type intersections `A & B` are *only* inhabited by the intersection of the sets of values of `A` and `B`. In general, with the merge operator, we can always find a term for any intersection type, even when the two types in the intersection are disjoint (i.e. when the sets of values denoted by the two types are disjoint). For example, `1,,true` has the type `Int & Bool`. In many classical intersection type systems without the merge operator, such type would not be inhabited [19]. Thus, the use of the term "intersection" is merely a historical legacy. The merge operator adds expressive power to calculi with intersection types. As we shall see, this added expressive power is useful to model several features of practical interest for programming languages.

Disjoint intersection types Oliveira et al. [17] solved the ambiguity problem by imposing a disjointness restriction on merges. Only types that are disjoint can be merged. For example, `Int` and `Bool` are disjoint, so the type `Int & Bool` is well-formed and `1,,true` is a valid term. Huang et al. [14] improved this calculus by introducing a type-directed operational semantics where types are used to assist reduction and proved its type soundness and determinism. Unfortunately, the restriction to disjoint intersection types, while allowing many of the original applications, rules out traditional function overloading (see Section 5 for more details).

2.2 Applications of the Merge Operator

To show that the merge operator is useful, we now cover four applications of the merge operator that have appeared in the literature: records and record projections, function overloading, return type function overloading and nested composition. All applications can be encoded by our calculus in Section 4.

Records and Record Projections The idea of using the merge operator to model record concatenation firstly appears in Reynold's work [23]. Records in our calculi are modelled as merges of multiple single-field records. Multi-field records can be viewed as syntactic sugar and `{x="hello", y="world"}` is simply `{x = "hello"},,{y = "world"}`. The behaviour of record projection is mostly standard in our calculi. After being projected by a label, the merged records will return the associated terms. For instance ($\hookrightarrow$ denotes reduce to).

```
({x = "hello"},,{y = "world"}).x ↪ "hello"
```

Function overloading Function overloading is a form of polymorphism where the implementation of functions can vary depending on different types of arguments that are applied by functions. There are many ways to represent types of overloaded functions. For example, suppose `show` is an overloaded function that can be applied to either integers or booleans. Haskell utilises type classes [25] to assign the type `Show a ⇒ a → String` to `show` with instances defined.

With intersection types, we can employ the merge operator [6,21] to define a simplified version of the overloaded `show` function. For instance, the `show` function below has type `(Int → String) & (Bool → String)`.

```
show : (Int → String) & (Bool → String) = showInt,,showBool
```

The behaviour of `show` is standard, acting as a normal function: it can be applied to arguments and the correct implementation is selected based on types.

```
show 1 ↪ "1"                    show true ↪ "true"
```

Return type overloading One common example of return type overloading is the `read :: Read a ⇒ String → a` function in Haskell, which is the reverse operation of `show` and parses a string into some other form of data. Like `show`, we can define a simplified version of `read` using the merge operator:

```
read : (String → Int) & (String → Bool) = readInt,,readBool
```

In Haskell, because the return type `a` cannot be determined by the argument, `read` either requires programmers to give an explicit type annotation, or needs to automatically infer the return type from the context. Our calculi work in a similar manner. Suppose that `succ` is the successor function on integers and `not` is the negation function on booleans, then we can write:

```
succ (read "1") ↪ 2             not (read "true") ↪ false
```

Nested Composition Simply stated, nested composition reflects distributivity properties of intersection types at the term level. When eliminating terms created by the merge operator (usually functions and records), the results extracted from nested terms will be composed. In the context of records, the distributive subtyping rule enabling this behaviour is $\{\mathsf{l} : \mathsf{A}\} \& \{\mathsf{l} : \mathsf{B}\} <: \{\mathsf{l} : \mathsf{A} \& \mathsf{B}\}$. With this rule we can have the following expression:

```
({x = "hello"},,{x = 1}).x ↪ "hello",,1
```

Note that here we *allow* repeated fields with the same name. One may worry about ambiguities but, with a disjointness restriction, we can only accept fields with the same labels if the types of the fields are disjoint. Nested composition is a key feature in *compositional programming* [29], which uses it to solve challenging modularity problems such as the Expression Problem [26], and to model forms of *family polymorphism* [11]. We refer interested readers to the work of Zhang et al. [29] for details.

Nested composition can also occur with functional intersections, using the subtyping rule $(A \to B) \,\&\, (A \to C) <: A \to (B \,\&\, C)$. With this rule, we can, for example, write the following program:

```
(succ,,intToDigit) 5 ↪ 6,,'5'
```

which applies two functions to the integer `5`. Note that here `intToDigit` takes an integer and returns a corresponding character. We will also see that nested composition enables overloaded functions to be curried.

2.3 Challenges in the Design of the Semantics

The goal of this work is to design an automatic type inference algorithm for applications and record projection and the corresponding dynamic semantics, so that the system supports all applications presented in the previous section. Unfortunately, designing the semantics of the merge operator poses significant challenges, which we explain in the rest of this section.

Inference of Projections and Applications In traditional type systems, in applications $e_1\ e_2$ or projections $e.l$, e_1 is expected to have an arrow type and e is expected to have a record type. Such convention, however, cannot apply to our system because certain forms of intersection types can also play the role of arrow or record types. In particular, such use cases of intersection types are helpful for modelling overloaded functions and multi-field records. For example, we know that `showInt` is one branch of `show` with the subtyping statement:

```
(Int → String) & (Bool → String) <: Int → String
```

From this example we can see that the dynamic semantics must somehow be type-dependent. In our work we follow the *type-directed operational semantics* (TDOS) [14] approach, which chooses between merged functions according to type information during runtime. However, existing TDOS approaches do not support overloading for two reasons. Firstly, TDOS requires merged functions to be disjoint with each other, but in this case the merged functions are not disjoint (i.e., `Int → String` is not disjoint with `Bool → String` because of the common return type `String`). Secondly, even if we would simply ignore the disjointness restriction, we would still need to put an explicit type annotation `Int → String` and write the program as `(show : Int → String) 1` to select the correct implementation to apply from the overloaded `show` function. This is because previous TDOS calculi have restricted application rules that cannot accommodate traditional overloading. Clearly, in a setting with overloading, having to write such explicit annotations would be unsatisfying. Therefore we wish to have an approach where we can write overloaded functions naturally.

A similar problem occurs using record projections in existing TDOS calculi. For instance, the type system of λ_{i+} [14] requires explicit annotations for projections of multi-field records with distinct labels, such as `({x = 1},,{y = true} : {x : Int}).x`. This is of course, quite unnatural to write. Although source languages targetting the TDOS calculi can eliminate the explicit use of such annotations at the source level, it would be better to address this problem directly in the TDOS.

Dynamic Semantics Giving a direct semantics to overloaded applications is a non-trivial problem. Thanks to the merge operator and the call-by-value strategy, our overloaded functions are expected to be in the form of nested merges according to the structure of types. So we can reason about the dynamic semantics as we deal with the types. Unfortunately, the distributivity of subtyping complicates the story. The challenge comes from the fact that in our setting overloaded functions are first-class. That is, they can be taken as arguments or returned as results. For instance, we can have:

```
pshow : Unit → (Int → String) & (Bool → String)
pshow = λx. show
```

In this situation, an overloaded function is wrapped with a lambda abstraction, while it should also be viewed as an overloaded function. For example, we expect the following to hold:

```
pshow unit 1 ↪ "1"          pshow unit true ↪ "true"
```

In the last two cases, with a traditional approach to applications, `pshow` is expected to have type `Unit → Int → String` and `Unit → Bool → String` respectively. From the perspective of intersection overloading, `pshow` should be of type `(Unit → Int → String) & (Unit → Bool → String)`, which, however, is different from the given type annotation. This alternative view of types and functions poses challenges to the design of the static as well as the dynamic semantics.

Ambiguities on the input types In languages like C++ and Java, overloading cannot be defined on return types, and ambiguities are detected when the input types of overloaded functions overlap. This is also a reason why many works model the inputs of overloaded functions as product types [15,6,9]. The advantages are obvious: it is easier to resolve the correct branch by only comparing the product types and types of arguments. The drawback of this model is that product types will prevent overloaded functions to be curried. This is because overloaded functions based on product types expect a tuple containing all the arguments and reject partial applications. The challenge of modelling overloaded curried functions is that partial applications may be insufficient to fully determine the implementation to take from the overloaded function. These pains can be alleviated using intersection types, the merge operator and the feature of nested composition.

```
f : Int → Int → Int          g : Int → Bool → Bool
```

For example, with `f,,g`, we can simply reason that the result of `(f,,g) 1 true` is `g 1 true`. The problem occurs in the partially applied term `(f,,g) 1`, for which there are two possible design choices. The first choice is to reject this application term since we cannot select between overloads, thus forbidding many use cases like this. Another choice is to apply `f` and `g` in parallel to `1`, resulting in `(f 1),,(g 1)`, which has the type `(Int → Int) & (Bool → Bool)`.

2.4 Key Ideas and Results

Applicative subtyping To help with the inference of the result types for applications and projections, we propose a new specialized subtyping algorithm for applications and projections. Specifically, conventional subtyping algorithms take two types as inputs, and return a boolean indicating whether two types are in a subtyping relation or not. We present an *applicative subtyping* algorithm, whose intuition is simple: given a functional type A (which may be an intersection of functions), and the type of an argument B, it tells whether this function can be applied to this argument and, if yes, it computes the output type. Similarly, given a record type A, and a label l, applicative subtyping tells whether this record can be projected by this label, and if yes, it computes the result type associated with this label. Basically, we try to solve the problem in the following subtyping form (denoted as <:), where we infer the type `?` given the argument type `Int`:

```
(Int → String) & (Bool → String) <: Int → ?
```

This problem can be split into two steps: first, check whether the application is well-typed, and if so, determine its output type. For the above example, `?` is expected to be `String`, as `Int` is an argument to `Int → String`. Record projection works similarly. `String & Int` should be derived as the result type for projection `({x = "hello"},,{x = 1}).x`.

```
{x : String} & {x : Int} <: {x : ?}
```

Applicative subtyping is used when typing applications and projections. Our algorithm adopts the notion of selectors S that abstract the arguments (as a type for applications, or a label for projections). The behaviour of applicative subtyping for intersection types is captured by a simple composition operator $\circledcirc$ which isolates particular design choices. In applicative subtyping, a possible result is that the application fails. We denote failure with a `.` symbol. We illustrate the results of applicative subtyping (denoted as $\ll$) for the above examples next.

```
(Int → String) & (Bool → String) ≪ Int    = String ◎ .
(String → Int) & (String → Bool) ≪ String = Int    ◎ Bool
{x : String)   & {y : String)    ≪ x      = String ◎ .
{x : String)   & {x :Int)        ≪ x      = String ◎ Int
```

There are two important things to notice here. Firstly, as discussed above, the second argument of $\ll$ is just the function argument type or a label, instead of the complete type that would be normally used in a subtyping comparison. Secondly, the results are given in terms of the composition operator $\circledcirc$. The composition operator abstracts behaviour that is specific to particular subtyping relations. When designing applicative subtyping, a desirable property is that it should be sound and complete with respect to subtyping. The soundness and completeness properties can be stated as follows (here we show the case for functions):

Lemma 1 (Soundness). *If* $\mathsf{A} \ll \mathsf{B} = \mathsf{C}$*, then* $\mathsf{A} <: \mathsf{B} \to \mathsf{C}$.

Lemma 2 (Completeness). *If* $\mathsf{A} <: \mathsf{B} \to \mathsf{C}$*, then* $\exists \mathsf{D}, \mathsf{A} \ll \mathsf{B} = \mathsf{D} \wedge \mathsf{D} <: \mathsf{C}$.

$$
\begin{aligned}
\cdot \circledcirc \cdot &= \cdot \\
A_1 \circledcirc \cdot &= A_1 \\
\cdot \circledcirc A_2 &= A_2 \\
A_1 \circledcirc A_2 &= A_1 \,\&\, A_2
\end{aligned}
$$

(a) Nested composition semantics.

$$
\begin{aligned}
\cdot \circledcirc \cdot &= \cdot \\
A_1 \circledcirc \cdot &= A_1 \\
\cdot \circledcirc A_2 &= A_2 \\
\mathsf{Amb} \circledcirc \mathsf{O} &= \mathsf{Amb} \\
\mathsf{O} \circledcirc \mathsf{Amb} &= \mathsf{Amb} \\
A_1 \circledcirc A_2 &= \mathsf{Amb}
\end{aligned}
$$

(b) Overloading semantics.

Fig. 1: Two possible composition operators, for calculi with distributive subtyping (on the left), and without distributive subtyping (on the right). The notation Amb denotes ambiguity, which represents a type-error, while the meta-variable O denotes any output type (i.e. either a type, failure or ambiguity).

Depending on the expressive power of the subtyping relation we need different implementations of $\circledcirc$ to satisfy soundness/completeness. In particular, whether or not the subtyping relation includes distributive subtyping rules affects the definition of $\circledcirc$. Figure 1 illustrates this difference. In a relation with distributive subtyping rules (such as $(A \to B) \,\&\, (A \to C) <: A \to (B \,\&\, C)$), the composition operator on the left (a) leads to a sound and complete definition of applicative subtyping. This operator, which is quite simple, allows combining results that arise from multiple branches. We say that this composition operator implements a *nested composition semantics*, since it allows combining multiple results. For instance, `Int` $\circledcirc$ `Bool` simply denotes `Int & Bool`, meaning that an applicative subtyping statement `(String` $\to$ `Int) & (String` $\to$ `Bool)` $\ll$ `String` succeeds, computing the output type `Int & Bool`. Without the distributive subtyping rule the composition operator on the left (a) is not sound with respect to subtyping. Instead we should use the implementation on the right (b), which will reject cases like `Int` $\circledcirc$ `Bool`, since such cases denote a form of ambiguity. We say that the composition operator on the right implements an *overloading semantics*, since if multiple implementations in an overloaded definition match with an argument (or a label), we reject the application. This is similar to traditional overloading mechanisms, which reject such cases as a form of ambiguity. In other words, in the overloading semantics, only one implementation can be selected from an overloaded definition. Note that the overloading semantics implementation can also be used in a calculus with distributivity, but this would lose the completeness property. One counter example is `(Int` $\to$ `String) & (Int` $\to$ `Int) <: Int` $\to$ `(String & Int)`, which holds according to the distributivity rule, but the applicative subtyping based on the overloading semantics will derive an ambiguity `Amb` error.

TDOS for Overloading For the semantics, we follow up the idea of typed-directed operational semantics [14] and define a new judgment that performs *applicative dispatching* to support overloading. At a high level, applicative dispatch reflects applicative subtyping in the dynamic semantics. As we analyzed

above, distributivity forbids overloaded functions to be exact nested merges, thus a canonical form of overloaded function should be settled. To solve this problem we use an explicit merge with extra annotations that play a role of "runtime types", which are used by applicative dispatching to select the correct branch during runtime.

Two Calculi We present two calculi to demonstrate the applicative subtyping and applicative dispatching. Both calculi utilize the composition operator (Figure 1a) since we want to allow nested composition, and explore curried and first-class overloaded functions. The first calculus embraces a simple design and adopts an unrestricted merge operator. All features mentioned above can be encoded in this calculus, but the calculus will have a non-deterministic semantics due to ambiguities. For ambiguities, we present a second calculus, which adopts a restricted merge operator: only terms with disjoint types can be merged. This calculus is deterministic but excludes certain forms of overloading, like the `show` function. Since `Int` $\rightarrow$ `String` is not disjoint with `Bool` $\rightarrow$ `String`, such merges will be rejected.

3 Applicative Subtyping

In this section, we first present the normal subtyping algorithm for intersection types and then present applicative subtyping.

3.1 Types and Subtyping

The types that we consider in this work are:

$$\begin{array}{ll} \text{Types} & A, B ::= \mathsf{Int} \mid \mathsf{Top} \mid A \to B \mid A \,\&\, B \mid \{l : A\} \\ \text{Ordinary Types} & A^o, B^o ::= \mathsf{Int} \mid \mathsf{Top} \mid A \to B^o \mid \{l : A^o\} \end{array}$$

A and B are metavariables which range over types. Int and Top are base types and Top is the supertype of all types. Compound types are function types $A \to B$, intersection types $A \,\&\, B$, and record types $\{l : A\}$. Ordinary types [8,14] are essentially types without intersection types, except for functions where intersection types can appear in argument types.

Algorithmic BCD subtyping Subtyping relations for intersection types can vary in whether distributivity rules are included or not. For calculi with intersection types, a common rule allows the intersection of arrow types to distribute over arrows. One well-known subtyping relation with such distributivity rule is BCD subtyping [3]. Huang et al. [14] provide a sound and complete algorithm for BCD subtyping by eliminating the transitivity rule, and employing the notions of ordinary types and splittable types. We present that subtyping relation in Fig. 2. Splittable types describe that types can be split into two simpler types and ordinary types are those which cannot be split. Rule Sub-And is the most interesting rule as it captures the distributivity of intersection types over function types and record types. This rule splits the type B into two types B_1 and B_2 and proceeds by testing whether A is a subtype of both B_1 and B_2.

$\boxed{A_1 \lhd A \rhd A_2}$ *(Splittable Types)*

$$\textsc{Sp-And}\ \frac{}{A \lhd A \,\&\, B \rhd B} \qquad \textsc{Sp-Arr}\ \frac{B_1 \lhd B \rhd B_2}{A \to B_1 \lhd A \to B \rhd A \to B_2} \qquad \textsc{Sp-Rcd}\ \frac{A_1 \lhd A \rhd A_2}{\{l : A_1\} \lhd \{l : A\} \rhd \{l : A_2\}}$$

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

$$\textsc{Sub-Int}\ \frac{}{\mathsf{Int} <: \mathsf{Int}} \qquad \textsc{Sub-Top}\ \frac{}{A <: \mathsf{Top}} \qquad \textsc{Sub-Arr}\ \frac{C <: A \qquad B <: D^o}{A \to B <: C \to D^o} \qquad \textsc{Sub-Rcd}\ \frac{A <: B^o}{\{l : A\} <: \{l : B^o\}}$$

$$\textsc{Sub-And}\ \frac{B_1 \lhd B \rhd B_2 \qquad A <: B_1 \qquad A <: B_2}{A <: B} \qquad \textsc{Sub-And-L}\ \frac{A <: C^o}{A \,\&\, B <: C^o} \qquad \textsc{Sub-And-R}\ \frac{B <: C^o}{A \,\&\, B <: C^o}$$

Fig. 2: Splittable Types and Algorithmic Subtyping

$$A_1 \to A_2 \ll B = A_2 \quad \textit{when } B <: A_1 \tag{1}$$
$$A_1 \to A_2 \ll B = . \quad \textit{when } \neg(B <: A_1) \tag{2}$$
$$\{l = A\} \ll l = A \tag{3}$$
$$\{l_1 = A\} \ll l_2 = . \quad \textit{when } l_1 \neq l_2 \tag{4}$$
$$A_1 \,\&\, A_2 \ll S = (A_1 \ll S) \circledcirc (A_2 \ll S) \tag{5}$$
$$A \ll S = . \quad \textit{otherwise} \tag{6}$$

Fig. 3: Applicative Subtyping

3.2 Applicative Subtyping

Applicative subtyping utilises the notion of *selectors* to find the correct output type from applicable types. We consider applicable types to be function or record types. This relation enables the type system to infer the type of applications and record projections, as shown in Section 4. We model the types of arguments and labels of record projections as selectors, and the outputs as being either a type or nothing (denoting the failure to find a suitable output type).

$$\text{Selectors} \quad S ::= A \mid l \qquad \text{Outputs} \quad O ::= . \mid A$$

The definition of applicative subtyping is given in Fig. 3. Selectors are used as the second parameter and propagate through the subtyping checks, until we reach arrow or record types. For arrow types, in rules (1) (2), we check the contravariant subtyping between input type A_1 and argument B. If successful, the output type A_2 is returned, otherwise we fail. For record types (3) (4), we check the equality between labels. If the labels are equal, we return the output type A, otherwise we fail. For the case of the intersection types $A_1 \,\&\, A_2$ (5), we introduce a composition operator $\circledcirc$ to combine two results which are derived from applying A_1 and A_2 with the same selector B. Rule (6) covers a number of

missing cases (such as $\mathsf{Int} \ll \mathsf{S}$) which will all fail. For simplicity of presentation we write those rules as a single rule (6).

The composition operator accepts output results and returns a new output result. For systems with BCD subtyping, which include distributivity rules, we use the composition operator implementing the nested composition semantics in Fig. 1a.

3.3 Metatheory

We proved the soundness and completeness of our applicative subtyping with respect to the normal subtyping. The decidability of applicative subtyping is straightforward since it is modelled as a structurally recursive function. We have two versions of soundness and completeness lemmas. The first version applies to the case where the supertype is a function:

Lemma 3 (Soundness (Function)). *If* $\mathsf{A} \ll \mathsf{B} = \mathsf{C}$, *then* $\mathsf{A} <: \mathsf{B} \to \mathsf{C}$.

Lemma 4 (Completeness (Function)). *If* $\mathsf{A} <: \mathsf{B} \to \mathsf{C}$, *then* $\exists \mathsf{D}, \mathsf{A} \ll \mathsf{B} = \mathsf{D} \wedge \mathsf{D} <: \mathsf{C}$.

The soundness lemma is intuitive. If the result of checking applicative subtyping with a subtype A and and input type B computes a type C then it should be the case that $\mathsf{A} <: \mathsf{B} \to \mathsf{C}$. For completeness we wish to show that if A is a subtype of a function type $\mathsf{B} \to \mathsf{C}$ then applicative subtyping will always be able to find some output type D which is a subtype of C.

The second version of the lemma, which applies to the case where the supertype is a record, is defined in a similar manner.

Lemma 5 (Soundness (Record)). *If* $\mathsf{A} \ll \mathsf{l} = \mathsf{B}$, *then* $\mathsf{A} <: \{\mathsf{l} : \mathsf{B}\}$.

Lemma 6 (Completeness (Record)). *If* $\mathsf{A} <: \{\mathsf{l} : \mathsf{B}\}$, *then* $\exists \mathsf{C}, \mathsf{A} \ll \mathsf{l} = \mathsf{C} \wedge \mathsf{C} <: \mathsf{B}$.

Remark Note that, if we would drop the distributivity of intersections over other constructs by removing the rules Sp-Arr and Sp-Rcd, then to have soundness and completeness we need to employ the composition operator implementing the overloading semantics to the right of Fig. 1. When using that composition operator, the soundness lemmas remain the same, but we need to adjust the completeness lemmas to consider the ambiguous cases. For instance, the completeness for the case of a function supertype would become:

Lemma 7 (Completeness). *If* $\mathsf{A} <: \mathsf{B} \to \mathsf{C}$, *then* $(\exists \mathsf{D}, \mathsf{A} \ll \mathsf{B} = \mathsf{D} \wedge \mathsf{D} <: \mathsf{C}) \vee \mathsf{A} \ll \mathsf{B} = \mathsf{Amb}$.

4 A Calculus with an Unrestricted Merge Operator

This section presents a type sound calculus that supports both intersection types and a merge operator. This calculus can be viewed as a variant of Dunfield's calculus (without union types) [9]. Our calculus employs a type-directed operational semantics [14] instead of using elaboration semantics as proposed by Dunfield and adopts applicative subtyping and distributive subtyping.

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

$$\text{T-Lit}\ \frac{}{\Gamma \vdash i \Rightarrow \mathsf{Int}} \qquad \text{T-Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A} \qquad \text{T-Lam}\ \frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.\, e : A \to B \Rightarrow A \to B} \qquad \text{T-Rcd}\ \frac{\Gamma \vdash e \Rightarrow A}{\Gamma \vdash \{l = e\} \Rightarrow \{l : A\}}$$

$$\text{T-App}\ \frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B \qquad A \ll B = C}{\Gamma \vdash e_1\, e_2 \Rightarrow C} \qquad \text{T-Proj}\ \frac{\Gamma \vdash e \Rightarrow A \qquad A \ll l = B}{\Gamma \vdash e.l \Rightarrow B}$$

$$\text{T-Mrg}\ \frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B}{\Gamma \vdash e_1 ,, e_2 \Rightarrow A \,\&\, B} \qquad \text{T-Ann}\ \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash e : A \Rightarrow A} \qquad \text{T-Sub}\ \frac{\Gamma \vdash e \Rightarrow A \qquad A <: B}{\Gamma \vdash e \Leftarrow B}$$

Fig. 4: Bi-directional typing. The bidirectional mode syntax is $\Leftrightarrow \ ::= \ \Leftarrow | \Rightarrow$.

4.1 Syntax

The syntax of this calculus is:

Expressions	$e ::= x \mid i \mid e : A \mid e_1\, e_2 \mid \lambda x.e : A \to B \mid e_1 ,, e_2 \mid \{l = e\} \mid e.l$
Raw Values	$p ::= i \mid \lambda x.e : A \to B$
Values	$v ::= p : A^o \mid v_1 ,, v_2 \mid \{l = v\}$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$

Most expressions are standard. Lambda expressions $\lambda x.\, e : A \to B$ are fully annotated because the operational semantics is type-directed. The expression $e_1 ,, e_2$ creates a merge of two expressions e_1 and e_2. The expression $\{l = e\}$ denotes a single-field record with label l and field e. The projection of records is represented by $e.l$. Raw values include integers and lambdas, and values are defined on raw values annotated with ordinary types, merges of values and records whose fields are values. We stratify raw values and values because we need to utilise annotations to adopt dispatching in the semantics. The ordinary restriction on values enforces a canonical form for overloaded functions. Overloaded functions will be reduced to explicit nested merges, even in settings with distribuivity.

4.2 Typing

Fig. 4 shows our bi-directional type system. Most of the rules are adapted from traditional bi-directional typing [10]. The novel rules are rules T-App and T-Proj, whose inferred type is derived from applicative subtyping.

Typing of Application and Projection Our approach to type applications [28] is to infer the type of functions and arguments at the same time, pass their types into applicative subtyping (A and B in rule T-App), and assign the computed result C to applications. This is because we allow intersection types to distribute over arrow types, thus the type of the function can be an arrow type or an intersection type. We cannot simply extract the input type of a function. Since

multi-field records are also intersection types in our system, the typing for projections (rule T-Proj) uses a similar idea to applications.

Examples We show an example of how the rule T-App works. Suppose that we have $\Gamma = \mathsf{f} : \mathrm{I} \to \mathrm{I} \to \mathrm{I}, \mathsf{g} : \mathrm{I} \to \mathrm{B} \to \mathrm{B}$. (I and B stand for $\mathtt{Int}$ and $\mathtt{Bool}$)

$$\dfrac{\dfrac{\Gamma \vdash (\mathsf{f},,\mathsf{g}) \Rightarrow (\mathrm{I} \to \mathrm{I} \to \mathrm{I}) \,\&\, (\mathrm{I} \to \mathrm{B} \to \mathrm{B}) \qquad \Gamma \vdash 2 \Rightarrow \mathrm{I}}{\Gamma \vdash (\mathsf{f},,\mathsf{g})\ 2 \Rightarrow (\mathrm{I} \to \mathrm{I}) \,\&\, (\mathrm{B} \to \mathrm{B})}\ \text{T-App} \qquad \Gamma \vdash \mathsf{true} \Rightarrow \mathrm{B}}{\Gamma \vdash (\mathsf{f},,\mathsf{g})\ 2\ \mathsf{true} \Rightarrow \mathrm{B}}\ \text{T-App}$$

Note that for space reasons we omit the applicative subtyping derivations here, which are straightforward. To infer the type of $(\mathsf{f},,\mathsf{g})\ 2\ \mathsf{true}$, we first infer both the type of $(\mathsf{f},,\mathsf{g})\ 2$ and true. The type of $(\mathsf{f},,\mathsf{g})\ 2$ is $(\mathtt{Int} \to \mathtt{Int}) \,\&\, (\mathtt{Bool} \to \mathtt{Bool})$. This result is computed from applicative subtyping with two inputs: type of function merges $\mathsf{f},,\mathsf{g}$ and type of 2. Later we use the computed result of $(\mathsf{f},,\mathsf{g})\ 2$ to derive our final type $\mathtt{Bool}$.

4.3 Semantics

This calculus adopts a type-directed operational semantics [14], where type annotations are used to cast terms instead of being erased after type checking.

Casting We introduce the casting judgment in Fig. 5. Judgment $v \longmapsto_A v'$ describes that value v is cast to value v' by type A, thus forcing the value to match the type structure of A. The casting rules are essentially the same as the rules proposed by Huang et al. [14]. Rules Ct-Mrg-L and Ct-Mrg-R state that merges will be cast to one result by ordinary types. For example, `show` will be cast to `showInt` by type `Int → String`.

Applicative Dispatching We introduce a new judgement called applicative dispatching (Fig. 5), which extends Huang et al's [14] *parallel application* judgement. In contrast to parallel application, we must also deal with overloading. Judgment $(v \bullet vl) \hookrightarrow e$ describes that value v is applied to value or label vl (i.e. $vl ::= v \mid l$) and then reduced to a term e. Rule App-Lam performs beta-reduction and appends an extra annotation D to enforce the output type of the application. Rule App-Proj simply extracts the value from the single record field. The interesting part is the remaining three rules for merges. The function $\langle vl \rangle$ simply extracts out the type of a value, to provide the types to be compared with applicative subtyping. To deal with overloading we need to introduce rules App-Mrg-L and App-Mrg-R, which allows a merge to be applied when only one of the values is applicable. The last rule, rule App-Mrg-P deals with the parallel application, where both values in the merge can be applied.

Operational Semantics We present our small-step reduction rules in Fig. 6. Rules Step-Int-Ann and Step-Arr-Ann append extra annotations to the partial value, in order to preserve the precise types at runtime. Rule Step-Pv-Split will split terms according to splittable types, forcing the type of each branch in merges to be ordinary. Rules Step-App and Step-Prj directly call applicative dispatching. Rule Step-Val-Ann triggers casting: v is cast to v' by type A.

$\boxed{v \longmapsto_A v'}$ *(Casting)*

$$\textsc{Ct-Int}\ \frac{}{i : A \longmapsto_{\mathsf{Int}} i : \mathsf{Int}} \qquad \textsc{Ct-Top}\ \frac{}{v \longmapsto_{\mathsf{Top}} \top : \mathsf{Top}} \qquad \textsc{Ct-Rcd}\ \frac{v \longmapsto_{A^o} v'}{\{l = v\} \longmapsto_{\{l:A^o\}} \{l = v'\}}$$

$$\textsc{Ct-Arr}\ \frac{E <: C \to D^o}{(\lambda x.\, e : A \to B) : E \longmapsto_{(C \to D^o)} (\lambda x.\, e : A \to D^o) : (C \to D^o)} \qquad \textsc{Ct-Mrg-L}\ \frac{v_1 \longmapsto_{A^o} v_1'}{v_1 ,, v_2 \longmapsto_{A^o} v_1'}$$

$$\textsc{Ct-Mrg-R}\ \frac{v_2 \longmapsto_{A^o} v_2'}{v_1 ,, v_2 \longmapsto_{A^o} v_2'} \qquad \textsc{Ct-And}\ \frac{A_1 \lhd A \rhd A_2 \quad v \longmapsto_{A_1} v_1 \quad v \longmapsto_{A_2} v_2}{v \longmapsto_A v_1 ,, v_2}$$

$\boxed{(v \bullet vl) \hookrightarrow e}$ *(Applicative Dispatching)*

$$\textsc{App-Lam}\ \frac{v \longmapsto_A v'}{((\lambda x.\, e : A \to B) : C \to D \bullet v) \hookrightarrow e[x \mapsto v'] : D} \qquad \textsc{App-Proj}\ \frac{}{(\{l = v\} \bullet l) \hookrightarrow v}$$

$$\textsc{App-Mrg-L}\ \frac{\langle v_2 \rangle \ll \langle vl \rangle = . \quad (v_1 \bullet vl) \hookrightarrow e}{((v_1 ,, v_2) \bullet vl) \hookrightarrow e} \qquad \textsc{App-Mrg-R}\ \frac{\langle v_1 \rangle \ll \langle vl \rangle = . \quad (v_2 \bullet vl) \hookrightarrow e}{((v_1 ,, v_2) \bullet vl) \hookrightarrow e}$$

$$\textsc{App-Mrg-P}\ \frac{\langle v_1 \rangle \ll \langle vl \rangle \neq . \quad \langle v_2 \rangle \ll \langle vl \rangle \neq . \quad (v_1 \bullet vl) \hookrightarrow e_1 \quad (v_2 \bullet vl) \hookrightarrow e_2}{((v_1 ,, v_2) \bullet vl) \hookrightarrow e_1 ,, e_2}$$

Fig. 5: Casting and Applicative Dispatching

Rule Step-Ann is a congruence rule with a restriction that e cannot be a raw value p. The remaining rules are normal congruence rules.

4.4 Type Soundness

For type soundness, we employ a proof technique similar to the one by Fan et al. [12]. First we need a number of results about the auxiliary relations used in reduction. We show some of the more interesting lemmas next:

Lemma 8 (Preservation (Applications and Projections)).
- *If* $\cdot \vdash v_1\ v_2 \Rightarrow A$ *and* $v_1 \bullet v_2 \hookrightarrow e$, *then* $\cdot \vdash e \Leftarrow A$.
- *If* $\cdot \vdash v.l \Rightarrow A$ *and* $v \bullet l \hookrightarrow e$, *then* $\cdot \vdash e \Leftarrow A$.

Lemma 9 (Progress (Applications and Projections)).
- *If* $\cdot \vdash v_1\ v_2 \Rightarrow A$, *then* $\exists e, v_1 \bullet v_2 \hookrightarrow e$
- *If* $\cdot \vdash v.l \Rightarrow A$, *then* $\exists e, v \bullet l \hookrightarrow e$.

Type soundness is proven via standard preservation and progress theorems.

Theorem 1 (Preservation). *If* $\cdot \vdash e \Leftrightarrow A$ *and* $e \longmapsto e'$, *then* $\cdot \vdash e' \Leftarrow A$.

Theorem 2 (Progress). *If* $\cdot \vdash e \Leftrightarrow A$, *then* e *is a value or* $\exists e', e \longmapsto e'$.

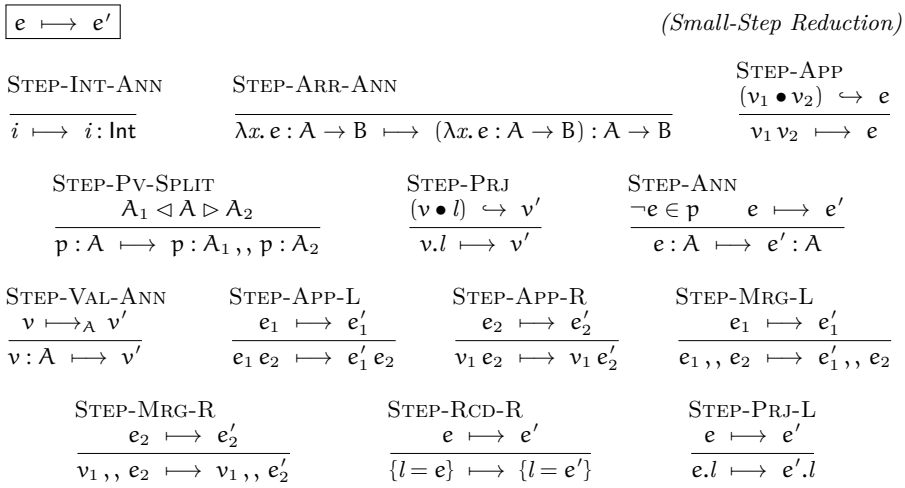

Fig. 6: Operational Semantics

5 A Calculus with a Disjoint Merge Operator

This section presents a second calculus with a disjointness restriction on merges [17] to recover determinism. This calculus forbids some cases of conventional overloading, but still supports the other features. We focus on the key differences to the previous calculus, since most rules and relations are the same. Compared to previous calculi with disjoint intersection types, the main novelty is the use of the applicative subtyping and dispatching relations, which enables support for record projections and a restricted form of overloading naturally (without redundant type annotations).

5.1 Disjointness

We employ the definition of disjointness proposed by Oliveira et al. [17]. Informally, if all common supertypes of two types are *top-like* types, we can conclude that the two types are disjoint. Top-like types are those who are supertypes of all types (e.g., Top, $\mathsf{Top} \,\&\, \mathsf{Top}$) and defined as:

$$\text{Top-like types} \qquad \rceil A \lceil \ ::= \mathsf{Top} \ \mid \ \rceil A \lceil \,\&\, \rceil B \lceil \mid A \to \rceil B \lceil \mid \{l : \rceil A \lceil\}$$

Note that the including of such types into top-like types is also part of the classical BCD subtyping relation [3]. A formal specification of disjointness is given below. There is a sound and complete set of algorithmic disjointness rules that conform to this specification. The interested reader can check existing work for the algorithmic rules [17,14]. For space reasons we omit them here.

Definition 1. (Disjointness) $A * B \triangleq \forall C$ *if* $A <: C \wedge B <: C$*, then* C *is top-like.*

In our calculus we allow merges of disjoint functions. Thus, types such as $\mathtt{Int} \to \mathtt{Int}$ or $\mathtt{Int} \to \mathtt{Bool}$ are disjoint. To include function types into our disjointness, types like $\mathtt{Int} \to \mathtt{Top}$ should be top-like and supertypes of all types, since otherwise $\mathtt{Int} \to \mathtt{Int}$ and $\mathtt{Int} \to \mathtt{Bool}$ cannot be disjoint according to our definition. However, this disjointness definition prevents some forms of overloading. For example, the type of `show` is `(Int → String) & (Bool → String)`, which will be rejected by the disjointness condition since `Int → String` is not disjoint with `Bool → String`. For those two types we can find a common supertype `Int & Bool → String`, which is not top-like. To see why we should prevent such merges assume that `show` is allowed, then `show (1,,true) : String` is ambiguous since the result can be either `"1"` or `"true"`. Note that some forms of overloading are still possible. For instance `succ,,not` will be accepted since `Int → Int` is disjoint with `Bool → Bool`.

We follow previous work on disjoint intersection types [4] and generalize our subtyping rule for rule S-Top to be $A <: \rceil B \lceil$ where $\rceil B \lceil$ means that B is a top-like type. Disjointness has important properties, which are helpful for the metatheory of the calculus. In particular, if two types are disjoint, their applicative subtyping results under the same partial types are also disjoint.

Lemma 10 (Applicative Subtyping and Disjointness). *If* $A * B$, $A \ll S = C_1$ *and* $B \ll S = C_2$, *then* $C_1 * C_2$.

Soundness and completeness of applicative subtyping. With the more general subtyping rule for top-like types, applicative subtyping remains sound (with lemmas 3, 5 in Section 3) with respect to subtyping. However, the completeness of our applicative subtyping needs to be slightly adapted.

Lemma 11 (Completeness of Applicative Subtyping). *If* $A <: B \to C$, *then* $(\exists D, A \ll B = D \wedge D <: C) \vee \mathsf{Top} <: C$.

In other words, applicative subtyping is complete except for the case where the output type is top-like. In such case applicative subtyping fails. Note though that this failure prevents strange programs from being type-checked. For example, subtyping has instances `Top <: A → Top`, allowing `(1 : Top) 2` to be well-typed, which would require special treatment in the typing rules. We reject such cases, making the typing rules simpler, and avoiding type-checking such programs.

5.2 Typing and Semantics

The main change in typing is that we add a disjoint premise in rule T-Mrg.

$$\frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B \qquad A * B}{\Gamma \vdash e_1 ,, e_2 \Rightarrow A \,\&\, B} \text{ T-Mrg}$$

Most changes in the dynamic semantics are related to top-like types. Basically we need some extra conditions in the rules testing whether or not types are top-like? However, apart from these minor changes, the rules remain essentially the same. For space reasons we omit the detailed rules here.

5.3 Type Soundness and Determinism

All properties, including subject reduction and type soundness shown in the first calculus, also hold in this calculus. We only focus on determinism here, which is the most interesting property of the calculus with disjointness.

Theorem 3 (Determinism). *If* $\cdot \vdash e \Leftrightarrow A$, $e \longmapsto e_1$ *and* $e \longmapsto e_2$, *then* $e_1 = e_2$.

6 Related Work

Intersection Types, Merges and Overloading Forsythe, introduced by Reynolds [21] has a restricted merge operator and its coherent semantics is formally proven. However, it does not account for overloaded functions since multiple functions are forbidden by merges. Pierce [18] introduced a `glue` construct in his calculus $F_\wedge$ as a language extension to support user-defined overloading and the types of overloaded functions are also modelled as intersection types. However his glue operator is unrestricted, leading to a non-deterministic semantics. Castagna et al. [6] gave a formalization to calculus for overloaded functions with subtyping. In his calculus, overloaded functions are defined as &-terms and their types are a finite list of arrow types with a consistency restriction. In overloaded applications, the "best-match" branch will be selected. The semantics is type-dependent, and overloaded applications rely on the runtime types, which is similar to our TDOS approach. Differently to our approach, nested composition is not supported in his calculus. Moreover, only one branch can be selected in the overloaded application, thus terms like `succ,,intToDigit` are rejected, forbidding currying on overloaded functions. In their work, records are encoded by lambda functions and multi-field records are overloaded functions.

Dunfield's calculus [9] is powerful enough to encode overloaded functions and record projection. Unlike our calculi, it does not support distributivity and nested composition. This means that overloaded functions do not interact nicely with currying. For example, to program `pshow unit 1` in her calculus, we should write `((pshow unit) : Int → Bool) 1`. As acknowledged by Dunfield, the semantics is not deterministic. This is similar to our first calculus in Section 4. To restrict the power of the merge operator and enable determinism, a disjointness restriction on merges has been proposed [17]. Closest to our work is the λ_i^+ calculus [14], which is a deterministic calculus with intersection types and a disjoint merge operator. There are two major differences between our work and λ_i^+. (1) Our first calculus utilizes an unrestricted merge operator, which allows *any* functions and records to be merged. (2) Our second calculus can be viewed as a variant of λ_i^+ that employs applicative subtyping and thus avoids many unnecessary annotations that are required in λ_i^+ since function overloading and record projection are not directly supported in λ_i^+. In λ_i^+, we would need a term with an explicit type annotation instead: `((succ,,not) : Int → Int) 1`. The rigid form of applications and projections in λ_i^+ prevents expressions such as `(succ,,not) 1`, which are not well-typed in λ_i^+.

In recent work, Rioux et al. [24] proposed a calculus with a disjoint merge operator that deals with union types and overloading. This is achieved with two more fine-grained disjointness relations called *mergeability* and *distinguishability*. Similarly to our calculus, they consider an expressive type-level dispatch relation that plays the same role as applicative subtyping in our calculus. Such dispatching relation supports union types, unlike our calculus. In terms of the operational semantics, there are significant differences between our work and Rioux et al.'s work. While their semantics still employs types at runtime, there is no casting relation. Instead there are patterns and co-patterns, which enforce runtime coercions via η-expansion. While overloading is supported, the disjointness relations are still not flexible enough to support return type overloading.

Semantic Subtyping Semantic subtyping [13] takes a different direction to type overloaded functions with intersection types and union types. In semantic subtyping the semantics of types is set-theoretic and subtyping relations are derived from the semantics. The type system features intersection types, union types and negation types. Overloaded functions are defined by a *typecase* primitive which is similar to the elimination of union types. For example, the type of `show` is `Int | Bool` → `String` (`|` denotes union types). The approach to semantic subtyping of overloaded functions is different from ours, since in our calculi (1) only intersection types are used to represent types of overloaded functions; and (2) overloaded functions can be introduced by simply merging functions.

7 Conclusion and Future Work

In this paper, we proposed applicative subtyping, a novel subtyping algorithm to infer the return types of application and projection. We also designed its corresponding judgment applicative dispatching in the dynamic semantics. Together these features enable expressive calculi with a merge operator. We present a type sound calculus that supports all features, but is non-deterministic, and a second deterministic calculus with a disjointness restriction supporting all features except for overloading. Future work includes finding a design that enables overloading, while preserving determinism. Furthermore we are interested in extending the calculus with disjoint polymorphism [2].

Acknowledgement

We thank the anonymous reviewers for their helpful comments and our colleagues of HKU Programming Languages Group for their discussions to understand this work better. We thank Chen Cui for his valuable feedback on the draft. This work has been sponsored by Hong Kong Research Grant Council projects number 17209519, 17209520 and 17209821.

References

1. Applicative Intersection Types (Artifact). Zenodo (Aug 2022). https://doi.org/10.5281/zenodo.7004695, https://doi.org/10.5281/zenodo.7004695

2. Alpuim, J., Oliveira, B.C.d.S., Shi, Z.: Disjoint polymorphism. In: European Symposium on Programming. pp. 1–28. Springer (2017)
3. Barendregt, H., Coppo, M., Dezani-Ciancaglini, M.: A filter lambda model and the completeness of type assignment1. The journal of symbolic logic **48**(4), 931–940 (1983)
4. Bi, X., Oliveira, B.C.d.S., Schrijvers, T.: The essence of nested composition. In: 32nd European Conference on Object-Oriented Programming (ECOOP 2018). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2018)
5. Bobrow, D.G., DeMichiel, L.G., Gabriel, R.P., Keene, S.E., Kiczales, G., Moon, D.A.: Common lisp object system specification. ACM Sigplan Notices **23**(SI), 1–142 (1988)
6. Castagna, G., Ghelli, G., Longo, G.: A calculus for overloaded functions with subtyping. Information and Computation **117**(1), 115–135 (1995)
7. Coppo, M., Dezani-Ciancaglini, M., Venneri, B.: Functional characters of solvable terms. Mathematical Logic Quarterly **27**(2-6), 45–58 (1981)
8. Davies, R., Pfenning, F.: Intersection types and computational effects. In: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming. pp. 198–208 (2000)
9. Dunfield, J.: Elaborating intersection and union types. Journal of Functional Programming **24**(2-3), 133–165 (2014)
10. Dunfield, J., Krishnaswami, N.: Bidirectional typing. ACM Computing Surveys (CSUR) **54**(5), 1–38 (2021)
11. Ernst, E.: Family polymorphism. In: European Conference on Object-Oriented Programming. pp. 303–326. Springer (2001)
12. Fan, A., Huang, X., Xu, H., Sun, Y., d. S. Oliveira, B.C.: Direct foundations for compositional programming. In: 36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany. LIPIcs, vol. 222, pp. 18:1–18:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)
13. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. Journal of the ACM (JACM) **55**(4), 1–64 (2008)
14. Huang, X., Zhao, J., Oliveira, B.C.: Taming the merge operator. Journal of Functional Programming **31** (2021)
15. Kaes, S.: Parametric overloading in polymorphic programming languages. In: European Symposium on Programming. pp. 131–144. Springer (1988)
16. Marntirosian, K., Schrijvers, T., Oliveira, B.C.d.S., Karachalias, G.: Resolution as intersection subtyping via modus ponens. Proc. ACM Program. Lang. **4**(OOPSLA) (nov 2020)
17. Oliveira, B.C.d.S., Shi, Z., Alpuim, J.: Disjoint intersection types. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. pp. 364–377 (2016)
18. Pierce, B.C.: Programming with intersection types and bounded polymorphism. Ph.D. thesis, Citeseer (1991)
19. Pottinger, G.: A type assignment for the strongly normalizable λ-terms. To HB Curry: essays on combinatory logic, lambda calculus and formalism pp. 561–577 (1980)
20. Rémy, D.: Type checking records and variants in a natural extension of ml. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 77–88 (1989)
21. Reynolds, J.C.: Preliminary design of the programming language forsythe (1988)

22. Reynolds, J.C.: The coherence of languages with intersection types. In: International Symposium on Theoretical Aspects of Computer Software. pp. 675–700. Springer (1991)
23. Reynolds, J.C.: Design of the programming language f orsythe. In: ALGOL-like languages, pp. 173–233. Springer (1997)
24. Rioux, N., Huang, X., Oliveira, B.C.d.S., Zdancewic: A bowtie for a beast. Tech. Rep. MS-CIS-22-02, Department of Computer and Information Science, University of Pennsylvania (2022)
25. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 60–76 (1989)
26. Wadler, P., et al.: The expression problem. Posted on the Java Genericity mailing list (1998)
27. Wand, M.: Type inference for record concatenation and multiple inheritance. Information and Computation **93**(1), 1–15 (1991)
28. Xie, N., Oliveira, B.C.d.S.: Let arguments go first. In: European Symposium on Programming. pp. 272–299. Springer (2018)
29. Zhang, W., Sun, Y., Oliveira, B.C.: Compositional programming. ACM Transactions on Programming Languages and Systems (TOPLAS) **43**(3), 1–61 (2021)