# Applicative Intersection Types

*by*

**Xue, Xu**
(薛旭)

A thesis submitted in partial fulfillment of the requirements for
the degree of Master of Philosophy
at The University of Hong Kong

January 2023

Abstract of thesis entitled
**"Applicative Intersection Types"**

Submitted by
**Xue, Xu**

for the degree of Master of Philosophy
at The University of Hong Kong
in January 2023

A new programming language is often elaborated by introducing different features. Some of those features are not orthogonal, and their combination risks breaking the safety of the whole system, especially in statically typed languages. Also, the newly coming feature is conventionally reasoned in isolation or in a rather minimised calculus. It is not surprising for language researchers and implementers to desire a general framework that contains features as much as possible and has salient extensibility. Intersection types are a nice fit for this purpose.

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 thesis, 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 formalised in the Coq theorem prover and we have a prototype implementation as well.

---

**An abstract of 260 words**

*To my beloved parents*

# Declaration

I declare that this thesis represents my own work, except where due acknowledgment is made, and that it has not been previously included in a thesis, dissertation or report submitted to this University or to any other institution for a degree, diploma or other qualifications.

. . . . . . . . . . . . . . . . . . . . . . . . . .

**Xue, Xu**

January 2023

# Acknowledgments

I thank:

- Bruno C. d. S. Oliveira, the best supervisor I have ever had;
- Ningning Xie, Jinxu Zhao, Xuejing Huang, Yaoda Zhou, Baber Rehman, Mingqi Xue, Yaozhu Sun, Wenjia Ye, Chen Cui, Jinhao Tan and Shengyi Jiang for their continuous Q&A support, generous advice, enlightenment, valuable answers to my numerous dumb questions, quick and clever suggestions in my research and life;
- Atsushi Igarashi, Sérgio Lourenço and Tsz Hon Yuen for their committee work of thesis feedback and examination;
- Jianqiao Lu, Hao Yu, Teng Zhong, Xinhao Zhang, Yuzhe Zhang, Minhe Zhang, Menghui Qin, Xu Wang and Shunran Zhang for their lasting friendship;
- Tian Lan, Zhi Li and Miumiu for their host of my first year in Hong Kong;
- My parents for their unconditional support.

# Contents

# List of Figures

# 1 Introduction

## 1.1 Motivation

Inventing new programming languages is no longer big news in recent years. For new programming languages, a paradigm or principle is usually established first, which will be elaborated by introducing different features. Some of those features are not orthogonal, and their combination risks breaking the safety of the whole system, especially in statically typed languages. On the other hand, programming languages are not designed at once. In reality, many features are extended after a prototype is settled. Thus extensibility is the primary concern when designing programming languages.

In a theoretical setting, the newly coming feature is conventionally reasoned in isolation or in a rather minimised calculus. The combination of the new features and other existing features will not be examined thoroughly, making more complete languages hard to model, maintain and extend. It is not surprising for language researchers and implementers to desire a general framework that contains features as much as possible and has salient extensibility. Intersection types are a nice fit to serve as a general framework to model many features, argued in Dunfield [2014],

> "our goal is to use intersections and unions as general mechanisms for encoding language features, so we really should do it in full generality, or not at all."

Intersection types [Barendregt et al. 1983; Coppo et al. 1981; Pottinger 1980; Reynolds 1991] have a long history in programming languages. They were originally introduced to assign meaning to canonical forms in strongly normalising lambda terms by Pottinger [1980]. Reynolds [1988] 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 [2014] 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* [Castagna et al. 1995; Dunfield 2014], *return type overloading* [Marntirosian et al. 2020], *extensible records* [Dunfield 2014; Reynolds 1988] and *nested composition* [Bi et al. 2018; Huang et al. 2021].

In the following part of this chapter, we give an overview of these features encoded by intersection types and show their interactions.

### 1.1.1 Overloading

Function overloading is a form of polymorphism where the implementation of functions can vary depending on the 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 [Wadler and Blott 1989] to assign the type `Show a` $\Rightarrow$ `a` $\rightarrow$ `String` to `show` with instances defined. The key idea of type classes is to combine the feature of parametric polymorphism and ad-hoc polymorphism by specifying constraints on the polymorphic variables. Practically in C++, overloaded functions are represented as a set of candidate functions. The arguments list and candidate functions will be used in overloading resolution based on a large group of sophisticated rules.

With intersection types, we can represent the overloaded `show` function in a relatively more straightforward way. For instance, the `show` function has the below type:

$$\texttt{show} : (\texttt{Int} \rightarrow \texttt{String}) \mathbin{\&} (\texttt{Bool} \rightarrow \texttt{String})$$

Generally speaking, the type of overloaded functions is the type intersection of their different branches. The function `show` has two branches: `showInt` and `showBool`, thus the type of their overloads is `Int` $\rightarrow$ `String` intersected with `Bool` $\rightarrow$ `String`.

### 1.1.2 Return type overloading

Return type overloading is another form of overloading. The common example is Haskell's `read :: Read a` $\Rightarrow$ `String` $\rightarrow$ `a`; it accepts a string and returns its parsed value (integers, booleans etc.). Their implementation is not determined in function application. Instead, it is determined by the surrounding type contexts. For example, `succ (read "1")` returns the integer `2` if `succ` is the successor function that takes an integer and return its successor. The type definition works the same with `show`:

$$\texttt{read} : (\texttt{String} \rightarrow \texttt{Int}) \mathbin{\&} (\texttt{String} \rightarrow \texttt{Bool})$$

### 1.1.3 Extensible Records

Records are useful for modelling features in Object-Oriented Programming. In record calculi, the records are interpreted as objects; labels are member names; associated values are attributes and methods. Inheritance can be modelled as concatenation of records [Cardelli and Mitchell 1991].

In calculi with intersection types, multi-field records can be viewed as syntactic sugar.

$$\{x : A, y : B\}\ \texttt{desugars to}\ \{x : A\}\ \&\ \{y : B\}$$

With this encoding, we obtain the width subtyping of records ($\{l_i : T_i\}^{i=1..n..n+k} <: \{l_i : T_i\}^{1..n}$) for free since it is subsumed by the subtyping of intersection types ($A \,\&\, B <: A$). For the depth subtyping, it is also easy to extend this rule into the subtyping of intersection types without affecting other rules. The permutation rule is subsumed by the commutativity of intersection types.

### 1.1.4 Nested Composition

Nested composition reflects the 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. Nested composition is not an independent feature; it comes into use when we combine it with another feature of intersection types (e.g., overloading and extensible records).

In the context of records, the distributive subtyping rule enabling nested composition is

$$\{l : A\}\ \&\ \{l : B\} <: \{l : A \,\&\, B\}$$

Nested record composition is a key feature in Compositional Programming (CP) [Zhang et al. 2021]. It plays an important role in solving challenging modularity problems such as the Expression Problem [Wadler 1998] and models forms of family polymorphism [Ernst 2001]. We show CP code here to illustrate how the Compositional Programming style solves the expression problem in two dimensions.

Pre-defined Language We start with defining the type `Eval` and the compositional interface `AddSig`. `Eval` defines a method `eval` which returns an integer as the output. `AddSig` works similarly to an interface in Java and declares two constructors `Lit` and `Add`. `Exp` works like a type parameter and can be instantiated by a type. For example, in the trait `evalAdd`, `Exp` is instantiated by `Eval`. We then build a concrete expression `expAdd`.

```
type Eval = { eval : Int };
type AddSig<Exp> = {
  Lit: Int → Exp;
  Add: Exp → Exp → Exp;
};
evalAdd = trait implements AddSig<Eval> ⇒ {
  (Lit     n).eval = n;
  (Add e1 e2).eval = e1.eval + e2.eval;
};

expAdd Exp = trait [self : AddSig<Exp>] ⇒ {
  exp = Add (Lit 4) (Lit 8);
};
```

Adding Operations  Adding new operations is straightforward: we define a new type `Print` and construct a trait `printAdd` by instantiating `Exp` by type `Print` in `AddSig`.

```
type Print = { print : String };
printAdd = trait implements AddSig<Print> ⇒ {
  (Lit     n).print = toString n;
  (Add e1 e2).print = "(" ++ e1.print ++ " + " ++ e2.print ++ ")";
};
```

Adding Datatypes  To add a new datatype, we extend the compositional interface `AddSig` with a new constructor: `Mul`. We can then implement the trait `evalMul` and `printMul` by instantiating the type parameter with `Eval` and `Print`. We then build a concrete expression `expMul`.

```
type MulSig<Exp> = AddSig<Exp> & {
  Mul : Exp → Exp → Exp;
};
evalMul = trait implements MulSig<Eval> inherits evalAdd ⇒ {
  (Mul e1 e2).eval = e1.eval * e2.eval;
};
printMul = trait implements MulSig<Print> inherits printAdd ⇒ {
  (Mul e1 e2).print = "(" ++ e1.print ++ " * " ++ e2.print ++ ")";
};
expMul Exp = trait [self : MulSig<Exp>] inherits expAdd @Exp ⇒ {
  override exp = Mul super.exp (Lit 4);
};
```

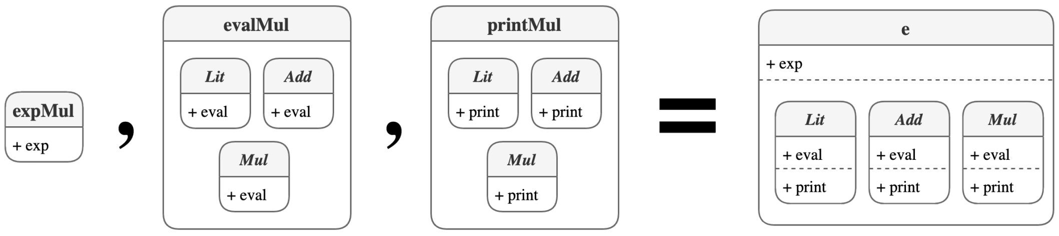

Figure 1.1: Nested Composition in Expression Problem

Nested Composition  To build a new expression which supports a new operation and a new datatype, we use the merge operator to compose them together. We demonstrate this idea in Figure 1.1. We can see that those three figures are composed nestedly with all the hierarchies combined together.

```
e = new evalMul , printMul , expMul @(Eval & Print);
```

Nested composition can also occur with functional intersections, using the below subtyping rule and enables the overloaded function to be curried.

$$(A \to B) \,\&\, (A \to C) <: A \to (B \,\&\, C)$$

## 1.2 Problems and Challenges

We listed some features that intersection types equipped with a merge operator can model. Nevertheless, no previous calculus supports all four features together. Some calculi enable function overloading [Castagna et al. 1995], but preclude return type overloading and nested composition. On the other hand, calculi with disjoint intersection types [Bi et al. 2018; Huang et al. 2021] support return type overloading and nested composition but disallow conventional functional overloading. Dunfield's calculus [Dunfield 2014] 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.

### 1.2.1 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:

$$(\texttt{Int} \to \texttt{String}) \,\&\, (\texttt{Bool} \to \texttt{String}) <: \texttt{Int} \to \texttt{String}$$

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) [Huang et al. 2021] 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 $\lambda_{i+}$ [Huang et al. 2021] 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.

### 1.2.2 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:

$$\texttt{pshow unit 1} \hookrightarrow \texttt{"1"} \qquad \texttt{pshow unit true} \hookrightarrow \texttt{"true"}$$

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.

### 1.2.3 Ambiguities in the Design Space

A particular challenge with overloading and merges is ambiguities. Ambiguities can happen both with record projections and/or overloading. Repeated labels are always a concern in designing the concatenation of records. Whether to allow repeated labels leads to different designs of records. Ambiguities in function overloading are more complicated. We identify two key problems of overloading ambiguities below.

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 [Castagna et al. 1995; Dunfield 2014; Kaes 1988]. 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.

$$f : \texttt{Int} \to \texttt{Int} \to \texttt{Int} \qquad g : \texttt{Int} \to \texttt{Bool} \to \texttt{Bool}$$

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)`.

Ambiguities on the output types Output types are usually not considered in common languages, especially when input types are not likely to cause ambiguities. Unfortunately, in a formal system with an unrestricted merge operator, we can always create a term fitting both branches. For example, we can apply `show` to `1,,true`, and the result is `"1",,"true"`, which potentially loses determinism, since term `"1",,"true"` can be reduced to `"1"` or `"true"` under the cast of `String`. In conclusion, whatever the input types are, there is always a risk to break determinism if the output types are the same under the context of an unrestricted merge operator. There is also a concern in modelling return type overloading since there exist ambiguities when the surrounding context cannot distinguish between instances.

## 1.3 Key Ideas and Contribution

This thesis 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 specialised 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 [Bobrow et al. 1988; Castagna et al. 1995; Dunfield 2014; Kaes 1988] only considers uncurried overloaded functions.

### 1.3.1 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`:

$$(\texttt{Int} \to \texttt{String}) \ \& \ (\texttt{Bool} \to \texttt{String}) <: \texttt{Int} \to ?$$

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`.

$$\{x : \texttt{String}\} \ \& \ \{x : \texttt{Int}\} <: \{x : ?\}$$

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 ⊚ 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 ≪) 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
```

Ambiuguities To deal with the ambiguities we mentioned in the last section, we isolate the behaviour when both branches can accept the selector type. We can reject or accept the below case when we adopt different composition operators, which will be expanded in the Chapter 3.

$$(\texttt{Int} \to \texttt{Int}) \ \& \ (\texttt{Int} \to \texttt{Bool}) \ll \texttt{Int} = \texttt{Int} \circledcirc \texttt{Bool}.$$

Another solution to avoid ambiguities and recover determinism is to adopt disjoint intersection types, which is mentioned in the next section and is explored in Chapter 5.

Correctness We prove the soundness and completeness of the applicative subtyping with regard to the normal subtyping. We have several variants of the metatheory and only show the function case here. The details will be expanded in Chapter 3.

**Lemma 1.1** (Soundness). *If $A \ll B = C$, then $A <: B \to C$.*

**Lemma 1.2** (Completeness). *If $A <: B \to C$, then $\exists D, A \ll B = D \wedge D <: C$.*

### 1.3.2 TDOS for Overloading

For the semantics, we follow up the idea of typed-directed operational semantics [Huang et al. 2021] 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.

### 1.3.3 Two Calculi

We present two calculi to demonstrate the applicative subtyping and applicative dispatching. 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 → String` is not disjoint with `Bool → String`, such merges will be rejected.

### 1.3.4 First-class Curried Overloaded Functions

First-class curried overloading is considered a novel feature when we designed this system. It appears when the system has both overloading and nested composition and is a powerful idiom where programmers can abstract over and return overloaded functions. For example, we create our overloaded function `show` by merging two implementations together. `pshow` is the `show` function wrapped in a thunk.

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

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

The newly created function `pshow` is still recognised as an overloaded function; it makes the selection when the second arguments are given. We see the examples below.

$$\texttt{pshow ()} \hookrightarrow \texttt{show} \quad \texttt{pshow () 1} \hookrightarrow \texttt{"1"} \quad \texttt{pshow () true} \hookrightarrow \texttt{"true"}$$

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 makes the overloaded functions flexible to program with.

### 1.3.5 Summary

In summary, the contributions of this thesis 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 specialised applicative subtyping relation to deal with the problem of inferring output 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.
- *Interpreter Implementation:* We implement our calculus as a Lisp dialect. We enrich the implementation with several primitives and syntactic sugar and employ a contract-based function to ensure the correctness of the interpreter.
- *Mechanical formalisation :* All the subtyping algorithms, calculi and proofs are formalised in the Coq theorem prover.

## 1.4 Outline and Published work

This thesis is organised as follows:

- Chapter 2 gives an introduction to basic concepts and notational preliminaries.
- Chapter 3 discusses application typing problems and challenges and presents several applicative subtyping designs.

- Chapter 4 presents type sound calculus with an unrestricted merge operator, which supports all features mentioned previously.
- Chapter 5 enrich the previous calculus with disjointness and prove its determinism.
- Chapter 6 gives interpreter implementation of our calculi.
- Chapter 7 shows the related work and their differences.
- Chapter 8 concludes this thesis and gives directions for future work.

PUBLICATIONS This thesis is partially based on the below publications.

Xu Xue, Bruno C. d. S. Oliveira and Ningning Xie "Applicative Intersection Types" In *The 20th Asian Symposium on Programming Languages and Systems* (APLAS 2022).

ARTIFACT The mechanical formalisation and interpreter implementation can be found at:

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

# 2 Background

In this chapter, basic concepts and notational preliminaries will be explained in detail. This part is helpful for readers to gain some intuition and references in order to better understand and evaluate the rest of the thesis. Section 2.1 introduces the Simply Typed $\lambda$-Calculus and the notion of subtyping, then discusses their combination with bidirectional typing: $\lambda_{\leq}$. Section 2.2 explains intersection types, the merge operator and their applications. Section 2.3 presents a calculus introduced by Dunfield, which serves as a starting point of our system. Section 2.4 introduces the main idea of type-directed operational semantics.

## 2.1 Simply Typed $\lambda$-Calculus with Subtyping

The $\lambda$-Calculus is a formalisation of a mathematical model that describes computation. It has inspired the designs of many mainstream programming languages (Python, JavaScript, etc.) and has also been adopted as the basis of functional programming languages such as Scheme, Haskell, etc.

### 2.1.1 Simply Typed $\lambda$-Calculus

The prototype of the $\lambda$-Calculus was firstly introduced by Church [1932] to serve as the foundation of mathematics. Its inconsistency was pointed out by Kleene and Rosser [1935]. Later, Church revised it, introduced untyped $\lambda$-Calculus and then augmented it with types to make it become logically consistent [Church 1940]. The $\lambda$-Calculus with simple types (base types and arrow types) is called the Simply Typed $\lambda$-Calculus (STLC).

The usage of types develops into the design of the type system, where the most fundamental property is type safety, described by Milner [1978]:

> Well-typed programs cannot "go wrong"[1].

Practically, type safety [2] ensures that the type-checker will detect all stuck terms before the evaluation. In STLC, well-typed programs are expressions which can be type-checked.

[1] "Wrong" does not mean that we can exclude all bad cases, only part of them, like add an integer to a string

[2] People usually treat type safety and type soundness as synonyms, but for clarity, we only use type soundness to describe the theorem that can be formally stated and proved.

Terms in the $\lambda$-Calculus are literals, variables, abstractions and application. Types are literal types and arrow types (also called function types). A context is a sequence of pairs between variables and their types.

$$\begin{array}{llcl} \text{Expressions} & e & := & i \mid x \mid \lambda x : A.\, e \mid e_1\, e_2 \\ \text{Types} & A, B & := & Int \mid A \to B \\ \text{Context} & \Gamma & := & \emptyset \mid \Gamma,\, x : A \end{array}$$

In syntax,

- $i$ is used for literals, for simplicity we only consider it as numbers.
- $\lambda$ is often used for the prefix name of a calculus, or a lambda abstraction in the $\lambda$-Calculus.
- $x, y, z$ are reserved for bound variables in the $\lambda$-Calculus.
- $A, B, C$ are reserved for type variables.
- $e_i$ is used for terms.
- $\Gamma$ describes a typing context, which is a sequence of pairs $(x, A)$.

With the syntax of expressions, types and context defined, we will introduce a relation called *typing assignment* in Figure 2.1, which is a 3-ary relation between context $\Gamma$, term $e$ and type $A$. Rule T-Lit states that literals have integer types. Rule T-Var says that variable $x$ has the type $A$ if the context $\Gamma$ has their association. Rule T-Lam is the rule for lambda abstractions: $\lambda x : A.\, e$ has the arrow type $A \to B$ if the body $e$ has the type $B$ under the context $\Gamma$ extended with a pair $x : A$. The application rule T-App derives the output type from $e_1$ and checks the equality between the input type and the type of argument $e_2$. With the typing rules, we will reject ill-typed terms like 1 2 (literal 1 is applied to argument 2) since 1 does not have a function type.

The typing assignment relation defines the statics of the language, while semantics defines its dynamics. There are many approaches to define semantics, including *denotational semantics* [Schmidt 1986], *operational semantics* [Plotkin 1981] and *axiomatic semantics* [Goguen et al. 1977]. This thesis only focuses on operational semantics, since it is closer to the computation intuition and implementation details.

Before presenting the reduction rules of the operational semantics, the normal form of this language should be established to characterise the end of the computation process. Reducible terms are called *redexes*. An expression containing no redexes is in *normal form*. The normal

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

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

$$\text{T-App}\ \frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B}$$

Figure 2.1: STLC Typing

$\boxed{e \longmapsto e'}$ *(Small-Step Reduction)*

$$\text{ST-Beta}\ \frac{}{(\lambda x : A.\, e)\, v \longmapsto e[x \mapsto v]} \qquad \text{ST-App-L}\ \frac{e_1 \longmapsto e_1'}{e_1\, e_2 \longmapsto e_1'\, e_2} \qquad \text{ST-App-R}\ \frac{e_2 \longmapsto e_2'}{v_1\, e_2 \longmapsto v_1\, e_2'}$$

Figure 2.2: STLC Operational Semantics

form defines the possible structure of termination of reduction, which is represented by *value* under the context of operational semantics. A value is either a literal or a lambda abstraction in the STLC.

$$\text{Values} \qquad v := i \mid \lambda x : A.\, e$$

In Figure 2.2, rule ST-App-L and rule ST-App-R are normal *congruence rules* and account for evaluating the arguments before they are substituted in the abstraction body. A more interesting rule is rule ST-Beta, which introduces a new operation: *substitution*. Substitution essentially describes substituting one expression for a variable in another expression. Dealing with capture-avoiding substitution often involves cumbersome reasoning, we will use the technique of *Locally Nameless Representation* [Charguéraud 2012] for the formalisation of our calculi in Coq.

### 2.1.2 Subtyping

Subtyping arises as one of the critical features in Object-Oriented Programming. It describes the behaviour (often called subtype polymorphism) that terms can be safely replaced by other

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

$$\textsc{S-Refl}\ \frac{}{A \leq A} \qquad \textsc{S-Trans}\ \frac{A \leq B \qquad B \leq C}{A \leq C} \qquad \textsc{S-Arr}\ \frac{B_1 \leq A_1 \qquad A_2 \leq B_2}{A_1 \to A_2 \leq B_1 \to B_2} \qquad \textsc{S-Top}\ \frac{}{A \leq \mathsf{Top}}$$

Figure 2.3: STLC Subtyping

terms typed with their subtype. Reflecting on the typing relation would be the typing subsumption rule.

$$\textsc{Ty-Sub}\ \frac{\Gamma \vdash e : A \qquad A \leq B}{\Gamma \vdash e : B}$$

Rule Ty-Sub says that if $e$ has the type $A$ and $A$ is the subtype of $B$, then $e$ has the type $B$. The intuition to understand the subsumption is to think of it as the feature of object-oriented programming, where the term typed with a subclass can also have the superclass type. We formally define the subtyping in Figure 2.3. $A \leq B$ is the subtyping relation, which can be viewed as a preorder of sets of two types.

Rule Sub-Refl and rule Sub-Trans define the properties of reflexivity and transitivity. Rule S-Arr describes the subtyping relation of arrow types: input types are contravariant and output types are covariant. Rule S-Top employs a new Top type, which is the supertype of all types.

### 2.1.3 Bidirectional Typing

Initially, a typing judgment only accounts for a type assignment system, which is often written in the form of

$$\Gamma \vdash e : A$$

Bidirectional typing introduces a notion of mode [Warren 1978] into typing: *synthesis mode* and *check mode*.

- *Synthesis Mode* ($\Rightarrow$)*:* Synthesize a type from an expression, which can be implemented as a infer function, with typing context $\Gamma$ and expression $e$ as inputs and the type $A$ as output.

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

$$\frac{}{\Gamma \vdash i \Rightarrow \mathsf{Int}}\ \textsc{Ty-Lit} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A}\ \textsc{Ty-Var} \qquad \frac{\Gamma, x : A \vdash e \Rightarrow B}{\Gamma \vdash \lambda x : A.\, e \Rightarrow A \to B}\ \textsc{Ty-Lam}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\, e_2 \Rightarrow B}\ \textsc{Ty-App} \qquad \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash e : A \Rightarrow A}\ \textsc{Ty-Ann} \qquad \frac{\Gamma \vdash e \Rightarrow A \qquad A \leq B}{\Gamma \vdash e \Leftarrow B}\ \textsc{Ty-Sub}$$

Figure 2.4: Bidirectional Typing

- *Check Mode ($\Leftarrow$):* Check an expression with a specific type, which can be implemented as a check function, with typing context $\Gamma$, the expression $e$ and the type $A$ as inputs and booleans indicating success or failure as output.

Bidirectional typing originated from Pierce and Turner [2000] and is extremely helpful for typing a system with subtyping by adding type annotations. The typing of STLC with subtyping can be introduced by bidirectionalizing each rule in Figure 2.4. There are two rules to trigger the check mode: rules Ty-App and Ty-Ann. Rule Ty-App expects $e_1$ to infer the arrow type and use the input type $A$ to check the argument $e_2$. Rule Ty-Ann uses the annotated type to check the term $e$.

In conclusion, there are several advantages of bidirectional typing:

- It is easy to extend the type system to support subtyping.
- The feature of local type inference enables simpler syntax and localises the error information.

### 2.1.4 Metatheory

In the setting of operational semantics, type soundness can be determined by preservation and progress [Wright and Felleisen 1994]. There are three properties we are interested in:

**Theorem 2.1** (Preservation). *If $\emptyset \vdash e : A$ and $e \longmapsto e'$, then $\emptyset \vdash e' : A$.*

Preservation tells that the type of expression will be preserved during the evaluation. This is not the case when there is subtyping, where the type may become smaller. We can slightly modify this theorem in the presence of subtyping.

**Theorem 2.2** (Preservation). *If $\emptyset \vdash e : A$ and $e \longmapsto e'$, then $\exists B$, $\emptyset \vdash e' : B$ and $B \leq A$.*

**Theorem 2.3** (Progress). *If $e$ is well-typed, then $e$ is a value or $\exists e', e \longmapsto e'$.*

The progress theorem ensures that any well-typed expression is either a value or makes a progress.

**Theorem 2.4** (Determinism). *If $e$ is well-typed, $e \longmapsto e_1$ and $e \longmapsto e_2$, then $e_1 = e_2$.*

Determinism is a strong property which requires each reduction step is deterministic.

## 2.2 Intersection Types and Merge Operator

We have already seen one compound type in STLC: arrow types. In this section, we introduce another one: intersection types. We then introduce the merge operator, which is able to create terms inhabited by intersection types.

### 2.2.1 Intersection Types

Intersection types (denoted as $A \,\&\, B$) were firstly introduced in Coppo et al. [1981] as a theoretical notion to describe the properties of $\lambda$ terms. Simply speaking, intersection types enable one term to have multiple types. As presented in typing rules, a term $e$ can have the intersection types if $e$ has both the type $A$ and the type $B$.

$$\textsc{\&-Intro}\ \frac{\Gamma \vdash e : A \qquad \Gamma \vdash e : B}{\Gamma \vdash e : A \,\&\, B} \qquad \textsc{\&-Elim-L}\ \frac{\Gamma \vdash e : A \,\&\, B}{\Gamma \vdash e : A} \qquad \textsc{\&-Elim-R}\ \frac{\Gamma \vdash e : A \,\&\, B}{\Gamma \vdash e : B}$$

If types are treated as a way to describe sets of terms, then intersection types are introduced to describe conjunctive forms. Calculi with intersection types will have many interesting properties. For example, they can type check terms which are impossible to be checked by simple types. The self-application term $\lambda x : A \,\&\, (A \to B).\ x\ x$ cannot be typed in STLC but it is a well-typed term in calculi with intersection types.

Although it is meaningless to find some terms which is the intersection of integers and booleans, things will become more interesting when we create intersection types of more complex terms (e.g., functions, records and interfaces). Intersection types have already been adopted by many programming languages, including TypeScript, Scala etc.

### 2.2.2 Merge Operator

The merge operator (denoted as $\,,,$) has been introduced by Reynolds [Reynolds 1988], and later refined by Dunfield [Dunfield 2014], 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 thesis, 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 [Coppo et al. 1981], 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 [Pottinger 1980]. 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.

### 2.2.3 Applications of 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 all be encoded by our calculus in Chapter 4.

Records and Record Projections The idea of using the merge operator to model record concatenation firstly appears in Reynolds [1997]. 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).

$$(\{\mathtt{x} = \mathtt{"hello"}\},,\{\mathtt{y} = \mathtt{"world"}\}).\mathtt{x} \hookrightarrow \mathtt{"hello"}$$

Function overloading Function overloading is a form of polymorphism where the implementation of functions can vary depending on the 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 [Wadler and Blott 1989] to assign the type $\mathtt{Show\ a} \Rightarrow \mathtt{a} \rightarrow \mathtt{String}$ to `show` with instances defined.

With intersection types, we can employ the merge operator [Castagna et al. 1995; Reynolds 1988] to define a simplified version of the overloaded `show` function. For instance, the `show` function below has type `(Int → String) & (Bool → String)`.

$$\texttt{show} : (\texttt{Int} \rightarrow \texttt{String}) \mathbin{\&} (\texttt{Bool} \rightarrow \texttt{String}) = \texttt{showInt,,showBool}$$

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.

$$\texttt{show 1} \hookrightarrow \texttt{1} \qquad \texttt{show true} \hookrightarrow \texttt{"true"}$$

Return type overloading One common example of return type overloading is the `read` function in Haskell,

$$\texttt{read} :: \texttt{Read a} \Rightarrow \texttt{String} \rightarrow \texttt{a}$$

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:

$$\texttt{read} : (\texttt{String} \rightarrow \texttt{Int}) \mathbin{\&} (\texttt{String} \rightarrow \texttt{Bool}) = \texttt{readInt,,readBool}$$

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 `succ` is the successor function on integers and `not` is the negation function on booleans, then we can write:

$$\texttt{succ (read "1")} \hookrightarrow \texttt{2} \qquad \texttt{not (read "true")} \hookrightarrow \texttt{false}$$

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 $\{l : A\} \mathbin{\&} \{l : B\} <: \{l : A \mathbin{\&} B\}$. With this rule we can have the following expression:

$$(\{\texttt{x = "hello"}\}\texttt{,,}\{\texttt{x = 1}\}).\texttt{x} \hookrightarrow \texttt{"hello",,1}$$

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* [Zhang et al. 2021], which uses it to solve challenging modularity problems such as the Expression Problem [Wadler 1998], and to model forms of *family polymorphism* [Ernst 2001]. We refer interested readers to the work of Zhang et al. [2021] 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 expression:

$$(\texttt{succ}\,,,\,\texttt{intToDigit})\;\texttt{5} \hookrightarrow \texttt{6}\,,,\,\texttt{"5"}$$

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 Dunfield's Calculus

In this section, we will give an introduction to Dunfield's calculus [Dunfield 2014], which aims to provide a general mechanism to subsume different features, including *overloading*, *records* and *heterogeneous data*. In her calculus, intersection types and merge operator are introduced in the syntax. For simplicity, we ignore the union types here.

### 2.3.1 Syntax

We first present the (source) syntax of the calculus.

$$\begin{array}{llcl}
\text{Expressions} & e & := & () \mid x \mid \lambda x.\, e \mid e_1\, e_2 \mid e_1 \,,,\, e_2 \\
\text{Types} & A, B & := & \top \mid A \to B \mid A \,\&\, B \\
\text{Context} & \Gamma & := & \emptyset \mid \Gamma,\, x : A \\
\text{Values} & v & := & x \mid () \mid \lambda x.\, e \mid v_1 \,,,\, v_2
\end{array}$$

The expressions $e$ are conventional except for the merge expression $e_1 \,,,\, e_2$, which describes that two expressions can be merged by a merge operator $,,$. The types $A$ are top types, function types and intersection types. Values are most standard, except for merges $v_1 \,,,\, v_2$ even though they can step further.

### 2.3.2 Subtyping and Typing

Subtyping is shown in Figure 2.5. The rules for arrow and intersection types are standard. The bidirectional typing is shown in Figure 2.6. Rule T-Mrg-Inf is the introduction typing rule for the merge operator, $e_1 \,,,\, e_2$ has the type $A_1 \,\&\, A_2$ if $e_1$ has the type $A_1$ and $e_2$ has the type $A_2$. Rule T-Mrg-Inf-K is the elimination rule, $e_1 \,,,\, e_2$ has the type $A$ if one of the

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

$$\frac{B_1 \leq A_1 \qquad A_2 \leq B_2}{A_1 \to A_2 \leq B_1 \to B_2}\ \textsc{S-Arr} \qquad \frac{}{A \leq \mathsf{Top}}\ \textsc{S-Top} \qquad \frac{A_k \leq B}{A_1 \,\&\, A_2 \leq B}\ \textsc{S-And-K} \qquad \frac{A \leq B_1 \qquad A \leq B_2}{A \leq B_1 \,\&\, B_2}\ \textsc{S-And}$$

Figure 2.5: Dunfield's Subtyping

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

$$\frac{x : A \mathit{\,in\,} \Gamma}{\Gamma \vdash x \Rightarrow A}\ \textsc{T-Var} \qquad \frac{\Gamma \vdash e_k \Leftarrow A}{\Gamma \vdash e_1\,,,\,e_2 \Leftarrow A}\ \textsc{T-Mrg-CHK-K} \qquad \frac{\Gamma \vdash e_k \Rightarrow A}{\Gamma \vdash e_1\,,,\,e_2 \Rightarrow A}\ \textsc{T-Mrg-Inf-K}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A_1 \qquad \Gamma \vdash e_2 \Rightarrow A_2}{\Gamma \vdash e_1\,,,\,e_2 \Rightarrow A_1 \,\&\, A_2}\ \textsc{T-Mrg-Inf} \qquad \frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.\, e \Leftarrow A \to B}\ \textsc{T-Lam}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\, e_2 \Rightarrow B}\ \textsc{T-App} \qquad \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash e : A \Rightarrow A}\ \textsc{T-Ann} \qquad \frac{\Gamma \vdash e \Rightarrow A \qquad A \leq B}{\Gamma \vdash e \Leftarrow B}\ \textsc{T-Sub}$$

Figure 2.6: Dunfield's Typing

branch ($e_1$ or $e_2$ has the type $A$). The design is flexible to encode many features: including function overloading and record projection, though leading to non-deterministic inference since it is not syntax-directed.

Feature Encoding The typing is very powerful and enables inference for the application. For example, `show` is the merge of `showInt` and `showBool`. We show the typing example for the `show 1`,

$$\frac{\dfrac{\emptyset \vdash \texttt{showInt} \Rightarrow \texttt{Int} \to \texttt{String}}{\emptyset \vdash \texttt{show} \Rightarrow \texttt{Int} \to \texttt{String}}\ \textsc{T-Mrg-Inf-K} \qquad \emptyset \vdash 1 \Leftarrow \texttt{Int}}{\emptyset \vdash \texttt{show}\ 1 \Rightarrow \texttt{String}}\ \textsc{T-App}$$

### 2.3.3 Elaboration Semantics

Dunfield's calculus employs a non-deterministic elaboration semantics. The calculus with intersection types can be elaborated to the calculus with product types. In the elaboration, the explicit elimination of product types will be inserted after the type checking. Note that there is also a operational semantics, though it simply helps check the correctness of the elaboration and does not actually model the computation process.

## 2.4 Type-Directed Operational Semantics

In classical calculi (e.g., *Simply Typed Lambda Calculi*), types are usually erased in the dynamic semantics. In contrast with *type-erasure semantics*, *type-dependent* (also known as *type-passing semantics*) makes use of the types in the evaluation to make run-time decisions.

In more realistic programming languages where types are required to assist the evaluation, the *elaboration semantics* is often adopted. That is to say, languages with runtime decisions are elaborated to another language without them. A notable example is type classes, which is an approach to support parametric overloading and has been incorporated as the main feature in languages like Haskell. The main idea is to translate a function typed with class into a function with a dictionary which will be passed at the runtime. This inspires many designs like *implicits* in Scala [Oliveira et al. 2010] and *instance arguments* in Agda [Devriese and Piessens 2011].

Type-dependent semantics still has its own advantages over the elaboration semantics, though the latter is usually chosen in a practical sense.

- Elaboration semantics is usually given indirectly and complicates the reasoning since we need to study both the semantics of source and target languages. Furthermore, the metatheory of coherence is quite challenging since the equivalence needs to be established.

- The standard determinism and subject-reduction can be proved using direct semantics, while those cannot be proven in elaboration semantics.

One recent work of type-dependent semantics is typed-directed operational semantics (TDOS) [Huang and Oliveira 2020]. It is firstly introduced for calculi with intersection types and a merge operator. It enables a natural proof of determinism and subjection-reduction of the calculi.

The main idea of TDOS is the utilisation of type annotations. Type annotations are discarded in many conventional languages with operational semantics. It instead will be used in two places in TDOS:

- *Trigger the Casting:* Annotating values will force them to be cast into the corresponding items according to the type provided.
- *Typed Reduction:* The casting judgment $v \hookrightarrow_A v'$ (originally called typed reduction) accept one input value $v$ and one input type $A$, then cast $v$ into another value $v'$ according to the type $A$. For example, `1,,true : Int` will be evaluated to `1`.

TDOS has also been studied to support gradual typing and assist its runtime checking [Ye et al. 2021].

# 3 Applicative Subtyping

In this section, we start with an introduction to classical subtyping algorithms for intersection types, discuss some design options to deal with the inference of applications, and then present applicative subtyping. Applicative subtyping is able to compute the output type of applications in the presence of overloading and nested composition. Then we make an analogy of record projections. Thus, in the context of calculi with intersection types, applicative subtyping provides a unified solution to infer types for applications and record projections.

Two versions of subtyping of intersection types are presented: one is the subtyping without distributivity and one is the BCD subtyping with distributivity included. Then we interpret each rule in terms of overloading and projection and describe several challenges of dealing with the inference of the application types and projection types. In the following, we develop our two main methods: application mode and applicative subtyping. Application mode can be treated as a stack-based resolution of applicative subtyping and corresponds to the subtyping without distributivity. Instead, applicative subtyping instantly computes the application types without the necessity of collecting all the argument types and corresponds to the distributive subtyping. We examine the metatheory of applicative subtyping in the last.

## 3.1 Types

We start with the syntax of types:

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

$A$ 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 $A$ & $B$. Ordinary types [Davies and Pfenning 2000; Huang et al. 2021] are essentially types without intersection types, except for functions where intersection types can appear in argument types. As we will see later, ordinary types are helpful in the definition of algorithmic subtyping.

## 3.2 Subtyping and 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. We present two versions of subtyping and then derive our approach, respectively.

### 3.2.1 Subtyping

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

$$\textsc{S-Int}\ \frac{}{\mathsf{Int} \leq \mathsf{Int}} \qquad \textsc{S-Top}\ \frac{}{A \leq \mathsf{Top}} \qquad \textsc{S-Arr}\ \frac{C \leq A \qquad B \leq D}{A \to B \leq C \to D} \qquad \textsc{S-And}\ \frac{A \leq B_1 \qquad A \leq B_2}{A \leq B_1 \,\&\, B_2}$$

$$\textsc{S-And-L}\ \frac{A \leq C}{A \,\&\, B \leq C} \qquad \textsc{S-And-R}\ \frac{B \leq C}{A \,\&\, B \leq C}$$

Figure 3.1: Subtyping

In Figure 3.1, we present the rules for subtyping. Rules S-Int and S-Top describes base types `Int` and `Top`, `Int` is the subtype of itself and `Top` is the supertype of *any* type. For functions, rule S-Arr says that $A \to B$ is the subtype of $C \to D$ if $A$ is the supertype of $C$ and $B$ is the subtype of $D$. On the other hand, the subtyping relation of their input types is contravariant and the subtyping relation of their output types is covariant.

Rules S-And, S-And-L, and S-And-R are used for intersections, type $A$ is the subtype of the intersection type $B_1 \,\&\, B_2$ if it is the subtype of both branches $B_1$ and $B_2$. The intersection type $A \,\&\, B$ is the subtype of $C$ if there exists one branch that is the subtype of $C$.

Remark We can prove that the subtyping relation is reflexive and transitive. Thus the following two rules are derivable from the rules in Figure 3.1.

$$\textsc{S-Refl}\ \frac{}{A \leq A} \qquad \textsc{S-Trans}\ \frac{A \leq B \qquad B \leq C}{A \leq C}$$

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

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

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

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

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

Figure 3.2: Splittable Types and BCD Subtyping

### 3.2.2 BCD Subtyping

One well-known subtyping relation with such a distributivity rule is BCD subtyping [Barendregt et al. 1983].

$$\textsc{S-Distr}\ \frac{}{(A \to B) \,\&\, (A \to C) \leq A \to (B \,\&\, C)}$$

Rule S-Distr says intersection types can distribute over function types. Huang et al. [2021] provide a sound and complete algorithm for BCD subtyping by eliminating the transitivity rule and employing the notions of ordinary types and splittable types. Splittable types describe that types can be split into two simpler types and ordinary types are those which cannot be split.

We present splittable types and the subtyping relation in Figure 3.2 (To distinguish between the above subtyping relation, we use $<:$ instead of $\leq$ here). Rule Sub-And is the most interesting rule as it captures the distributivity of intersection types over function 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 $B_2$, and says $A$ is the subtype of $B$ if $A$ are subtypes of its splits $A_1$ and $A_2$. Rules Sub-And-L and Sub-And-R describe that intersection type $A \,\&\, B$ is the subtype of ordinary type $C$ if one of them is the subtype of it.

Metatheory The algorithmic BCD subtyping has the following interesting properties:

**Lemma 3.1** (Generalisation of subtyping).

- *If* $B <: D$ *and* $C <: A$*, then* $A \to B <: C \to D$*.*
- *If* $A <: C$*, then* $A \,\&\, B <: C$*.*
- *If* $B <: C$ *, then* $A \,\&\, B <: C$*.*

**Lemma 3.2** (Reflexivity of BCD Subtyping). $A <: A$*.*

**Lemma 3.3** (Transitivity of BCD Subtyping). *If* $A <: B$ *and* $B <: C$*, then* $A <: C$*.*

## 3.3 Intersection Subtyping and Overloading

In this section, we will study the connection between intersection subtyping and function overloading in detail. Since the two variants of subtyping are shown above, we provide two interpretations of overloading. One is the overloading resolution with all the arguments given, which fits into our familiar knowledge of overloading. Another interpretation is rather simple and novel: curried overloading.

### 3.3.1 Interpretation of Subtyping Rules

Intersection types are naturally useful for representing types of overloaded functions. The type of overloaded functions is the *and (&)* combination of types from each instance. For example, if `show` has two instances: `showInt` and `showBool`. Then the type of `show` is `(Int → String) & (Bool → String)`.

As we mentioned above, subtyping describes the relation between types. The subtyping of intersection types is useful for telling us information about overloading. Note that in the relation $A \leq B$, the left side $A$ plays the role of (overloaded) functions, and the right side $B$ plays the role of arguments and results (take it for granted for now).

- Rule S-Arr tells that for a function having the type $A \to B$, with an argument type $C$ and a result type $D$ given, we check their correctness via the subtyping relations.
- Rules S-And-L and S-And-R tells that $C$ can be derived out if some of $A \,\&\, B$ can derive out the $C$. This is also helpful for telling us whether one branch belongs to the overloading instances. For example, we can know that `showInt` is one branch of `show` with the subtyping statement: `(Int → String) & (Bool → String) <: Int → String`.

- Rule S-Distr is the interesting rule. After this rule is added, we can know $(A \to B) \,\&\, (A \to C)$ is isomorphic with $A \to (B \,\&\, C)$. Reflecting in values (terms), $\lambda x.\ e_1 ,, \lambda x.\ e_2$ should be "isomorphic" with $\lambda x.\ (e_1 ,, e_2)$ in some sense. This will introduce the new notion, we name it *first-class curried overloaded function*.

### 3.3.2 Intersection Subtyping, Partially

Previously we assumed that the subtyping relation builds on the fact that the result type for applications is known. However, this is not the case in overloaded applications where only the argument types are known. Basically, we try to solve the problem in the following form, where we infer the type `?` given the argument type `Int`:

$$(\texttt{Int} \to \texttt{String}) \,\&\, (\texttt{Bool} \to \texttt{String}) <: \texttt{Int} \to ?$$

We then introduce the relation $A <: P \rightsquigarrow O$ where $P$ stands for *partial types* and $O$ stands for *output types*. The original subtyping can be treated as a special case when $A <: B \rightsquigarrow \emptyset$. The formal definition of $P$ and $O$ is shown as follows,

| | |
|---|---|
| Partial Types | $P ::= A \mid A_{\to}$ |
| Output Types | $O ::= A \mid \emptyset$ |

Then, the corresponding rules are specially adapted for partial types,

$$\textsc{S-Arr-Par}\ \frac{C <: A}{A \to B <: C_{\to} \rightsquigarrow B}$$

$$\textsc{S-And-L-Par}\ \frac{A <: C_{\to} \rightsquigarrow D}{A \,\&\, B <: C_{\to} \rightsquigarrow D} \qquad \textsc{S-And-R-Par}\ \frac{B <: C_{\to} \rightsquigarrow D}{A \,\&\, B <: C_{\to} \rightsquigarrow D}$$

With new rules, we can conclude some interesting cases. For example, the result type of `showInt,,showBool 1` is `String`.

$$\dfrac{\dfrac{Int <: Int}{Int \to String <: Int_{\to} \rightsquigarrow String}\ \textsc{S-Arr-Par}}{(Int \to String) \,\&\, (Bool \to String) <: Int_{\to} \rightsquigarrow String}\ \textsc{S-And-L-Par}$$

The challenging part is how to deal with the case `f,,g 1` where `f` has the type `Int → Int → Int` and `g` has the type `Int → Bool → Bool`[1]. According to our rules shown, there are two possible results: `Int → Int` and `Bool → Bool`. Both are not ideal. We then can have two choices here:

- *Reject* because it cannot make a choice. Then dealing with the case `f,,g 1 true`, where `g` can be selected according to the two arguments `1` and `true`, is still a problem.

- *Accept* and derive the result `(Int → Int) & (Bool → Bool)`, which corresponds to the distributivity rule in subtyping.

We will develop two choices in section 3.5 and 3.6.

## 3.4 Challenges of Application Typing

In a traditional type system with bidirectional typing, we expect that our typing rule for applications is in the following form.

$$\frac{\Gamma \vdash e_1 \Rightarrow A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\; e_2 \Rightarrow B}\ \textsc{T-App}$$

Rule T-App does two things. The first is to check whether $e_1\; e_2$ is well-typed, which requires $e_1$ to infer the arrow type, and $e_2$ can be checked by the input from the arrow type. Second, it infers the type for the application term (e.g., $B$ in this case).

The story becomes different when we introduce intersection types, which may cause the type of $e_1$ to be an intersection type, which is

$$\frac{\Gamma \vdash e_1 \Rightarrow A \,\&\, B \qquad \Gamma \vdash e_2 \Leftarrow ?}{\Gamma \vdash e_1\; e_2 \Rightarrow ?}\ \textsc{T-App}$$

In this situation, we cannot simply extract the input type from $A \,\&\, B$ to check $e_2$, and cannot derive the result type for the application term.

[1]Merge operator has higher precedence than application.

## 3.5 Application Mode

To address this problem, we borrow the idea of an *application mode* [Xie and Oliveira 2018]. At first, this idea looks promising, but a more careful look reveals some issues.

The Application Mode Xie and Oliveira [2018] suggested an alternative approach to bidirectional type checking that employs a generalisation of the inference mode. The key idea is to use the type from arguments to infer the function type. In their work, the inference focuses on the higher-rank polymorphic function. The application mode generalises the synthesis mode, and $\Gamma \vdash e \Rightarrow A$ is syntactic sugar when the argument stack $\Psi$ is empty. With the application mode the rule for applications is:

$$\frac{\Gamma \vdash e_2 \Rightarrow A \qquad \Gamma \mid \Psi, A \vdash e_1 \Rightarrow A \to B}{\Gamma \mid \Psi \vdash e_1\, e_2 \Rightarrow B}\ \text{T-App}$$

The motivating example for demonstrating the advantages of the application mode is to type check the term $(\lambda x.\, x)\ 1$. In a traditional system with bidirectional typing, $(\lambda x.\, x)$ requires an explicit annotation since the untyped lambda abstract should be in check mode. Under the application mode, we can give the typing derivations for this term.

$$\frac{. \vdash 1 \Rightarrow \texttt{Int} \qquad . \mid \texttt{Int} \vdash \lambda x.\, x \Rightarrow \texttt{Int} \to \texttt{Int}}{. \vdash (\lambda x.\, x)\ 1 \Rightarrow \texttt{Int}}\ \text{T-App}$$

Here we can make an analogy and use this technique to infer the function type with intersection types. Since intersection types $A\ \&\ B$ play the role of finite polymorphic type. For instance, `show 2` should select branch `showInt` and has the type `String`.

$$\frac{. \vdash 1 \Rightarrow \texttt{Int} \qquad . \mid \texttt{Int} \vdash \texttt{showInt,,showBool} \Rightarrow \texttt{Int} \to \texttt{String}}{. \vdash \texttt{showInt,,showBool}\ 1 \Rightarrow \texttt{String}}\ \text{T-App}$$

We present a design of a stack-based approach in Figure 3.3. Rule As-Refl is the reflexivity rule: type $A$ returns the type itself when the argument is empty. Rule As-Arr keeps consuming the arguments in the stack by eliminating the input types of arrow types. Rules As-And-L and As-And-R derive the result $D$ from either the left or right branches. Note that we have a special judgment $\Psi \vdash A <: B$ in the premise as negation, which is the binary version of $\Psi \vdash A <: B$ and the negation $\Psi \vdash A$ says any branches in $A$ cannot accept the arguments

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

$$\textsc{AS-Refl}\quad \frac{}{\cdot \vdash A <: A} \qquad \textsc{AS-Arr}\quad \frac{C <: A \qquad \Psi \vdash B <: D}{\Psi, C \vdash A \to B <: C \to D} \qquad \textsc{AS-And-L}\quad \frac{\Psi, C \vdash A <: D \qquad \neg\, \Psi, C \vdash B}{\Psi, C \vdash A \,\&\, B <: D}$$

$$\textsc{AS-And-R}\quad \frac{\Psi, C \vdash B <: D \qquad \neg\, \Psi, C \vdash A}{\Psi, C \vdash A \,\&\, B <: D}$$

Figure 3.3: Applicative Subtyping (Stack)

in the stack. By this design we can accept `f,,g 1 true` and reject `f,,g 1`. We show the example below.

$$\dfrac{\dfrac{\texttt{Int} <: \texttt{Int} \qquad \dfrac{\texttt{Bool} <: \texttt{Bool} \qquad \dfrac{}{\cdot \vdash \texttt{Bool} <: \texttt{Bool}}\,\textsc{AS-Refl}}{\texttt{Bool} \vdash \texttt{Bool} \to \texttt{Bool} <: \texttt{Bool}}\,\textsc{AS-Arr}}{\texttt{Bool}, \texttt{Int} \vdash (\texttt{Int} \to \texttt{Bool} \to \texttt{Bool}) <: \texttt{Bool}}\,\textsc{AS-Arr}}{\texttt{Bool}, \texttt{Int} \vdash (\texttt{Int} \to \texttt{Int} \to \texttt{Int}) \,\&\, (\texttt{Int} \to \texttt{Bool} \to \texttt{Bool}) <: \texttt{Bool}}\,\textsc{AS-And-R}$$

The accompanying typing for the application mode is shown in Figure 3.4. Rule Ty-Var resolves the stack by computing the result for variables. Rule Ty-Lam is the standard rule for lambda abstractions. Rule Ty-Lam-S resolves the stack by checking the types and types in the stack. Rule Ty-App is the only rule to collect the types of arguments and push them to the stack $\Psi$. Rule Ty-Ann resolves the stack and derives the result. Rule Ty-Mrg is the standard rule. Rule Ty-Mrg infers the type which may come from either left or right branches according to the stack given.

Challenges in Dynamic Semantics The reason that the application mode has not been adopted in the calculi is that it is challenging to design the corresponding dynamic semantics. In the statics, the idea is to collect all the types of arguments and then make the selection, from outside to inside. Suppose $f$ and $g$ are functions with the following types.

$$\texttt{f} : \texttt{Int} \to \texttt{Int} \to \texttt{Int} \qquad \texttt{g} : \texttt{Int} \to \texttt{Bool} \to \texttt{Bool}$$

To infer the type `f,,g 1 true`, we will do three steps:

- First, we encounter `true` and push its type `Bool` into the stack.

$\boxed{\Gamma \mid \Psi \vdash e \Leftrightarrow A}$ *(Applicative Typing)*

$$\textsc{Ty-Lit}\quad \frac{}{\Gamma \mid \cdot \vdash i \Rightarrow \mathsf{Int}} \qquad \textsc{Ty-Var}\quad \frac{x : A \in T \qquad \Psi \vdash A <: B}{\Gamma \mid \Psi \vdash x \Rightarrow A} \qquad \textsc{Ty-Lam}\quad \frac{\Gamma, x : A \mid \cdot \vdash e \Rightarrow B}{\Gamma \mid \cdot \vdash \lambda x.\, e : A \to B \Rightarrow A \to B}$$

$$\textsc{Ty-Lam-S}\quad \frac{\Gamma, x : A \mid \cdot \vdash e \Rightarrow B \qquad \Psi, C \vdash A \to B <: D}{\Gamma \mid \Psi, C \vdash \lambda x.\, e : A \to B \Rightarrow D}$$

$$\textsc{Ty-App}\quad \frac{\Gamma \mid \cdot \vdash e_2 \Rightarrow A \qquad \Gamma \mid \Psi, A \vdash e_1 \Rightarrow A \to B}{\Gamma \mid \Psi \vdash e_1\, e_2 \Rightarrow B}$$

$$\textsc{Ty-Ann}\quad \frac{\Gamma \mid \cdot \vdash e \Rightarrow C \qquad C <: A \qquad \Psi \vdash A <: B}{\Gamma \mid \Psi \vdash e : A \Rightarrow B} \qquad \textsc{Ty-Mrg}\quad \frac{\Gamma \mid \cdot \vdash e_1 \Rightarrow A \qquad \Gamma \mid \cdot \vdash e_2 \Rightarrow B}{\Gamma \mid \cdot \vdash e1, , e2 \Rightarrow A \,\&\, B}$$

$$\textsc{Ty-Mrg-S-L}\quad \frac{\Gamma \mid \Psi, A \vdash e_1 \Rightarrow C \qquad \Gamma \mid \cdot \vdash e_2 \Rightarrow B}{\Gamma \mid \Psi, A \vdash e1, , e2 \Rightarrow C} \qquad \textsc{Ty-Mrg-S-R}\quad \frac{\Gamma \mid \Psi, A \vdash e_2 \Rightarrow C \qquad \Gamma \mid \cdot \vdash e_1 \Rightarrow B}{\Gamma \mid \Psi, A \vdash e1, , e2 \Rightarrow C}$$

Figure 3.4: Applicative Typing

- Next, we analyse 1 and push its type `Int` into the stack.
- Last, using the contents of the stack, we infer the type of `f,,g` is `Int` $\to$ `Bool` $\to$ `Bool`.

The order of type analysis is reversed from the operational semantics based on the call-by-value strategy. In dynamic semantics of the $\lambda$-calculus, `f,, g 1` should be evaluated first to a value and we know nothing about the second argument.

## 3.6 Applicative Subtyping

Applicative subtyping is another approach to infer the types of applications and projections, which is different from application mode in three ways.

- Applicative subtyping corresponds to the subtyping relation with distributivity, while the application mode does not support distributivity.
- Applicative subtyping resolves the argument instantly and one-by-one, while the latter resolves the argument stack after collecting all the arguments.

$$A_1 \to A_2 \ll B = A_2 \qquad \textit{when } B <: A_1 \tag{3.1}$$
$$A_1 \to A_2 \ll B = . \qquad \textit{when } \neg(B <: A_1) \tag{3.2}$$
$$A_1 \,\&\, A_2 \ll S = (A_1 \ll S) \circledcirc (A_2 \ll S) \tag{3.3}$$
$$A \ll S = . \qquad \textit{otherwise} \tag{3.4}$$

Figure 3.5: Applicative Subtyping

- The design of operational semantics of applicative subtyping is relatively easier while the latter is challenging.

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. The typing still adopts bi-directional typing, and unlike the traditional rules for applications and projections, we also infer the argument types to help the inference of application type. As we can see below, applicative subtyping is used in the rule T-App and help compute the result of applications.

$$\textsc{T-App}\quad \frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B \qquad A \ll B = C}{\Gamma \vdash e_1\, e_2 \Rightarrow C}$$

We model the types of arguments as selector types, and the outputs as being either a type or nothing (denoting the failure to find a suitable output type).

| | |
|---|---|
| Selector Types | $S ::= A$ |
| Outputs | $O ::= . \mid A$ |

The definition of applicative subtyping is given in Figure 3.5. It has the form $A \ll S = C$ and it reads as the type $A$ can be selected by the selector type $S$ and derive the result type $C$.

Selector types are used as the second parameter and propagate through the subtyping checks, until we reach arrow or record types. For arrow types, in rules (3.1) (3.2), 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 the case of the intersection types $A_1 \,\&\, A_2$ (3.3), we introduce a composition operator $\circledcirc$ to combine two results which are derived from applying $A_1$ and $A_2$ with the same selector type $B$. Rule (3.4) covers a number of missing cases (such as `Int` $\ll S$) which will all fail. For simplicity of presentation, we write those rules as a single rule (3.4).

The composition operator $\circledcirc$ accepts output results and returns a new output result. The isolation of the composition operator gives us the ability to model different forms of applicative subtyping. We present two design options below.

Nested Composition Semantics Implementation For systems with BCD subtyping, which includes distributivity rules, we use the combinator implementing the nested composition semantics.

$$A_1 \circledcirc A_2 = A_1 \,\&\, A_2 \tag{3.5}$$
$$A_1 \circledcirc . \;\;\; = A_1 \tag{3.6}$$
$$. \circledcirc A_2 = A_2 \tag{3.7}$$
$$. \circledcirc . \;\;\; = . \tag{3.8}$$

If two outputs are both types, we return their intersection types $A_1 \,\&\, A_2$ in (3.5). This rule models the behaviour of nested composition, combining two results. In (3.6) and (3.7), if we have at least one type, we just return that type. Finally, (3.8) means if neither output is a type then we have to return nothing. Intuitively, we apply the arguments to different branches of the overloaded function, if fail we ignore the result and if successful we will take the results out and combine them using the & type constructor.

Overloading Semantics Implementation It is a rather restricted form of applicative subtyping. We name it the overloading semantics implementation since if multiple implementations in an overloaded definition match with an argument, 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.

$$A_1 \circledcirc A_2 = Amb \tag{3.9}$$
$$A_1 \circledcirc . \;\;\; = A_1 \tag{3.10}$$
$$. \circledcirc A_2 = A_2 \tag{3.11}$$
$$. \circledcirc . \;\;\; = . \tag{3.12}$$

$$\begin{aligned}
&(Int \to String)\ \&\ (Bool \to String) \ll Int\\
\textit{by } (3.3) &\hookrightarrow (Int \to String) \ll Int \circledcirc (Bool \to String) \ll Int\\
\textit{by } (3.1)\ (3.2) &\hookrightarrow String \circledcirc .\\
\textit{by } (3.8) &\hookrightarrow String
\end{aligned}$$

$$\begin{aligned}
&(Int \to Int)\ \&\ (Int \to Int) \ll Int\\
\textit{by } (3.3) &\hookrightarrow (Int \to Int) \ll Int \circledcirc (Int \to Int) \ll Int\\
\textit{by } (3.1) &\hookrightarrow Int \circledcirc Int\\
\textit{by } (3.12) &\hookrightarrow Amb
\end{aligned}$$

Figure 3.6: Examples of Applicative Subtyping with different composition operators

The rule 3.12 says that if two outputs are both types, we return a type error `Amb` to denote the ambiguity. This rule models the behaviour of overloading ambiguity checking in programming languages.

Remark The approach of applicative subtyping with overloading semantics implementation is more rigid than the application mode version. This is because it will reject the unambiguous applications like `f,,g 1 true` since the first argument cannot make the selection.

Examples We give examples of our two design options in Figure 3.6. The first is the inference of application term `show 1` and the return type is `String`. The second is the inference of absurd case `succ,,pred 1`, and the result is a type error (`Amb`). The ideas are to recursively delegate the selector type to the left and right branches. Then for the first, it combines the left and right results. For the second, it outputs a `Amb` since both branches can accept the argument, which leads to ambiguity.

## 3.7 Record Projection

In this section, we employ applicative subtyping to record projection and show that applicative subtyping can also be used to compute the result type of projections.

Intersection types can naturally support records with a single field record type added. For records with multiple fields, we use the intersection type constructor to concatenate them. For instance, $\{l_1 : A, l_2 : B, l_3 : C\}$ is essentially $\{l_1 : A\}$ & $\{l_2 : B\}$ & $\{l_3 : C\}$.

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

Further we integrate the rule S-Rcd for records in the subtyping: $\{l : A\}$ is the subtype of $\{l : B\}$ if $A$ is the subtype of $B$. To let the intersection types also distribute over record types, we extend the splittable types with the record case.

$$\frac{A <: B}{\{l : A\} <: \{l : B\}}\ \textsc{S-Rcd} \qquad \frac{A_1 \lhd A \rhd A_2}{\{l : A_1\} \lhd \{l : A\} \rhd \{l : A_2\}}\ \textsc{Sp-Rcd}$$

In company with the rules for intersection types, encoded record types have the subtyping properties developed in classical record calculi [Cardelli and Mitchell 1991]: width subtyping, depth subtyping and permutation.

The essential operation for records is projection, given a label and record, returns the values associated with the label. This problem is the same with the overloaded application: given an argument and overloaded functions, the result after applying the correct branch should be returned. We can reuse our applicative subtyping by abstracting the label as selector types and extending the new rules. For record types (3.15) (3.16), we check the equality between labels. If the labels are equal, we return the output type $A$, otherwise we fail.

$$\text{Selector Types} \quad S ::= A \mid \boxed{l}$$

$$\begin{array}{rlr} A_1 \to A_2 \ll B = A_2 & \quad \mathit{when}\ B <: A_1 & (3.13) \\ A_1 \to A_2 \ll B = . & \quad \mathit{when}\ \neg(B <: A_1) & (3.14) \\ \{l = A\} \ll l \;= A & & (3.15) \\ \{l_1 = A\} \ll l_2 = . & \quad \mathit{when}\ l_1 \neq l_2 & (3.16) \\ A_1 \,\&\, A_2 \ll S \;= (A_1 \ll S) \circledcirc (A_2 \ll S) & & (3.17) \\ A \ll S \;= . & \quad \mathit{otherwise} & (3.18) \end{array}$$

Examples We illustrate the ideas by examples $\{x = 1, y = \texttt{true}\}.x$. The return type is `Int`. We first recursively call both branches with the selector type (label `x`) and use the composition operator to combine their results. The left compares the equality and returns its associated type `Int`; the right outputs a failure since label `y` is not equal to `x`.

$$
\begin{aligned}
& \{x : Int\} \mathbin{\&} \{y : Bool\} \ll x \\
\textit{by } (3.17) &\hookrightarrow \{x : Int\} \ll x \circledcirc \{y : Bool\} \ll x \\
\textit{by } (3.15)\ (3.16) &\hookrightarrow Int \circledcirc . \\
\textit{by } (3.10) &\hookrightarrow Int
\end{aligned}
$$

## 3.8 Metatheory

We proved the soundness and completeness of our applicative subtyping with respect to 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 3.4** (Soundness (Function)). *If $A \ll B = C$, then $A <: B \to C$.*

*Proof.* Induction on the $A \ll B$. □

**Lemma 3.5** (Completeness (Function)). *If $A <: B \to C$, then $\exists D, A \ll B = D \land D <: C$.*

*Proof.* Induction on the subtyping $A <: B \to C$. □

The soundness lemma is intuitive. If the result of checking applicative subtyping with a subtype $A$ and input type $B$ computes a type $C$ then it should be the case that $A <: B \to C$. For completeness, we wish to show that if $A$ is a subtype of a function type $B \to C$ 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 3.6** (Soundness (Record)). *If $A \ll l = B$, then $A <: \{l : B\}$.*

**Lemma 3.7** (Completeness (Record)). *If $A <: \{l : B\}$, then $\exists C, A \ll l = C \land C <: B$.*

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. 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 3.8** (Completeness). *If $A <: B \to C$, then $(\exists D, A \ll B = D \wedge D <: C) \vee A \ll B = Amb$.*

# 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) [Dunfield 2014]. Our calculus employs a type-directed operational semantics [Huang et al. 2021] instead of using elaboration semantics as proposed by Dunfield and adopts applicative subtyping and distributive subtyping. Overloaded functions, nested composition and multi-field records can all be encoded in this calculus. In contrast, in Dunfield's calculus distributivity and nested composition are not supported. Like Dunfield's calculus, the presence of unrestricted merges makes the semantics non-deterministic. We address the issue of non-determinism in the next section.

## 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. The lambda expression $\lambda x.\, e : A \to B$ is 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 $e.l$. Contexts are standard: $\Gamma$ is a list of bound variables $x$ and their types $A$.

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 seman-

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

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

$$\textsc{T-Rcd}\ \frac{\Gamma \vdash e \Rightarrow A}{\Gamma \vdash \{l = e\} \Rightarrow \{l : A\}} \qquad \textsc{T-App}\ \frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B \qquad A \ll B = C}{\Gamma \vdash e_1\, e_2 \Rightarrow C}$$

$$\textsc{T-Proj}\ \frac{\Gamma \vdash e \Rightarrow A \qquad A \ll l = B}{\Gamma \vdash e.l \Rightarrow B} \qquad \textsc{T-Mrg}\ \frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B}{\Gamma \vdash e_1 \,,, \, e_2 \Rightarrow A \,\&\, B} \qquad \textsc{T-Ann}\ \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash e : A \Rightarrow A}$$

$$\textsc{T-Sub}\ \frac{\Gamma \vdash e \Rightarrow A \qquad A <: B}{\Gamma \vdash e \Leftarrow B}$$

Figure 4.1: Bi-directional Typing.

tics. 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 distributivity.

## 4.2 Typing

Figure 4.1 shows our bi-directional type system. Most of the rules are adapted from traditional bi-directional typing [Dunfield and Krishnaswami 2021]. The novel rules are rules T-App and T-Proj, whose inferred type is derived from applicative subtyping.

### 4.2.1 Typing of Application and Projection

The common approach to typing applications is to infer the function type first, use the input type to check the arguments, and assign the output type to applications. Our approach to type applications [Xie and Oliveira 2018] is to infer 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. We infer the types of $e$ and get the label $l$, pass it to applicative subtyping and derive the result type for projections.

### 4.2.2 Examples

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

$$\dfrac{\dfrac{\Gamma \vdash \texttt{(f,,g)} \Rightarrow (I \to I \to I) \,\&\, (I \to B \to B) \qquad \Gamma \vdash 2 \Rightarrow I}{\Gamma \vdash \texttt{(f,,g)}\ \texttt{2} \Rightarrow \boxed{(I \to I) \,\&\, (B \to B)}}\ \textsc{T-App} \qquad \Gamma \vdash \texttt{true} \Rightarrow B}{\Gamma \vdash \texttt{(f,,g)}\ \texttt{2}\ \texttt{true} \Rightarrow \boxed{B}}\ \textsc{T-App}$$

To infer the type of `(f,,g) 2 true`, we first infer both the type of `(f,,g) 2` and `true`. The type of `(f,,g) 2` is `(Int → Int) & (Bool → Bool)`. This result is computed from applicative subtyping with two inputs: type of function merges `f,,g` and type of `2`. Later we use the computed result of `(f,,g) 2` to derive our final type `Bool`.

### 4.2.3 Properties of Typing

Since our typing rules adopt techniques of bidirectional typing, there are some properties:

**Lemma 4.1** (Uniqueness of Synthesis). *If* $\Gamma \vdash e \Rightarrow A$ *and* $\Gamma \vdash e \Rightarrow B$*, then* $A = B$.

**Lemma 4.2** (Checking Subsumption). *If* $\Gamma \vdash e \Leftarrow A$ *and* $A \leq B$*, then* $\Gamma \vdash e \Leftarrow B$.

## 4.3 Semantics

This calculus adopts a type-directed operational semantics [Huang et al. 2021], where type annotations are used to cast terms instead of being erased after type checking.

### 4.3.1 Casting

We introduce the casting judgment in Figure 4.2. Judgment $v \longmapsto_A v'$ describes that value $v$ is cast to another 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. [2021]. Rules Ct-Mrg-L and Ct-Mrg-R state that merges will be cast to one result by ordinary types. For example, `showInt,,showBool` will be cast to `showInt` by type `Int → String`.

**Lemma 4.3** (Preservation of Casting). *If* $\Gamma \vdash v \Rightarrow B$ *and* $v \longmapsto_A v'$*, then* $\Gamma \vdash v' \Leftarrow A$.

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

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

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

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

Figure 4.2: Casting

The preservation of castings states that the structure of the cast result $v'$ will match the type provided.

**Lemma 4.4** (Progress of Casting). *If* $\cdot \vdash v \Leftarrow A$*, then* $\exists v', v \longmapsto_A v'$.

The progress lemma of castings states that a well-typed value can be cast to another value. More specifically if this value can be checked by the type $A$, then it can be cast by the type $A$. For example `showInt,,showBool` can be checked by the type `Int` $\to$ `String`, then it can be cast by this type.

## 4.3.2 Applicative Dispatching

We introduce a new judgement called applicative dispatching (Figure 4.3), which extends Huang et al. [2021] *parallel application* judgement. The idea of the parallel application is simple: function merges will be parallel applied to arguments. In contrast to parallel application, we must also deal with overloading. Judgment $(v \bullet vl) \hookrightarrow e$ describes that value $v$ is applied to selectors $vl$ and then reduced to a term $e$.

$$\text{Selectors} \quad vl ::= v \mid l$$

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 or a label, to provide the types to be compared with

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

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

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

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

Figure 4.3: Applicative Dispatching

applicative subtyping. For label, $\langle l \rangle$ simply returns the original value, and for values, $\langle v \rangle$ extracts the annotation from terms. It can be defined as a structurally recursive function.

$$\begin{aligned}
\langle l \rangle &= l \\
\langle p : A^o \rangle &= A^o \\
\langle v_1 ,, v_2 \rangle &= \langle v_1 \rangle \,\&\, \langle v_2 \rangle \\
\langle \{l = v\} \rangle &= \{l : \langle v \rangle\}
\end{aligned}$$

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.

**Lemma 4.5** (Preservation of Applicative Dispatching (Function and Records)).

- *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* $\cdot \vdash e \Leftarrow A$.

The preservation lemma of applicative dispatching states that for a well-typed application or projection term, their types will be preserved in the evaluation of the applicative dispatching judgment.

**Lemma 4.6** (Progress of Applicative Dispatching (Function and Records)).

- *If* $\cdot \vdash v_1 \ v_2 \Rightarrow A$*, then* $\exists e, v_1 \bullet v_2 \hookrightarrow e$

$\boxed{e \longmapsto e'}$ *(Small-Step Reduction)*

$$\textsc{Step-Int-Ann}\ \frac{}{i \longmapsto i : \mathsf{Int}} \qquad \textsc{Step-Arr-Ann}\ \frac{}{\lambda x.\, e : A \to B \longmapsto (\lambda x.\, e : A \to B) : A \to B} \qquad \textsc{Step-App}\ \frac{(v_1 \bullet v_2) \hookrightarrow e}{v_1\, v_2 \longmapsto e}$$

$$\textsc{Step-Pv-Split}\ \frac{A_1 \lhd A \rhd A_2}{p : A \longmapsto p : A_1 ,, p : A_2} \qquad \textsc{Step-Prj}\ \frac{(v \bullet l) \hookrightarrow v'}{v.l \longmapsto v'} \qquad \textsc{Step-Ann}\ \frac{\neg e \in p \qquad e \longmapsto e'}{e : A \longmapsto e' : A}$$

$$\textsc{Step-Val-Ann}\ \frac{v \longmapsto_A v'}{v : A \longmapsto v'} \qquad \textsc{Step-App-L}\ \frac{e_1 \longmapsto e_1'}{e_1\, e_2 \longmapsto e_1'\, e_2} \qquad \textsc{Step-App-R}\ \frac{e_2 \longmapsto e_2'}{v_1\, e_2 \longmapsto v_1\, e_2'} \qquad \textsc{Step-Mrg-L}\ \frac{e_1 \longmapsto e_1'}{e_1 ,, e_2 \longmapsto e_1' ,, e_2}$$

$$\textsc{Step-Mrg-R}\ \frac{e_2 \longmapsto e_2'}{v_1 ,, e_2 \longmapsto v_1 ,, e_2'} \qquad \textsc{Step-Rcd-R}\ \frac{e \longmapsto e'}{\{l = e\} \longmapsto \{l = e'\}} \qquad \textsc{Step-Prj-L}\ \frac{e \longmapsto e'}{e.l \longmapsto e'.l}$$

Figure 4.4: Operational Semantics

- *If* $\cdot \vdash v.l \Rightarrow A$, *then* $\exists e, v \bullet l \hookrightarrow e$.

The progress lemma of applicative dispatching states that well-typed application or projection terms are able to be evaluated by the applicative dispatching judgment.

### 4.3.3 Operational Semantics

We present our small-step reduction rules in Figure 4.4. 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 $A$. 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

We show the preservation lemmas of casting and applicative dispatching using the checking mode. That is to say, the types are not exactly preserved in the evaluation. Instead, we can only ensure that types can become smaller and is the subtype of original types. However, our

$\boxed{A \approx B}$ *(Isomorphic Subtyping)*

$$\textsc{ISub-Refl}\quad \frac{}{A \approx A} \qquad \textsc{ISub-Rcd}\quad \frac{A \approx B}{\{l : A\} \approx \{l : B\}} \qquad \textsc{ISub-And}\quad \frac{B_1 \lhd B \rhd B_2 \qquad A_1 \approx B_1 \qquad A_2 \approx B_2}{A_1 \,\&\, A_2 \approx B}$$

Figure 4.5: Isomorphic Subtyping

calculus is more strict than that, the isomorphic form of the types will be preserved in the reduction. We show the isomorphic subtyping in Figure 4.5 and prove a more strict version of preservation.

### 4.4.1 Isomorphic Subtyping

Isomorphic subtyping is a restricted relation to describe relations between types. We show the rules in Figure 4.5. Rule ISub-Refl describes that types are isomorphic with themselves. Rule ISub-Rcd gives a special treatment for record types: $\{l : A\} \approx \{l : B\}$ if $A \approx B$. The reason why we distinguish function types and record types here is that we have slightly different evaluation strategies for functions and records. For functions, we append an annotation to it and make a split if applicable. For records, we evaluate the inner expression into values. Rule ISub-And captures the essence of the isomorphic subtyping: the intersection types are isomorphic subtyping to their corresponding splittable types. For example, `(Int → String) & (Int → Bool)` is isomorphic subtyping to `Int → (String & Bool)`.

The isomorphic subtyping has the following properties:

**Lemma 4.7** (Transitivity of Isomorphic Subtyping). *If $A \approx B$ and $B \approx C$, then $A \approx C$.*

*Proof.* Induction on the size of the type $B$. □

**Lemma 4.8** (Soundness of Isomorphic Subtyping). *If $A \approx B$, then $A <: B$ and $B <: A$*

**Lemma 4.9** (Applicative Subtyping and Isomorphic Subtyping (Function and Records)).

- *If $A \ll B = C$, $A' \approx A$ and $B' \approx B$, then $\exists C', A' \ll B' = C' \wedge C' \approx C$.*
- *If $A \ll l = B$, $A' \approx A$, then $\exists B', A' \ll l = B' \wedge B' \approx B$.*

### 4.4.2 Type Soundness

Type soundness is proven via standard preservation and progress theorems.

**Theorem 4.10** (Preservation). *If* $\cdot \vdash e \Leftrightarrow A$ *and* $e \longmapsto e'$, *then* $\cdot \vdash e' \Rightarrow A' \wedge A' \approx A$.

**Theorem 4.11** (Progress). *If* $\cdot \vdash e \Leftrightarrow A$, *then* $e$ *is a value or* $\exists e', e \longmapsto e'$.

**Corollary 4.12** (Type Soundness). *If* $\cdot \vdash e \Leftrightarrow A$ *and* $e \longmapsto_* e'$, *then* $e'$ *will not get stuck.*

# 5 A Calculus with A Disjoint Merge Operator

This section presents a second calculus with a disjointness restriction on merges [Oliveira et al. 2016] to recover determinism. This calculus forbids some cases of conventional overloading but still supports the other features. We focus on the key differences from 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. [2016]. 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 that are supertypes of all types (e.g., `Top`, `Top & Top`) and defined in Figure 5.1.

The inclusion of such types into top-like types is also part of the classical BCD subtyping relation [Barendregt et al. 1983]. A formal specification of disjointness is given below. There is a sound and complete set of algorithmic disjointness rules that conform to this specification. We present it in Figure 5.2.

**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 and types such as `Int` $\rightarrow$ `Int` or `Int` $\rightarrow$ `Bool` are disjoint. To include function types into our disjointness, types like `Int` $\rightarrow$ `Top` should be top-like and supertypes of all types, since otherwise `Int` $\rightarrow$ `Int` and `Int` $\rightarrow$ `Bool` cannot be disjoint according to our definition. We follow previous work on disjoint intersection types [Bi et al. 2018] and generalize our subtyping rule for rule S-Top.

$$\text{S-Top}\quad \frac{\rceil B \lceil}{A <: B}$$

$\boxed{\rceil A \lceil}$ *(Top-like Types)*

$$\textsc{Tl-Top}\ \frac{}{\rceil \mathsf{Top} \lceil} \qquad \textsc{Tl-And}\ \frac{\rceil A \lceil \quad \rceil B \lceil}{\rceil A \,\&\, B \lceil} \qquad \textsc{Tl-Arr}\ \frac{\rceil B \lceil}{\rceil A \to B \lceil} \qquad \textsc{Tl-Rcd}\ \frac{\rceil A \lceil}{\rceil \{l : A\} \lceil}$$

Figure 5.1: Top-like types

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

$$\textsc{Dj-Top-L}\ \frac{}{\mathsf{Top} * A} \qquad \textsc{Dj-Top-R}\ \frac{}{A * \mathsf{Top}} \qquad \textsc{Dj-Int-Arr}\ \frac{}{\mathsf{Int} * A \to B} \qquad \textsc{Dj-Arr-Int}\ \frac{}{A \to B * \mathsf{Int}} \qquad \textsc{Dj-Arr-Arr}\ \frac{B * D}{A \to B * C \to D}$$

$$\textsc{Dj-Rcd-Eq}\ \frac{A * B}{\{l : A\} * \{l : B\}} \qquad \textsc{Dj-Rcd-Neq}\ \frac{l_1 \neq l_2}{\{l_1 : A\} * \{l_2 : B\}} \qquad \textsc{Dj-Int-Rcd}\ \frac{}{\mathsf{Int} * \{l : A\}} \qquad \textsc{Dj-Rcd-Int}\ \frac{}{\{l : A\} * \mathsf{Int}}$$

$$\textsc{Dj-Arr-Rcd}\ \frac{}{A \to B * \{l : C\}} \qquad \textsc{Dj-Rcd-Arr}\ \frac{}{\{l : A\} * B \to C} \qquad \textsc{Dj-And-L}\ \frac{A * C \quad B * C}{A \,\&\, B * C} \qquad \textsc{Dj-And-R}\ \frac{A * B \quad A * C}{A * B \,\&\, C}$$

Figure 5.2: Algorithmic Disjointness

Disjointness has important properties, which are helpful for the metatheory of calculus. In particular, if two types are disjoint, their applicative subtyping results under the same partial types are also disjoint.

**Lemma 5.1** (Covariance of Disjointness). *If $A * B$ and $B <: C$, then $A * C$.*

**Lemma 5.2** (Applicative Subtyping and Disjointness). *If $A * B$, $A \ll S = C_1$ and $B \ll S = C_2$, then $C_1 * C_2$.*

## 5.2 Disjointness and Applicative Subtyping

With the more general subtyping rule for top-like types, applicative subtyping remains sound (with Lemma 3.4, Lemma 3.6 in Chapter 3) with respect to subtyping. However, the completeness of our applicative subtyping needs to be slightly adapted.

**Lemma 5.3** (Completeness of Applicative Subtyping). *If $A <: B \to C$, then $(\exists D, A \ll B = D \wedge D <: C) \vee Top <: C$.*

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.

## 5.3 Typing

The main change in typing is that we add a disjoint premise in rule T-Mrg. We show the full typing rules in Figure 5.3. Specially rule T-Mrg-Val is used to prove the consistency of the evaluation and can be ignored in the implementation.

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

Remark Disjointness has its advantage over the coherence property, however, it rejects some useful functions such as `show`. The side condition $A * B$ forbids the term typed with `(Int → String) & (Bool → String)` since they are not disjoint according to the definition in Figure 5.2.

## 5.4 Dynamic Semantics

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. We show two judgments in Figure 5.4 and Figure 5.5.

Since our value defines as raw values carrying annotation, the special term for Top is unnecessary since we can use any constant terms (1 in our rules) with top-like types to replace them. Rule Ct-Top says that values will be cast to a constant value $1 : A^o$ (we demonstrate 1 here for simplicity) by an ordinary top-like type.

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

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

$$\textsc{T-Rcd}\quad \frac{\Gamma \vdash e \Rightarrow A}{\Gamma \vdash \{l = e\} \Rightarrow \{l : A\}} \qquad \textsc{T-App}\quad \frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B \qquad A \ll B = C}{\Gamma \vdash e_1\, e_2 \Rightarrow C}$$

$$\textsc{T-Proj}\quad \frac{\Gamma \vdash e \Rightarrow A \qquad A \ll l = B}{\Gamma \vdash e.l \Rightarrow B} \qquad \textsc{T-Mrg}\quad \frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B \qquad A * B}{\Gamma \vdash e_1 \,,,\, e_2 \Rightarrow A \,\&\, B}$$

$$\textsc{T-Mrg-Val}\quad \frac{\cdot \vdash u_1 \Rightarrow A \qquad \cdot \vdash u_2 \Rightarrow B \qquad u_1 \approx u_2}{\Gamma \vdash u_1 \,,,\, u_2 \Rightarrow A \,\&\, B} \qquad \textsc{T-Ann}\quad \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash e : A \Rightarrow A}$$

$$\textsc{T-Sub}\quad \frac{\Gamma \vdash e \Rightarrow A \qquad A <: B}{\Gamma \vdash e \Leftarrow B}$$

Figure 5.3: Bidirectional Typing (Disjointness)

## 5.5 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.

**Lemma 5.4** (Determinism of Casting). *If* $\cdot \vdash v \Rightarrow A$, $v \longmapsto_B v_1$ *and* $v \longmapsto_B v_2$, *then* $v_1 = v_2$.

**Lemma 5.5** (Determinism (Applications and Projections)).

- *If* $\cdot \vdash v_1\, v_2 \Rightarrow A$, $v_1 \bullet v_2 \hookrightarrow e_1$ *and* $v_1 \bullet v_2 \hookrightarrow e_2$, *then* $e_1 = e_2$.
- *If* $\cdot \vdash v.l \Rightarrow A$, $v \bullet l \hookrightarrow e_1$ *and* $v \bullet l \hookrightarrow e_2$, *then* $e_1 = e_2$.

**Theorem 5.6** (Determinism). *If* $\cdot \vdash e \Leftrightarrow A$, $e \longmapsto e_1$ *and* $e \longmapsto e_2$, *then* $e_1 = e_2$.

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

$$\textsc{Ct-Int}\ \frac{A <: \mathsf{Int}}{i : A \longmapsto_{\mathsf{Int}} i : \mathsf{Int}} \qquad \textsc{Ct-Top}\ \frac{\rceil A^O \lceil}{v \longmapsto_A 1 : A^O}$$

$$\textsc{Ct-Arr}\ \frac{\neg\rceil D^o \lceil \qquad E <: C \to D^o}{(\lambda x.\, e : A \to B) : E \longmapsto_{(C \to D^o)} (\lambda x.\, e : A \to D^o) : (C \to D^o)}$$

$$\textsc{Ct-Rcd}\ \frac{\neg\rceil A^O \lceil \qquad v \longmapsto_{A^O} v'}{\{l = v\} \longmapsto_{\{l : A^O\}} \{l = v'\}} \qquad \textsc{Ct-Mrg-L}\ \frac{v_1 \longmapsto_{A^O} v_1'}{v_1 \,,,\, v_2 \longmapsto_{A^O} v_1'} \qquad \textsc{Ct-Mrg-R}\ \frac{v_2 \longmapsto_{A^O} v_2'}{v_1 \,,,\, v_2 \longmapsto_{A^O} v_2'}$$

$$\textsc{Ct-And}\ \frac{A_1 \lhd A \rhd A_2 \qquad v \longmapsto_{A_1} v_1 \qquad v \longmapsto_{A_2} v_2}{v \longmapsto_A v_1 \,,,\, v_2}$$

Figure 5.4: Casting (Disjointness)

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

$$\textsc{App-TopLike}\ \frac{\rceil A \lceil}{((p : A) \bullet v) \hookrightarrow 1 : \mathit{tail}\, A} \qquad \textsc{App-Proj}\ \frac{}{(\{l = v\} \bullet l) \hookrightarrow v}$$

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

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

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

Figure 5.5: Applicative Dispatching (Disjointness)

$\boxed{e \longmapsto e'}$ *(Small-Step Reduction)*

$$\textsc{Step-Int-Ann}\quad \frac{}{i \longmapsto i : \mathsf{Int}} \qquad \textsc{Step-Arr-Ann}\quad \frac{}{\lambda x.\, e : A \to B \longmapsto (\lambda x.\, e : A \to B) : A \to B}$$

$$\textsc{Step-Pv-Split}\quad \frac{A_1 \lhd A \rhd A_2}{p : A \longmapsto p : A_1 \,,,\, p : A_2} \qquad \textsc{Step-Papp}\quad \frac{(v_1 \bullet v_2) \hookrightarrow e}{v_1\, v_2 \longmapsto e}$$

$$\textsc{Step-Pproj}\quad \frac{(v \bullet l) \hookrightarrow v'}{v.l \longmapsto v'} \qquad \textsc{Step-Val-Ann}\quad \frac{v \longmapsto_A v'}{v : A \longmapsto v'}$$

$$\textsc{Step-App-L}\quad \frac{e_1 \longmapsto e_1'}{e_1\, e_2 \longmapsto e_1'\, e_2} \qquad \textsc{Step-App-R}\quad \frac{e_2 \longmapsto e_2'}{v_1\, e_2 \longmapsto v_1\, e_2'}$$

$$\textsc{Step-Mrg-Parallel}\quad \frac{e_1 \longmapsto e_1' \qquad e_2 \longmapsto e_2'}{e_1 \,,,\, e_2 \longmapsto e_1' \,,,\, e_2'}$$

$$\textsc{Step-Mrg-L}\quad \frac{e_1 \longmapsto e_1'}{e_1 \,,,\, v_2 \longmapsto e_1' \,,,\, v_2} \qquad \textsc{Step-Mrg-R}\quad \frac{e_2 \longmapsto e_2'}{v_1 \,,,\, e_2 \longmapsto v_1 \,,,\, e_2'} \qquad \textsc{Step-Ann}\quad \frac{\neg e \in p \qquad e \longmapsto e'}{e : A \longmapsto e' : A}$$

$$\textsc{Step-Rcd}\quad \frac{e \longmapsto e'}{\{l = e\} \longmapsto \{l = e'\}} \qquad \textsc{Step-Prj}\quad \frac{e \longmapsto e'}{e.l \longmapsto e'.l}$$

Figure 5.6: Operational Semantics (Disjointness)

# 6 Implementation

This chapter gives an interpreter implementation of our calculus in Chapter 4. Unlike traditional approaches of using datatypes to represent the types, expressions etc., we take a more direct and relatively simple representation: symbols. All the algorithms are computed via symbolic pattern matching and dispatching. This approach saves us from writing a parser of our languages since we adopt S-expressions [McCarthy 1960] and treat this implementation as a dialect of Lisp, though statically typed.

## 6.1 Statics

We start with the definition of types and terms and then introduce the subtyping algorithm. In the last part, we implement type inference and type checking.

### 6.1.1 Types and Terms

First, we define our syntax as several contracts (aka. predicates). For `type?`, we have basic types (integer, floating number, boolean, top) and compound types (function types, intersection types and record types). For simplicity, the label is just interpreted as a number.

```
(define label? number?)
(define (type? t)
  (match t
    ['int                                                   #t]
    ['float                                                 #t]
    ['bool                                                  #t]
    ['top                                                   #t]
    [`(→ ,(? type?) ,(? type?))                             #t]
    [`(& ,(? type?) ,(? type?))                             #t]
    [`(* ,(? label?) ,(? type?))                            #t]
    [_                                                      #f]))
```

For terms, we have variables, numbers, booleans, lambda abstractions, function applications, merges (e.g., `(m e1 e2)`), annotated terms, records (e.g., `(~> l e)`) and record projections (e.g., `(<~ e l)`) and primitive additions (e.g., `int+, flo+`).

```
(define (expr? e)
  (match e
    [(? symbol?)                                                    #t]
    [(? exact-integer?)                                             #t]
    [(? flonum?)                                                    #t]
    ['#t                                                            #t]
    ['#f                                                            #t]
    [`(λ (,(? symbol?) : ,(? type?)) ,(? expr?) ,(? type?))         #t]
    [`(,(? expr?) ,(? expr?))                                       #t]
    [`(m ,(? expr?) ,(? expr?))                                     #t]
    [`(: ,(? expr?) ,(? type?))                                     #t]
    [`(~> ,(? label?) ,(? expr?))                                   #t]
    [`(<~ ,(? expr?) ,(? label?))                                   #t]
    [`(int+ ,(? expr?) ,(? expr?))                                  #t]
    [`(flo+ ,(? expr?) ,(? expr?))                                  #t]
    [_                                                              #f]))
```

### 6.1.2 Subtyping

After the syntax is settled, we now define our subtyping and applicative subtyping algorithms. We first define the ordinary judgment as a contract and let splittable types be a function that accepts a type and returns a pair of types. The contract of the split function describes that it accepts a non-ordinary type and returns two types.

```
(define/contract (ord? t)
  (→ type? boolean?)
  (match t
    ['int                                                           #t]
    ['float                                                         #t]
    ['bool                                                          #t]
    ['top                                                           #t]
    [`(→ ,_ ,B)                                                (ord? B)]
    [`(* ,_ ,A)                                                (ord? A)]
    [_                                                              #f]))

(define/contract (spl t)
  (→ (and/c type? (not/c ord?)) (cons/c type? type?))
  (match t
    [`(→ ,A ,B)              (let ([Bs (spl B)])
                              `((→ ,A ,(car Bs)) . (→ ,A ,(cdr Bs))))]
    [`(* ,l ,A)              (let ([As (spl A)])
                                `((* ,l ,(car As)) . (* ,l ,(cdr As))))]
    [`(& ,A ,B)                                               `(,A . ,B)]
    [_                                       (error "fail to split" t)]))
```

The subtyping relation is modelled as a function that takes two types as inputs and returns a boolean specifying if two types have the subtyping relation. The first four branches deal with basic types and top types. For compound types, we check whether they are ordinary types and then dispatch them into different options. Simply speaking, we keep splitting types on the right side until they are ordinary; then we apply rules for arrows and records to them.

```
(define/contract (sub? t1 t2)
  (→ type? type? boolean?)
  (match* (t1 t2)
    [('int 'int)                                                     #t]
    [('float 'float)                                                 #t]
    [('bool 'bool)                                                   #t]
    [(_ 'top)                                                        #t]
    [(`(→ ,A ,B) `(→ ,C ,(? ord? D)))      (and (sub? C A) (sub? B D))]
    [(`(* ,l ,A) `(* ,l ,(? ord? B)))                       (sub? A B)]
    [(A (? (not/c ord?) B)) (let ([Bs (spl B)])
                           (and (sub? A (car Bs)) (sub? A (cdr Bs))))]
    [(`(& ,A ,B) (? ord? C))                (or (sub? A C) (sub? B C))]
    [(_ _)                                                           #f]))
```

## 6.1.3 Applicative Subtyipng

Our selector types are types or labels, and we define them by combining contracts `types?` and `label?`. For output types to denote the success or failure, we combine `type?` and `fail?`. There is only one inhabitant (`#f`) for the contract `fail?`.

The composition operator is simply implemented as a function that takes two outputs and returns an output type. `asub` is the function for applicative subtyping; it takes a regular type and a selector type, then returns the result type if success; otherwise it returns `#f` as a failure.

```
(define selector? (or/c type? label?))
(define (fail? t) (equal? t #f))
(define output? (or/c type? fail?))

(define/contract (comb o1 o2)
  (→ output? output? output?)
  (match* (o1 o2)
    [((? type? A1) (? type? A2))                           `(& ,A1 ,A2)]
    [((? fail?) (? type? A))                                              A]
    [((? type? A) (? fail?))                                              A]
    [((? fail?) (? fail?))                                              #f]))

(define/contract (asub t s)
  (→ type? selector? output?)
  (match* (t s)
    [(`(→ ,A1 ,A2) (? type? B)) #:when (sub? B A1)                      A2]
    [(`(→ ,A1 ,A2) (? type? B))                                         #f]
    [(`(* ,l1 ,A) (? label? l2)) #:when (equal? l1 l2)                    A]
    [(`(* ,l1 ,A) (? label? l2))                                        #f]
    [(`(& ,A1 ,A2) S)                       (comb (asub A1 S) (asub A2 S))]
    [(_ _)                                                              #f]))
```

## 6.1.4 Typing

The bidirectional typing is implemented as two functions: `infer` and `check`. The function `infer` takes a term and an environment as inputs and returns a type as an output. That is to say, the type of it is `(→ expr? list? type?)`. The function `check` takes three arguments: a term, a variable environment and a type that needs to be checked. It returns a boolean value specifying whether the check success.

```
(define/contract (check e t env)
  (→ expr? type? list? boolean?)
  (let ([A (infer e env)]) (sub? A t)))
(define/contract (infer e env)
  (→ expr? list? type?)
  (match e
    [(? exact-integer?)                                                'int]
    [(? flonum?)                                                     'float]
    ['#t                                                              'bool]
    ['#f                                                              'bool]
    [(? symbol?)                                            (lookup env e)]
    [`(λ (,x : ,A) ,e ,B) #:when (check e B (cons `(,x ,A) env))
                                                                  `(→ ,A ,B)]
    [`(~> ,l ,e)                                      `(* ,l ,(infer e env))]
    [`(: ,e ,A) #:when (check e A env)                                     A]
    [`(,e1 ,e2)            (let ([A (infer e1 env)] [B (infer e2 env)])
                                        (or (asub A B) (error "app")))]
    [`(<~ ,e ,l)           (let ([A (infer e env)])
                                       (or (asub A l) (error "proj")))]
    [`(m ,e1 ,e2)          (let ([A (infer e1 env)] [B (infer e2 env)])
                                                         `(& ,A ,B))]
    [`(int+ ,e1 ,e2) #:when (and (check e1 'int env)
                                 (check e2 'int env))              'int]
    [`(flo+ ,e1 ,e2) #:when (and (check e1 'float env)
                                 (check e2 'float env))          'float]
    [_                 (error "cannot infer the type of" e "under" env)]))
```

## 6.2 Dynamics

For the dynamic semantics, we first define two contracts: raw value and value. Then we specify that a casting function accepts a value and a type, it returns the result value if successful, otherwise it fails. The function `disp` implements our applicative dispatching; it accepts a value and a selector and returns an expression.

```
(define/contract (pvalue? e) ...)
(define/contract (value? e) ...)
(define/contract (cast e t)
  (→ value? type? (or/c value? fail?))
  (match* (e t)
    [(`(: ,n ,A) 'int) #:when (sub? A 'int)                  `(: ,n int)]
    [(`(: ,n ,A) 'float) #:when (sub? A 'float)            `(: ,n float)]
    [(`(: #t ,A) 'bool) #:when (sub? A 'bool)               '(: #t bool)]
    [(`(: #f ,A) 'bool) #:when (sub? A 'bool)               '(: #f bool)]
    [(v  'top)                                                '(: 1 top)]
    [(`(: (λ (,x : ,A) ,e ,B) ,E) `(→ ,C ,(? ord? D)))
     #:when (sub? E `(→ ,C ,D))     `(: (λ (,x : ,A) ,e ,D) (→ ,C ,D))]
    [(`(~> ,l ,v) `(* ,l ,(? ord? A)))              `(~> ,l ,(cast v A))]
    [(`(m ,v1 ,v2) (? ord? A)) #:when (cast v1 A)            (cast v1 A)]
    [(`(m ,v1 ,v2) (? ord? A)) #:when (cast v2 A)            (cast v2 A)]
    [(v (? (not/c ord?) A))
      (let ([As (split A)]) `(m ,(cast v (car As)) ,(cast v (cadr As))))]
    [(_ _)                                                           #f]))

(define/contract (disp v vl)
  (→ value? (or/c label? value?) expr?)
  (match v
    [`(: (λ (,x : ,A) ,e ,B) (→ ,C ,D))
                                    `(: ,(subst e x (cast vl A)) ,D)]
    [`(~> ,l ,v)  #:when (equal? l vl)                                v]
  [`(m ,v1 ,v2) #:when (not (asub (pt v2)  (at vl)))      (disp v1 vl)]
  [`(m ,v1 ,v2) #:when (not (asub (pt v1)  (at vl)))      (disp v2 vl)]
  [`(m ,v1 ,v2) #:when (and (asub (pt v1)  (at vl))
                            (asub (pt v2)  (at vl)))
                                      `(m ,(disp v1 vl) ,(disp v2 vl))]))
```

To evaluate the term, we choose the small-step approach. First, we develop a `step` function which models our operational semantics. Then the `eval` function evaluates the terms multiple times until they have a canonical form.

```
(define/contract (step e)
  (→ expr? expr?)
  (match e
    [(? exact-integer? n)                                      `(: ,n int)]
    [(? flonum? n)                                           `(: ,n float)]
    ['#t                                                     '(: #t bool)]
    ['#f                                                     '(: #f bool)]
    [`(λ (,x : ,A) ,e ,B)            `(: (λ (,x : ,A) ,e ,B) (→ ,A ,B))]
    [`(: ,(? pvalue? p) ,(? (not/c ord?) A))    (let ([As (spl A)])
                                `(m (: ,p ,(car As)) (: ,p ,(cdr As))))]
    [`(<~ ,(? value? v) ,(? label? l))                          (disp v l)]
    [`(<~ ,(? (not/c value?) e) ,(? label? l))        `(<~ ,(step e) ,l)]
    [`(~> ,(? label? l) ,(? (not/c value?) e))        `(~> ,l ,(step e))]
    [`(,(? value? v) ,(? value? vl))                           (disp v vl)]
    [`(: ,(? value? v) ,A)                                      (cast v A)]
    [`(: ,(? (not/c pvalue?) e) ,A)                      `(: ,(step e) ,A)]
    [`(,(? (not/c value?) e1) ,e2)                       `(,(step e1) ,e2)]
    [`(,(? value? v) ,e2)                                 `(,v ,(step e2))]
    [`(m ,e1 ,(? value? v))                            `(m ,(step e1) ,v)]
    [`(m ,(? value? v) ,e2)                            `(m ,v ,(step e2))]
    [`(m ,e1 ,e2)                              `(m ,(step e1) ,(step e2))]
    [`(int+ ,(? (not/c value?) e1) ,e2)            `(int+ ,(step e1) ,e2)]
    [`(int+ ,(? value? v) ,(? (not/c value?) e2))  `(int+ ,v ,(step e2))]
    [`(int+ ,(? value? v1) ,(? value? v2))                    (plus v1 v2)]
    [`(flo+ ,(? (not/c value?) e1) ,e2)            `(flo+ ,(step e1) ,e2)]
    [`(flo+ ,(? value? v) ,(? (not/c value?) e2))  `(flo+ ,v ,(step e2))]
    [`(flo+ ,(? value? v1) ,(? value? v2))                    (plus v1 v2)]))

(define/contract (eval e)
  (→ any/c value?)
  (let ([e (desugar e)])
    (when (infer e '()) (if (value? e) e (eval (step e))))))
```

## 6.3 Tour

This section gives a tour of our language and each line is documented in the comments.

```
;; simple literal
42 42.2 #t #f

;; λ abstraction
(λ (x : int) x int)

;; function application
((λ (x : int) x int) 1)

;; record creation, for simplicity, we use numbers to represent the label
(~> 42 #t)

;; record projection
(<~ (~> 42 #t)
    42)

;; merge two values
(m 1 #t)

;; merge two functions
(m (λ (x : int) x int)
   (λ (x : bool) x bool))

;; merged function can be applied
((m (λ (x : int) x int)
    (λ (x : bool) x bool))
 1)

;; merge two records
(m (~> 1 #t)
   (~> 2 #f))

;; merged records can be selected by label
(<~ (m (~> 1 #t)
       (~> 2 #f))
    1)

;; merged arguments can also be automatically selected
```

```
((λ (x : int) x int)
 (m 1 #t))

;; an expression can be annotated
(: (λ (x : int) x int)
   (→ int int))

;; annotate a "value" can force a downcast/upcast
(: (: 1 int)
   (& int int)) ;; ⇒ duplicate a number

(: (: (λ (x : int) x int) (→ int int))
   (& (→ int int)
      (→ int int))) ;; ⇒ duplicate a function

(: (m (λ (x : int) x int)
      (λ (x : bool) x bool))
   (→ bool bool)) ;; ⇒ downcast to a boolean identity function

;; use int+ to add integers
(int+ 1 3)
;; use flo+ to add floats
(flo+ 1.0 2.1)

((λ (x : int) (int+ x x) int) 1)
((λ (x : float) (flo+ x x) float) 1.1)

;; overload int+ and flo+ to create a polymorphic "double" function
((m (λ (x : int) (int+ x x) int)
    (λ (x : float) (flo+ x x) float))
 1)

;; variadic version of merge operator
(<~ (M (~> 1 #t)
       (~> 2 #f)
       (~> 3 #t))
    2)

;; another syntactic sugar for records
(R (1 #t) (43 #f) (99 #t))
```

# 7 Related Work

## 7.1 Intersection Types, Merges and Overloading

Forsythe, introduced by Reynolds [1988], 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 [1991] 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, the metatheory of the glue operator has not been studied.

Directly computing the type with arguments and functions in the application rule is not new. Originally it appears as the function *apptype* in the work of Freeman and Pfenning [1991]. The idea of *apptype* is simple: if the function has the type $(A_1 \to B_1)$ & $(A_2 \to B_2)$ & ... & $(A_n \to B_n)$ and the argument has the type $A$, the result type will be derived.

$$\underset{i|A\leq A_i}{\&B_i}$$

The algorithm is based on the refinement types which are rewritten into *union normal form* and *intersection normal form*, where the distributivity of intersection types will not appear. Also, the *apptype* algorithm is integrated into a type inference algorithm, the correctness of the algorithm is implied by the correctness of the inference and corresponding dynamic semantics is not designed.

Dunfield's calculus [Dunfield 2014] 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` $\to$ `Bool) 1`. As acknowledged by Dunfield, the semantics is not deterministic. This is similar to our first calculus in Chapter 4. To restrict the power of the merge operator and enable determinism, a disjointness restriction on merges has been proposed [Oliveira et al. 2016]. Closest to our work is the $\lambda_i^+$ calculus [Huang et al. 2021], which is a deterministic calculus with intersection types and a disjoint merge operator. There are two major differences between our work and $\lambda_i^+$. (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 $\lambda_i^+$ that employs applicative subtyping and thus avoids many unnecessary annotations that are required in $\lambda_i^+$ since function overloading and record projection are not directly supported in $\lambda_i^+$. In $\lambda_i^+$, we would need a term with an explicit type annotation instead: `((succ,,not) : Int → Int) 1`. The rigid form of applications and projections in $\lambda_i^+$ prevents expressions such as `(succ,,not) 1`, which are not well-typed in $\lambda_i^+$.

In recent work, [Rioux et al. 2022] 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 $\eta$-expansion. While overloading is supported, the disjointness relations are still not flexible enough to support return type overloading.

## 7.2 $\lambda\&$-Calculus

Castagna et al. [1995] gave a formalisation to calculus for overloaded functions with subtyping. In his calculus, overloaded functions are defined as &-terms, and their application is specially treated as a new application term since they adopt two different mechanisms. The syntax of his calculus is defined as follows. The most interesting part is highlighted where $M\&^V M$ is used for adding new instances for overloaded functions and $M \bullet M$ is used for overloaded applications.

$$\text{Terms} \quad M ::= x^V \mid \lambda x^V.\, M \mid M \cdot M \mid \epsilon \mid M\&^V M \mid M \bullet M$$

The types of overloaded functions are a finite list of arrow types with a consistency restriction.

$$\text{PreTypes} \quad V ::= A \mid V \to V \mid \{V_1' \to V_1'', ..., V_n' \to V_n''\}$$

For typing rules, the new introduction rule and elimination rule are added.

$$\{\}\textsc{Intro}\ \frac{\vdash M : W_1 \leq \{U_i \to V_i\}_{i\leq(n-1)} \qquad \vdash N : W_2 \leq U_n \to V_n}{\vdash (M\&^{\{U_i\to V_i\}_{i\leq n}} N) : \{U_i \to V_i\}_{i\leq n}}$$

$$\{\}\textsc{Elim}\ \frac{\vdash M : \{U_i \to V_i\}_{i\in I} \qquad \vdash N : U \qquad U_j = min_{i\in I}\{U_i | U \leq U_i\}}{\vdash M \bullet N : V_j}$$

Unlike our strategy, all the fitting branches will be accepted or rejected when ambiguities are detected. This typing rule will infer the result type of the "best-match" branch and apply that branch at runtime. The semantics is type-dependent, and overloaded applications rely on the runtime types, which is similar to our TDOS approach.

The confluence and subject-reduction are proved in his calculus. Differently to our approach, the 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 his work, records are encoded by lambda functions, and multi-field records are overloaded functions.

## 7.3 Type Classes

Type classes [Wadler and Blott 1989] enable ad-hoc polymorphism by adding constraints to polymorphic types, which is completely different from our approach. The type signature of overloaded functions is defined in a class, which is an interface that states the types of overloaded functions. We can simply add the implementation of overloaded functions by writing instances. The overloaded application will be made by instance resolution. The expressiveness of type classes is more powerful than our approach. For example,

```
show :: Show a ⇒ a → String
show [1,2] -- evaluated to "[1,2]"
```

It also works fine with the curried overloading. For example, addition can be defined as [1]

```
class Addable a b c where
    add :: a → b → c
```

Each instance will create an instance of overloaded function `add`,

```
instance Addable Int Int Int where
    add x y = x + y
```

[1]with MultiParamTypeClasses, FlexibleContexts extensions

```
instance Addable Int Bool Int where
    add x y = if y then x + 1 else x

curried :: Addable Int b c ⇒ b → c
curried = add (1 :: Int)
```

One of the interesting ways to compare type classes with our approach is nested composition. For example, if we abstract `show` function in a thunk, the class `Show` will appear at the most left. In the view of the distributivity, the result `pshow ()` and `pshow () 1` illustrate that the class `Show` can distribute over the each instance of `pshow`.

```
pshow :: Show a ⇒ p → a → String
pshow = \x → show
```

Another point to be noted is that constrained by the HM type system, type classes in Haskell do not support subtyping.

## 7.4 Semantic Subtyping

Semantic subtyping [Frisch et al. 2008] 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 then subtyping relations are derived from the semantics. The type system features intersection types, union types and negation types. Overloaded functions can be defined by a *typecase* primitive which is similar to the elimination of union types. For example, the type of `show` is `Int | Bool → String`

The typing for application is standard in their calculi [Castagna et al. 2022], and different implementations are dispatched by the explicit type given by the programmer. For example, the function `show` is defined as:

```
let show = fun x → if x is Int then showInt x else showBool x
```

Also, `succ,,not` has the type `(Int → Int) & (Bool → Bool)`.

```
let succnot = fun x → if x is Int then succ x else not x
```

The approach to semantic subtyping of overloaded functions is different from ours since in our calculi:

- only intersection types are used to represent types of overloaded functions; and
- overloaded functions can be introduced by simply merging.

## 7.5 Multimethods and Multiple Dispatching

Calculi with subtyping (e.g., Featherweight Java [Igarashi et al. 2001]) often feature dispatching since types of terms will become more precise during the reduction. The function call will be resolved by the actual types of arguments (referred to as runtime types). Multimethods describe that the dispatching strategy is based on multiple arguments. It has already been adopted by many languages, including Common Lisp Object System (CLOS) [DeMichiel and Gabriel 1987], Dylan [Shalit 1996], Julia [Bezanson et al. 2017] etc.

Zappa Nardelli et al. [2018] formalised Julia's multiple dispatching by an expressive subtyping relation. The most specific method (function) will be picked based on the type of arguments. Union types and parametric constructors have been used in their approach, however intersection types are not considered.

In multimethods, arguments are resolved at once and the "best-match" branch will be selected according to specializers. However, in our calculi, we resolve arguments one by one and all branches that can apply are selected and then results will be combined. The resolution in dynamic semantics is based on annotations.

# 8 Conclusion and Future Work

## 8.1 Conclusion

In this thesis, 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 some forms of overloading.

## 8.2 Future Work

We give some directions for future work below.

### 8.2.1 Application Mode

Though *applicative subtyping* is a rather elegant solution to enable the overloaded application, record projection and nested composition, *application mode* still has its advantages: it fits into the setting of ordinary overloading and has the potential for better type inference and best-match strategy, which will be expanded in later paragraphs.

We discussed some design challenges in Section 3.5. The ideal solution is to adopt an uncommon evaluation order and to reduce function application into the canonical form like $f\ a_1\ a_2 ... a_n$. The syntax of $f$ and $a_i$ should be defined in a delicate way.

Potential to reduce annotations In the calculi shown, our lambda syntax is fully annotated and rather rigid to write, abandoning the major advantages of bidirectional typing. As discussed in Section 3.5, we can utilise the application mode and allow us to write the expression like $(\lambda x.\ x, , \lambda x.\ (x+1))\ 1$. The future work is on how to design a corresponding dynamic semantics for application mode and improve the typing rule to allow the unannotated syntax.

### 8.2.2 Bidirectional Typing

Since our approach infers both the types of function and arguments, which is different from the traditional bidirectional typing for the application case.

$$\frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B \qquad A \ll B = C}{\Gamma \vdash e_1\, e_2 \Rightarrow C}\ \textsc{T-App}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \rightarrow B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\, e_2 \Rightarrow B}\ \textsc{T-App}$$

This approach sacrifices the possibility of writing a smarter high-order function application. For example $\texttt{map}\ (\lambda x.\ \texttt{true})\ [1, 2]$ (notice that the lambda function is not annotated).

$$\texttt{map} : (\texttt{Int} \rightarrow \texttt{Bool}) \rightarrow [\texttt{Int}] \rightarrow [\texttt{Bool}]$$

In our calculi, we need to write $\texttt{map}\ (\lambda x.\ \texttt{true} : \texttt{Int} \rightarrow \texttt{Bool})\ [1, 2]$ where the annotation is not necessary from intuition. A line of future work is on how to retain the advantage of *check mode* and also provide the information of arguments to help infer the application term.

### 8.2.3 Ambiuguities

In Chapter 3, we discussed the ambiguities and provided a rather naive solution to prevent those. However, whether *overloding semantics* or *disjoint intersection types* will reject the expressions we consider valid and unambiguous.

Future work includes finding a design that enables overloading while preserving determinism. There are several places to do this. One is that we can extend the disjointness specification and adapt it in case of overloading. Another is we add the extra condition check in applicative subtyping. For the latter, we gave several design choices, though none of them serves as a perfect design where ambiguous cases are forbidden while all applicable cases can be retained.

### 8.2.4 Best-Match Approach

Multimethods are related to our work and based on the precise runtime types. Usually, multimethods adopt a best-match approach, and so is the work of Castagna et al. [1995]. In our calculi shown, all the possible branches will be selected. However, with application mode, after all arguments are collected, we have the potential to model the “best-match” approach and also reuse the same approach in the dynamic semantics.

### 8.2.5 Disjoint Polymorphism

The combination of disjoint intersection types, merge operator and parametric polymorphism empowers the disjoint polymorphism [Alpuim et al. 2017], which lets programmers specify the constraints of the type variables. Disjoint polymorphism can encode many valuable features: dynamic mixins, polymorphic records, row and bounded polymorphism [Xie et al. 2020]. In the future, we are interested in investigating how to apply disjoint polymorphism in our calculi.

### 8.2.6 Compile to Racket

We give an interpreter implementation in Chapter 6, which heavily relies on the Racket's runtime contract checking. This approach differs from practical languages in two aspects:

- Type checking should be statically done at compile time.
- Overloading application and record projection should be statically resolved.

In the current implementation, we only utilise minimal parts of the functionalities of Racket's macro system and delegate the parsing and evaluation to the functions. We can refer to sophisticated techniques [Chang et al. 2017; King 2020] to employ the macro system to deal with type checking and overloading resolution. However, they all do not account for the elaboration semantics of the language. The future work is on investigating Racket's macro system to deal with elaboration and building the compiler implmentation of our languages.

# Bibliography

[Citing pages are listed after each reference.]

João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. 2017. Disjoint polymorphism. In *European Symposium on Programming*. Springer, 1–28. [cited on page 71]

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A filter lambda model and the completeness of type assignment1. *The journal of symbolic logic* 48, 4 (1983), 931–940. [cited on pages 1, 27, and 49]

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 59, 1 (2017), 65–98. [cited on page 67]

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)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. [cited on pages 1, 5, and 49]

Daniel G Bobrow, Linda G DeMichiel, Richard P Gabriel, Sonya E Keene, Gregor Kiczales, and David A Moon. 1988. Common lisp object system specification. *ACM Sigplan Notices* 23, SI (1988), 1–142. [cited on page 8]

Luca Cardelli and John C Mitchell. 1991. Operations on records. *Mathematical structures in computer science* 1, 1 (1991), 3–48. [cited on pages 3 and 37]

Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. 1995. A calculus for overloaded functions with subtyping. *Information and Computation* 117, 1 (1995), 115–135. [cited on pages 1, 5, 7, 8, 20, 64, and 70]

Giuseppe Castagna, Mickaël Laurent, Kim Nguyen, and Matthew Lutze. 2022. On type-cases, union elimination, and occurrence typing. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 75. [cited on page 66]

Stephen Chang, Alex Knauth, and Ben Greenman. 2017. Type systems as macros. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 694–705. [cited on page 71]

Arthur Charguéraud. 2012. The locally nameless representation. *Journal of automated reasoning* 49, 3 (2012), 363–408. [cited on page 15]

Alonzo Church. 1932. A set of postulates for the foundation of logic. *Annals of mathematics* (1932), 346–366. [cited on page 13]

Alonzo Church. 1940. A formulation of the simple theory of types. *The journal of symbolic logic* 5, 2 (1940), 56–68. [cited on page 13]

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 1981. Functional characters of solvable terms. *Mathematical Logic Quarterly* 27, 2-6 (1981), 45–58. [cited on pages 1, 18, and 19]

Rowan Davies and Frank Pfenning. 2000. Intersection types and computational effects. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 198–208. [cited on page 25]

Linda G DeMichiel and Richard P Gabriel. 1987. The common lisp object system: An overview. In *European Conference on Object-Oriented Programming*. Springer, 151–170. [cited on page 67]

Dominique Devriese and Frank Piessens. 2011. On the bright side of type classes: instance arguments in Agda. *ACM SIGPLAN Notices* 46, 9 (2011), 143–155. [cited on page 23]

Jana Dunfield. 2014. Elaborating intersection and union types. *Journal of Functional Programming* 24, 2-3 (2014), 133–165. [cited on pages 1, 5, 7, 8, 18, 21, 41, and 63]

Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional typing. *ACM Computing Surveys (CSUR)* 54, 5 (2021), 1–38. [cited on page 42]

Erik Ernst. 2001. Family polymorphism. In *European Conference on Object-Oriented Programming*. Springer, 303–326. [cited on pages 3 and 21]

Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 268–277. [cited on page 63]

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM (JACM)* 55, 4 (2008), 1–64. [cited on page 66]

Joseph A Goguen, James W Thatcher, Eric G Wagner, and Jesse B Wright. 1977. Initial algebra semantics and continuous algebras. *Journal of the ACM (JACM)* 24, 1 (1977), 68–95. [cited on page 14]

Xuejing Huang and Bruno C. d. S. Oliveira. 2020. A type-directed operational semantics for a calculus with a merge operator. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. [cited on page 23]

Xuejing Huang, Jinxu Zhao, and Bruno C. d. S. Oliveira. 2021. Taming the Merge Operator. *J. Funct. Program.* 31 (2021), e28. [cited on pages 1, 5, 6, 10, 25, 27, 41, 43, 44, and 63]

Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 3 (2001), 396–450. [cited on page 67]

Stefan Kaes. 1988. Parametric overloading in polymorphic programming languages. In *European Symposium on Programming*. Springer, 131–144. [cited on pages 7 and 8]

Alexis King. 2020. Hackett-lib. `https://pkgs.racket-lang.org/package/hackett-lib` [cited on page 71]

Stephen C Kleene and J Barkley Rosser. 1935. The inconsistency of certain formal logics. *Annals of Mathematics* (1935), 630–636. [cited on page 13]

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), 30 pages. [cited on page 1]

John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM* 3, 4 (1960), 184–195. [cited on page 55]

Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences* 17, 3 (1978), 348–375. [cited on page 13]

Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type classes as objects and implicits. *ACM Sigplan Notices* 45, 10 (2010), 341–360. [cited on page 23]

Bruno C. d. S. Oliveira, Zhiyuan Shi, and Joao Alpuim. 2016. Disjoint intersection types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 364–377. [cited on pages 49 and 63]

Benjamin C Pierce. 1991. *Programming with intersection types and bounded polymorphism*. Ph.D. Dissertation. Carnegie Mellon University. [cited on page 63]

Benjamin C Pierce and David N Turner. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (2000), 1–44. [cited on page 17]

Gordon D Plotkin. 1981. *A structural approach to operational semantics*. Aarhus university. [cited on page 14]

Garrel Pottinger. 1980. A type assignment for the strongly normalizable $\lambda$-terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism* (1980), 561–577. [cited on pages 1 and 19]

John C Reynolds. 1988. Preliminary design of the programming language Forsythe. (1988). [cited on pages 1, 18, 20, and 63]

John C Reynolds. 1991. The coherence of languages with intersection types. In *International Symposium on Theoretical Aspects of Computer Software*. Springer, 675–700. [cited on page 1]

John C Reynolds. 1997. Design of the Programming Language F orsythe. In *ALGOL-like languages*. Springer, 173–233. [cited on page 19]

Nicholas Rioux, Xuejing Huang, Bruno C. d. S. Oliveira, and Steve Zdancewic. 2022. *A Bowtie for a Beast*. Technical Report MS-CIS-22-02. Department of Computer and Information Science, University of Pennsylvania. [cited on page 64]

David A Schmidt. 1986. *Denotational semantics: a methodology for language development*. William C. Brown Publishers. [cited on page 14]

Andrew Shalit. 1996. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison Wesley Longman Publishing Co., Inc. [cited on page 67]

Philip Wadler. 1998. The expression problem. *Posted on the Java Genericity mailing list* (1998). [cited on pages 3 and 21]

Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 60–76. [cited on pages 2, 19, and 65]

David HD Warren. 1978. Applied logic: its use and implementation as a programming tool. (1978). [cited on page 16]

Andrew K Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and computation* 115, 1 (1994), 38–94. [cited on page 17]

Ningning Xie and Bruno C. d. S. Oliveira. 2018. Let Arguments Go First. In *European Symposium on Programming*. Springer, 272–299. [cited on pages 31 and 42]

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)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. [cited on page 71]

Wenjia Ye, Bruno C. d. S. Oliveira, and Xuejing Huang. 2021. Type-Directed Operational Semantics for Gradual Typing. In *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. [cited on page 24]

Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. 2018. Julia subtyping: a rational reconstruction. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27. [cited on page 67]

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. [cited on pages 3 and 21]