Taming Intersection Types and the Merge Operator

Xuejing Huang

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

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.

Xuejing Huang
January 2023

Abstract of thesis entitled

"Taming Intersection Types and the Merge Operator"

Submitted by

Xuejing Huang

for the degree of Doctor of Philosophy

at The University of Hong Kong

in January, 2023

For modern programming languages, a systematic approach for establishing usability and reliability is to first have a formalized and verified core language guarded by a type system. This dissertation proposes a *type-directed operational semantics* (TDOS) approach to model *disjoint intersection types*, and studies three calculi λ_i, λ_i^+, and F_i^+ serving as a new foundation for *Compositional Programming*.

Compositional Programming is a recently proposed programming style. Its prototype language CP supports multiple inheritance in a type-safe way and provides simple solutions to modularity problems that are hard for conventional object-oriented programming and functional programming languages. At the core of CP is F_i^+, a polymorphic calculus that supports a *symmetric merge operator* with subtyping. The merge operator generalizes record concatenation to any type, enabling expressive forms of object composition.

Due to its flexibility and ambiguity, the merge operator lacks a satisfying direct operational semantics. Prior systems usually define the runtime semantics indirectly by elaborating the source expressions into a target language. In contrast, the TDOS approach gives a semantics to F_i^+ directly. Besides being *deterministic* and *type sound*, the new calculus supports additional features such as *recursion* and *impredicative polymorphism*.

As an essential part of the TDOS design, we show a novel algorithm for the subtyping of intersection types with distributive laws. In this formulation, types that decompose into two smaller parts are characterized by *splittable types*. This allows a simple proof of transitivity and the modular addition of distributivity rules without pre-processing on types. We then extend this idea to union types and present an algorithmic formulation of subtyping based on the *minimal relevant logic* **B+**. (268 words)

Acknowledgements

I would like to express my deepest gratitude to my supervisor Bruno C. d. S. Oliveira, for his advice, insights, and patience. I almost knew nothing about programming language theory before I met Bruno and it was him taught me where to start and how to do research. Bruno always encourages me when we meet problems and he is open to different opinions. I really enjoy discussing and brainstorming with him and I am grateful to all the guidance he gave me.

I also want to thank my thesis advisory committee and examination committee, especially Prof. Frank Pfenning and Prof. Ondřej Lhoták, for providing valuable feedback and suggestions.

Additionally, this endeavor would not have been possible without the contribution from my coauthors: Jinxu Zhao, Andong Fan, Han Xu, and Yaozhu Sun. Special thanks to Weixin Zhang and Yaozhu Sun for developing the CP language, and João Alpuim, Zhiyuan Shi, Xuan Bi, Ningning Xie for their previous work on disjoint intersection types. I would like to extend my sincere thanks to Baber Rehman, Wenjia Ye, Nick Rioux, and Steve Zdancewic. It is my pleasure to collaborate with you.

Thanks should also go to other people I have been working with in the HKU PL Group: Tomas Tauber, Huang Li, Haoyuan Zhang, Yanlin Wang, Yanpeng Yang, Mingqi Xue, Xu Xue, Chen Cui, Jinhao Tan, Shengyi Jiang, Yaoda Zhou, and Litao Zhou. I have received a lot of feedback and learnt a great deal through our group seminars.

I would like to thank people in the PL community, which is not big, but very warm, including the anonymous reviewers and readers of our papers, and many lovely people I met in OPLSS 2017, the beginning of my journey in PL research.

I thank my parents and everyone in my family for their support, including our beloved cat Xiaohei who passed away before the beginning of 2023.

Especially, I thank my boyfriend Bingchen Gong, for his love, encouragement, and patience. Thank you for always being here by my side.

Last several years were very hard, for Hong Kong, and for the world. Through it all, we remain hopeful, and will be more brave. “If there is no struggle, there is no progress.”

Contents

CHAPTER 1

Introduction

A programming language does not only serves as a bridge between human and machines, but also plays a key role for programmers to communicate and to organize their own ideas. While natural languages can be ambiguous, programming languages are supposed to be well-defined and have a clear interpretation. For execution, we want programs to always behave as expected, and do not cause errors. A runtime error could be fatal as all immediate computation results will be lost and the program stops to provide service. For coding, we like languages with intuitive notations, supporting concise and reusable programs. Types provide abstraction, and help classify various data and regulate programs. For example, by specifying the addition function only takes two integers, a language can prevent users to add a integer to a character in before running the program, for which the expected behavior is unclear. Or, in a object-oriented programming language, users can group a sort of data (objects) by characterizing their common properties in the class definition, and be free to forget the implementation details when using the class. For modern programming languages, a systematic approach to establishing the usability and reliability is to have a formalized and verified core language guarded by a type system. Such a type system handles the interaction of all features in one language, making sure they do not conflict, and guarantees the safety of programs, like the famous slogan “well-typed programs cannot go wrong" [Mil78].

In this dissertation, we focus on the core calculus of programming languages, including type systems, operational semantics, subtyping and related algorithms, towards a type-safe and modular programming language.

1.1 Motivations

Terms are characterized by types. On the other hand, types can be interpreted as a set of terms that inhabit it. In some systems, one term can have multiple types, and some of the sets of types are fully contained by another set. Usually, it is the set inclusion relation that defines the

subtyping relation among types. Compared with its supertype, a subtype includes less terms and therefore captures the properties of its inhabited terms more precisely. Any term that can be typed by a subtype can also be typed by the supertype. Thus it is always "better" to use a subtype to type a term when it is possible.

Some work on type systems focuses on enhancing the expressiveness of types. With universal quantified types, parametric polymorphism [Gir72; Rey74] can assign the most specific type to the identity function $\lambda x.\, x : \forall \alpha.\, \alpha \to \alpha$. With intersection types, as a form of finitary polymorphism, the same function can have type $(\mathsf{Int} \to \mathsf{Int}) \,\&\, (\mathsf{Bool} \to \mathsf{Bool})$, or $(\mathsf{Pos} \to \mathsf{Pos}) \,\&\, (\mathsf{Neg} \to \mathsf{Neg})$ which is a refinement of $\mathsf{Int} \to \mathsf{Int}$. (Here Pos and Neg stands for positive integers and negative integers). When types carry more information, some previously ill-typed terms become well-typed. For example, the intersection type discipline accepts all normalizing terms, including $(\lambda x.\, x\, x)\ (\lambda x.\, x)$, which can also be typed with universal quantified types.

On the other hand, ad hoc polymorphism, or overloading, allows one name (usually a function) to be associated with multiple implementations of different types. This feature can be encoded by a merge construct [CGL95]. A merge operator [Rey88] composes expressions, and the aggregated expression can directly behave like a sub-expression without destructors, as shown in the following program:

$$\mathsf{let}\ x : \mathsf{Int} \,\&\, \mathsf{Bool}\ =\ 1 \,,, \mathsf{true}\ \mathsf{in}\ (x + 1,\ \mathsf{not}\ x)$$

Here ,, stands for the merge operator while , is used to construct pairs. The variable x has the intersection type $\mathsf{Int} \,\&\, \mathsf{Bool}$. Later uses of x can treat it either as an integer or as a Boolean. For this example, the executing result is the pair $(2,\ \mathsf{false})$. Note that in an intersection type system with conventional terms, $\mathsf{Int} \,\&\, \mathsf{Bool}$ is uninhabited, so is $(A \to A) \,\&\, (A \to A \to A)$. Because the two parts of the intersection share no common normal forms. With such a construct that merges computationally different terms, these two types become inhabited.

There are various practical applications for the merge operator. As Dunfield argues, the merge operator provides "general mechanisms that subsume many different features" [Dun14]. Besides overloading, merges of single-field records can encode *multi-field records*, and merges of functions can be used as *an elimination construct of union types*. Furthermore, when we restrict our attention to the concatenation of records, which the merge operator generalizes, the combination of record concatenation and subtyping paves the ground for encoding expressive forms of *multiple inheritance* [Wan89; Rém95; PZ04; Zwa97].

More recently, the merge operator has been used in calculi with *disjoint intersection types* [OSA16]. The disjointness restriction means that the two values being merged have distinct types. Such a variant of the symmetric merge operator has been used to encode several non-trivial object-oriented features, which enable highly dynamic forms of object composition that are not available in current mainstream languages such as Scala or Java. These include

first-class traits [BO18], *dynamic mixins* [AOS17], and forms of *family polymorphism* [BOS18]. These features enable widely used and expressive techniques for object composition used by JavaScript programmers (and programmers in other dynamically typed languages), but in a completely *statically type-safe* manner. Following is an example of first-class traits in the CP language [ZSO21], which is based on disjoint intersection types.

```
// addId takes a trait as an argument, and returns another trait
addId (base : Trait<Person>) (idNumber : Int) =
trait [self : Person] inherits base ⇒ {
  // dynamically inheriting from an unknown person
  id = idNumber
};
```

Similarly to classes in JavaScript, first-class traits can be passed as arguments, returned as results, and can be constructed dynamically (at run-time). In the program above inheritance is encoded as a merge in the core language used by CP.

Despite over 30 years of research, the semantics of the merge operator has proved to be quite elusive. This is perhaps not too surprising. It is well-known that, in the closely related area of record calculi, the combination of record concatenation and subtyping is non-trivial [CM91]. Since the merge operator for intersection types generalizes record concatenation and calculi with intersection types naturally give rise to subtyping, the semantics of the merge operator will clearly face similar problems that appear in record calculi with subtyping.

Because of its pervasive importance, we would expect a simple and clear *direct* semantics for calculi with a merge operator. After all, this is what we get for other foundational calculi such as the *simply-typed lambda calculus*, *System F*, *System* F_ω, the *calculus of constructions*, *System* $F_{<:}$, *Featherweight Java* and others. While for the merge operator there have been efforts in the past to define direct operational semantics, these efforts have placed severe limitations that disallow many of the previously discussed applications or they lacked important properties. Reynolds was the first to look at this problem, but in his calculus the merge operator is severely limited [Rey91]. Castagna, Ghelli, and Longo studied another calculus, where only merges of functions are possible [CGL95]. Pierce was the first to briefly consider a calculus with an unrestricted merge operator (called *glue* in his work) [Pie91]. He discussed an extension to $F_\wedge$ with a merge operator but he did not study the dynamic semantics with the extension. Finally, Dunfield goes further and presents a direct operational semantics for a calculus with an unrestricted merge operator [Dun14]. However, the problem is that subject reduction and determinism are lost.

Dunfield also presents an alternative way to give the semantics for a calculus with the merge operator *indirectly by elaboration* to another calculus. This elaboration semantics is type-safe and offers a reasonable implementation strategy, and it is also employed in more recent work on the merge operator with disjoint intersection types. However, the elaboration semantics

has two major drawbacks. Firstly, reasoning about the elaboration semantics is more involved: to understand the semantics of programs with the merge operator we have to understand the translation and semantics of the target calculus. This complicates informal and formal reasoning. Secondly, in calculi defined by elaboration, we want to have *coherence* [Rey91], which is a property that ensures that the meaning of a program is not ambiguous. Dunfield's elaboration semantics is not coherent. To fix this, calculi with disjoint intersection types have to impose some restrictions. However, even with such restrictions, coherence comes at a high price: the calculi and proof techniques employed to prove coherence are complex, and can only deal with terminating programs. The latter is a severe limitation in practice!

1.2 Overview

Mainly, this dissertation address two problems:

- Finding algorithmic subtyping formulations with distributivity rules
- Designing a type-directed operational semantics for calculi with the merge operator

Our focus This dissertation proposes a *type-directed operational semantics* (TDOS) for calculi with intersection types and a merge operator. Our calculi address two key difficulties in the dynamic semantics of calculi with a merge operator. The first difficulty is the type-dependent nature of the merge operator. Using type annotations as casts to guide reduction addresses this difficulty, and paves the way to prove *type soundness*. The second difficulty is that a fully unrestricted merge operator is inherently ambiguous. For instance the merge $1 \,,, 2$ can evaluate to both 1 and 2. To obtain a deterministic reduction semantics, our type systems employ a disjointness restriction that is used in calculi using disjoint intersection types, and a new notion *consistency*, which relaxes the restriction on merges.

In the core of our reduction is the casting of terms. Our casting rules mirror the structure of subtyping definition. That is because casts correspond to coercions in the elaboration semantics, which are generated by subtyping derivation. While coercions are conversion functions, the reduction rules of casting directly convert terms. As we want the reduction rules to be executable, the corresponding subtyping formulation has to be algorithmic. Besides, one subtyping judgment can generate different coercions because it has multiple derivations. To ensure the elaboration semantics is unambiguous, the coherence property requires that the elaboration results of the same term are always equivalent, even when they contain different coercions. Determinism in TDOS offers the same guarantee that coherence offers in an elaboration semantics. We will show that when two casting rules overlap, for which case in the subtyping two different coercions can be produced, the resulting term is unique. As a result,

the TDOS approach deals with recursion and impredicative polymorphism in a straightforward way.

This thesis has two parts. One focus is to formulate subtyping with intersection and union types in an algorithmic style. The other is to design calculi that support intersection types and the merge operator. We will start from the subtyping algorithms and then talk about three calculi: λ_i, a minimal calculus with the merge operator and a deterministic direct operational semantics. λ_i^+, which extends λ_i by BCD-style distributive subtyping rules, and also adds support for record types and *nested composition*. F_i^+, which extends λ_i^+ by disjoint polymophism. The subtyping relation in λ_i is relatively simple while λ_i^+ and λ_i^+ make use of our algorithmic formulation of subtyping.

Our calculi follow the idea of disjoint intersection types [OSA16; BOS18; Bi+19]. They are based on the existing work: λ_i of [OSA16], λ_i^+ of Bi, Oliveira, and Schrijvers, and F_i^+ of Bi et al. While our calculi share similar type systems with these calculi, we are the first to employ the direct operational semantics approach to disjoint intersection types. Previous calculi use the framework proposed by Dunfield where the semantics is defined by elaboration. In the elaboration semantics, terms in the source language are translated into a standard target language via the typing derivation. A coercion is produced by each subtyping derivation to help converting terms. A key property in such system is to show that all coercions of the same subtyping judgment produce coherent results. It, then causes difficulty on the proof of coherence. With the previous proof approach, some restrictions are needed in the systems: No recursion, no impredicative polymorphism.

Summary of contributions In summary, the contributions of this thesis are as follows:

- **A novel algorithmic formulation of subtyping relations for intersection and union types with distributivity:** By generalizing the conjunction and disjunction introduction rule in subtyping, our formulation embeds the challenging distributivity rules in a modular way and eliminates the transitivity rule to obtain algorithmic properties. We present two formulations that are sound and complete to the original BCD subtyping and the subtyping based on **B+** logic respectively.

- **The λ_i, λ_i^+, and F_i^+ calculi and their TDOS:** We propose a type-directed operational semantics for calculi with intersection types and a merge operator, where type annotations are interpreted as casts. For each of the three calculi, we show the reduction is *deterministic* and prove the *type soundness* property.

- **Support for non-terminating programs and impredicative polymorphism:** With a direct operational semantics, our proof methods can deal with recursion and impredicative polymorphism, while the prior calculi with disjoint intersection types do not support them due to limitations of the proof approaches for coherence.

- **Casting and consistency:** We propose the reduction rules of casting and the notion of consistency, which are useful to obtain determinism and type soundness.

- **Coq formalization:** All the results presented in this paper have been formalized in the Coq theorem prover. The proofs for Chapter 3 and Chapter 4 can be found at `https://github.com/XSnow/DistributingTypes`. The proofs for Chapter 5 and Chapter 6 can be found at `https://github.com/XSnow/TamingMerge`. The proofs for Chapter 7 can be found at `https://github.com/XSnow/CP-Foundations`.

Roadmap The structure of the thesis is as follows:

Chapter 2 is about background of intersection types and the merge operator, including the history and applications of disjoint intersection types.

The following technical sections are divided into two parts.

The first part of the technical sections studies subtyping algorithms.

Chapter 3 focus on the subtyping of intersection types. Especially, it shows how we eliminate the transitivity rule to obtain algorithmic formulations.

Chapter 4 extends the discussion to union types. None of the three calculi supports union types. So this part can be ignored for readers who want to focus on the development of these calculi.

The second part utilizes the algorithmic formulation of subtyping in three calculi.

Chapter 5, **Chapter 6**, and **Chapter 7** presents the λ_i calculus, λ_i^+ calculus, and F_i^+ calculus respectively.

Chapter 8 reviews related work. We compare λ_i with the calculi proposed by Dunfield and Oliveira, Shi, and Alpuim (the original λ_i calculus). In short, all programs that are accepted by the original λ_i calculus can type-check with our type system, and the semantics of λ_i is sound with respect to Dunfield's semantics.

Chapter 9 discusses some potential directions for future work.

Prior publications This thesis is based on these previously published papers:

- Xuejing Huang, and Bruno C. d. S. Oliveira. 2020. "A Type-Directed Operational Semantics for a Calculus with a Merge Operator". In *European Conference on Object-Oriented Programming (ECOOP)*.

- Xuejing Huang, Jinxu Zhao, and Bruno C. d. S. Oliveira. 2021. "Taming the Merge Operator". In *Journal of Functional Programming (JFP)*.

- Xuejing Huang, and Bruno C. d. S. Oliveira. 2021. "Distributing Intersection and Union Types with Splits and Duality (Functional Pearl)". 2021. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

- Andong Fan*, Xuejing Huang*, Han Xu, Yaozhu Sun, and Bruno C. d. S. Oliveira. “Direct Foundations for Compositional Programming”. 2022. In *European Conference on Object-Oriented Programming (ECOOP)*. (Co-first authors)

CHAPTER 2

Background

Intersection types have been studied since the 1970s. Saying that a term has an intersection type $A \& B$ means it has both type A and type B. In the beginning, they were introduced to type systems as a correspondence of conjunction. While intersection types enhance the expressiveness of types, and help characterize all normalizing terms [CD78; Pot80; BCD83], they do not correspond to conjunction from the view of propositions as types [How80], i.e. the provability of propositions correspond to the inhabitation of types. The conjunction of two provable propositions are always provable. But the intersection of two inhabited types is not inhabited unless there exists some term that belongs to both types. Compared with product types, which are thought to be the correspondence of conjunction, the two propositions in an intersection type must be proved by the same program, rather than two programs embedded in a pair.

The term-level construct of intersection, merge operator, was later proposed for the imperative language Forsythe [Rey88]. A key advantage of the merge operator is its generality and the ability to model various programming language features. However, it poses significant challenges to type system and semantics design. In this chapter, we briefly review some related work for background information:

- Section 2.1 is about the early history of intersection types. It aims to provide justification for intersection subtyping.
- Section 2.2 shows the merge operator and its non-deterministic semantics proposed by Dunfield [Dun14].
- Section 2.3 discuss the information hiding problem in subtyping.
- Section 2.4 presents the disjoint restriction that rejects ambiguous merges proposed by Oliveira, Shi, and Alpuim [OSA16].

- Section 2.5 demonstrates some language features encoded via the merge operators in a Compositional Programming language.

2.1 Intersection Type Discipline and BCD-Subtyping

Following types à la Curry [CF58], lambda abstractions do not come with annotations, therefore a function can have different types.

$$\frac{\Gamma\,,\, x{:}A \vdash e : B}{\Gamma \vdash \lambda x.\, e : A \to B}\ {\to}\textsc{intro}$$

Curry's system has the principal typing property: for any typeable term, a *best* type can be found, from which all the types that are assignable to the term can be generated. Commonly, the *betterness* makes a partial order of types. That is the subtyping relation, written as $A \leqslant B$. The subtype A is more specific and contains more information, while the supertype B is more general and have more terms inhabited. In a type system with subtyping, the principal type of a term is a subtype of any type that is assignable to it.

In this system, every typeable term is guaranteed to reduce to a normal form. But the reverse does not hold. In other words, types are preserved in reduction but not in expansion. The type assignment system is conservative and rejects some strongly normalizing terms. A typical example is the self application function:

$$\lambda x.\, x\, x$$

The variable x needs to have a function type in its first occurrence and an argument type in its second occurrence. To better describe such polymorphism and capture these strongly normalizing terms that cannot be accepted by previously existing systems, started from the 1970s, multiple researchers, including Coppo, Dezani-Ciancaglini, Sallé, Pottinger, Venneri, and Barendregt, extended Curry's system with intersection types, a new kind of type [CD80; CDV81; CDS79; CDV80; Pot80; BCD83]. They gave the introduction and elimination rules for intersection types as follows.

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash e : B}{\Gamma \vdash e : A \,\&\, B}\ \&\,\textsc{intro} \qquad \frac{\Gamma \vdash e : A \,\&\, B}{\Gamma \vdash e : A}\ \&\,\textsc{elim-l} \qquad \frac{\Gamma \vdash e : A \,\&\, B}{\Gamma \vdash e : B}\ \&\,\textsc{elim-r}$$

The previous self application term now has type $A \,\&\, (A \to B) \to B$.

Note that we use $\&$ as the binary constructor for intersection type (and $|$ for union type). We follow the convention that intersections have higher precedence than arrows.

$$\textit{Type} \qquad A, B, C ::= \mathbb{A} \mid \top \mid A \to B \mid A \,\&\, B$$

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

$$\text{OS-REFL}\ \frac{}{A \leqslant A} \qquad \text{OS-TRANS}\ \frac{A \leqslant B \qquad B \leqslant C}{A \leqslant C} \qquad \text{OS-TOP}\ \frac{}{A \leqslant \top} \qquad \text{OS-TOPARR}\ \frac{}{\top \leqslant \top \to \top} \qquad \text{OS-ARR}\ \frac{B_1 \leqslant A_1 \qquad A_2 \leqslant B_2}{A_1 \to A_2 \leqslant B_1 \to B_2}$$

$$\text{OS-AND}\ \frac{A \leqslant B \qquad A \leqslant C}{A \leqslant B \,\&\, C} \qquad \text{OS-ANDL}\ \frac{}{A \,\&\, B \leqslant A} \qquad \text{OS-ANDR}\ \frac{}{A \,\&\, B \leqslant B} \qquad \text{OS-DISTARR}\ \frac{}{(A \to B) \,\&\, (A \to C) \leqslant A \to B \,\&\, C}$$

Figure 2.1: The BCD-subtyping with intersections and distributivity.

Such a system type checks all strongly normalizing terms. But it is then too powerful for its type inference to be decidable.

A subtyping relation is later introduced by Barendregt, Coppo, and Dezani-Ciancaglini with a more enhanced type assignment system [BCD83], shown in Figure 2.1. $\mathbb{A}$ denotes a countable set of type atoms, and $\top$ is the universal type, which is the top element in subtyping.

Rules OS-AND, OS-ANDL, and OS-ANDR correspond to the introduction and elimination typing rules for intersection types. Together they axiomatize that $A \,\&\, B$ is the greatest lower bound of A and B. OS-DISTARR is its signature rule. It states that arrow distributes over intersection. The rule OS-TOPARR is also interesting: in combination with the transitivity rule, it makes $\top \leqslant A \to \top$ admissible. With BCD subtyping, we can further justify the property $\top \leqslant A \to \top$ by the distributivity rule (extended to multiple components):

$$(A \to B_1) \,\&\, (A \to B_2) \,\&\, \ldots \,\&\, (A \to B_n) \leqslant A \to B_1 \,\&\, B_2 \,\&\, \ldots \,\&\, B_n$$

The $\top$ type is commonly treated as an intersection of zero types. Therefore, when $n = 0$, the above subtyping judgment becomes $\top \leqslant A \to \top$.

Two types can also be considered equivalent when they are subtypes of each other. In this case, we write $A \equiv B$. For example, $\mathsf{Int} \equiv \mathsf{Int} \,\&\, \mathsf{Int}$ and $\mathsf{Int} \,\&\, \mathsf{Bool} \equiv \mathsf{Bool} \,\&\, \mathsf{Int}$.

Definition 2.1 (Type Equivalence)**.** $A \equiv B \triangleq A \leqslant B$ and $B \leqslant A$.

Strictly speaking, the subtyping of intersection types is a *preorder* on types since equivalent but syntactically different types exist. But if we divide types into equivalence classes, the subtyping can can be considered as a partial order relation.

2.2 The Merge Operator and Dunfield's Semantics

The *merge operator* for intersection types was firstly introduced by Reynolds in the Forsythe language over 30 years ago [Rey88]. It takes two terms e_1 and e_2 of some types A and B, to create a new term $e_1 \,,, e_2$ that can behave both as a term of type A and as a term of type B. It has since been studied, refined and used in some language designs by multiple researchers [Pie91; CGL95; Dun14; OSA16].

$$\mathsf{let}\ x : \mathsf{Int} \,\&\, \mathsf{Bool} = \ 1 \,,, \mathsf{true}\ \mathsf{in}\ (x + 1,\ \mathsf{not}\ x)$$
$$\mathsf{let}\ x : (\mathsf{Int}\ ,\ \mathsf{Bool}) = \ (1\ ,\ \mathsf{true})\ \mathsf{in}\ (\mathsf{fst}\ x + 1\ ,\ \mathsf{not}\ (\mathsf{snd}\ x))$$

The term-level construct makes intersection types very similar to product types. For instance, the above two programs behave identically. We use $(e_1\ ,\ e_2)$ for products and $(A\ ,\ B)$ for the product type. The major difference between the two programs is that the elimination of products is explicit. Projections are used to extract the inner component with its location specified: fst (or proj_1) takes the first one while snd (or proj_2) takes the second one.

In contrast, the extraction of values from merges is implicit. As demonstrated by the above program, it is often driven by the types of the terms. But terms can have the same type. A key problem with the merge operator is ambiguity. For instance, for the expression:

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

the result is ambiguous (it could be 4 or 5) since we could extract either 1 or 2 from the merge to add to 3.

In a biased setting, one side of the merge has higher priority than the other side. For example, in Reynolds's merge, if there are multiple functions, the rightmost one overrides others. Dunfield proposed an unbiased merging construct to better align intersection types with logical conjunction [Dun14]. On top of the previously shown three typing rules for intersection types, she added the following two rules for merges.

$$\frac{\Gamma \vdash e_1 : A}{\Gamma \vdash e_1 \,,, e_2 : A}\ \textsc{merge-l} \qquad \frac{\Gamma \vdash e_2 : A}{\Gamma \vdash e_1 \,,, e_2 : A}\ \textsc{merge-r}$$

Although no intersection type is directly involved in these two rules, with the help of the introduction rule of intersection, merge can have an intersection type. This unbiased merge operator has a flexible behavior, as defined by Dunfield.

$$\frac{e_1 \rightsquigarrow e_1'}{e_1 \,,, e_2 \rightsquigarrow e_1' \,,, e_2} \qquad \frac{e_2 \rightsquigarrow e_2'}{e_1 \,,, e_2 \rightsquigarrow e_1 \,,, e_2'} \qquad \frac{}{e_1 \,,, e_2 \rightsquigarrow e_1} \qquad \frac{}{e_1 \,,, e_2 \rightsquigarrow e_2} \qquad \frac{}{e \rightsquigarrow e \,,, e}$$

There is no evaluation priority between the two components in a merge. Besides, $e_1 \,,, e_2$ (including values) can step, to its left subexpression or the right one. Other values can step further as well, by the last rule, which allows any expression to split into two. As Dunfield noted, the operational semantics is nondeterministic and does not preserve types.

Problem 1: no subject reduction. Since the reduction is oblivious of types, a term can reduce to two terms with potentially different (and unrelated) types. For instance:

$$1 \,,, \mathsf{true} \rightsquigarrow 1 \qquad\qquad 1 \,,, \mathsf{true} \rightsquigarrow \mathsf{true}$$

In Dunfield's calculus the term $1 \,,, \mathsf{true}$ can have multiple types, including Int or Bool or Int & Bool or Bool & Int. Not all types that can be assigned to a term lead to type-preserving reductions. For instance, if the term is given the type Int, then the second reduction above does not preserve the type. What is worse, a well-typed expression can reduce to an ill-typed expression by dropping the wrong part:

$$(1 \,,, \lambda x.\, x + 1)\, 2 \rightsquigarrow 1\, 2$$

Problem 2: non-determinism. The choice between a merge always has two options. Even in type-preserving reductions, a reduced term can lead to two other terms of the same type with different meanings.

$$1 \,,, 2 \rightsquigarrow 1 \qquad\qquad 1 \,,, 2 \rightsquigarrow 2$$

There is even a third option to duplicate the whole merge:

$$1 \,,, 2 \rightsquigarrow (1 \,,, 2) \,,, (1 \,,, 2)$$

In other words, the semantics is non-deterministic.

Dunfield's system uses elaboration typing to translate merges to products and insert projections. The elaboration semantics, although is type-safe, also suffers from this issue. The merge $1 \,,, 2$ can elaborate to 1 or 2 when checked against Int by the typing rules. Her implementation prioritizes the left part, resulting in a biased merge operator.

2.3 The Information Hiding Problem in Subtyping

The complications of subtyping. An intuition behind subtyping is that a subtype provides more information for reasoning about a term. Then, naturally, we expect any term of a subtype can be safely used as a term of its supertype, which is called the principle of safe substitution [Pie02]. The following subsumption rule, commonly seen in type systems with subtyping, supports this principle.

$$\frac{\Gamma \vdash e : A \qquad A <: B}{\Gamma \vdash e : B}\ \textsc{subsumption}$$

However, subtyping and the subsumption rule enable a program to "forget" about some static information about the types of values. Since the extraction of values from merges is type-directed, such loss of type information can affect the search for the value. Consider the following program:

$$\mathsf{let}\ x : \mathsf{Bool}\ =\ \mathsf{true}\ ,,\ 1\ \mathsf{in}\ (2\ ,,\ x) + 3$$

Note that here we view $\mathsf{true}\ ,,\ 1$ as a value. The merge has type $\mathsf{Bool}\ \&\ \mathsf{Int}$, but because of subtyping it also has type Bool, the type for x. In a naive operational semantics, for the program above, we would eventually reach a point where we would need to extract a value from the merge $2\ ,,\ \mathsf{true}\ ,,\ 1$. This merge has two conflicting integers values.

In a language employing a disjointness restriction the merge $2\ ,,\ \mathsf{true}\ ,,\ 1$ ought to be rejected, but such a merge only appears at run-time. In the program itself all merges are disjoint: $\mathsf{true}\ ,,\ 1$ is disjoint; and $2\ ,,\ x$ is also disjoint since x has type Bool. Thus the program should type-check! One possibility would be to abort the program at run-time with a disjointness error. However, this would defeat the main purpose of the disjointness restriction, which is to provide a way to *statically* prevent ambiguity.

A language offering an asymmetric merge operator would have other issues. Assuming that the merge operator would be right-biased (giving preference to the values on the right side), then a programmer may expect that because x has type Bool, $(2\ ,,\ x) + 3$ should evaluate to 5. However such *static reasoning* is not synchronized with the run-time behavior, since x contains the integer 1 and therefore the result of the evaluation would be 4.

From another perspective, we could expect that a valid optimization of the program above is to replace the expression $(2\ ,,\ x) + 3$ by $2 + 3$, since the static type of x has no integer type. This optimization would be valid (for both symmetric and asymmetric merges) if the origin of runtime values can be determined statically by looking at the types. However this is not the case if we simply employ a naive semantics: we statically know that the merge contains an integer 2, but at runtime 1 is extracted instead.

Note that the disjointness constraint assumes the type we know is complete: a term of type Bool does not contain Int. But this is not true if we can directly use $\mathsf{true}\ ,,\ 1$ when the context expects a Bool. This is to say, we cannot have the principle of safe substitution, which is based on the *inclusive semantics* of subtyping: a subtype is a subset of its supertype, when considering types as sets of all inhabited values. Reynolds proposed the *coercive semantics* [Rey88] that interprets subtyping as implicit conversions. His type system still has the subsumption rule, but the meaning of a program depends on its typing derivation and it is unambiguous. Although his

system has an asymmetric merge operator, the idea of using subtyping to generate coercions and defining the semantics of programs with the access of typing derivation inspires later work on the merge operator.

Record concatenation and subtyping. The problems with the merge operator and subtyping are closely related to the problem of typing record concatenation in the presence of subtyping. The latter is well-acknowledged to be a difficult problem in the design of record calculi [CM91]. Foundational work on programming languages in the end of the 80s and the early 90s looked at this problem because the combination record concatenation with subtyping was perceived as a way to extend lambda calculi with support for OOP. In essence, since objects in OOP can be viewed as records, it is natural to look for a language that supports records. Furthermore, record concatenation would provide support for encoding *multiple inheritance*, which entails composing several objects/records together. Finally, subtyping is perceived as a key feature of OOP and should be supported as well. Unfortunately, the problem was found to be quite challenging, for very similar reasons to those that make the interaction of the merge operator with subtyping difficult. This should not come as a surprise, since the merge operator can generalize record concatenation. To see the relationship between the two problems, consider the following variant of the previous program with records:

$$\mathsf{let}\ x{:}\{l{:}\mathsf{Bool}\}\ =\ \{m = 1\} \,,,\, \{l = \mathsf{true}\}\ \mathsf{in}\ (\{m = 2\} \,,,\, x).m + 3$$

In this variant, x is a record with the static type $\{l{:}\mathsf{Bool}\}$, but having an extra field m that is hidden by subtyping. The record x is then merged with the record $\{m = 2\}$. Statically this merge seems safe, since the static types of both records do not share record labels in common. However, when doing the field lookup for m at runtime there would be two fields m with different values (once again assuming a naive semantics). In essence, we would have the same problems as with the earlier variant of the program without records.

2.4 Disjoint Intersection Types

Ideally, components of a merge are commutative. They cannot be selected via tags or locations. But this does not mean there is no information for the elimination of merges. For example, since the context $[\cdot] + 3$ expects an integer and 1 is the only integer part in $1 \,,,\, \mathsf{true}$, there is no ambiguity for $(1 \,,,\, \mathsf{true}) + 3$ evaluates to 4. That is to say, the ill-typed possibility $\mathsf{true} + 3$ is ignored. Following this approach, the elimination of merges can be deterministic as long as terms in a merge are always distinguishable by types. Terms like $1 \,,,\, 2$ should be rejected because it provides two different values to the same type Int, and is thus potentially ambiguous.

Such restriction on merges is conducted via a check on types called *disjointness* ($A * B$ means

A and B are disjoint), introduced by Oliveira, Shi, and Alpuim [OSA16].

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

Note that in the above typing rule of merges, $\Rightarrow$ denotes the *synthesis mode* in *bidirectional typing* [PT98]. In typing judgements with such a mode, types are synthesized from the term and the typing environment, rather than provided as inputs for the term to be checked against. The synthesized type is usually the principal type, and describes all the components contained by the expression exhaustively. For example, the term $1 \,,, \mathsf{true}$ has $\mathsf{Int} \,\&\, \mathsf{Bool}$ as its synthesized type. If we are going to merge it with another term e, knowing that the synthesized type of e does not have an Int or Bool part is enough to avoid conflicts.

2.4.1 Disjointness

To check whether two types are disjoint is to check whether they overlap. Here, overlapping is defined on subtyping. Any part of an intersection is its supertype. So the shared part of two overlapping types is their common supertype. For example, Int is the smallest supertype of $\mathsf{Int} \,\&\, \mathsf{Bool}$ and $\mathsf{Char} \,\&\, \mathsf{Int}$. If two types are disjoint (e.g. $(\mathsf{Int} \,\&\, \mathsf{Char}) * \mathsf{Bool}$), they should have no common part, and their corresponding values never conflict (e.g. $1 \,,, \text{'c'}$ and true).

In general, knowing the A is a subtype of B means that any term of A can be used as (or converted to) a term of B. Therefore, if e_1 and e_2 have type A_1 and A_2, and they are both subtypes of B, in a context expects a term of B either e_1 or e_2 can satisfy it. In this case, we say A_1 and A_2 overlaps on B.

Definition 2.2 (Disjointness specification)**.** If A is disjoint with B (written as $A * B$), any common supertypes they have must be equivalent to $\top$.

The top type $\top$ is the supertype of every type. It can be the empty type, or unit type, which has only one kind of value. When a context wants a term of a top-equivalent (or top-like) type, it is viewed as that the context needs no information from the term. Therefore it is harmless for multiple terms in a merge match a top-like type at the same time, and we know every term matches it.

In this thesis, top-like types include certain form of function types, and function types are disjoint if their return types are. In contrast, a more restricted definition of top-like types only include $\top$ and intersections composed by $\top$. Consequently, a merge cannot contain more than one function. This is because we can always construct a common supertype $A_1 \,\&\, B_1 \to \top$ for any two function types $A_1 \to A_2$ and $B_1 \to B_2$. Note that such types are supertype of $\top$ in BCD-style subtyping.

2.4.2 Calculi with Disjoint Intersection Types

Via the disjointness restriction on types, ambiguous terms are avoided, therefore the original λ_i calculus [OSA16] obtains a type-safe and coherent elaboration semantics. That is, an expression corresponds to an unique result after translation and evaluation. Besides the introduction rule for merges, it also enforces disjointness via type well-formedness. Intersections are always restricted not to overlap. For example, $1 : \mathsf{Int} \,\&\, \mathsf{Int}$ is rejected by the type system. Because of this restriction, the proof of coherence in the original λ_i is still relatively simple.

Likewise, in the following work on the F_i calculus [AOS17], which extends λ_i with disjoint polymorphism that allows universal quantified types with a disjointness constraint, *all* intersections must be *disjoint*. However, the disjointness restriction causes difficulties because it breaks *stability of type substitutions*. Stability in a polymorphic type system ensures that if a polymorphic type is well-formed then any instantiation of that type is also well-formed. F_i can only prove a restricted version of stability, which makes its metatheory non-trivial.

Disjointness of all well-formed intersections is only sufficient but not necessary restriction to ensure an unambiguous semantics. The λ_i^+ calculus [BOS18] relaxes the restriction without introducing ambiguity. $1 : \mathsf{Int} \,\&\, \mathsf{Int}$ is allowed, as λ_i^+ employs the disjointness restriction *only* on merges, but otherwise allows *unrestricted intersections*. Unfortunately, this comes at a cost: it is much harder to prove the coherence of elaboration. Both λ_i^+ and F_i^+ [Bi+19] (a calculus derived from F_i that allows unrestricted intersections) deal with this problem by establishing coherence using contextual equivalence and a *logical relation* [Tai67; Plo73; Sta85]. The proof method, however, cannot deal with non-terminating programs. In fact, none of the existing calculi with disjoint intersection types supports recursion, which is a severe restriction.

Let us take a closer look of the coherence property, taking the elaboration semantics of the original F_i^+ as an example. Expressions in F_i^+ are translated into F_{co}, which has no subtyping or intersection types, and it has a conventional operational semantics. Coherence states that although the semantics depends on type derivation, all valid derivations produce equivalent results. For example, the two possible elaborations for the same F_i^+ source expression into F_{co} are contextually equivalent:

$$1 : \mathsf{Int} \,\&\, \mathsf{Int} : \mathsf{Int} \;\rightsquigarrow\; \mathsf{fst}\ (1, 1)$$
$$1 : \mathsf{Int} \,\&\, \mathsf{Int} : \mathsf{Int} \;\rightsquigarrow\; \mathsf{snd}\ (1, 1)$$

Two typing derivations lead to two elaborations in this example, which pick different sides of the merge. However, both elaborated expressions will be reduced to 1 eventually.

2.5 Application of the Merge Operator: CP Examples

Compositional Programming [ZSO21] is a recently proposed modular programming paradigm. CP is a prototype language for Compositional Programming. The semantics of CP and its notion of traits is defined via an elaboration to the core calculus F_i^+ [Bi+19]: a polymorphic core language with a merge operator and disjoint intersection types. The merge operator naturally enables a form of multiple inheritance, and a powerful form of dynamic inheritance where inherited implementations can be parameterized, by integrating record concatenation and subtyping.

Next we will introduce the basic reusable unit in CP: *traits*, which are usually type-checked against compositional interfaces. And we will show a modular solution to the *Expression Problem*, which illustrates some key features in Compositional Programming.

2.5.1 Typed First-Class Traits

Traits in Object-Oriented Programming provide a model of multiple inheritance. Both traits and *mixins* [BC90; FKF98] encapsulate a collection of related methods to be added to a class. The main difference between traits and mixins has to do with how conflicts are dealt with. Mixins use the order of composition to determine which implementation to pick in the case of conflicts. Traits require programmers to explicitly resolve the conflicts instead, and reject compositions with conflicts. In essence, this difference is closely related to the choice of a symmetric or asymmetric model for the merge operator. Symmetric merges with disjoint intersection types are closely related to traits because merges with conflicts are rejected, and the composition is associative and commutative (just like the composition for traits). Asymmetric merges are closer to mixins, giving preference to one of the implementations in the case of conflicts. We point the reader to Schärli et al. paper for an extensive discussion of the qualities of the trait model and a comparison with the mixin model [Sch+03].

The CP language supports *first-class traits* [BO18]. It essentially adopts the original trait model, but traits in CP are statically typed and support dynamic inheritance. The merge operator in F_i^+ is key for CP to model trait composition. Our examples next are adapted from Bi, Oliveira, and Schrijvers [BOS18].

A simple trait in CP is:

```
type Editor = {
  on_key : String → String;
  do_cut : String;
  show_help : String
};

type Version = {
```

```
  version : String
};

editor = trait [self: Editor & Version] ⇒ {
  on_key (key : String) = "Pressing " ++ key;
  do_cut                = on_key "C-x" ++ " for cutting text";
  show_help             = "Version: " ++ self.version ++ " Basic usage..."
};
```

A trait can be viewed as a function taking a self argument and producing a record. In this example, the record, which contains three fields, is encoded as a merge of three single field records. Because all the fields have distinct field names, the merge is disjoint and the definition is accepted. Methods in CP can be dynamically dispatched, as usual in OOP languages. For instance, in the trait `editor`, the `do_cut` method calls the `on_key` method via the self reference and it is dynamically dispatched. Moreover, traits in CP have a self type annotation similar to Scala [Ode+04]. In this example, the type of the self reference is the intersection of two record types `Editor` and `Version`. Note that `show_help` is defined in terms of an *undefined* `version` method. Usually, in a statically typed language like Java, an abstract method is required, making `editor` an abstract class. Instead, CP encodes abstract methods via self-types. The requirements stated by the type annotation of self must be satisfied when later composing `editor` with other traits, i.e. an implementation of the method `version` should be provided.

First-class traits and dynamic inheritance. The interesting features in CP are that traits are *first-class* and inheritance can be *dynamic*. The next example shows such features:

```
type Spelling = {check : String};

spell (base : Trait<Editor & Version ⇒ Editor>) = trait [self : Editor &
  Version] implements Editor inherits base ⇒ {
  override on_key (key : String) = "Process " ++ key ++ " on spell editor";
  check = super.on_key "C-c"  ++ " for spelling check"
};
```

The `spell` function takes a trait as an argument and returns a trait as a result. Thus, since traits can be passed as arguments and returned as results they are first-class (just like lambda functions in functional programming). The new trait adds a `check` method and overrides the `on_key` method of the `base` trait. The argument `base` is a trait of type `Trait<Editor & Version ⇒ Editor>`, where the two types denote trait requirements and functionality respectively. As we can see from its definition, trait `editor` matches that type. Note that unlike mainstream OOP languages like Java, the inherited trait (which would correspond to a superclass in Java) is *parameterized*, thus enabling dynamic inheritance. In CP the choice of the inherited trait (i.e.

the superclass) can happen at run-time, unlike in languages with static inheritance (such as Java or Scala).

Multiple inheritance. The following trait illustrates multiple inheritance in CP:

```
version = trait ⇒ {
   version = "0.2"
 };

 spell_editor = trait [self : Editor & Version & Spelling] inherits spell editor ,
   version ⇒ {}

 (new spell_editor).check --> "Pressing C-c for spelling check"
```

The trait `spell_editor` inherits from both `spell editor` and `version`. The latter defines an implementation for the field `version`. Finally an object `editor1` can be created from the trait `spell_editor`.

2.5.2 Expression Problem

To demonstrate the capabilities of Compositional Programming, we show how to solve a variant of the *Expression Problem* [Wad98] in the CP language. The Expression Problem is a classic challenge about the extensibility of a programming language. In the expression problem, a data type of expressions is defined, with several cases (literals and additions in the following code) associated with some operations (e.g. evaluation). There are two directions to extend the data type: adding a new case and adding a new operation. In a solution both extensions should be independently defined, and it should be possible to combine them to close the diamond. In a typical OOP language, class inheritance makes it easy to add a new case to the data type, while extending in the other direction in a modular and type-safe way remains hard.

Our solution is adapted from the original one by Zhang et al. [ZSO21]. In this variant, in addition to the usual challenge of extensibility in multiple directions, we also consider the problem of *context evolution* [LHJ95; SO11], so the interpreter may require different contextual information for different features of the interpreter.

Examples are based on a simple expression language, and the goal is to perform various operations over it, such as evaluation and free variable bookkeeping. The expression language consists of numbers, addition, variables, and let-bindings. Besides CP code, we also provide analogous Haskell code in the initial examples so that readers can connect them with existing concepts in functional languages.

Compositional interfaces First, we define the compositional interface for numeric literals and addition. The compositional interface at the top of Figure 2.2a is similar to Haskell's

```
type NumSig<Exp> = {
  Lit : Int → Exp;
  Add : Exp → Exp → Exp;
};

type Eval Ctx = { eval : Ctx → Int };
evalNum Ctx = trait implements NumSig<Eval
    Ctx> ⇒ {
  (Lit     n).eval _   = n;
  (Add e1 e2).eval ctx =
     e1.eval ctx + e2.eval ctx;
};
```

(a) CP code.

```
data Exp where
  Lit :: Int → Exp
  Add :: Exp → Exp → Exp


type Eval ctx = ctx → Int

eval :: Exp → Eval ctx
eval (Lit     n) _   = n
eval (Add e1 e2) ctx =
   eval e1 ctx + eval e2 ctx
```

(b) Haskell counterpart.

Figure 2.2: Initial expression language: numbers and addition.

algebraic data type at the top of Figure 2.2b. `Exp` is a special kind of type parameter in CP called a *sort*, which serves as the return type of both constructors `Lit` and `Add`. Sorts will be instantiated with concrete representations later. Internally, sorts are handled differently from normal type parameters [ZSO21]. In accordance with the compositional interface, we can then define how to evaluate the expression language.

Polymorphic contexts As shown in the middle of Figure 2.2a, the type `Eval` declares a method `eval` that takes a context and returns an integer. `Ctx` is a type parameter that can be instantiated later, enabling particular traits to assume particular contextual information for the needs of various features. The technique is called *polymorphic contexts* [ZSO21] in Compositional Programming.

Compositional traits The trait `evalNum` in Figure 2.2a is parametrized by a type parameter `Ctx`. Note that, in CP, type parameters always start with a capital letter, while regular parameters are lowercase. The trait `evalNum` implements the compositional interface `NumSig` by instantiating it with the sort `Eval Ctx`. In this trait, we use a lightweight syntax called *method patterns* to define how to evaluate different expressions. Such a definition is analogous to pattern matching in Figure 2.2b. Since `Lit` and `Add` do not need to be conscious of any information in the context, the type parameter `Ctx` is unconstrained. The only thing that we can do to the polymorphic context is either to ignore it (like in `Lit`) or to pass it to recursive calls (like in `Add`).

More expressions Adding more constructs to the expression language is awkward in Haskell because algebraic data types are *closed*. However, language components can be modularly declared in CP. Two new constructors, `Let` and `Var`, are declared in the second compositional interface `VarSig`, as shown in Figure 2.3. Then the two traits implement `VarSig` using method

```
type VarSig<Exp> = {
  Let : String → Exp → Exp → Exp;
  Var : String → Exp;
};

type Env = { env : String → Int };
evalVar (Ctx*Env) = trait implements VarSig<Eval (Env&Ctx)> ⇒ {
  (Let s e1 e2).eval ctx = e2.eval
    { ctx with env = insert s (e1.eval ctx) ctx.env };
  (Var       s).eval ctx = lookup s ctx.env;
};
```

Figure 2.3: Adding more expressions: variables and let-bindings.

patterns for the new constructors. Since the two new expressions need to inspect or update some information in the context, we expose the appropriate `Env` part to `evalVar`, while the remaining context is kept polymorphic. This is achieved with the *disjointness constraint* [AOS17] `Ctx*Env` in `evalVar`. A disjointness constraint denotes that the type parameter `Ctx` is disjoint to the type `Env`. In other words, types that instantiate `Ctx` cannot overlap with the type `Env`. Also note that the notation `{ ctx with env = ... }` denotes a *polymorphic record update* [CM91]. In the code for let-expressions, we need to update the environment in the recursive calls to extend it with a new entry for the let-variable.

Intersection types Independently defined interfaces can be composed using *intersection types*. For example, `ExpSig` below is an intersection of `NumSig` and `VarSig`, containing all of the four constructors:

```
type ExpSig<Exp> = NumSig<Exp> & VarSig<Exp>;
--               = { Lit : ...; Add : ...; Let : ...; Var : ... };
```

More operations Not only can expressions be modularly extended, but we can easily add more operations. In Figure 2.4, a new trait `fv` modularly implements a new operation that records free variables in an expression. Here, `union` and `delete` are two library functions for arrays. It is very similar to defining a new function by pattern matching in Haskell as before. The modular definition of `fv` is quite natural in functional programming, but it is hard in traditional object-oriented programming. We have to modify the existing class definitions and supplement them with a method. This is typical of the well-known Expression Problem. In summary, we have shown that Compositional Programming can solve both dimensions of this problem: adding expressions and operations.

Dependency injection Besides the Expression Problem, Figure 2.4 also shows another significant feature of CP: dependency injection. In `evalWithFV`, a new implementation of

```
type FV = { fv : [String] };
fv = trait implements ExpSig<FV> ⇒ {
  (Lit       n).fv = [];
  (Add   e1 e2).fv = union e1.fv e2.fv;
  (Let s e1 e2).fv = union e1.fv (delete s e2.fv);
  (Var       s).fv = [s];
};

evalWithFV (Ctx*Env) = trait implements ExpSig<FV ⇒ Eval (Env&Ctx)> ⇒ {
  (Lit       n).eval _   = n;
  (Add   e1 e2).eval ctx = e1.eval ctx + e2.eval ctx;
  (Let s e1 e2).eval ctx = if elem s e2.fv
    then e2.eval { ctx with env = insert s (e1.eval ctx) ctx.env }
    else e2.eval ctx;
  (Var       s).eval ctx = lookup s ctx.env;
};
```

Figure 2.4: Adding more operation: free variable bookkeeping and another version of evaluation.

evaluation is defined with a dependency on free variables. The method pattern for `Let` will check if `s` appears as a free variable in `e2`. If so, it evaluates `e1` first as usual; otherwise, we do not need to do any computation or update the environment since `s` is not used at all. Note that the compositional interface `ExpSig` is instantiated with two types separated by a fat arrow (⇒) (⇒ was originally denoted by `%` in Zhang et al.'s implementation of CP). `FV` on the left-hand side is the dependency of `evalWithFV`. In other words, the definition of `evalWithFV` depends on another trait that implements `ExpSig<FV>`. The static type checker of CP will check this fact later at the point of trait instantiation. With such dependency injection, we can call `e2.fv` even if `evalWithFV` does not have an implementation of `fv`. In other words, `evalWithFV` depends only on the interface of `fv` (the type `FV`), but not any concrete implementation.

Self-type annotations Before we show how to perform the new version of the evaluation over the whole expression language, we want to create a repository of expressions for later use. We expect that these expressions are unaware of any concrete operation, so we use a polymorphic `Exp` type to denote some abstract type of expressions. The code that creates the repository of expressions is[1]:

```
repo Exp = trait [self : ExpSig<Exp>] ⇒ {
  num = Add (Lit 4) (Lit 8);
  var = Let "x" (Lit 4) (Let "y" (Lit 8) (Add (Var "x") (Var "y")));
};
```

To make constructors available from the compositional interface, we add a *self-type annotation*

[1] In Zhang et al.'s original work [ZSO21], the `new` operator must be added before every constructor. But the new implementation [SDO22] implicitly insert `new`.

to the trait `repo`. The self type annotation `[self : ExpSig<Exp>]` imposes a requirement that the `repo` should finally be merged with some trait implementing `ExpSig<Exp>`. This requirement is also statically enforced by the static type checker of CP. This is the second mechanism in Compositional Programming to modularly inject dependencies.

Nested trait composition With the language components ready, we can compose them using the merge operator, which in the CP language is denoted as a single comma (`,`). First, we show how to compose the old version of the evaluation:

```
exp = new repo @(Eval Env) , evalNum @Env , evalVar @Top;
exp.var.eval { env = empty }  -⟶ 12
```

Since the context has evolved after we add variables, we pass different type arguments to the two traits to make the final context consistent. The final context type is `Env`, so we pass `Env` to `evalNum` and `Top` to `evalVar`. Type arguments are prefixed by `@` in CP. A more interesting example is to merge the new version of evaluation with free variable bookkeeping:

```
exp' = new repo @(Eval Env & FV) , evalWithFV @Top , fv;
exp'.var.eval { env = empty }  -⟶ 12
```

After the trait composition, both operations (`eval` and `fv`) are available for expressions that are built with the four constructors (`Lit`, `Add`, `Let`, and `Var`). Note that here `fv` satisfies the dependency of `evalWithFV`. If no implementation of the type `FV` is present in the composition, there would be a type error, since the requirement for `evalWithFV` would not be satisfied. The composition of the three traits is *nested* because the two methods nested in the four constructors are composed, as visualized in Figure 2.5.

This is allowed because with BCD-style subtyping [BCD83] type constructors, like functions or records distribute over intersections. Since traits in CP are in essence records of functions, such distributivity enables a view in Figure 2.5: components that are nested inside the traits being composed are themselves recursively composed.

With nested trait composition [BOS18], the Expression Problem is elegantly solved in Compositional Programming. Moreover, we allow context evolution using a relatively simple way with polymorphic contexts.

Impredicative polymorphism In ML-like type systems, there is usually a *predicativity* restriction imposed on polymorphism. It means that a type variable cannot be instantiated as a polymorphic type. While the original formulation of F_i^+ also has this restriction, our variant of F_i^+ supports *impredicative polymorphism*. Therefore, CP supports the creation of objects with polymorphic methods, similar to most OOP languages with generics where classes can contain polymorphic methods (like Java). For example:

```
type Poly = { id : forall A. A → A };
```

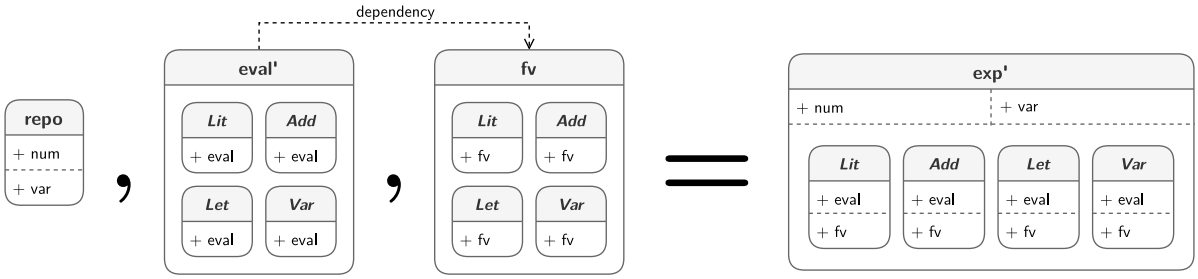

Figure 2.5: Visualization of nested composition.

```
idTrait = trait implements Poly ⇒ { id = ΛA. \(x:A) → x };

(new idTrait).id @Poly  -- impredicative
```

Part I

Distributive Subtyping

CHAPTER 3

Developing Subtyping Algorithms for Intersection Types with Distributivity

Subtyping relations for intersection types can vary in expressive power. Compared with the basic subtyping we have seen in last chapter, advanced relations usually include distributive rules. A typical example is the signature rule in the subtyping relation of Barendregt, Coppo, and Dezani-Ciancaglini [BCD83] (BCD subtyping).

$$\text{OS-distArr}\quad \frac{}{(A \to B) \,\&\, (A \to C) \leqslant A \to B \,\&\, C}$$

Here the function type constructor is distributive over intersection. In BCD-style subtyping, if more type constructors exist, intersections usually distribute over them as well, like record types. Especially, if union types are involved, there can be more distributivity rules, as we will see in the next chapter.

Although known to be decidable [KT95; RU11; Sta15], distributivity makes it hard to design a simple algorithm for BCD subtyping. In particular, in previous work [BOS18; Pie89; Bes+16; BRD19; Sie19], the distributivity rule leads to non-modular algorithmic formulations where many standard subtyping rules have to be changed due to distributivity.

We propose a modular and algorithmic BCD formulation. The key idea is to use the novel notion of *splittable types*, which are types that can be split into an intersection of two simpler types. We show basic properties of our formulation, including transitivity and inversion lemmas, and conclude that it is sound and complete with respect to the declarative BCD subtyping. Of particular interest is our transitivity proof. This proof is remarkably simple in comparison with other proofs in the literature due to a semantic characterization of types using splittable and ordinary types, which is used as the inductive argument for transitivity.

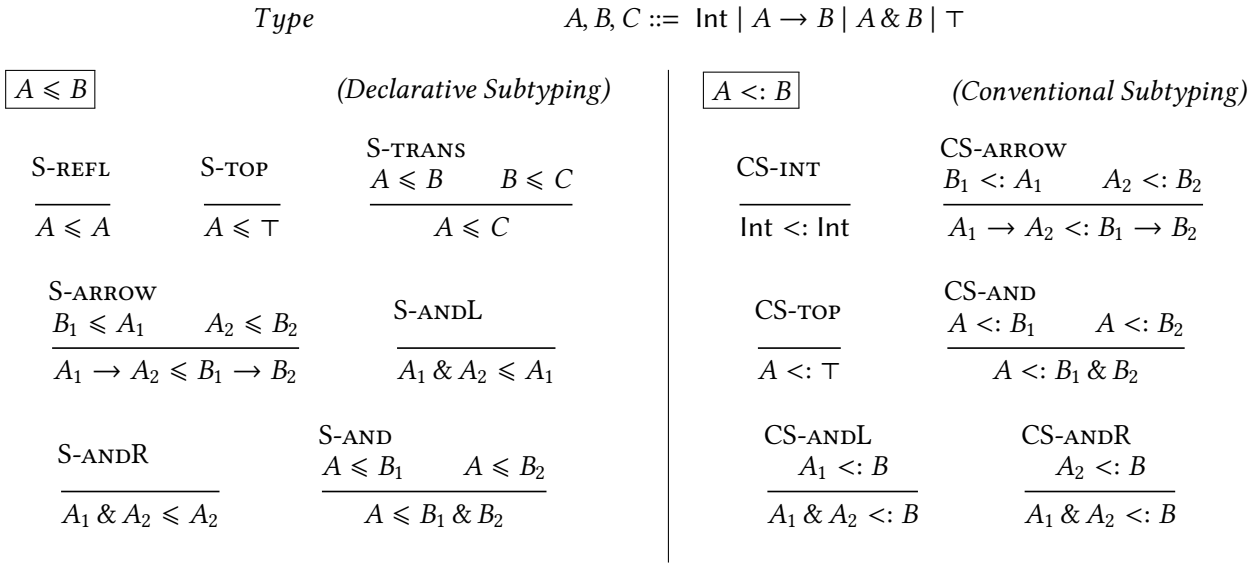

Figure 3.1: The conventional algorithmic subtyping with intersections (without distributivity), compared with the declarative rules.

3.1 Background: Transitivity Elimination

To design an algorithmic system for the declarative subtyping relation (discussed later in Section 7.2.2), the key challenge is to eliminate the transitivity rule. In this section, we first discuss how the explicit transitivity rule is avoided in the simple subtyping relation with intersection types and no distributivity, and then discuss where this approach fails when a distributive law is included. We follow the convention that intersections have a higher precedence than unions, and arrows have the lowest precedence.

3.1.1 Conventional Subtyping with Intersection Types

A subset of the BCD subtyping is shown at the left of Figure 3.1. After we exclude the two rules related to distributivity, it is equivalent to the simple subtyping on its right. It has a transitivity rule (S-TRANS). For any given goal $A \leqslant C$, there is no restriction and no clue for the intermediate type B. A direct implementation may enumerate all possible types, and never ends when the goal is unprovable. Especially, if it always tries S-REFL immediately after S-TRANS, B will be unified to be A, and the goal does not change. Such an infinite loop happens even for admissible goals.

Compared with the declarative rules, the algorithmic system, at the right of Figure 3.1, has no explicit transitivity rule, and the reflexivity rule is specialized to the primitive type Int. Reflexivity is straightforward to obtain for any form of types, including the top type, function types, and intersection types. In the remaining rules, only CS-ANDL and CS-ANDR are changed, which can be viewed as the rule S-ANDL and the rule S-ANDR with transitivity built-in. For

example, use S-andL to construct the first premise in S-trans, we can get CS-andL.

$$\textsc{S-trans}\ \dfrac{\textsc{S-andL}\ \dfrac{}{A_1 \,\&\, A_2 <: A_1} \qquad A_1 <: C}{A_1 \,\&\, A_2 <: C} \quad \Rightarrow \quad \textsc{CS-andL}\ \dfrac{A_1 <: C}{A_1 \,\&\, A_2 <: C}$$

In this case, such an algorithmic formulation follows from a common strategy for transitivity elimination: pushing transitivity into other rules.

However, one question still remains: why does not transitivity also extend to other positions like the following rule?

$$\textsc{S-trans}\ \dfrac{A <: C_1 \,\&\, C_2 \qquad \dfrac{\textsc{S-andL}}{C_1 \,\&\, C_2 <: C_1}}{A <: C_1} \quad \Rightarrow \quad \textsc{CS-andL-sub}\ \dfrac{A <: C_1 \,\&\, C_2}{A <: C_1}$$

The answer is because such a rule is admissible in the system: Rule CS-and is the only rule that can be used to prove the premise of rule CS-andL-sub, and it already implies the conclusion. Besides, adding rule CS-andL-sub would break the subformula property and lead to a non-algorithmic system.

3.1.2 Adding Distributivity: the Simple Approach to Transitivity Elimination Fails

Next we show the two rules that we omitted from BCD subtyping in Figure 3.1.

$$\textsc{OS-distArr}\ \dfrac{}{(A \to B) \,\&\, (A \to C) \leqslant A \to B \,\&\, C} \qquad \textsc{OS-topArr}\ \dfrac{}{\top \leqslant \top \to \top}$$

The second rule implies $\top \leqslant A \to \top$, as we have explained in Section 2.1.

Let us add rule OS-distArr back to the declarative system. Following the same method, it seems that we can obtain an algorithmic formulation by directly pushing transitivity into the rules. That is, by adding the following two rules.

$$\textsc{CS-distArrR-sub}\ \dfrac{C <: (A \to B_1) \,\&\, (A \to B_2)}{C <: A \to (B_1 \,\&\, B_2)} \qquad \textsc{CS-distArrR-super}\ \dfrac{A \to (B_1 \,\&\, B_2) <: C}{(A \to B_1) \,\&\, (A \to B_2) <: C}$$

Rule CS-distArrR-sub enables, for instance, arrows with a different input type to unify first,

and then applying distributivity on the result:

$$\textsc{cs-distArrR-sub}\ \dfrac{\textsc{cs-and}\ \dfrac{\textsc{cs-andL}\ \dfrac{\textsc{cs-arrow}\ \dfrac{\textsc{cs-andL}\ \dfrac{A_1 <: A_1}{A_1 \,\&\, A_2 <: A_1} \qquad B_1 <: B_1}{A_1 \to B_1 <: A_1 \,\&\, A_2 \to B_1}}{(A_1 \to B_1) \,\&\, (A_2 \to B_2) <: A_1 \,\&\, A_2 \to B_1} \qquad \dfrac{\ldots}{\ldots <: A_1 \,\&\, A_2 \to B_2}\ \textsc{cs-andR}}{(A_1 \to B_1) \,\&\, (A_2 \to B_2) <: (A_1 \,\&\, A_2 \to B_1) \,\&\, (A_1 \,\&\, A_2 \to B_2)}}{(A_1 \to B_1) \,\&\, (A_2 \to B_2) <: A_1 \,\&\, A_2 \to B_1 \,\&\, B_2}$$

For simplification, we end at reflexivity in the derivation above. Meanwhile, rule CS-distArrR-super can apply distributivity to nested arrows:

$$\textsc{cs-distArrR-super}\ \dfrac{\textsc{cs-arrow}\ \dfrac{A <: A \qquad \dfrac{B \to C_1 \,\&\, C_2 <: B \to C_1 \,\&\, C_2}{(B \to C_1) \,\&\, (B \to C_2) <: B \to C_1 \,\&\, C_2}\ \textsc{cs-distArrR-super}}{A \to (B \to C_1) \,\&\, (B \to C_2) <: A \to B \to C_1 \,\&\, C_2}}{(A \to B \to C_1) \,\&\, (A \to B \to C_2) <: A \to B \to C_1 \,\&\, C_2}$$

Unfortunately, that is not enough. Transitivity can extend to both sides of the distributivity rule together. Consequently, the declarative system (Figure 4.1 extended by rule S-distArrR) can apply distributivity to nested arrows with different input types, as the following example shows.

$$\begin{aligned}
&(A_1 \to B \to C_1) \,\&\, (A_2 \to B \to C_2) \\
&\leqslant (A_1 \,\&\, A_2 \to B \to C_1) \,\&\, (A_1 \,\&\, A_2 \to B \to C_2) && \text{by rule S-and, rule S-arrow, and other rules} \\
&\leqslant A_1 \,\&\, A_2 \to (B \to C_1) \,\&\, (B \to C_2) && \text{by rule S-distArrR} \\
&\leqslant A_1 \,\&\, A_2 \to B \to C_1 \,\&\, C_2 && \text{by rule S-arrow with rule S-distArrR}
\end{aligned}$$

An attempt at providing an equivalent algorithmic formulation with rule CS-distArrR-sub and rule CS-distArrR-super fails to accept such subtyping statement. The derived result only matches rule CS-andL and rule CS-andR, but both of them drop part of the subtype, making it impossible to reach the result. Without losing expressive power, we cannot extend the system directly in this way.

Ordinary type $A^\circ, B^\circ, C^\circ ::= \mathsf{Int} \mid \top \mid A \to B^\circ$

$\boxed{B \lhd A \rhd C}$ *(Splittable Types for BCD)*

$$\textsc{bcd-Sp-and}\ \frac{}{A \lhd A \,\&\, B \rhd B} \qquad \textsc{bcd-Sp-arrow}\ \frac{B_1 \lhd B \rhd B_2}{A \to B_1 \lhd A \to B \rhd A \to B_2}$$

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

$$\textsc{TL-top}\ \frac{}{\rceil \top \lceil} \qquad \textsc{TL-arrow}\ \frac{\rceil B \lceil}{\rceil A \to B \lceil} \qquad \textsc{TL-and}\ \frac{\rceil A \lceil \qquad \rceil B \lceil}{\rceil A \,\&\, B \lceil}$$

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

$$\textsc{bcd-AS-int}\ \frac{}{\mathsf{Int} <: \mathsf{Int}} \qquad \textsc{bcd-AS-top}\ \frac{\rceil B^\circ \lceil}{A <: B^\circ} \qquad \textsc{bcd-AS-arrow}\ \frac{B_1 <: A_1 \qquad A_2 <: B_2^\circ}{A_1 \to A_2 <: B_1 \to B_2^\circ}$$

$$\textsc{bcd-AS-and}\ \frac{B_1 \lhd B \rhd B_2 \qquad A <: B_1 \qquad A <: B_2}{A <: B} \qquad \textsc{bcd-AS-andL}\ \frac{A_1 <: B^\circ}{A_1 \,\&\, A_2 <: B^\circ} \qquad \textsc{bcd-AS-andR}\ \frac{A_2 <: B^\circ}{A_1 \,\&\, A_2 <: B^\circ}$$

Figure 3.2: The algorithmic BCD subtyping with intersection types and distributivity.

3.2 A Simple and Modular Formulation of BCD with Splittable Types

Luckily there are other ways to interpret the distributivity rules. In this section, we will present the algorithmic formulation in Figure 3.2. It is equivalent to the declarative rules for the conventional subtyping in Figure 3.1 plus the two distributivity rules (OS-distArr and OS-topArr). As a step towards transitivity elimination, we decompose any type A that is equivalent to an intersection type $B \,\&\, C$ into two conjuncts B and C. If such treatment is possible, we call A *splittable*; otherwise A is *ordinary*.

Ordinary types. Conventionally, *ordinary types* are types that are not intersections, including all function types [DP00]. When there is no distributivity, an intersection is a subtype of an ordinary type only if an ordinary component of the intersection is a subtype of the ordinary type. Therefore the subtyping rules can be viewed as two parts: one tears intersection types apart; the other only deal with ordinary types. However, in the presence of rule OS-distArr,

$(A \to B_1) \,\&\, (A \to B_2)$ is a subtype of $A \to B_1 \,\&\, B_2$ while neither part of it is a subtype: that is both $A \to B_1 \leqslant A \to B_1 \,\&\, B_2$ and $A \to B_2 \leqslant A \to B_1 \,\&\, B_2$ do not hold in general. To fix the broken property and maintain the boundary of intersection types and ordinary types, we tighten the definition: a function type is ordinary only if its return type is ordinary. Types like $A \to B_1 \,\&\, B_2$ are treated like intersections.

Splittable types. The type splitting relation, also shown in Figure 3.2, can be viewed as taking an input type A, and returning two types B and C, such that $B \,\&\, C$ is equivalent to A. The relation extends the decomposition of intersections to types that are equivalent to intersections via distributivity rules. According to rule OS-distArr, we know that an arrow type can split if its result type can (rule bcd-Sp-arrow). The other direction $A \to B_1 \,\&\, B_2 \leqslant (A \to B_1) \,\&\, (A \to B_2)$ is derivable even without the distributive law. The following lemma provides some justification for type splitting. It is proven by a routine induction on the splittable premise.

Lemma 3.1 (Type splitting loses no information)**.** If $B \lhd A \rhd C$ then $A \equiv B \,\&\, C$.

Three important properties related to ordinary and splittable types are:

Lemma 3.2 (Ordinary types do not split)**.** For any ordinary type A, A is not splittable.

Lemma 3.3 (Types are ordinary or splittable)**.** For any type A, either A is ordinary or A is splittable, and it is decidable.

Lemma 3.4 (Splitting is deterministic)**.** For any splittable type A, if $B_1 \lhd A \rhd C_1$ and $B_2 \lhd A \rhd C_2$, then $B_1 = B_2$ and $C_1 = C_2$.

The modular BCD subtyping algorithm The main idea for the algorithmic formulation of subtyping, shown at the bottom of Figure 3.2, is that the right-hand side type B keeps splitting until it becomes ordinary. When it splits, rule bcd-AS-and is applied, which works in the same way as rule CS-and when B is an intersection type. The most interesting case is when B is a splittable function type, for example, $B := B_1 \to B_2 \,\&\, B_3$. Type B can be split into $B_1 \to B_2$ and $B_1 \to B_3$. Therefore, the premises of $A <: B$ are $A <: B_1 \to B_2$ and $A <: B_1 \to B_3$, or equivalently, $A <: (B_1 \to B_2) \,\&\, (B_1 \to B_3)$. Thus we are able to conclude $A <: B$ with a combination of rule S-trans and rule S-distArrR in declarative subtyping. That is to say, while rule bcd-AS-and combines rule S-trans and rule S-and, it also takes rule S-distArrR into consideration implicitly. The ordinary-type conditions eliminate some overlapping between the rules: we can see that when B is splittable, only rule bcd-AS-and can be applied, since rule bcd-AS-andL and rule bcd-AS-andR require B to be ordinary. However, dropping such conditions does not alter the expressive power: it leads to an equivalent system (but with more overlapping).

The previous failed example can now be derived, as its right-hand side type, although not matched by rule S-distArrR, is captured by type splitting. Due to space limitations, we

omit the type $(A_1 \to B \to C_1) \,\&\, (A_2 \to B \to C_2)$, which is unchanged across the application of rule BCD-AS-AND, in its premises. We also employ the rules without the ordinary-type conditions in the derivation for simplification. The main derivation is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{BCD-AS-ARROW}\ \dfrac{\cdots}{A_1 \to B \to C_1 <: A_1 \,\&\, A_2 \to B \to C_1}}{\text{BCD-AS-ANDL}\ \ldots <: A_1 \,\&\, A_2 \to B \to C_1} \quad D \quad \dfrac{\text{BCD-AS-ARROW}\ \dfrac{\cdots}{\ldots \to C_2 <: \ldots \to C_2}}{\ldots <: \ldots \to C_2}\ \text{BCD-AS-ANDR}}{\text{BCD-AS-AND}\ (A_1 \to B \to C_1) \,\&\, (A_2 \to B \to C_2) <: A_1 \,\&\, A_2 \to B \to C_1 \,\&\, C_2}$$

The missing subderivation D for type splitting is:

$$\dfrac{\dfrac{\text{BCD-SP-AND}\ \dfrac{}{C_1 \lhd C_1 \,\&\, C_2 \rhd C_2}}{\text{BCD-SP-ARROW}\ B \to C_1 \lhd B \to C_1 \,\&\, C_2 \rhd B \to C_2}}{\text{BCD-SP-ARROW}\ A_1 \,\&\, A_2 \to B \to C_1 \lhd A_1 \,\&\, A_2 \to B \to C_1 \,\&\, C_2 \rhd A_1 \,\&\, A_2 \to B \to C_2}$$

Top-like types. As suggested by its name, a top-like type is both a supertype and a subtype of $\top$. They can be easily distinguished by the rules in the middle of Figure 3.2. Besides $\top$, top-like types contain intersection types like $\top \,\&\, \top$. Notably, rule TL-ARROW allows arrow types to be top-like when their return types are top-like. This enlargement is to cope with rule OS-TOPARR. Rule BCD-AS-TOP says that a top-like type is a supertype of any type. It can be justified by the following theorem.

Theorem 3.1 (BCD-AS-TOP in declarative BCD)**.** If $\rceil B \lceil$, then $A \leqslant B$.

3.2.1 Modularity

A more declarative (and modular) formulation of subtyping is to omit each ordinary-type condition in Figure 3.2. Note that here we employ the term "modularity" to mean that existing subtyping rules do not need to be changed because of a new feature (in this case distributivity).

Our first observation is that omitting the ordinary-type conditions does not change expressiveness.

Lemma 3.5 (Modular BCD-AS-TOP)**.** If $\rceil B \lceil$, then $A <: B$.

Lemma 3.6 (Modular BCD-AS-ARROW)**.** If $B_1 <: A_1$, $A_2 <: B_2$, then $A_1 \to A_2 <: B_1 \to B_2$.

Lemma 3.7 (Modular BCD-AS-ANDL)**.** If $A <: C$, then $A \,\&\, B <: C$.

Lemma 3.8 (Modular BCD-AS-ANDR)**.** If $B <: C$, then $A \,\&\, B <: C$.

With these lemmas, the two formulations (with and without ordinary-type conditions) are proved to be sound and complete with respect to each other. Thus, compared to the conventional simple subtyping relation at the right of Figure 3.1, the modular BCD subtyping relation only replaces rule CS-AND by rule BCD-AS-AND, and replaces rule CS-TOP by rule BCD-AS-TOP to enable BCD distributivity. The new subtyping rules generalize the previous ones.

It is possible to have an equivalent alternative approach for adding BCD distributivity (rule BCD-AS-AND) without modifying the existing rules. One just needs to keep the old rule CS-AND and add rule BCD-AS-AND-ALT:

$$\frac{A <: B_1 \qquad A <: B_2}{A <: B_1 \,\&\, B_2}\ \text{CS-AND} \qquad \frac{C_1 \lhd B_2 \rhd C_2 \qquad A <: B_1 \to C_1 \qquad A <: B_1 \to C_2}{A <: B_1 \to B_2}\ \text{BCD-AS-AND-ALT}$$

Similarly, an alternative to rule BCD-AS-TOP and the top-like relation ($\lceil A \rceil$) is to use the following two rules:

$$\frac{}{A <: \top}\ \text{CS-TOP} \qquad \frac{\top <: C}{A <: B \to C}\ \text{BCD-AS-TOP-ALT-ARR}$$

The first rule is just the standard rule for top types, while the second rule is a special rule which deals with top-like function types.

Both alternative approaches replace one rule in our modular subtyping relation by two, while keeping the expressiveness of subtyping unchanged. Rule BCD-AS-AND and rule BCD-AS-TOP in our modular BCD subtyping are generalizations of the designs that would use 2 rules instead, which is why we choose them to be in our system.

It is also worth mentioning that our algorithmic relation keeps the simple judgment form $A <: B$, thus the system is easier to extend with orthogonal features, which have been presented with a subtyping relation of that form. Some BCD subtyping formulations require a different form to the subtyping relation [BOS18; Pie89; Bes+16; BRD19].

3.3 Metatheory of Modular BCD

A benefit of our new formulation of BCD subtyping is that the metatheory is remarkably simple. The metatheory of BCD subtyping has been a notoriously difficult topic of research.

Inversion lemmas. Given that our algorithmic relations are not entirely syntax-directed, several inversion lemmas indicate that the algorithm and the declarative system behave similarly.

Lemma 3.9 (Inversion on left split)**.** If $A <: B^\circ$ and $A_1 \lhd A \rhd A_2$ then $A_1 <: B^\circ$ or $A_2 <: B^\circ$.

$\boxed{\vdash_{\&} A}$ *(Proper Types)*

$$\frac{}{\vdash_{\&} \mathsf{Int}}\ \text{RTY-INT} \qquad \frac{}{\vdash_{\&} \top}\ \text{RTY-TOP} \qquad \frac{\vdash_{\&} A \qquad \vdash_{\&} B^{\circ}}{\vdash_{\&} A \to B^{\circ}}\ \text{RTY-ORDFUN} \qquad \frac{B \lhd A \rhd C \qquad \vdash_{\&} B \qquad \vdash_{\&} C}{\vdash_{\&} A}\ \text{RTY-SPLIT}$$

Figure 3.3: Proper types.

Lemma 3.10 (Inversion on right split)**.** If $A <: B$ and $B_1 \lhd B \rhd B_2$ then $A <: B_1$ and $A <: B_2$.

Both lemmas are easily proven by induction on the subtyping premises.

Transitivity. Since the transitivity rule is eliminated in algorithmic systems, we need to show that the transitivity lemma holds. This property is critical but difficult for any BCD formulation without the transitivity axiom built-in.

Lemma 3.11 (Transitivity of modular BCD)**.** If $A <: B$ and $B <: C$ then $A <: C$.

To prove the transitivity lemma, one might try at first to proceed by induction on B. However, that does not succeed, since our algorithm is not entirely syntax-directed. In particular, the behavior of the subtyping algorithm is determined by whether the type on the right is ordinary or splittable (but not simply the syntax form of the type). For example, in the case where $A <: B$ is derived by rule BCD-AS-AND, B can be split into two parts $B_1 \lhd B \rhd B_2$, yet B_1 and B_2 cannot be applied to the induction hypothesis, simply because they may not be components of type B. On the other hand, assuming one does induction on the two subtyping derivations, it is a tricky case when $A <: B$ is derived by rule BCD-AS-AND and $B <: C$ is derived by rule BCD-AS-ARROW. The former splits type B while the latter decomposes it as a function type, and they do not match.

To overcome this problem we would like to treat any splittable type similarly to an intersection type [1]. Therefore, we need a proper characterization of the type structure, so that the induction hypothesis on splittable types is always as desired. The relation defined in Figure 3.3 defines the so-called *proper types*. Proper types act as an alternative inductive definition for types, distinguishing types based on whether they are ordinary or splittable. The following lemma shows that the definition is general: any type is a proper type.

Lemma 3.12 (Types are proper types)**.** For any type A, $\vdash_{\&} A$.

With the new definition for types, we are ready to prove the transitivity lemma. Induction is performed on the relation $\vdash_{\&} B$ which is obtained easily on type B through Lemma 3.12. The induction then breaks into several cases:

[1] An alternative approach is to define a size measure for types and do induction on the sum of the sizes of types.

- Int and $\top$ are easy base cases.
- When B is a function type constructed by rule RTY-ORDFUN, a nested induction on the premise $B <: C$ gives three sub-cases.
  - Sub-cases rule BCD-AS-TOP and rule BCD-AS-AND are easy to prove by induction hypothesis.
  - Sub-case rule BCD-AS-ARROW is then able to finish by another nested induction on the other premise $A <: B$.
- The last case is when B is a splittable type ($B_1 \lhd B \rhd B_2$), where we know that $A <: B_1$ and $A <: B_2$ by Lemma 3.10. Let us do induction on $\vdash_{\&} C$.
  - If C is an ordinary type, by Lemma 3.9, either $B_1 <: C$ or $B_2 <: C$ holds. In both cases, applying the induction hypothesis of B finishes the proof.
  - Otherwise, assuming $C_1 \lhd C \rhd C_2$, we apply Lemma 3.10 to $B <: C$ and get $B <: C_1$ and $B <: C_2$. Via the induction hypothesis of C we can obtain $A <: C_1$ and $A <: C_2$ and reach the goal by rule BCD-AS-AND.

Equivalence to declarative BCD. Thanks to the simple judgment form used in our algorithm, the soundness and completeness theorems are stated directly as follows.

Theorem 3.2 (Soundness of modular BCD)**.** If $A <: B$ then $A \leqslant B$.

The soundness theorem only relies on Lemma 3.1 and Theorem 3.1.

The completeness theorem is also easy to show with the help of transitivity (Lemma 3.11), by induction on the premise.

Theorem 3.3 (Completeness of modular BCD)**.** If $A \leqslant B$ then $A <: B$.

To sum up, our novel formulation of BCD subtyping adds the function distributivity feature in a modular way, and the metatheory is straightforward to establish with the notion of proper types.

3.4 Implementation

Finally, in Figure 3.4, we present a Haskell implementation of the subtyping rules in Figure 3.2. We model types as:

```
data Type = TInt | TTop | TArrow Type Type | TAnd Type Type
```

```
-- ordinary type
ordinary :: Type → Bool
ordinary a = split a == Nothing

-- split type
split :: Type → Maybe (Type, Type)
split (TAnd a b) = Just (a, b)                                  -- Bsp-and
split (TArrow a b)                                              -- Bsp-arrow
  | Just (b1, b2) <- split b
  = Just (TArrow a b1, TArrow a b2)
  split _ = Nothing

-- check whether the given type is top-like
checkTopLike :: Type → Bool
checkTopLike TTop         = True                                  -- TL-top
checkTopLike (TArrow a b) = checkTopLike b                        -- TL-arrow
checkTopLike (TAnd a b)   = checkTopLike a && checkTopLike b  -- TL-and
checkTopLike _            = False

-- subtyping
checkSub :: Type → Type → Bool
checkSub TInt TInt = True                       -- bcd-AS-int
checkSub a b
  | checkTopLike b == True = True               -- bcd-AS-top
  | Just (b1, b2) <- split b                    -- bcd-AS-and
  = checkSub a b1 && checkSub a b2
checkSub (TAnd a1 a2) b                         -- bcd-AS-andL bcd-AS-andR
  = checkSub a1 b || checkSub a2 b
checkSub (TArrow a1 a2) (TArrow b1 b2)          -- bcd-AS-arrow
  = checkSub b1 a1 && checkSub a2 b2
checkSub _ _ = False
```

Figure 3.4: Haskell implementation of BCD subtyping.

A type is either ordinary or splittable. The `split` function in Figure 3.4 is based on the definition of the type splitting relation. It returns the split results if the input type is splittable. The `ordinary` function makes use of the fact that the set of ordinary types is complementary to the set of splittable types. It can also be implemented by analyzing the form of the type.

The `checkSub` function takes two types and decides whether the first input is a subtype of the second one. The first two cases correspond to rule BCD-AS-INT and rule BCD-AS-TOP. Then the case corresponding to rule BCD-AS-AND handles all cases of which the second input is splittable. Since the code executes sequentially, the second input is guaranteed to be ordinary after that. Following are the cases corresponding to rule BCD-AS-ANDL and rule BCD-AS-ANDR. When the first input is an intersection type, it is necessary to try both rules before returning `False`. In the end, both types must be arrow types and must satisfy rule BCD-AS-ARROW if the subtyping holds.

Summary In this chapter we presented a subtyping algorithm design that uses a type-splitting operation to convert types to an equivalent intersection type. The subtyping algorithm uses type splitting whenever an intersection type is expected in the conventional algorithm for subtyping without distributivity, and therefore handles distributivity smoothly and modularly. Later we will extend this algorithmic formulation to record types for the λ_i^+ calculus (Chapter 6) and to universal quantified types with disjoint polymorphism for the F_i^+ calculus (Chapter 7).

CHAPTER 4

Subtyping with Union Types

In this chapter, we will see how the ideas of splittable types can be extended into a more complex setting with union types and additional distributivity rules. Then, using another technique called duotyping, we will exploit the fundamental dualities between intersection and union types to further unify the rules in the system. In the end, we will present a compact functional implementation of the subtyping algorithm. Along the way, we show various results regarding the metatheory of the system, including soundness, completeness and decidability.

4.1 Overview

Subtyping relations with intersections and unions have deep connections to logic, which follow from the *Curry-Howard isomorphism* [How80]. Types can be interpreted as propositions: intersections are interpreted as conjunctions, unions as disjunctions and functions types as implications. Furthermore, from the perspective of logic, the subtyping problem is essentially the problem of determining *logical entailment*: does a logical statement follow from another one? Where in logic one may write $P \vdash Q$ for logical entailment, with subtyping one writes $P <: Q$ to denote that Q is a supertype of (or follows from) P. Naturally, algorithms for deciding logical entailment have applications to other areas, such as theorem proving [Sto19]. One particular subtyping relation of practical interest for programming languages is closely related to the basic positive logic **B+** of Routley and Meyer [RM72]. The connection to programming languages is due to Bakel et al. [Bak+00], who have shown a type assignment system that corresponds to the **B+** logic. Logic **B+** is also called the *minimal relevant logic* since it is the minimal (or the weakest) relevant logic system that is complete for the Routley-Meyer ternary relational semantics. In the minimal relevant logic **B+**, there are two axioms that can be interpreted as the two distributivity rules above. Languages that have some form of distributivity rules include Ceylon, ℂDuce, Julia and Scala 3 (or Dotty). The subtyping relations used in Ceylon and ℂDuce, for instance, include all the rules of the minimal relevant logic.

In this chapter, we continue with the idea of *splittable types*. With it extended to union types, we obtain a novel algorithmic formulation of a powerful subtyping relation with union and intersection types that is based on the minimal relevant logic. Unlike many normalization-based algorithms, our new subtyping algorithm works directly on source types. In other words, there is only one step in our algorithm without any pre-processing phase. Besides, we employ another recent idea *duotyping* [OCR20], which provides a generalization of subtyping with a mode. This mode allows exploiting fundamental dualities between union and intersection types and their subtyping rules. This leads to a consistent and symmetrical design for the rules, and also benefits both metatheory and implementation. We will first present the subtyping based on minimal relevant logic then we show how it is derived straightforwardly from the duotyping formulation.

4.2 Subtyping based on Minimal Relevant Logic

In this section, we show two equivalent subtyping relations (one declarative and another algorithmic) for a variant of minimal relevant logic subtyping [RM72; Bak+00]. The main novelty over the BCD subtyping relation presented in Chapter 3 are the addition of union types, and extra distributivity rules. We show that the idea of splittable types smoothly extends to deal with those features. The algorithmic formulation is derived from a formulation based on duotyping that will be presented in Section 4.3, but the presentation in this section is understandable independently of duotyping.

4.2.1 Declarative subtyping

The declarative subtyping rules in Figure 4.1 extend the rules on the left of Figure 3.1. The addition of union types ($A \mid B$) and the bottom type ($\bot$) brings the following new rules.

1. Rule S-BOT, similarly to rule S-TOP, defines the bottom type as the lowermost bound among types. $\bot$ has no inhabited values. In other words, it is the empty type from the set-theoretic view of types. In a system where arrow types are interpreted as logical implications, the bottom type can be used to encode negation as $A \to \bot$.

2. Rule S-OR, rule S-ORL, and rule S-ORR define basic subtyping for unions, similarly to the three intersection rules. In a language with union types, a term of type A can be transformed into any union type containing A. From the point of view of proof theory, having the proof of either A or B is enough to construct a proof of $A \mid B$. That is to say, a union type is a supertype of its components. Moreover, a union type is a subtype of some type if both its components are subtypes of that type.

$$\textit{Type} \qquad A, B, C ::= \mathsf{Int} \mid A \to B \mid A \,\&\, B \mid (A \mid B) \mid \top \mid \bot$$

$\boxed{A \leqslant B}$ *(Declarative Subtyping Extension)*

$$\textsc{S-bot}\ \frac{}{\bot \leqslant A} \qquad \textsc{S-or}\ \frac{A_1 \leqslant B \qquad A_2 \leqslant B}{A_1 \mid A_2 \leqslant B} \qquad \textsc{S-orL}\ \frac{}{B_1 \leqslant B_1 \mid B_2} \qquad \textsc{S-orR}\ \frac{}{B_2 \leqslant B_1 \mid B_2}$$

$$\textsc{S-distArrR}\ \frac{}{(A \to B_1) \,\&\, (A \to B_2) \leqslant A \to B_1 \,\&\, B_2} \qquad \textsc{S-distArrR-rev}\ \frac{}{A \to B_1 \,\&\, B_2 \leqslant (A \to B_1) \,\&\, (A \to B_2)}$$

$$\textsc{S-distArrL}\ \frac{}{(A_1 \to B) \,\&\, (A_2 \to B) \leqslant A_1 \mid A_2 \to B} \qquad \textsc{S-distArrL-rev}\ \frac{}{A_1 \mid A_2 \to B \leqslant (A_1 \to B) \,\&\, (A_2 \to B)}$$

$$\textsc{S-distOr}\ \frac{}{(A_1 \mid B) \,\&\, (A_2 \mid B) \leqslant (A_1 \,\&\, A_2) \mid B} \qquad \textsc{S-distAnd}\ \frac{}{(A_1 \mid A_2) \,\&\, B \leqslant (A_1 \,\&\, B) \mid (A_2 \,\&\, B)}$$

Figure 4.1: Declarative subtyping rules (extends the left part of Figure 3.1).

3. The remaining six rules are related to the distributivity of intersections and unions over other constructs. Rule S-distArrR is part of the BCD subtyping. Along with rule S-distArrL, the two rules distribute arrows over intersections and unions, respectively. These two rules have two corresponding reversed rules (rule S-distArrR-rev and rule S-distArrL-rev). Note that the latter two rules are not necessary since they can be derived from other rules, but we present them here because in Section 4.3.1 they will play a role in our reformulation of the subtyping relation using *duotyping* [OCR20]. Rule S-distArrR-rev and rule S-distArrL-rev, in combination with the previous two rules, illustrate that the two types in the subtyping relation are isomorphic (i.e. they are subtypes of each other).

4. The interaction between intersection types and union types is described by the rule S-distOr and the rule S-distAnd. They can distribute over each other. The reversed rules are derivable and therefore omitted. To be noted, it would not affect the whole system to drop one of the two rules (either rule S-distOr or rule S-distAnd): in the presence of one of the two rules, the other rule can be derived from the other subtyping rules.

Remark It is should be noted that, in combination with transitivity, more general subtyping rules become derivable. For instance, an intersection of any two arrow types has a supertype

that is a combination of them:

$$\text{S-distArr-gen}\quad \frac{}{(A_1 \to A_2) \,\&\, (B_1 \to B_2) \leqslant A_1 \,\&\, B_1 \to A_2 \,\&\, B_2}$$

In other words, rule S-distArr-gen is not restricted to types that share the same input type (as the rule S-distArrR). Similarly, the subtype of an intersection of two arrow types can be obtained from an arrow type that takes the union of the input types of the arrow types:

$$\text{S-distArr-rev-gen}\quad \frac{}{A_1 \mid B_1 \to A_2 \,\&\, B_2 \leqslant (A_1 \to A_2) \,\&\, (B_1 \to B_2)}$$

4.2.2 Algorithmic Subtyping: Adding Union Types and More Distributivity

Now we are ready to move on to the design of algorithmic subtyping for Figure 4.1. We first cover the extended definitions of ordinary and splittable types here, in which the initial definition is revised, and a dual version is defined for union types.

Intersection-ordinary and intersection-splittable types With union types and the bottom type taken into consideration, we need to revise the previous definitions. Firstly we rename them as *intersection-ordinary types* (A°, top of Figure 4.2) and *intersection-splittable types* ($B \lhd A \rhd C$, in the middle of Figure 4.2). Originally, "ordinary" was used to describe the lack of "intersections", and "splittable" meant a type that has two parts connected by an intersection. Now we make this explicit in the names. Intersection-ordinary types include (non-splittable) union types, but exclude intersection types at the top level (although intersection types can appear in some nested positions inside the types). In contrast, intersection-splittable types include top-level intersections and some union types. Mainly, there are four changes in the new definition.

1. The bottom type is ordinary like the top type.
2. Due to the distributivity of union over intersections (rule S-distOr and rule S-distAnd), some unions are also isomorphic to intersections, for example the type $(A_1 \,\&\, A_2) \mid B$ is isomorphic to $(A_1 \mid B) \,\&\, (A_2 \mid B)$. That is to say, the former can be split into $A_1 \mid B$ and $A_2 \mid B$. Splitting the union type $A \mid B$, first tries to split A by rule SpI-orL, and only moves to B if A cannot be split (rule SpI-orR). Thus only unions whose components are both ordinary can be treated as ordinary types.
3. Rule SpI-arrowL is brought by rule S-distArrL and rule S-distArrL-rev: if its input type is a union, an arrow type can be converted into an intersection type, and therefore it

$$
\begin{array}{lr}
\text{Intersection-Ordinary Types} & A^\circ, B^\circ, C^\circ ::= \mathsf{Int} \mid \top \mid \bot \mid A^\bullet \to B^\circ \mid (A^\circ \mid B^\circ) \\
\text{Union-Ordinary Types} & A^\bullet, B^\bullet, C^\bullet ::= \mathsf{Int} \mid \top \mid \bot \mid A \to B \mid A \,\&\, B \\
\text{Double-Ordinary Types} & A^{\circ\bullet}, B^{\circ\bullet}, C^{\circ\bullet} ::= \mathsf{Int} \mid \top \mid \bot \mid A^\bullet \to B^\circ
\end{array}
$$

$\boxed{B \lhd A \rhd C}$ *(Intersection-Splittable Types)*

$$
\textsc{SpI-and}\ \frac{}{A \lhd A \,\&\, B \rhd B} \qquad
\textsc{SpI-arrowR}\ \frac{B_1 \lhd B \rhd B_2}{A \to B_1 \lhd A \to B \rhd A \to B_2} \qquad
\textsc{SpI-arrowL}\ \frac{A_1 \blacktriangleleft A \blacktriangleright A_2}{A_1 \to B^\circ \lhd A \to B^\circ \rhd A_2 \to B^\circ}
$$

$$
\textsc{SpI-orL}\ \frac{A_1 \lhd A \rhd A_2}{A_1 \mid B \lhd A \mid B \rhd A_2 \mid B} \qquad
\textsc{SpI-orR}\ \frac{B_1 \lhd B \rhd B_2}{A^\circ \mid B_1 \lhd A^\circ \mid B \rhd A^\circ \mid B_2}
$$

$\boxed{B \blacktriangleleft A \blacktriangleright C}$ *(Union-Splittable Types)*

$$
\textsc{SpU-or}\ \frac{}{A \blacktriangleleft A \mid B \blacktriangleright B} \qquad
\textsc{SpU-andL}\ \frac{A_1 \blacktriangleleft A \blacktriangleright A_2}{A_1 \,\&\, B \blacktriangleleft A \,\&\, B \blacktriangleright A_2 \,\&\, B} \qquad
\textsc{SpU-andR}\ \frac{B_1 \blacktriangleleft B \blacktriangleright B_2}{A^\bullet \,\&\, B_1 \blacktriangleleft A^\bullet \,\&\, B \blacktriangleright A^\bullet \,\&\, B_2}
$$

Figure 4.2: Ordinary and splittable types.

is not ordinary. In the ordinary rule for arrow types, the new version has more restrictions: the input type of an intersection-ordinary arrow type must not be a union-like type, i.e. it is *a union-ordinary type* (Figure 4.2).

4. To be noted, a side condition B° is added in rule SpI-arrowL to ensure that the relation can be used as a deterministic function, like in rule SpI-orR. Although we prioritize rule SpI-orL and rule SpI-arrowR here, we believe some other arrangements are also feasible.

Union-ordinary and union-splittable types Figure 4.2 also presents the definition of union-ordinary types ($A^\bullet$) and union-splittable types ($B \blacktriangleleft A \blacktriangleright C$). A union-splittable type is isomorphic to the union of its split results, and union-ordinary types are those that cannot be split. Such a definition is almost the dual of the intersection- rules: just exchange intersection and union, and switch intersection- and union- judgments.

The key difference is that no arrow types are union-splittable and they are all union-ordinary. Correspondingly, splitting union types lacks arrow-related rules, and all function types are union-ordinary. The source of the difference is that both of the two distributivity

arrow rules in the declarative system (rule S-distArrR and rule S-distArrL) relate arrow types with intersection types but not union types. To have an exact dual, the union-ordinary and union-splittable types definitions would lead to the following two rules:

$$\text{S-distArrR-union}\quad \frac{}{A \to B_1 \mid B_2 \leqslant (A \to B_1) \mid (A \to B_2)} \qquad \text{S-distArrL-union}\quad \frac{}{A_1 \,\&\, A_2 \to B \leqslant (A_1 \to B) \mid (A_2 \to B)}$$

However, such rules do not lead to valid coercions from the coercive subtyping point of view where unions are interpreted as sums and intersections are interpreted as products. If we added these two rules, then some arrow types could be union-splittable.

While a type must be either intersection- (union-) ordinary or intersection- (union-) splittable, the two sets of ordinary (and splittable) definitions are overlapping. We use $A^{\circ\bullet}$ to denote types that are both intersection-ordinary and union-ordinary. The following table presents some examples of each of the four kinds of types:

	Intersection-	Union-
Ordinary	Int, $\top$, $\bot$	Int, $\top$, $\bot$
	$\mathsf{Bool} \,\&\, \mathsf{String} \to \mathsf{Int} \mid \mathsf{Char}$	$\mathsf{Bool} \,\&\, \mathsf{String} \to \mathsf{Int} \mid \mathsf{Char}$
	$\mathsf{Int} \mid \mathsf{Char}$	$\mathsf{Int} \,\&\, \mathsf{Char}$
	~~$A \,\&\, B$~~	~~$A \mid B$~~
	~~$\mathsf{Int} \to A \,\&\, B$~~	$\mathsf{Int} \to A \mid B$
Splittable	$A \lhd A \,\&\, B \rhd B$	$A \blacktriangleleft A \mid B \blacktriangleright B$
	$A \mid \mathsf{Int} \lhd (A \,\&\, B) \mid \mathsf{Int} \rhd B \mid \mathsf{Int}$	$A \,\&\, B \blacktriangleleft (A \,\&\, B) \mid \mathsf{Int} \blacktriangleright \mathsf{Int}$
	$A \mid B \lhd (A \mid B) \,\&\, \mathsf{Int} \rhd \mathsf{Int}$	$A \,\&\, \mathsf{Int} \blacktriangleleft (A \mid B) \,\&\, \mathsf{Int} \blacktriangleright B \,\&\, \mathsf{Int}$
	$\mathsf{Int} \to A \lhd \mathsf{Int} \to A \,\&\, B \rhd \mathsf{Int} \to B$	~~$\mathsf{Int} \to A \blacktriangleleft \mathsf{Int} \to A \mid B \blacktriangleright \mathsf{Int} \to B$~~

The examples using a ~~strikeout~~ font represent negative examples: that is types that do not conform to the definition. For instance, in the first cell, $\mathsf{Int} \mid \mathsf{Char}$ is intersection-ordinary, while $A \,\&\, B$ is not. Types that are ordinary from both perspectives include Int, $\top$, $\bot$, and all intersection-ordinary arrow types. Such arrow types can contain intersections in negative positions, or unions in positive positions, like $\mathsf{Bool} \,\&\, \mathsf{String} \to \mathsf{Int} \mid \mathsf{Char}$. In contrast, only some union types and intersection types are both intersection- or union- splittable, as demonstrated by the second and third lines in the cell of the splittable types. Once we have $A_1 \lhd A \rhd A_2$ and $B_1 \blacktriangleleft A \blacktriangleright B_2$, we know that A is isomorphic to $A_1 \,\&\, A_2$ and $B_1 \mid B_2$. Via the subtyping rules in Figure 4.1, we can obtain $A_i \leqslant B_j$ $(i, j = 1, 2)$. The last examples for splittable types correspond to the last examples for ordinary types. They emphasize the asymmetry between intersection-splittable types and union-splittable types, while other examples highlight the

$\boxed{A <: B}$ *(Algorithmic Subtyping for* **B+** *Logic)*

$$\textsc{bp-as-int}\ \frac{}{\mathsf{Int} <: \mathsf{Int}} \qquad \textsc{bp-as-top}\ \frac{}{A <: \top} \qquad \textsc{bp-as-bot}\ \frac{}{\bot <: A}$$

$$\textsc{bp-as-arrow}\ \frac{B_1^\bullet <: A_1^\bullet \qquad A_2^\circ <: B_2^\circ}{A_1^\bullet \to A_2^\circ <: B_1^\bullet \to B_2^\circ}$$

$$\textsc{bp-as-and}\ \frac{B_1 \lhd B \rhd B_2 \qquad A <: B_1 \qquad A <: B_2}{A <: B} \qquad \textsc{bp-as-andL}\ \frac{A_1 \lhd A \rhd A_2 \qquad A_1 <: B^\circ}{A <: B^\circ}$$

$$\textsc{bp-as-andR}\ \frac{A_1 \lhd A \rhd A_2 \qquad A_2 <: B^\circ}{A <: B^\circ} \qquad \textsc{bp-as-or}\ \frac{A_1^\circ \blacktriangleleft A^\circ \blacktriangleright A_2^\circ \qquad A_1^\circ <: B^\circ \qquad A_2^\circ <: B^\circ}{A^\circ <: B^\circ}$$

$$\textsc{bp-as-orL}\ \frac{B_1^\circ \blacktriangleleft B^\circ \blacktriangleright B_2^\circ \qquad A^{\circ\bullet} <: B_1^\circ}{A^{\circ\bullet} <: B^\circ} \qquad \textsc{bp-as-orR}\ \frac{B_1^\circ \blacktriangleleft B^\circ \blacktriangleright B_2^\circ \qquad A^{\circ\bullet} <: B_2^\circ}{A^{\circ\bullet} <: B^\circ}$$

Figure 4.3: Algorithmic subtyping rules.

symmetric parts.

Algorithmic subtyping Compared to the modular BCD subtyping in Figure 3.2, we have a rule BP-AS-BOT for the bottom type, and three more rules for union-splittable types (rule BP-AS-OR, rule BP-AS-ORL, and rule BP-AS-ORR) in our algorithmic system in Figure 4.3. Similarly to the modular BCD subtyping, the three distributivity rules in the declarative system in Figure 4.1 (rule S-DISTARRR, rule S-DISTARRL, and rule S-DISTOR), are covered with the rule BP-AS-AND by splitting the supertype. Besides this, rule BP-AS-ANDL and rule BP-AS-ANDR generalize the subtype to an intersection-splittable type, while the modular BCD subtyping uses an intersection type. That is because rule BP-AS-ARROW is restricted to only handle ordinary types with the two intersection-ordinary premises, and intersection-splittable types need help from other rules. In rule BP-AS-ORL and rule BP-AS-ORR, the subtype in the conclusion is not splittable in either way. Compared to the rules for intersection-splittable types, the three rules have additional restrictions on types to avoid overlapping with them. These restrictions on ordinary types divide rules into groups and make an order among them, except for rule BP-AS-TOP and rule BP-AS-BOT, which can still overlap with other rules. Such pre-conditions help to implement an algorithm with less backtracking, but we can drop them and the subtyping system would remain equivalent in terms of expressive power.

Example Let us demonstrate the algorithmic formulation with an example.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Int} <: \mathsf{Int}}\,\textsc{bp-as-int} \qquad \Delta_3}{\mathsf{Int} <: A^{\circ\bullet} \mid \mathsf{Int}}\,\textsc{bp-as-orR} \qquad \Delta_2}{(A^{\circ\bullet} \mid B^{\circ\bullet}) \,\&\, \mathsf{Int} <: A^{\circ\bullet} \mid \mathsf{Int}}\,\textsc{bp-as-andR} \qquad \Delta_1 \qquad \dfrac{\Delta_2 \qquad \dfrac{\dots}{A^{\circ\bullet} \mid B^{\circ\bullet} <: A^{\circ\bullet} \mid B^{\circ\bullet}}\,\textsc{bp-as-or}}{(A^{\circ\bullet} \mid B^{\circ\bullet}) \,\&\, \mathsf{Int} <: A^{\circ\bullet} \mid B^{\circ\bullet}}\,\textsc{bp-as-andL}}{(A^{\circ\bullet} \mid B^{\circ\bullet}) \,\&\, \mathsf{Int} <: A^{\circ\bullet} \mid (\mathsf{Int} \,\&\, B^{\circ\bullet})}\,\textsc{bp-as-and}$$

The missing subderivation Δ_1 for type splitting is:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Int} \lhd \mathsf{Int} \,\&\, B^{\circ\bullet} \rhd B^{\circ\bullet}}\,\textsc{SpI-and}}{A^{\circ\bullet} \mid \mathsf{Int} \lhd A^{\circ\bullet} \mid (\mathsf{Int} \,\&\, B^{\circ\bullet}) \rhd A^{\circ\bullet} \mid B^{\circ\bullet}}\,\textsc{SpI-orR}$$

Subderivations Δ_2 and Δ_3 are:

$$\dfrac{}{A^{\circ\bullet} \mid B^{\circ\bullet} \lhd (A^{\circ\bullet} \mid B^{\circ\bullet}) \,\&\, \mathsf{Int} \rhd \mathsf{Int}}\,\textsc{SpI-and} \qquad\qquad \dfrac{}{A^{\circ\bullet} \blacktriangleleft A^{\circ\bullet} \mid \mathsf{Int} \blacktriangleright \mathsf{Int}}\,\textsc{SpU-or}$$

In the next section, we will see the duality of intersection and union types more clearly.

4.3 Duotyping Based on Minimal Relevant Logic

The algorithmic subtyping relation in Section 4.2.2 was not designed from first principles. Instead, it is derived (in a straightforward way) from another definition that exploits the duality between unions and intersections (as well as top and bottom). The approach that exploits duality is the so-called *duotyping* [OCR20]. The key idea is to generalize the subtyping relation with a third argument, which is the mode of the relation (subtyping or supertyping). Then many dual rules can be expressed as a single rule in the duotyping relation, and the dual rules are ensured (by construction) to be designed in a consistent way. In turn, this leads to an implementation approach that can exploit duotyping and modes, to reduce the number of cases, definitions and code. Our implementation in Section 4.4 will use duotyping.

This section shows the duotyping definitions from which the subtyping relations in Figure 4.3 were derived, and discusses the respective metatheory, including transitivity as well as various soundness and completeness theorems among the different relations. We opted to present the traditional formulation of subtyping first because the formulation of duotyping, while more compact, is also more abstract and can be harder to grasp than the more concrete subtyping formulation. We first reformulate declarative subtyping in terms of duotyping,

and then exploit the duotyping structure found in the declarative formulation to design the algorithmic formulation.

4.3.1 Declarative Duotyping

Before developing the algorithmic version of duotyping we first show a duotyping version of declarative subtyping. Duotyping is particularly useful when the types in the language have multiple dual constructs, which is precisely the case here: we have both intersections and unions, as well as top and bottom types. Thus, understanding the subtyping relation from the point of view of duotyping can shed new light over the various rules used in subtyping, and give us some hints regarding the design of an algorithmic version.

The key idea of duotyping The key idea in duotyping is to have a generalization of the subtyping relation that takes an extra mode as an argument:

$$\textit{Mode} \qquad\qquad \diamond ::= \ \prec \mid \succ$$

The mode can either be subtyping ($\prec$) or supertyping ($\succ$). A duotyping judgement $A \diamond B$ can therefore be interpreted as both A is a subtype of B ($A \prec B$), and A is a supertype of B ($A \succ B$), depending on which mode is chosen. This extra mode is helpful to generalize dual rules for dual constructs.

Auxiliary functions Before we dive into the duotyping rules, some auxiliary definitions need to be introduced. At the top of Figure 4.4, we restate some of the auxiliary functions as described in the original duotyping work. For example, to combine the conventional subtyping rules for top and bottom (rule S-TOP and rule S-BOT), a function $\rceil\diamond\lceil$ parameterized by the mode $\diamond$ is necessary. The intersection and union constructors are also dual, and they can be selected using the function $(A \diamond_? B)$. Finally, there is a flip function to invert the mode.

Declarative duotyping With these helper functions, the declarative duotyping relation is defined in the bottom part of Figure 4.4. For instance, using $\rceil\diamond\lceil$ we can capture the two subtyping rules for top and bottom with the following duotyping rule:

$$\textsc{Duo-bound}\ \frac{}{A \diamond \rceil\diamond\lceil}$$

Some rules (rule DUO-REFL, rule DUO-TRANS and rule DUO-ARROW) are direct generalizations from the original subtyping rules in Figure 3.1 and Figure 4.1, since those rules are reversible (i.e. they work in both modes). One unique rule in the duotyping formulation is the duality

$$\rceil \prec \lceil = \top \qquad (A \prec_? B) = A \,\&\, B \qquad \overline{\prec} = \succ$$
$$\rceil \succ \lceil = \bot \qquad (A \succ_? B) = A \mid B \qquad \overline{\succ} = \prec$$

$\boxed{A \diamond B}$ *(Declarative Duotyping)*

$$\textsc{Duo-dual}\ \frac{B \mathbin{\overline{\diamond}} A}{A \diamond B} \qquad \textsc{Duo-refl}\ \frac{}{A \diamond A} \qquad \textsc{Duo-trans}\ \frac{A \diamond B \qquad B \diamond C}{A \diamond C} \qquad \textsc{Duo-bound}\ \frac{}{A \diamond \rceil \diamond \lceil} \qquad \textsc{Duo-arrow}\ \frac{A_1 \mathbin{\overline{\diamond}} B_1 \qquad A_2 \diamond B_2}{A_1 \to A_2 \diamond B_1 \to B_2}$$

$$\textsc{Duo-and}\ \frac{A \diamond B \qquad A \diamond C}{A \diamond (B \diamond_? C)} \qquad \textsc{Duo-andL}\ \frac{}{(A \diamond_? B) \diamond A} \qquad \textsc{Duo-andR}\ \frac{}{(A \diamond_? B) \diamond B} \qquad \textsc{Duo-distArrR}\ \frac{}{(A \to B) \,\&\, (A \to C) \diamond A \to B \,\&\, C}$$

$$\textsc{Duo-distArrL}\ \frac{}{(A \to C) \,\&\, (B \to C) \diamond A \mid B \to C} \qquad \textsc{Duo-distOr}\ \frac{}{((A_1 \mathbin{\overline{\diamond}_?} B) \diamond_? (A_2 \mathbin{\overline{\diamond}_?} B)) \diamond ((A_1 \diamond_? A_2) \mathbin{\overline{\diamond}_?} B)}$$

Figure 4.4: Declarative duotyping and auxiliary functions (top).

rule (rule Duo-dual), which transforms a subtyping judgement into a supertyping one, or vice versa. The remaining rules match with two original subtyping rules: rule Duo-and, rule Duo-andL, and rule Duo-andR unify the three intersection rules and three union rules (rule S-and, rule S-andL, rule S-andR, rule S-or, rule S-orL, and rule S-orR). Rule Duo-distArrR and rule Duo-distArrL are for distributing arrows over unions and intersections. As mentioned in Section 7.2.2, their dual rules are absent, therefore cannot be further unified. The rule Duo-distOr is the generalization of rule S-distOr and rule S-distAnd. Compared to the rules in Figure 4.1, the duotyping version is pleasingly more compact (11 rules versus 17 rules). More importantly, the dual relationship between various rules in Figure 4.1 is now made explicit in the formalization.

4.3.2 Algorithmic Duotyping

The rules for the algorithmic duotyping system are presented in Figure 4.5.

$\boxed{\mathsf{Ordinary}^{\diamond} A}$ *(Ordinary Types)*

$$\textsc{duo-Ord-top}\ \frac{}{\mathsf{Ordinary}^{\diamond} \top} \qquad \textsc{duo-Ord-bot}\ \frac{}{\mathsf{Ordinary}^{\diamond} \bot} \qquad \textsc{duo-Ord-int}\ \frac{}{\mathsf{Ordinary}^{\diamond} \mathsf{Int}} \qquad \textsc{duo-Ord-arrowU}\ \frac{}{\mathsf{Ordinary}^{\succ} A \to B}$$

$$\textsc{duo-Ord-arrowI}\ \frac{\mathsf{Ordinary}^{\succ} A \qquad \mathsf{Ordinary}^{\prec} B}{\mathsf{Ordinary}^{\prec} A \to B} \qquad \textsc{duo-Ord-or}\ \frac{\mathsf{Ordinary}^{\diamond} A \qquad \mathsf{Ordinary}^{\diamond} B}{\mathsf{Ordinary}^{\diamond} (A \mathbin{\overline{\diamond}_?} B)}$$

$\boxed{B \lhd A \mathbin{\diamond} \rhd C}$ *(Splittable Types)*

$$\textsc{duo-Sp-and}\ \frac{}{A \lhd (A \mathbin{\diamond_?} B) \mathbin{\diamond} \rhd B} \qquad \textsc{duo-Sp-arrowR}\ \frac{B_1 \lhd B \mathbin{\diamond_{\prec}} \rhd B_2}{A \to B_1 \lhd A \to B \mathbin{\diamond_{\prec}} \rhd A \to B_2} \qquad \textsc{duo-Sp-arrowL}\ \frac{\mathsf{Ordinary}^{\prec} B \qquad A_1 \lhd A \mathbin{\diamond_{\succ}} \rhd A_2}{A_1 \to B \lhd A \to B \mathbin{\diamond_{\prec}} \rhd A_2 \to B}$$

$$\textsc{duo-Sp-orL}\ \frac{A_1 \lhd A \mathbin{\diamond} \rhd A_2}{(A_1 \mathbin{\overline{\diamond}_?} B) \lhd (A \mathbin{\overline{\diamond}_?} B) \mathbin{\diamond} \rhd (A_2 \mathbin{\overline{\diamond}_?} B)} \qquad \textsc{duo-Sp-orR}\ \frac{\mathsf{Ordinary}^{\diamond} A \qquad B_1 \lhd B \mathbin{\diamond} \rhd B_2}{(A \mathbin{\overline{\diamond}_?} B_1) \lhd (A \mathbin{\overline{\diamond}_?} B) \mathbin{\diamond} \rhd (A \mathbin{\overline{\diamond}_?} B_2)}$$

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

$$\textsc{duo-AS-int}\ \frac{}{\mathsf{Int} \mathbin{\diamond_a} \mathsf{Int}} \qquad \textsc{duo-AS-bound}\ \frac{}{A \mathbin{\diamond_a} \rceil \diamond \lceil} \qquad \textsc{duo-AS-arrow}\ \frac{\mathsf{Ordinary}^{\prec} A_1 \to A_2 \qquad \mathsf{Ordinary}^{\prec} B_1 \to B_2 \qquad A_1 \mathbin{\overline{\diamond}_a} B_1 \qquad A_2 \mathbin{\diamond_a} B_2}{A_1 \to A_2 \mathbin{\diamond_a} B_1 \to B_2}$$

$$\textsc{duo-AS-dual}\ \frac{\mathsf{Ordinary}^{\diamond} A \qquad \mathsf{Ordinary}^{\diamond} B \qquad B \mathbin{\overline{\diamond}_a} A}{A \mathbin{\diamond_a} B} \qquad \textsc{duo-AS-and}\ \frac{B_1 \lhd B \mathbin{\diamond} \rhd B_2 \qquad A \mathbin{\diamond_a} B_1 \qquad A \mathbin{\diamond_a} B_2}{A \mathbin{\diamond_a} B}$$

$$\textsc{duo-AS-andL}\ \frac{\mathsf{Ordinary}^{\diamond} B \qquad A_1 \lhd A \mathbin{\diamond} \rhd A_2 \qquad A_1 \mathbin{\diamond_a} B}{A \mathbin{\diamond_a} B} \qquad \textsc{duo-AS-andR}\ \frac{\mathsf{Ordinary}^{\diamond} B \qquad A_1 \lhd A \mathbin{\diamond} \rhd A_2 \qquad A_2 \mathbin{\diamond_a} B}{A \mathbin{\diamond_a} B}$$

Figure 4.5: Algorithmic duotyping.

Ordinary and splittable types By parametrizing over the mode $\diamond$, the two ordinary types in Figure 4.2 can be merged. Instantiated $\diamond$ by subtyping, $\mathsf{Ordinary}^{\prec} A$ defines the intersection-ordinary relation, which characterize A° defined on the top of Figure 4.2. Its supertyping dual, $\mathsf{Ordinary}^{\succ} A$, matches with the previously defined union-ordinary types $A^{\bullet}$. A similar unification applies to splittable types. $B \lhd A \prec \rhd C$ is for intersection-splittable types, while $B \lhd A \succ \rhd C$ is for union-splittable ones. Only arrow-related rules (rule DUO-ORD-ARROWI, rule DUO-ORD-ARROWU, rule DUO-SP-ARROWR, rule DUO-SP-ARROWL) are specific to a particular mode (subtyping or supertyping), but otherwise, all the other rules are generic on the mode.

Duotyping In total there are 7 rules in the algorithmic duotyping system, compared to 10 in the subtyping system in Figure 4.3.

Duotyping reorganizes and unifies the rules in a more abstract style. All rules, except rule DUO-AS-INT, rule DUO-AS-ARROW and rule DUO-AS-DUAL unify a pair of subtyping rules. In essence one of the pairs is simply the result of flipping arguments and the mode. For instance, rule BP-AS-TOP flips rule BP-AS-BOT (and vice-versa), rule BP-AS-ANDL flips rule BP-AS-ORL (and vice-versa) and so on. While the definition of type splitting is extended, the subtyping rules are very similar to the modular BCD subtyping rules in Figure 3.2. Besides the generalization of mode, the key difference is that the left-hand side type in rule DUO-AS-ANDL and rule DUO-AS-ANDR is splittable rather than being restricted to intersections. As a consequence, rule DUO-AS-ARROW has an additional condition.

Strictly speaking, the inclusion of the duality rule (rule DUO-AS-DUAL) means that the system is not fully algorithmic. A naive implementation could apply the duality rule indefinitely, thus resulting in a non-terminating function. However, the set of rules with the duality rule is morally algorithmic. As illustrated by Oliveira, Cui, and Rehman [OCR20], in implementation we can use a Boolean flag to prevent repeated flipping with a case similar to the duality rule. We will see this approach in Section 4.4 when we illustrate our Haskell implementation.

From duotyping to subtyping: deriving the subtyping rules. As we have mentioned, the algorithmic subtyping rules can be derived from the duotyping formulation by specializing the mode. Take rule DUO-AS-ANDL as an example:

DUO-AS-ANDL-SUB

$$\frac{\mathsf{Ordinary}^{\prec} B \qquad A_1 \lhd A \prec \rhd A_2 \qquad A_1 \prec_a B}{A \prec_a B}$$

DUO-AS-ANDL-SUPER

$$\frac{\mathsf{Ordinary}^{\succ} B \qquad A_1 \lhd A \succ \rhd A_2 \qquad A_1 \succ_a B}{A \succ_a B}$$

The rules above are the result of specializing rule DUO-AS-ANDL to subtyping and supertyping, respectively. To make them compatible with the subtyping style, all judgments in supertyping mode need to be flipped via the duality rule. After that, each instances agree with rule BP-AS-ANDL and rule BP-AS-ORL respectively. Note that rule BP-AS-ORL has more restrictions ($\mathsf{Ordinary}^{\prec} A$ and $\mathsf{Ordinary}^{\prec} B$) than rule DUO-AS-ANDL-SUPER. These two conditions are essentially the ordinary conditions that arise from the use of the duality rule to convert supertyping into subtyping.

The order of rules Except for rule DUO-AS-INT and rule DUO-AS-BOUND, other rules have duotyping judgements as premises, which means the algorithm will search further after it matches its goal with the rule. Without the gray-highlighted conditions, it is possible for one duotyping judgement to satisfy multiple rules at the same time, i.e. rule DUO-AS-AND and rule DUO-AS-DUAL for $A_1 \mid A_2 \prec_a B_1 \,\&\, B_2$. Thus we use the ordinary-type conditions in gray to make these recursive rules disjoint and sort them in the following order: 1) rule DUO-AS-AND; 2) rule DUO-AS-ANDL and rule DUO-AS-ANDR; 3) rule DUO-AS-DUAL and rule DUO-AS-ARROW. Then a subtyping (or supertyping) judgement cannot match later rules once it satisfies the conditions of one rule (regardless of the satisfaction of the subtyping premises). For example, rule DUO-AS-DUAL cannot be used to flip $A_1 \mid A_2 \prec_a B$ unless $\mathsf{Ordinary}^{\prec} B$. In that case, rule DUO-AS-AND will be applied to the flipped goal $B \succ_a A_1 \mid A_2$, as $A_1 \lhd A_1 \mid A_2 \rhd_{\succ} A_2$. To justify the order, we prove the following lemmas:

Lemma 4.1 (Inversions on Splittable Types)**.** Assuming $A \diamond_a B$,

- if $B_1 \lhd B \rhd_{\diamond} B_2$ then $A \diamond_a B_1$ and $A \diamond_a B_2$.
- if $A_1 \lhd A \rhd_{\diamond} A_2$ and $\mathsf{Ordinary}^{\diamond} B$ then $A_1 \diamond_a B_1$ or $A_2 \diamond_a B_2$.
- if $A_1 \lhd A \rhd_{\overline{\diamond}} A_2$ then $A_1 \diamond_a B$ and $A_2 \diamond_a B$.
- if $\mathsf{Ordinary}^{\overline{\diamond}} A$ and $B_1 \lhd B \rhd_{\overline{\diamond}} B_2$ then $A \diamond_a B_1$ or $A \diamond_a B_2$

An inversion lemma tells us that it is safe to prioritize certain rule in some cases. The first one is for rule DUO-AS-AND: if a subtyping (or supertyping) judgement holds, and its right-hand side type is splittable under the mode, then it must satisfy rule DUO-AS-AND. The second one is for rule DUO-AS-ANDL and rule DUO-AS-ANDR. It has an extra condition $\mathsf{Ordinary}^{\diamond} B$, which means the judgement does not meet the conditions of rule DUO-AS-AND. The next two are for the rule DUO-AS-DUAL, with it we can unfold the duotyping rules by mirroring rule DUO-AS-AND, rule DUO-AS-ANDL and rule DUO-AS-ANDR with extra conditions. For instance, the dual

rule of rule DUO-AS-AND would be:

$$\textsc{duo-AS-unfold-or}\ \frac{\mathsf{Ordinary}^{\diamond} A \qquad \mathsf{Ordinary}^{\diamond} B \qquad A_1 \lhd A \underset{\diamond}{\rhd} A_2 \qquad A_1 \diamond_a B \qquad A_2 \diamond_a B}{A \diamond_a B}$$

In short, there are two principles: rule DUO-AS-AND before rule DUO-AS-ANDL and rule DUO-AS-ANDR; the dual of rule DUO-AS-AND before the dual of rule DUO-AS-ANDL and rule DUO-AS-ANDR. Since the order we choose obeys the principle, we can prove the duotyping system with the gray conditions is equivalent to the system without such conditions. The same applies to the derived subtyping system (Figure 4.3). Violating the principles may lead to false-negative results (i.e. an incomplete implementation with respect to the algorithmic specification). For example, if an algorithm tries rule DUO-AS-ANDL and rule DUO-AS-ANDR before rule DUO-AS-AND, it will reject $\mathsf{Int} \,\&\, \mathsf{Char} \prec_a \mathsf{Int} \,\&\, \mathsf{Char}$ because both rule DUO-AS-ANDL and rule DUO-AS-ANDR fail.

4.3.3 Metatheory

Here we present some theorems that connect the two systems in subtyping style (introduced in Section 7.2.2) with the two systems in duotyping style.

Theorem 4.1 (Equivalence of Declarative Systems)**.** For any type A B,

- if $A \diamond B$, then $\diamond = \prec$ and $A \leqslant B$, or $\diamond = \succ$ and $B \leqslant A$.
- if $A \leqslant B$, then $A \prec B$ and $B \succ A$.

The first property in the above theorem states the declarative duotyping system (Figure 4.4) is sound with respect to the declarative subtyping (Figure 4.1), no matter whether the judgement is in subtyping or supertyping mode. The second property is the completeness of the duotyping system.

After proving that the declarative subtyping relation is equally transformed into duotyping, we show that the algorithmic duotyping system can be mapped into the subtyping in Figure 4.3 as well. Firstly, the definitions of ordinary and splittable types are equivalent to the intersection- and union- ordinary types and splittable types defined in Figure 4.2.

Lemma 4.2 (Equivalence of Ordinary and Splittable Types)**.** For any type A,

- $\exists B^{\circ}$, $A = B^{\circ}$ if and only if $\mathsf{Ordinary}^{\prec} A$.
- $\exists B^{\bullet}$, $A = B^{\bullet}$ if and only if $\mathsf{Ordinary}^{\succ} A$.
- $B_1 \lhd A \rhd B_2$ if and only if $B_1 \lhd A \underset{\prec}{} \rhd B_2$.

- $B_1 \blacktriangleleft A \blacktriangleright B_2$ if and only if $B_1 \lhd A_{\succ} \rhd B_2$.

Then the soundness and completeness of the algorithmic subtyping (Figure 4.3) regarding to the algorithmic duotyping (Figure 4.5) can be established.

Theorem 4.2 (Equivalence of the Algorithmic Systems)**.** For any type A B,

- if $A <: B$, then $A \prec_a B$ and $B \succ_a A$.
- if $A \diamond_a B$, then $\diamond = \prec$ and $A <: B$, or $\diamond = \succ$ and $B <: A$.

Properties for the algorithmic system With a set of duotyping rules, one can reason about not only subtyping and supertyping, but also two modes together, which helps to unify theorems and proofs. Here we build the theorems on the two duotyping systems. Thanks to the equivalence between subtyping and duotyping systems, these theorems justify the algorithmic subtyping system (Figure 4.3) with respect to the declarative subtyping system (Figure 4.1) as well.

One of the key properties that validate the algorithmic system is the equivalence to the declarative system.

Theorem 4.3 (Soundness and Completeness of Algorithmic Duotyping)**.** $A \diamond_a B$ if and only if $A \diamond B$.

To establish it, reflexivity and transitivity are a must.

Theorem 4.4 (Reflexivity of the Algorithmic Duotyping)**.** $A \diamond_a A$.

During the proof we need to consider whether a type can be split or not. The process relies on two facts: First, types can be divided into ordinary types and splittable ones under any mode. Second, type splitting produces unique results.

Lemma 4.3 (Types are Either Ordinary or Splittable)**.** For any type A and any mode $\diamond$,

- $\mathsf{Ordinary}^{\diamond} A$ or $B \lhd A_{\diamond} \rhd C$ for some type B and C.
- $\mathsf{Ordinary}^{\diamond} A$ and $B \lhd A_{\diamond} \rhd C$ cannot both hold.

Lemma 4.4 (Determinism of Type Splitting)**.** If $A_1 \lhd A_{\diamond} \rhd A_2$ and $B_1 \lhd A_{\diamond} \rhd B_2$ then $A_1 = A_2$ and $B_1 = B_2$.

With reflexivity, the soundness of type splitting can be obtained directly. This suggests that the intersection (or union, according to the mode) of the splitting results is isomorphic to the original type.

Lemma 4.5 (Soundness of Splitting)**.** If $B_1 \lhd A_{\diamond} \rhd B_2$ then $A \diamond_a (B_1 \diamond_? B_2)$ and $(B_1 \diamond_? B_2) \overline{\diamond}_a A$.

Table 4.1: Summary of the proof scripts.

File	SLOC	Description
TypeSize.v	55	Defines the size of type for induction measures.
Definitions.v	397	Contains definitions for all relations. It is generated by the tool Ott [Sew+07].
Duotyping.v	966	Contains Lemma 4.1, Theorem 4.3, Theorem 4.4, Lemma 4.3, Lemma 4.4, Lemma 4.5, and Theorem 4.5.
Equivalence.v	159	Relates the two declarative systems and the two algorithmic systems, respectively. It contains Theorem 4.1, Lemma 4.2, and Theorem 4.2.
Subtyping.v	663	Contains some lemmas about the two subtyping systems. Three of them are used in the proof of Theorem 4.2
DistAnd.v	28	Justifies one statement in the paper. It shows that the rule S-distAnd (in Figure 4.1) is omittable.
DistSubtyping.v	832	A stand-alone file, which contains the two subtyping systems in Section 7.2.2, as well as related proofs that algorithmic subtyping (Figure 4.3) is decidable and equivalent to the declarative system (Figure 4.1).
Total	3,100	(2,240 excluding the last two files)

Compared with reflexivity, transitivity is straightforward since the ordinary conditions eliminate most overlapping.

Theorem 4.5 (Transitivity of the Algorithmic Duotyping)**.** If $A \diamond_a B$ and $B \diamond_a C$ then $A \diamond_a C$.

Decidability is the other key property. Its proof replays the algorithm in duotyping style.

Theorem 4.6 (Decidability of the Algorithmic Duotyping)**.** It is decidable whether $A \diamond_a B$.

4.3.4 Coq Formalization and Proof Statistics

All the lemmas and theorems are formalized and verified in the Coq proof assistant [Coq21]. We use *LibTactics.v* from the TLC Coq library [CP], which defines a collection of general-purpose tactics. In the formalization, a variant of algorithmic duotyping in Figure 4.5 is formalized where the rule duo-AS-dual is eliminated and dual rules are made explicit. The two variants (with and without dual rules) are proved to be equivalent in Coq, and some of the lemmas of algorithmic duotyping are proved using this variant. We also provide a stand-alone file for the two subtyping systems in Section 7.2.2. The proof scripts include 3,100 lines of code. Table 4.1 provides a brief summary of the files in the formalization, their number of source lines of code (SLOC), and a brief description of the content.

```
split :: Mode → Type → Maybe (Type, Type)
split MSub (TArrow a b)                          -- duo-Sp-arrowR
  | Just (b1, b2) <- split MSub b
  = Just (TArrow a b1, TArrow a b2)
split MSub (TArrow a b)                          -- duo-Sp-arrowL
  | Just (a1, a2) <- split MSuper a
  = Just (TArrow a1 b, TArrow a2 b)
split mode (TOp m' a b)                          -- duo-Sp-and
  | mode == m'
  = Just (a, b)
split mode (TOp m' a b)                          -- duo-Sp-orL
  | Just (a1, a2) <- split mode a
  = Just (TOp m' a1 b, TOp m' a2 b)
split mode (TOp m' a b)                          -- duo-Sp-orR
  | Just (b1, b2) <- split mode b
  = Just (TOp m' a b1, TOp m' a b2)
split _ _ = Nothing
```

Figure 4.6: Haskell implementation of splittable types in the algorithmic duotyping system

4.4 A Functional Implementation in Haskell

After working out through splittable types and duotyping, we now show the Haskell implementation of the algorithmic duotyping formulation presented in Figure 4.5. Our implementation exploits the extra ordinary-type conditions in the duotyping rules to avoid too much backtracking, and thus result in a more efficient implementation.

4.4.1 Abstract Syntax and Modes

The datatype definitions for the implementation are:

```
data Mode = MSub | MSuper
          deriving (Eq, Show)

data Type = TInt | TTop | TBot | TArrow Type Type | TOp Mode Type Type
          deriving (Eq, Show)
```

The datatype `Mode` models the two modes, which stand for the two directions of duotyping judgements: `MSub` for subtyping; and `MSuper` for supertyping. The datatype `Type` models the abstract syntax of types. The first four constructors directly correspond to the integer, top, bottom and arrow types. In the last constructor, we make use of the mode to unify intersection and union types. Specifically, `TOp MSub A B` means $A \,\&\, B$, and `TOp MSuper A B` means $A \mid B$.

4.4.2 Type Splitting

Figure 4.6 shows the implementation of type splitting. The type splitting function follows the formalization directly. The mode specifies whether it is for intersection- (`MSub`) or union- (`MSuper`) splittable types. The function splits the given type when possible and returns `Nothing` if the type is ordinary and cannot be split. The actual implementation makes interesting use of *pattern guards* [EP00]. For instance, in the first case we have to analyse the result of `split MSub b` to decide whether to execute the code on the right side of = or fail and move to the next case. If the pattern `Just (b1,b2)` does not match the result then we fail and move to the next case. Regarding the order of cases, we can divide them (denoted by the corresponding rules) into three groups based on the form of the split type: 1) rule DUO-SP-ARROWR and rule DUO-SP-ARROWL; 2) rule DUO-SP-AND; 3) rule DUO-SP-ORL and rule DUO-SP-ORR. While the order inside the group is restricted by the rules, the order across groups does not matter. Actually, the precedence among rules is merely assigned to avoid non-determinism of the split result, and the order itself is insignificant.

4.4.3 Duotyping and Subtyping

Figure 4.7 shows the implementation of duotyping. The code uses auxiliary functions for flipping modes and selecting $\top$ or $\bot$ by mode, which are trivial to implement:

```
flipmode :: Mode → Mode
flipmode MSub = MSuper
flipmode MSuper = MSub

select :: Mode → Type
select MSub   = TTop
select MSuper = TBot
```

The main function `check` takes two types and a mode as inputs (following the duotyping judgment $A \diamond_a B$), and one additional Boolean flag. The output is a Boolean which denotes if the judgement holds. The mode is flipped when it does not fit with the code corresponding to rule DUO-AS-INT, rule DUO-AS-BOUND, rule DUO-AS-AND, rule DUO-AS-ANDL, and rule DUO-AS-ANDR, according to rule DUO-AS-DUAL. In such a case we recheck the resulting judgement, which is equivalent to check the initial judgement by the dual of the above rules. A Boolean flag is used to make sure such flipping only happens at most once for one judgement. This implementation approach for duotyping follows the approach proposed by Oliveira, Cui, and Rehman [OCR20]. The arrow rule is the last one to be checked because flipping the goal does not affect it. At that point, we know that types on both sides are fully ordinary, and both of them are not `Int`. Thus, if rule DUO-AS-ARROW fails, a negative result will be returned. Finally, to obtain a function that checks the subtyping of two types we can simply have:

```
check :: Mode → Type → Type → Bool → Bool
check _ TInt TInt _ = True                                          -- duo-AS-int
check mode _ t _                                                    -- duo-AS-bound
  | select mode == t
  = True
check mode a b _                                                    -- duo-AS-and
  | Just (b1, b2) <- split mode b
  = (check mode a b1 False) && (check mode a b2 False)
check mode a b _                                                    -- duo-AS-andL
    duo-AS-andR
  | Just (a1, a2) <- split mode a
  = (check mode a1 b False) || (check mode a2 b False)
check mode a b False = check (flipmode m) b a True                  -- duo-AS-dual
check mode (TArrow a1 a2) (TArrow b1 b2) _                          -- duo-AS-arrow
  = (check (flipmode m) a1 b1 False) && (check mode a2 b2 False)
check _ _ _ _ = False
```

Figure 4.7: Haskell implementation of the duotyping checking algorithm

```
sub :: Type → Type → Bool
sub a b = check MSub a b False
```

4.4.4 Eliminating Backtracking

Note that no backtracking is employed during the process, except for the rule DUO-AS-ANDL and rule DUO-AS-ANDR: both rules need to be considered if the first attempt fails. The lack of other forms of backtracking is justified by our duotyping rules with ordinary conditions, which follows Lemma 4.1. Rules that involve no recursion are put at the start of the function. Rule DUO-AS-BOUND only returns a positive result so it is always safe to prioritize it among overlapping rules. Rule DUO-AS-DUAL overlaps with rule DUO-AS-ARROW, but in that case, after rule DUO-AS-DUAL, the flipped goal only matches with rule DUO-AS-ARROW, which makes no difference to directly applying rule DUO-AS-DUAL. Meanwhile, with the current order that we use, the ordinary-type conditions are guaranteed by previous rules which handle splittable types, and therefore not appear in the code.

An alternative implementation is to check rule DUO-AS-BOUND and its dual (for $\top$ and $\bot$) and rule DUO-AS-ARROW before the rules for splittable types (regardless of the ordinary conditions), which can potentially save some space and time. It can be justified by the following inversion lemma:

Lemma 4.6 (Inversion of DUO-AS-ARROW)**.** If $A_1 \to A_2 \diamond_a B_1 \to B_2$ then $A_1 \overline{\diamond}_a B_1$ and $A_2 \diamond_a B_2$.

If both types in a duotyping judgment are arrow types, their input types and output types must satisfy the duotyping relation respectively as required by the rule DUO-AS-ARROW. That means it is safe to put the arrow case before others in the Haskell implementation.

Summary This chapter shows a new algorithm for deciding subtyping (and logical entailment) in the presence of union types, intersection types and distributivity rules. Such algorithms are known to be challenging to implement and formalize. Most previous work has addressed similar problems using a pre-processing step to transform types into a normal form, before comparing types for subtyping. Here we present a new algorithm that directly compares source types for subtyping, without a pre-processing phase.

Splittable types are key to our algorithm. From the last chapter to this one, we illustrate that splittable types can scale up to systems with union types and additional distributivity rules. Moreover, *duotyping* proposed by Oliveira, Cui, and Rehman helped in designing a very symmetric formulation of algorithmic subtyping. One interesting aspect revealed by duotyping is that minimal relevant logic is not fully symmetric from the point of view of duality. As discussed in Section 4.2.2, minimal relevant logic lacks some "dual" axioms, making the subtyping rules not completely dual. This was a surprise to us. Although the absence of bottom types in minimal relevant logic created an obvious imbalance with respect to duality (which is easy to correct), we only detected the later issue with the duotyping design. Nonetheless, duotyping was still helpful to organize many of the other rules and relations, and leads to an implementation that can exploit duality to avoid extra code for dual cases. Overall we believe that both splittable types and duotyping are helpful in the design of expressive subtyping relations, and hope that this work encourages further exploration and use of both ideas.

Part II

Calculi with the Merge Operator

CHAPTER 5

The Basic System: λ_i and its Type-Directed Operational Semantics

This chapter presents the type system and the operational semantics of the λ_i calculus. It captures the basic functionality of the merge operator with a simple subtyping relation of intersection types, similar to the subtyping in Figure 3.1. This calculus is a variant of the original λ_i calculus [OSA16] (which is inspired by the calculus of Dunfield [Dun14]) with *fixpoints* and explicitly annotated lambdas instead of unannotated ones. Explicit annotations are necessary for the type-directed operational semantics of λ_i to preserve determinism. The TDOS can handle non-terminating programs, while some calculi using elaboration and coherence proofs [BOS18; Bi+19] do not support non-terminating programs. Dunfield's calculus supports recursion, but its elaboration semantics is incoherent.

5.1 Overview

A key advantage of the merge operator is its generality and the ability to model various programming language features, which we have seen in Section 2.5. However, there are challenging problems arising from the merge operator. In particular, the combination of the merge operator and subtyping is problematic. In this section, we revisit those challenges.

5.1.1 A Type-Driven Semantics for Type Preservation

As we have discussed in Section 2.2, the direct semantics of merges proposed by Dunfield is non-deterministic and lacks type safety. An essential problem is that the semantics cannot ignore the types if the reduction is meant to be type-preserving. Dunfield notes that "*For type preservation to hold, the operational semantics would need access to the typing derivation*". To avoid run-time type checking, we design an explicitly typed calculus λ_i and use contextual

type information like annotations to guide reduction. Nevertheless, it is easy to design source languages that infer some of the type annotations and insert them automatically to create valid λ_i terms as we will see in Section 8.2.1. We discuss the main challenges and key ideas of the design of λ_i next.

Annotations and type-driven reduction. A merge like $1 \,,, \mathsf{true}$ has multiple meanings under different types (e.g. Int or Bool). Eventually, we have to extract some components via the elimination of merges, which is a key issue when designing a direct operational semantics for a calculus with the merge operator. A non-deterministic semantics could allow $e_1 \,,, e_2 \hookrightarrow e_1$ and $e_1 \,,, e_2 \hookrightarrow e_2$ without any constraints.

$$1 \,,, \mathsf{true} \rightsquigarrow 1 \qquad 1 \,,, \mathsf{true} \rightsquigarrow \mathsf{true}$$

To obtain a non-ambiguous and type-safe semantics, we uses (up)casts to ensure that values have the right form during reduction. One can use an explicit type annotation to select some particular components from a term. While both the above reductions are valid in Dunfield's semantics, in λ_i they are only triggered when the expected type of the context is explicit, like:

$$(1 \,,, \mathsf{true}) : \mathsf{Int} \hookrightarrow 1 \qquad (1 \,,, \mathsf{true}) : \mathsf{Bool} \hookrightarrow \mathsf{true}$$

Casting can be viewed as a type-guided version of Dunfield's operational semantics. The type information "filters" reductions that are invalid due to a type mismatch and lead to a deterministic result.

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

Note that in λ_i the expression $1 \,,, \mathsf{true}$ without any type annotation is a value and does not reduce.

Casting in action. Written as $v \hookrightarrow_A v'$, the casting relation takes a value v and a type A as inputs, and produces a value v' of A as output. Casting is used when we want some value to match a type. Specifically, we consider two contexts: $[\cdot] : A$ and $(\lambda x.\, e : A \to B)[\cdot]$.

$$\textsc{Step-annov}\ \frac{v \hookrightarrow_A v'}{v : A \hookrightarrow v'} \qquad \textsc{Step-beta}\ \frac{v \hookrightarrow_A v'}{(\lambda x.\, e : A \to B)\ v \hookrightarrow (e[x \mapsto v']) : B}$$

We have seen that when reduction meets a value v with a type annotation A, it further reduces v against the type A. Consider a simple merge of primitive values $1 \,,, \mathsf{true} \,,, \text{'c'}$ with an

annotation Int & Char. Using rule Step-annov, casting is invoked, resulting in:

$$1 \,,, \mathsf{true} \,,, \text{`c'} \hookrightarrow_{\mathsf{Int}\,\&\,\mathsf{Char}} 1 \,,, \text{`c'}$$

We can also cast the same value by an equivalent type but where the two types in the intersection are interchanged:

$$1 \,,, \mathsf{true} \,,, \text{`c'} \hookrightarrow_{\mathsf{Char}\,\&\,\mathsf{Int}} \text{`c'} \,,, 1$$

The two valid reductions illustrate the ability of casting to create a value that matches exactly with the shape of the type.

Casting enables us to drop certain parts from a term. Very often, it is necessary for us to do so to satisfy the disjointness constraint and obtain an unambiguous merge.

Consider a function $\lambda x : \mathsf{Int}.\ x \,,, \mathsf{false}$. For its body to be well-typed, x cannot contain a boolean. Hence, when the function is applied to $1 \,,, \mathsf{true}$, we cannot directly substitute the argument in (which leads to $(1 \,,, \mathsf{true} \,,, \mathsf{false}) : \mathsf{Int}\,\&\,\mathsf{Bool}$). Instead, it is cast to Int to resolve the potential conflict.

$$\begin{aligned} & ((\lambda x : \mathsf{Int}.\ x \,,, \mathsf{false}) : \mathsf{Int}\,\&\,\mathsf{Bool} \to \mathsf{Int}\,\&\,\mathsf{Bool})\ (1 \,,, \mathsf{true}) \\ \hookrightarrow\ & (1 \,,, \mathsf{false}) : \mathsf{Int}\,\&\,\mathsf{Bool} \\ \hookrightarrow\ & 1 \,,, \mathsf{false} \end{aligned}$$

Note that, even an identity function could change its input value.

$$(\lambda x.\ x : \mathsf{Int} \to \mathsf{Int})\ (1 \,,, \text{`c'}) \hookrightarrow 1 : \mathsf{Int}$$

That may look strange from the view of the subtyping models used in conventional OOP languages, where upcasting has no runtime impact. However, instead of the inclusive semantics, we employ the coercive semantics of subtyping like the language Forsythe [Rey88]. The subtyping relation of two types does not imply the subset relation between their set interpretations (the set of its inhabited values), but rather corresponds to an implicit conversion function. A value of a subtype cannot be directly used as a value of its supertype, only after conversion it becomes a value of the supertype.

Previous work on intersection types with the merge operator [Dun14; OSA16; AOS17; BOS18; Bi+19] employ an elaboration semantics with coercive subtyping: subtyping triggers coercions, and such coercions are used by elaboration to transform values. For example, a merge is translated into a pair, and the coercion from Int & Char to Int is a projection on the pair which takes the first component. Our casting reduction also reflects the operational effect of subtyping and changes the underlying value.

5.1.2 The Challenges of Functions

Some of the hardest challenges in designing the semantics of λ_i involve functions.

Return types matter. Unlike primitive values, we cannot tell the type of a function by its form. Although the input type annotation of lambdas helps beta reduction, it is not enough to distinguish among multiple functions in a merge (e.g. $(\lambda x.\, x + 1) \,,,\, (\lambda x.\, \mathsf{true})$) without run-time type checking. To be able to select the right function from a merge, in λ_i, all functions are annotated with both the input and output types. With such annotations we can deal with programs like:

$$((\lambda f.\, f\ 1 : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}))\ ((\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int}) \,,,\, (\lambda x.\, \mathsf{true} : \mathsf{Int} \to \mathsf{Bool}))$$

In this program we have a lambda that takes a function f as an argument and applies it to 1. The lambda is applied to the merge of two functions of types $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Int} \to \mathsf{Bool}$. We select the wanted function by comparing their type annotations to the target type in casting. Otherwise runtime type checking would be necessary to recover the full type of functions.

Annotation refinement. Subtyping of intersection types leads to selecting and dropping components from merges. On the other hand, the subtyping of arrow types requests for type refinements on lambda expressions. Consider a single function $\lambda x.\, x \,,,\, \mathsf{false} : \mathsf{Int} \to \mathsf{Int}$ & Bool to be cast by type Int & $\mathsf{Bool} \to \mathsf{Int}$. To let the function return an integer when applied to a merge of type Int & Bool, we must change either the lambda body or the embedded annotation. Since reducing under a lambda body is not allowed in call-by-value, λ_i adopts the latter option, and treats the input and output annotations differently. The input annotation should not be changed as it represents the expectation of the function and helps to adjust the input value before substitution. The next example demonstrates that refining the input type annotation could result in ambiguity (having both true and false in one merge), which we prevent using rule Step-beta.

$$\begin{array}{ll} & (\lambda x.\, x \,,,\, \mathsf{false} : \mathsf{Int} \to \mathsf{Int}\ \&\ \mathsf{Bool} : \mathsf{Int}\ \&\ \mathsf{Bool} \to \mathsf{Int})\ (1 \,,,\, \mathsf{true}) \\ \hookrightarrow & \{\ \text{wrong step here: not keeping the input annotation in casting}\ \} \\ & (\lambda x.\, x \,,,\, \mathsf{false} : \mathsf{Int}\ \&\ \mathsf{Bool} \to \mathsf{Int})\ (1 \,,,\, \mathsf{true}) \\ \hookrightarrow & \{\ \text{by rule Step-beta}\ \} \\ & (1 \,,,\, \mathsf{true} \,,,\, \mathsf{false}) : \mathsf{Int} \qquad \text{(Does not type-check!)} \end{array}$$

The output annotation, in contrast, must be replaced by Int, representing a future reduction to be done after substitution. The output of the application then can be thought of as an integer and can be safely merged with another Boolean. The next example illustrates how λ_i correctly deals with annotation refinements:

$$
\begin{array}{ll}
 & ((\lambda x.\, x \,,,\, \mathsf{false} : \mathsf{Int} \to \mathsf{Int} \,\&\, \mathsf{Bool} : \mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{Int})\ (1 \,,,\, \mathsf{true})) \,,,\, \mathsf{true} \\
\hookrightarrow & \{ \text{keep the input annotation and change the output one} \} \\
 & ((\lambda x.\, x \,,,\, \mathsf{false} : \mathsf{Int} \to \mathsf{Int})\ (1 \,,,\, \mathsf{true})) \,,,\, \mathsf{true} \\
\hookrightarrow & \{ \text{by rule } \textsc{Step-beta} \} \\
 & (1 \,,,\, \mathsf{false}) : \mathsf{Int} \,,,\, \mathsf{true} \\
\hookrightarrow & \{ \text{by rule } \textsc{Step-annov} \} \\
 & 1 \,,,\, \mathsf{true}
\end{array}
$$

This example is similar to the previous one but, additionally, we merge the expression with true to demonstrate that the output type after beta-reduction, will filter the resulting merge.

Some calculi avoid the problem of function annotation refinement by treating annotated lambdas as values. For example, the target language of the original λ_i^+ [BOS18] does not reduce a value wrapped by a coercion in a function form. In the blame calculus [WF09], a value with a cast from an arrow type to another arrow type is still a value.

5.1.3 Disjoint Intersection Types and Consistency for Determinism

Even if the semantics is type-directed and it rules out reductions that do not preserve types, it can still be non-deterministic. To solve this problem, we employ the disjointness restriction that is used in calculi with disjoint intersection types [OSA16] and the novel notion of *consistency*. Both disjointness and consistency play a fundamental role in the proof of determinism.

Disjointness. Two types are disjoint (written as $A * B$), if any common supertypes that they have are *top-like types* (i.e. supertypes of any type; written as $\rceil C \lceil$).

Definition 5.1 (Disjoint specification). $A * B \triangleq \forall C$ if $A \leqslant C$ and $B \leqslant C$ then $\rceil C \lceil$

If two types are disjoint (e.g. $(\mathsf{Int} \,\&\, \mathsf{Char}) * \mathsf{Bool}$), their corresponding values do not overlap (e.g. $1 \,,,\, \text{'c'}$ and true). The only exceptions are top-like types, as they are disjoint with any type [AOS17]. Since every value of a top-like type has the same effect, casting unifies them to a fixed result. Thus the disjointness check in the following typing rule guarantees that e_1 and e_2 can be merged safely, without any ambiguities. For example, this typing rule does not accept $1 \,,,\, 2$ or $\mathsf{true} \,,,\, 1 \,,,\, \mathsf{false}$, as two subterms of the merge have overlapped types (in this case, the same type Int and Bool, respectively).

$$
\textsc{Typ-merge}\quad
\frac{\Gamma \vdash e_1 \implies A \qquad \Gamma \vdash e_2 \implies B \qquad A * B}{\Gamma \vdash e_1 \,,,\, e_2 \implies A \,\&\, B}
$$

Note that in this rule, $\Rightarrow$ denotes the *synthesis mode* in *bidirectional typing*. In typing judgements with such a mode, types are synthesized from the term, rather than provided to be checked.

Consistency. Recall the rule DStep-split in Dunfield's semantics: $e \rightsquigarrow e \,,, e$. It duplicates terms in a merge. Similar things can happen in our casting relation if the type has overlapping parts, which is allowed, for example, in an expression $1 : \mathsf{Int} \,\&\, \mathsf{Int}$. Note that in this expression the term 1 can be given type annotation $\mathsf{Int} \,\&\, \mathsf{Int}$ since $\mathsf{Int} \leqslant \mathsf{Int} \,\&\, \mathsf{Int}$. During reduction, casting is eventually used to create a value that matches the shape of type $\mathsf{Int} \,\&\, \mathsf{Int}$ by duplicating the integer:

$$1 \hookrightarrow_{\mathsf{Int} \,\&\, \mathsf{Int}} 1 \,,, 1$$

Note that the disjointness restriction does not allow sub-expressions in a merge to have the same type: $1 \,,, 1$ cannot type-check with rule Typ-merge. To retain *type preservation*, there is a special typing rule for merges of values, where a novel consistency check is used (written as $v_1 \approx_{spec} v_2$):

$$\frac{\cdot \vdash v_1 \Rightarrow A \qquad \cdot \vdash v_2 \Rightarrow B \qquad v_1 \approx_{spec} v_2}{\Gamma \vdash v_1 \,,, v_2 \Rightarrow A \,\&\, B} \text{ Typ-mergev}$$

This rule is not designed to accept more programs written by users. Instead, it takes care of expressions like $1 \,,, 1$ that may appear at runtime. Mainly, consistency allows values to have overlapped parts as far as they are syntactically equal. For example, $1 \,,, \text{true}$ and $1 \,,, \text{'c'}$ are consistent, since the overlapped part Int in both of merges has the same value. true and 'c' are consistent because they are not overlapped at all. But $1 \,,, \text{true}$ and 2 are *not consistent*, as they have different values for the same type Int. When two values have disjoint types, they must be consistent. For merges of such values, both rule Typ-mergev and rule Typ-merge can be applied, and the types always coincide. In λ_i, consistency is defined in terms of casting: [1]

Definition 5.2 (Consistency)**.** Two values v_1 and v_2 are said to be consistent (written $v_1 \approx_{spec} v_2$) if, for any type A, the casting result for the two values is the same.

$$v_1 \approx_{spec} v_2 \triangleq \forall A \text{ if } v_1 \hookrightarrow_A v_1' \text{ and } v_2 \hookrightarrow_A v_2' \text{ then } v_1' = v_2'$$

Although the specification of consistency is decidable and an equivalent algorithmic definition exists, an algorithmic definition is not required. In practice, in a programming language implementation, the rule Typ-mergev may be omitted, since, as stated, its main purpose is to ensure that run-time values are type-preserving. On the other hand, after preservation is proved in metatheory, run-time values are guaranteed to be well-typed and therefore there is no need to employ run-time type checking.

Note that the original λ_i [OSA16] is stricter than our variant of λ_i and forbids any intersection types which are not disjoint. That is to say, the term $1 : \mathsf{Int} \,\&\, \mathsf{Int}$ is not well-typed because the intersection $\mathsf{Int} \,\&\, \mathsf{Int}$ is not disjoint. In the original λ_i [OSA16] calculus disjointness checking

[1] A similar restriction was proposed by Reynolds but was not used in Forsythe's type system [Rey88].

is done by defining type well-formedness and forbidding all intersections of two non-disjoint types. However this approach is more conservative and less expressive.

The idea of allowing unrestricted intersections, while only having the disjointness restriction for merges, was first employed in the original λ_i^+ calculus [BOS18]. λ_i follows such an idea and $1 : \mathsf{Int} \,\&\, \mathsf{Int}$ is well-typed in λ_i. Allowing unrestricted intersections adds extra expressive power. For instance, in calculi with polymorphism, unrestricted intersections can be used to encode *bounded quantification* [CW85], whereas with disjoint intersections only such an encoding does not work [Bi+19; Xie+20]. Various authors, including Pierce and Castagna have (informally) observed that some form of bounded quantification can be encoded via (unrestricted) intersection types [Pie91; CX11]. Xie et al. formalize this encoding precisely [Xie+20]. For further details on this encoding, as well as to why unrestricted intersections are needed, we refer to their work.

5.2 Syntax, Subtyping and Typing

This section presents the type system of λ_i.

5.2.1 Syntax

The syntax of λ_i is:

Types	A, B	::=	$\mathsf{Int} \mid \top \mid A \to B \mid A \,\&\, B$
Expressions	e	::=	$x \mid i \mid \top \mid e : A \mid e_1\ e_2 \mid \lambda x.\, e : A \to B \mid e_1 \,,, e_2 \mid \mathbf{fix}\ x : A.\, e$
Values	v	::=	$i \mid \top \mid \lambda x.\, e : A \to B \mid v_1 \,,, v_2$
Contexts	Γ	::=	$\cdot \mid \Gamma, x : A$
Typing modes	$\Leftrightarrow$	::=	$\Rightarrow \mid \Leftarrow$

Types. Meta-variables A and B range over types. Two basic types are included: the integer type Int and the top type $\top$. Function types $A \to B$ and intersection types $A \,\&\, B$ can be used to construct compound types.

Expressions. Meta-variable e ranges over expressions. Expressions include some standard constructs: variables (x); integers (i); a canonical top value $\top$; annotated expressions ($e : A$); and application of a term e_1 to term e_2 (denoted by $e_1\ e_2$). Lambda abstractions ($\lambda x.\, e : A \to B$) must have a type annotation $A \to B$, meaning that the input type is A and the output type is B. The expression $e_1 \,,, e_2$ is the merge of expressions e_1 and e_2. Finally, fixpoints $\mathbf{fix}\ x : A.\, e$ (which also require a type annotation) model recursion.

Values, contexts, and typing modes. Meta-variable v ranges over values. Values include integers, the top value $\top$, lambda abstractions, and merges of values. Typing context Γ tracks

$\boxed{A <: B}$ *(Subtyping in λ_i)*

$$\textsc{Sub-z}\ \frac{}{\mathsf{Int} <: \mathsf{Int}} \qquad \textsc{Sub-top}\ \frac{\rceil B \lceil}{A <: B} \qquad \textsc{Sub-arrow}\ \frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2}$$

$$\textsc{Sub-andl1}\ \frac{A_1 <: A_3}{A_1 \,\&\, A_2 <: A_3} \qquad \textsc{Sub-andl2}\ \frac{A_2 <: A_3}{A_1 \,\&\, A_2 <: A_3} \qquad \textsc{Sub-andr}\ \frac{A_1 <: A_2 \qquad A_1 <: A_3}{A_1 <: A_2 \,\&\, A_3}$$

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

$$\textsc{TL-top}\ \frac{}{\rceil \top \lceil} \qquad \textsc{TL-arrow}\ \frac{\rceil B \lceil}{\rceil A \to B \lceil} \qquad \textsc{TL-and}\ \frac{\rceil A \lceil \qquad \rceil B \lceil}{\rceil A \,\&\, B \lceil}$$

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

$$\textsc{D-topL}\ \frac{}{\top *_a A} \qquad \textsc{D-topR}\ \frac{}{A *_a \top} \qquad \textsc{D-andL}\ \frac{A_1 *_a B \qquad A_2 *_a B}{A_1 \,\&\, A_2 *_a B} \qquad \textsc{D-andR}\ \frac{A *_a B_1 \qquad A *_a B_2}{A *_a B_1 \,\&\, B_2} \qquad \textsc{D-IntArr}\ \frac{}{\mathsf{Int} *_a A_1 \to A_2}$$

$$\textsc{D-arrInt}\ \frac{}{A_1 \to A_2 *_a \mathsf{Int}} \qquad \textsc{D-arrArr}\ \frac{A_2 *_a B_2}{A_1 \to A_2 *_a B_1 \to B_2}$$

Figure 5.1: Subtyping, top-like types, and disjointness of λ_i.

bound variables (x) with their type A. $\Leftrightarrow$ stands for the mode of a bidirectional typing judgment: $\Rightarrow$ is the synthesis mode; $\Leftarrow$ is the checking mode. They differ on whether the type is an output (inferred) or an input (to be checked). The details of bidirectional type checking will be discussed in Section 5.2.3.

5.2.2 Subtyping and Disjointness

Subtyping. The subtyping rules are shown at the top of Figure 5.1. Here we follow the formalization by Davies and Pfenning, except that we generalize rule Sub-top to allow any *top-like types* to be supertypes of any type. The original subtyping relation is known to be reflexive and transitive [DP00]. We proved the reflexivity and transitivity of the extended subtyping relation as well.

Top-like types and arrow types. As suggested by its name, a top-like type is both a supertype and a subtype of $\top$. Besides $\top$, top-like types contain intersection types like $\top \,\&\, \top$. In the middle of Figure 5.1 is its formal definition. Notably, rule TL-ARROW allows arrow types to be top-like when their return types are top-like. This enlargement of top-like types is inspired by the following rule in BCD-style subtyping [BCD83]:

$$\frac{}{\top \leqslant \top \to \top}\ \text{BCD-TOPARR}$$

We will come back to our motivation for allowing such top-like types in Section 5.2.3.

Disjointness. Section 5.1.3 presented the specification of disjointness. Such a specification is a liberal version of the original definition in λ_i. In our definition A and B can be *top-like types*, which was forbidden in λ_i. An equivalent algorithmic definition of disjointness ($A *_a B$) is presented on the bottom of Figure 5.1, which is the same as the definition in the original λ_i^+ calculus.

Lemma 5.1 (Disjointness properties)**.** Disjointness satisfies:

1. $A * B$ if and only if $A *_a B$.
2. if $A * (B_1 \to C)$ then $A * (B_2 \to C)$.
3. if $A * B \,\&\, C$ then $A * B$ and $A * C$.

5.2.3 Bidirectional Typing

We use a bidirectional type system for λ_i to avoid a general subsumption rule, which causes ambiguity in the presence of a merge operator. A bidirectional type system has two kinds of typing judgements, each associated with one mode. The checking judgement $\Gamma \vdash e \Leftarrow A$ says that in the typing environment Γ the expression e can be checked against type A, while the synthesis judgment $\Gamma \vdash e \Rightarrow A$ infers the type A from Γ and e. Unlike the original type system of λ_i (Figure 8.2), types have no well-formedness restriction; and expressions like $1 : \mathsf{Int} \,\&\, \mathsf{Int}$ are allowed. This generalization is inspired by the original λ_i^+ calculus [BOS18], which is the first to introduce *unrestricted intersections* to a calculus which supports disjoint intersection types.

In Figure 5.2, most typing rules directly follow the bidirectional type system of the original λ_i, including the merge rule TYP-MERGE, where disjointness is used. When two expressions have disjoint types, any parts from each of them do not overlap in the types. Therefore, their merge does not introduce ambiguity. With this restriction, rule TYP-MERGE does not accept expressions like $1 \,,,\, 2$ or even $1 \,,,\, 1$. On the other hand, the novel rule TYP-MERGEV allows

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

$$\textsc{AD-arr}\ \frac{}{A \to B \rhd A \to B} \qquad \textsc{AD-topArr}\ \frac{}{\top \rhd \top \to \top}$$

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

$$\textsc{Typ-top}\ \frac{}{\Gamma \vdash \top \Rightarrow \top} \qquad \textsc{Typ-lit}\ \frac{}{\Gamma \vdash i \Rightarrow \mathsf{Int}} \qquad \textsc{Typ-var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A} \qquad \textsc{Typ-abs}\ \frac{\Gamma, x{:}A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.\, e{:}A \to B \Rightarrow A \to B}$$

$$\textsc{Typ-app}\ \frac{\Gamma \vdash e_1 \Rightarrow C \qquad C \rhd A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\, e_2 \Rightarrow B} \qquad \textsc{Typ-merge}\ \frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B \qquad A * B}{\Gamma \vdash e_1 \,,, e_2 \Rightarrow A \,\&\, B}$$

$$\textsc{Typ-anno}\ \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash (e{:}A) \Rightarrow A} \qquad \textsc{Typ-fix}\ \frac{\Gamma, x{:}A \vdash e \Leftarrow A}{\Gamma \vdash \mathbf{fix}\ x{:}A.\, e \Rightarrow A} \qquad \textsc{Typ-mergev}\ \frac{\cdot \vdash v_1 \Rightarrow A \qquad \cdot \vdash v_2 \Rightarrow B \qquad v_1 \approx_{spec} v_2}{\Gamma \vdash v_1 \,,, v_2 \Rightarrow A \,\&\, B}$$

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

Figure 5.2: Typing of λ_i.

consistent values to be merged regardless of their types. It accepts $1 \,,, 1$ while still rejecting $1 \,,, 2$. The consistency specification used in rule Typ-mergev is given in Definition 5.2. It is for values only, and values are closed. Therefore the premises should have an empty context (denoted by $\cdot$). As discussed in Section 5.1.3, together the two rules support the determinism and type preservation of the TDOS, and rule Typ-mergev does not need to be included in an implementation. The type system with the remaining rules is algorithmic. The rule Typ-fix is new and allows fixpoints. In addition, two rules are revised: rule Typ-abs and rule Typ-app. Since lambdas are now fully annotated in the current system, rule Typ-abs is changed from checking to synthesis mode. Next, we will discuss how rule Typ-app is generalized with the applicative distributivity relation.

Applicative distribution and rule Typ-app**.** The top of Figure 5.2 shows the applicative distribution relation, which relates a type with one of its arrow supertypes. Applicative distribution is used in rule Typ-app, where a term is expected to play the role of a function. Therefore a term of type $\top$ can be used as if it has type $\top \to \top$, and be applied to any terms. For example, $\top\ 1$ is allowed and it evaluates to $\top$.

Top-like types and merges of functions. We can finally come back to the motivation to allow arrow types in top-like types and depart from Dunfield's calculus. *If no arrow types are top-like*, two arrow types $A_1 \to A_2$ and $B_1 \to B_2$ are never disjoint in terms of Definition 5.1, as they have a common supertype $A_1 \,\&\, B_1 \to \top$. Consequently, we can never create merges with more than one function, which is quite restrictive. For Dunfield this is not a problem, because she does not have the disjointness restriction. So her calculus supports merges of any functions (but its elaboration semantics is incoherent). In the original λ_i an ad-hoc solution is proposed, by forcing the matter and employing the syntactic definition of top-like types in Figure 5.1 in disjointness, while keeping the standard rule $A \leqslant \top$ in subtyping. However this means that top-like function types are not supertypes of $\top$, which contradicts the intended meaning of a top-like type. In contrast, the approach we take in λ_i is to change the rule Sub-top in subtyping. Now $\top \leqslant (A_1 \,\&\, B_1 \to \top)$ is derivable and $A_1 \,\&\, B_1 \to \top$ is genuinely a top-like type. In turn, this makes merges of multiple functions typeable without losing the intuition behind top-like types.

Checked subsumption. Unlike many calculi where there is a general subsumption rule that can apply anywhere, λ_i employs bidirectional type checking, where subsumption is controlled. The subsumption (rule Typ-sub) is in checking mode only. The checking mode is explicitly triggered by a type annotation, either via the rule Typ-anno, rule Typ-abs or rule Typ-fix. The annotation rule Typ-anno acts as explicit subsumption and assigns supertypes to expressions, providing a suitable type annotation.

Calculi with a merge operator are incompatible with a general subsumption rule because it cancels disjointness checking.

For example, with a general subsumption rule, we can directly use $1 \,,,\, \mathsf{true}$ as a term of type Int since $\mathsf{Int} \,\&\, \mathsf{Bool} \leqslant \mathsf{Int}$. Then, merging $1 \,,,\, \mathsf{true}$ with the term false would type-check since disjointness simply checks whether the static types of merging terms are disjoint, and Int is disjoint with Bool. But now, the merge contains two booleans, which would lead to ambiguity if later we wish to extract a boolean value from the merge without access of typing derivation.

We should also remark that this issue of incompatibility with a general subsumption rule is not unique to calculi with a merge operator. It shows up, for instance, in calculi with *gradual typing* [ST07] and calculi with *record concatenation* and subtyping [CM91].

Typing properties. The bidirectional type-checking system has some properties that are important for the type soundness proof presented in Section 5.3. Firstly, each term has only one synthesized type. Secondly, any well-typed term has a synthesized type, which is the principal type. Thirdly, the type in a checking judgement can be replaced by a supertype.

Lemma 5.2 (Synthesis uniqueness)**.** If $\Gamma \vdash e \implies A$ and $\Gamma \vdash e \implies B$, then $A = B$.

Ordinary type $\quad A^\circ, B^\circ, C^\circ ::= \mathsf{Int} \mid \top \mid A \to B$

$\boxed{v \hookrightarrow_A v'}$ *(Casting in λ_i)*

$$\textsc{Cast-lit}\ \frac{}{i \hookrightarrow_{\mathsf{Int}} i} \qquad \textsc{Cast-top}\ \frac{\rceil A^\circ \lceil}{v \hookrightarrow_{A^\circ} \top} \qquad \textsc{Cast-arrow}\ \frac{\neg\rceil A_2 \lceil \quad A_1 <: B_1 \quad B_2 <: A_2}{\lambda x.\, e : B_1 \to B_2 \hookrightarrow_{(A_1 \to A_2)} \lambda x.\, e : B_1 \to A_2}$$

$$\textsc{Cast-mergevl}\ \frac{v_1 \hookrightarrow_{A^\circ} v_1'}{v_1 \,,, v_2 \hookrightarrow_{A^\circ} v_1'} \qquad \textsc{Cast-mergevr}\ \frac{v_2 \hookrightarrow_{A^\circ} v_2'}{v_1 \,,, v_2 \hookrightarrow_{A^\circ} v_2'} \qquad \textsc{Cast-and}\ \frac{v \hookrightarrow_A v_1 \quad v \hookrightarrow_B v_2}{v \hookrightarrow_{A \,\&\, B} v_1 \,,, v_2}$$

Figure 5.3: The definition of casting in λ_i.

Lemma 5.3 (Synthesis principality)**.** If $\Gamma \vdash e \Leftarrow A$ then there exists type B, s.t. $\Gamma \vdash e \Rightarrow B$ and $B \leqslant A$.

Lemma 5.4 (Checking subsumption)**.** If $\Gamma \vdash e \Leftarrow A$ and $A \leqslant B$, then $\Gamma \vdash e \Leftarrow B$.

5.3 A Type-Directed Operational Semantics for λ_i

This section introduces the type-directed operational semantics for λ_i. It makes use of the contextual type information arising from type annotations to cast values in the reduction process, forcing the value to match the type structure. We show that reduction is *deterministic* and *type sound*. That is to say, there is only one way to reduce an expression according to the small-step relation, and the process preserves types and never gets stuck.

5.3.1 Casting of Values

The casting relation $v \hookrightarrow_A v'$ reduces the value v under type A, producing a value v' that has type A. It arises when given a value v of some type, where A is a supertype of the type of v, and v needs to be converted to a value compatible with the supertype A.

In λ_i, values and types have a strong correspondence. If a value is well-typed, its principal type can be told directly by looking at its syntactic form. Casting gives a runtime interpretation to subtyping, and the rules of are aligned in a one-by-one correspondence with the subtyping formulation. While subtyping states what kind of conversions are valid at the type level, casting gives an operational meaning for such conversions on values.

Figure 5.3 shows the casting relation. Rule Cast-top expresses the fact that a top-like type is the supertype of any type, which means that any value can be reduced under it. The top-like type is restricted to be *ordinary*, to avoid overlapping with the rule Cast-and. In this basic

system, our definition of ordinary types is the same as the one given by Davies and Pfenning: types that are not intersections are ordinary.

The rule Cast-top indicates that under such a type, any value reduces to the top value $\top$. Recall that the top-like definition in Figure 5.1 includes arrow types whose return type is top-like, thus the rule Cast-top covers values with such top-like arrow types as well. Rule Cast-lit expresses that an integer value reduced under the supertype Int is just the integer value itself. Rule Cast-arrow states that a lambda value $\lambda x.\, e : A_1 \to A_2$, under a *non-top-like type* $B_1 \to B_2$, evaluates to $\lambda x.\, e : A_1 \to B_2$ if $B_1 <: A_1$ and $A_2 <: B_2$. The restriction that $B_1 \to B_2$ is not top-like avoids overlapping with rule Cast-top. Importantly rule Cast-arrow changes the return type of lambda abstractions, and keeps the input type, since it is needed at run-time (by rule Step-beta which is discussed in Section 5.3.3).

Intersections and merges. In the remaining rules, we first decompose intersections. Then we only need to consider ordinary types. We take care of the value by going through every merge, until both the value and type are in a basic form. Rule Cast-mergevl and rule Cast-mergevr are a pair of rules for reducing merges under an ordinary type. Since the type is not an intersection, the result contains no merge. Usually, we need to select between the left part and right part of a merge according to the type. The values of disjoint types do not overlap on non-top-like types. For example, $1 \,,, (\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) \hookrightarrow_{\mathsf{Int}} 1$ selects the left part. For top-like types, no matter which rule is applied, the reduction result is determined by the type only, as the rule Cast-top suggests.

Rule Cast-and is the rule that deals with intersection types. It says that if a value v can be reduced to v_1 under type A, and can be reduced to v_2 under type B, then its reduction result under type $A \,\&\, B$ is the merge of two results $v_1 \,,, v_2$. Note that this rule may *duplicate values*. For example $1 \hookrightarrow_{\mathsf{Int}\,\&\,\mathsf{Int}} 1 \,,, 1$. Such duplication requires special care, since the merge violates disjointness. The specially designed typing rule (rule Typ-mergev) uses the notion of consistency (Definition 5.2) instead of disjointness to type-check a merge of two values. Note also that such duplication implies that sometimes it is possible to use either rule Cast-mergevl or rule Cast-mergevr to reduce a value. For example, $1 \,,, 1 \hookrightarrow_{\mathsf{Int}} 1$. The consistency restriction in rule Typ-mergev ensures that no matter which rule is applied in such a case, the result is the same.

Example. A larger example to demonstrate how casting works is:

$$\begin{aligned} & (\lambda x.\, (x \,,, \text{`c'}) : \mathsf{Int} \to \mathsf{Int}\,\&\,\mathsf{Char}) \,,, (\lambda x.\, x : \mathsf{Bool} \to \mathsf{Bool}) \,,, 1 \\ \hookrightarrow_{\mathsf{Int}\,\&\,(\mathsf{Int}\to\mathsf{Int})} \;& 1 \,,, (\lambda x.\, (x \,,, \text{`c'}) : \mathsf{Int} \to \mathsf{Int}) \end{aligned}$$

The initial value is the merge of two lambda abstractions and an integer. The target type is $\mathsf{Int}\,\&\,(\mathsf{Int} \to \mathsf{Int})$. Because the target type is an intersection, casting first employs rule Cast-

AND to decompose the intersection into Int and Int → Int. Under type Int the value reduces to 1, and under type Int → Int it will reduce to $\lambda x.\, x \,,,$ 'c' : Int → Int. Therefore, we obtain the merge $1 \,,, (\lambda x.\, x \,,,$ 'c' : Int → Int) with type Int & (Int → Int).

Basic properties of casting. Some properties can be proved directly by induction on the derivation of casting. First, when casting against a top-like type, the result only depends on the type, as a top-like type is isomorphic to the unit type. Second, casting produces the same result whenever it is done directly or indirectly. Assume v has type C, then the conversion from C to B has an equivalent effect as the composition of the conversion from C to A and the conversion from A to B. This corresponds to the coherence property of subtyping. Third, if a well-typed value can be cast by some type, its synthesized type must be a subtype of that type. The three properties are formally stated next:

Lemma 5.5 (Casting against top-like types)**.** If $\rceil A \lceil$, $v_1 \hookrightarrow_A v_1'$, and $v_2 \hookrightarrow_A v_2'$ then $v_1' = v_2'$.

Lemma 5.6 (Commutativity of casting)**.** If $v \hookrightarrow_A v_1$, and $v_1 \hookrightarrow_B v_2$, then $v \hookrightarrow_B v_2$.

Lemma 5.7 (Up-casting only)**.** If $v \hookrightarrow_A v'$ and $\cdot \vdash v \Rightarrow B$, then $B \leqslant A$.

Note that Lemma 5.7 relates casting and subtyping.

5.3.2 Consistency, Determinism and Type Soundness of Casting

Consistent values, as specified in Definition 5.2, introduce no ambiguity in casting. If two consistent values both can reduce under a type, they should produce the same result. The *consistency* restriction ensures that duplicated values in a merge type-check, but it still rejects merges with different values of the same type. A value of a top-like type is consistent with any other value. It can only be cast by top-like types, which leads to a fixed result decided by the type.

Relating disjointness and consistency. Assuming that the synthesized types of two values are disjoint, from Lemma 5.7, we can conclude that when the two values both cast by a type, that type must be a common supertype of their principal types, which is known to be top-like. Furthermore, Lemma 5.5 implies that their reduction results are always the same under such top-like types, so they are consistent. The above discussion concludes that values with disjoint types evaluate to the same result under the same type, i.e. they are consistent. This is captured by the following lemma:

Lemma 5.8 (Consistency of disjoint values)**.** If $A * B$, $\cdot \vdash v_1 \Rightarrow A$, and $\cdot \vdash v_2 \Rightarrow B$ then $v_1 \approx_{spec} v_2$.

Determinism of casting. The merge construct makes it hard to design a deterministic operational semantics. Disjointness and consistency restrictions prevent merges like $1 \,,\, 2$, and bring the possibility to deal with merges based on types. Casting takes a well-typed value, which, if it is a merge, must be consistent (according to Lemma 5.8). When the two casting rules for merges (rule Cast-mergevl and rule Cast-mergevr) overlap, no matter which one is chosen, either value reduces to the same result due to consistency. Indeed our casting relation always produces a unique result for any legal combination of the input value and type. This serves as a foundation for the determinism of the operational semantics.

Lemma 5.9 (Determinism of casting)**.** For every well-typed v (that is there is some type B such that $\cdot \vdash v \Rightarrow B$), if $v \hookrightarrow_A v_1$ and $v \hookrightarrow_A v_2$ then $v_1 = v_2$.

Runtime subtyping. While most casting rules produce values of the target type (in synthesis mode), two rules are more relaxed. Rule Cast-top offers $\top$ for any top-like types. Rule Cast-arrow keeps the original input annotation in the reduced lambda abstraction:

$$(\lambda x.\, x \,,\, 2 : \mathsf{Char} \to \mathsf{Char} \,\&\, \mathsf{Int}) \hookrightarrow_{(\mathsf{Char} \,\&\, \mathsf{Int} \to \mathsf{Char})} \lambda x.\, x \,,\, 2 : \mathsf{Char} \to \mathsf{Char}$$

We use *runtime subtype* to characterize the relation between the synthesized type of the casting result and the casting type. Defined in Figure 5.4, runtime subtyping is a restricted form of subtyping. Roughly, it only allows subtyping in contravariant positions except for top-like types.

Runtime subtyping is introduced because we need to find a middle point between equality and subtyping to describe how casting preserves the reduction type (Lemma 5.11). If A is the reduction type and B is the type of the output value in the casting relation, we cannot simply say that $B = A$. But knowing only that $B <: A$ is not enough. The initial value already has a subtype of A, so this does not even prevent directly using it as the result. In fact, for runtime subtyping, we expect the inclusive semantics: a value of subtype can be directly treated as a value of its supertype. Runtime subtyping ensures that the casting result behaves like a term of the casting type, and it keeps transitivity as well. Thus after multiple steps of reduction, the ultimate result still has a runtime subtype in terms of the type of the initial expression. Therefore the preservation property of λ_i is safely relaxed, to allow the expression type to become more and more specific during reduction.

Type soundness of casting. Via the transitivity lemma (Lemma 5.6) and the determinism lemma (Lemma 5.9), we obtain the following property: any reduction results of the given value are consistent.

Lemma 5.10 (Consistency after casting)**.** If v is well-typed, and $v \hookrightarrow_A v_1$, and $v \hookrightarrow_B v_2$ then $v_1 \approx_{spec} v_2$.

$\boxed{A \ll B}$ *(Runtime Subtyping)*

$$\frac{}{A \ll A}\ \textsc{RSub-z} \qquad \frac{B_1 <: A_1 \qquad A_2 \ll B_2}{A_1 \to A_2 \ll B_1 \to B_2}\ \textsc{RSub-arr}$$

$$\frac{A_1 \ll B_1 \qquad A_2 \ll B_2}{A_1 \,\&\, A_2 \ll B_1 \,\&\, B_2}\ \textsc{RSub-and} \qquad \frac{\rceil A \lceil}{\top \ll A}\ \textsc{RSub-top}$$

Figure 5.4: Runtime subtyping of λ_i.

The lemma shows that the reduction result of rule Cast-and is always made of consistent values, which is needed in type preservation via the typing rule Typ-mergev. Then a (generalized) type preservation lemma on casting can be proved.

Lemma 5.11 (Preservation of casting)**.** If $\cdot \vdash v \Leftarrow A$ and $v \hookrightarrow_A v'$, then $\exists B, \cdot \vdash v' \Rightarrow B$ and $B \ll A$.

In general, this lemma shows that casting produces well-typed values: it shows that if a value is checked by type A and it can be cast by A then the reduced value is always well-typed, and its synthesized type B is a *runtime subtype* of A. What is more, casting is guaranteed to progress for a given value and a type it can be checked against. That is to say, from a well-typed value, we can derive the existence of a casting judgement and the well-typedness of the reduction result.

Lemma 5.12 (Progress of casting)**.** If $\cdot \vdash v \Leftarrow A$, then $\exists v', v \hookrightarrow_A v'$.

Fewer checks on casting. In rule Cast-arrow (in Figure 5.3), the premise $A_1 <: B_1$ is redundant for reduction. Since we only care about well-typed terms being reduced, such a check has already been guaranteed by typing. Therefore an actual implementation could omit that check. The reason why we keep the premise is that casting plays another role in our metatheory: it allows us to define consistency. Consistency is defined for any (untyped) values, and the extra check there tightens up the definition of consistency. With the premise, casting directly implies a subtyping relation between the type of the reduced value and the reduction type. (See Lemma 5.7: If $v \hookrightarrow_A v'$, and $\cdot \vdash v \Rightarrow B$, then $B \leqslant A$).

One could wonder if this property is unnecessary because it may be derived by type preservation of reduction. Note that whenever casting is called in a reduction rule, the subtyping relation can be obtained from the typing derivation of the reduced term. For example, reducing $v : A$ will cast v under A. If $v : A$ is well-typed, then we could in principle prove that the type of v is a subtype of A. Unfortunately, the above proof is hard to attain in practice. Because type preservation depends on consistency, and consistency is defined by casting. Once the subtyping property relies on type preservation, there is a cyclic dependency between the properties.

$\boxed{e \hookrightarrow e'}$ *(Reduction)*

$$\textsc{Step-top}\ \frac{}{\top\ v \hookrightarrow \top} \qquad \textsc{Step-beta}\ \frac{v \hookrightarrow_A v'}{(\lambda x.\, e : A \to B)\ v \hookrightarrow (e[x \mapsto v']) : B} \qquad \textsc{Step-annov}\ \frac{v \hookrightarrow_A v'}{v : A \hookrightarrow v'} \qquad \textsc{Step-appl}\ \frac{e_1 \hookrightarrow e_1'}{e_1\ e_2 \hookrightarrow e_1'\ e_2}$$

$$\textsc{Step-appr}\ \frac{e_2 \hookrightarrow e_2'}{v_1\ e_2 \hookrightarrow v_1\ e_2'} \qquad \textsc{Step-mergel}\ \frac{e_1 \hookrightarrow e_1'}{e_1 \,,, e_2 \hookrightarrow e_1' \,,, e_2} \qquad \textsc{Step-merger}\ \frac{e_2 \hookrightarrow e_2'}{v_1 \,,, e_2 \hookrightarrow v_1 \,,, e_2'} \qquad \textsc{Step-anno}\ \frac{e \hookrightarrow e'}{e : A \hookrightarrow e' : A}$$

$$\textsc{Step-fix}\ \frac{}{(\mathbf{fix}\ x : A.\, e) \hookrightarrow (e[x \mapsto (\mathbf{fix}\ x : A.\, e)]) : A}$$

Figure 5.5: Call-by-value reduction of λ_i.

5.3.3 Reduction

The reduction rules are presented in Figure 5.5. Recall that rule Typ-app is generalized using applicative distribution. Correspondingly, the top value consumes every input it meets using rule Step-top. Pierce and Steffen employ a similar rule in a calculus with higher-order subtyping [PS97]. Rule Step-beta and rule Step-annov are the two rules relying on casting. Rule Step-beta casts the argument value before substitution. The rule Step-annov interprets the annotation that wraps a value as type casting.

We choose to allow $\top$ act like an applicative term (rule Step-top) because we want a unified result for casting under any top-like type, for determinism purpose. For example, since we accept programs like $v_1 : (\mathsf{Int} \to \top)\ 1$, v_1 will be reduced to $\top$ at runtime, and finally $\top\ 1$ to $\top$. Corresponding, in typing we convert $\top$ to $\top \to \top$ (rule AD-topArr to accept $\top\ 1$, keeps type preservation.

Metatheory of reduction. When designing the operational semantics of λ_i, we want it to have two properties: *determinism of reduction* and *type soundness*. That is to say, there is only one way to reduce an expression according to the small-step relation, and the process preserves types and never gets stuck. Similar lemmas on casting were already presented, which are necessary for proving the following theorems, mainly in cases related to rule Step-annov and rule Step-beta.

Theorem 5.1 (Determinism of $\hookrightarrow$)**.** If $\cdot \vdash e \Leftarrow A$, $e \hookrightarrow e_1$, $e \hookrightarrow e_2$, then $e_1 = e_2$.

The preservation theorem states that during reduction, the program is always well-typed, and the reduced expression can be checked against the original type.

Theorem 5.2 (Type preservation of $\hookrightarrow$)**.** If $\cdot \vdash e \Leftrightarrow A$, and $e \hookrightarrow e'$ then $\cdot \vdash e' \Leftarrow A$.

This theorem is a corollary of the following lemma:

Lemma 5.13 (Generalized type preservation of $\hookrightarrow$)**.** If $\cdot \vdash e \Leftrightarrow A$, and $e \hookrightarrow e'$ then $\exists B$, $\cdot \vdash e' \Leftrightarrow B$ and $B \ll A$.

The lemma has a similar structure to Lemma 5.11: the type of the reduced result is a runtime subtype (Figure 5.4) of the target type. Note that $\Leftrightarrow$ in this and the following lemmas is a meta-variable for typing mode. It means both checking and synthesis mode work for it, as long as the conclusion and the premise have the same mode. To prove Lemma 5.13, the substitution lemma has to be adapted. The substituted term is allowed to have a runtime subtype of the expected type. The type of the result, accordingly, is a subtype of the initial type. For example, a lambda of type $\mathsf{Int} \to \mathsf{Int}$ can be used when a term of $\mathsf{Int} \,\&\, \mathsf{Char} \to \mathsf{Int}$ is expected. It can be viewed as a combination of type narrowing via runtime subtyping and the conventional substitution lemma.

Lemma 5.14 (Substitution preserves types)**.** For any expression e, if $\Gamma_1, x{:}B, \Gamma_2 \vdash e_1 \Leftrightarrow A$ and $\Gamma_2 \vdash e_2 \Rightarrow B'$, $B' \ll B$, then $\Gamma_1, \Gamma_2 \vdash e_1[x \mapsto e_2] \Leftrightarrow A'$ and $A' \ll A$.

Finally, the progress theorem promises that reduction never gets stuck. Its proof relies on the progress lemma of casting.

Theorem 5.3 (Progress of $\hookrightarrow$)**.** If $\cdot \vdash e \Leftarrow A$, then e is a value or $\exists e'$, $e \hookrightarrow e'$.

Summary In this work, we showed how a type-directed operational semantics allows us to address the ambiguity problems of calculi with a merge operator. Therefore, with the TDOS approach, we can answer the question of how to give a direct operational semantics for both the general merge operator in a setting with intersection. Next we will extend the system to support record concatenation and more advanced subtyping.

CHAPTER 6

The Nested Composition Calculus λ_i^+

In this chapter, we will introduce the λ_i^+ calculus, which extends λ_i with distributivity rules for subtyping and nested composition[1]. Like λ_i, the λ_i^+ calculus is essentially a TDOS variant of the calculus with the same name introduced by Bi, Oliveira, and Schrijvers. In the original λ_i^+ calculus the semantics is defined by elaboration [BOS18]. The subtyping relation for λ_i^+ is based on the subtyping relation of Barendregt, Coppo, and Dezani-Ciancaglini [BCD83], which is presented in Chapter 3. λ_i^+ has record types and supports record concatenation via the merge operator. While it is quite similar to the λ_i calculus, BCD subtyping empowers λ_i^+ so that a merge of functions can act as a function, and a merge of records can act like a record. We will see how this behavior leads to changes in the typing and reduction rules.

6.1 Overview

Although the subtyping of λ_i allows multiple functions in a merge, it lacks the distributive subtyping rule for intersection types that we have seen in Chapter 3. Distributivity in a calculus with a merge operator is interesting because it enables nested composition, which essentially reflects the distributivity seen at the type level into the term level. For instance, the distributive property of functions over intersections enables subtyping statements such as:

$$(A_1 \to A_2) \,\&\, (B_1 \to B_2) \leqslant A_1 \,\&\, B_1 \to A_2 \,\&\, B_2$$

Therefore, a merge of two functions can be treated as a single function where the inputs and outputs of the two original functions have intersection types.

[1] The application of nested composition was discussed in Section 2.5.2.

Revisiting the examples with merges and subtyping. Recall the first example presented in Section 2.3, rewritten here to use a lambda abstraction instead of a let expression:

$$(\lambda x.\,(2 \,,,\, x) + 3 : \mathsf{Bool} \to \mathsf{Int})\ (\mathsf{true} \,,,\, 1)$$

As argued in Section 2.3, in the inclusive semantics of subtyping, examples like the above are problematic since they can lead to non-disjoint merges appearing at runtime. In λ_i, with the TDOS approach, the full reduction steps for this example are:

$$\begin{array}{lll}
 & (\lambda x.\,(2 \,,,\, x) + 3 : \mathsf{Bool} \to \mathsf{Int})\ (\mathsf{true} \,,,\, 1) & \\
\hookrightarrow & ((2 \,,,\, \mathsf{true}) + 3) : \mathsf{Int} & \text{by rule } \textsc{Step-beta} \\
\hookrightarrow & 5 : \mathsf{Int} & \text{reduction for } + \\
\hookrightarrow & 5 & \text{by rule } \textsc{Step-annov}
\end{array}$$

Firstly, the input value is filtered by casting against the input type Bool. Importantly, *only* the selected part true is substituted in the body of the lambda during beta-reduction. Then, the expression $(2 \,,,\, \mathsf{true}) + 3$ evaluates to 5. In the process, + acts like a lambda with annotation $\mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$. Finally, the return type Int filters the result, which does not change it in this case.

The second example in Section 2.3 with records is slightly more involved. Especially because multi-field records are encoded as merges, and such a merge can be directly projected. We will show the reduction of that example by the end of this section.

Splittable arrow types. Without distributivity, if an arrow type is a supertype of an intersection of multiple types, then it must be a supertype of one of those types. Conversely, when cast a merge by an arrow type, we will obtain a single function (one of the components of the merge) as a result. However, in λ_i^+, we are now faced with the following kind of casting due to the change in subtyping:

$$\begin{array}{ll}
 & (\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) \,,,\, (\lambda x.\, \mathsf{true} : \mathsf{Int} \to \mathsf{Bool}) \,,,\, (\lambda x.\, \text{`c'} : \mathsf{Int} \to \mathsf{Char}) \\
\hookrightarrow_{\mathsf{Int} \to \mathsf{Int}\ \&\ \mathsf{Char}} & (\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) \,,,\, (\lambda x.\, \text{`c'} : \mathsf{Int} \to \mathsf{Char})
\end{array}$$

Even though we are doing casting under an arrow type, we do not obtain a function as a result. Instead what we have is a merge of two functions. This is because, with the distributivity of arrow types, multiple components present in a merge can contribute to the final result. For instance, in the reduction above both the first and the last lambdas must be present to ensure that the resulting value "behaves" as a function of type $\mathsf{Int} \to \mathsf{Int}\ \&\ \mathsf{Char}$.

Parallel application. One consequence of allowing merges to be a normal form of arrow types is that a beta reduction for applications is not enough, since merges of functions can also

be applied to values. We use a relation called the parallel application to deal with applications of merges to another value. Parallel application distributes the input to every lambda in a merge, and beta reduces them in parallel. From the point of view of the small-step semantics, the parallel application process, like casting, is finished in a single step, like the following example.

$$\begin{aligned}&(\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int} \,,,\, (\lambda x.\, \mathsf{true} : \mathsf{Int} \to \mathsf{Bool} \,,,\, \lambda x.\, \text{`c'} : \mathsf{Int} \to \mathsf{Char}))\ 2 \\ \hookrightarrow\ &(2 + 1) : \mathsf{Int} \,,,\, (\mathsf{true} : \mathsf{Bool} \,,,\, \text{`c'} : \mathsf{Char})\end{aligned}$$

Generalized consistency. In λ_i^+, a merge of function values, once applied to an argument, can step to a merge of expressions. In the previous example, for instance, one of the components in the resulting merge is $(2 + 1) : \mathsf{Int}$, which is an expression but not a value. This raises a challenge to the consistency definition employed in λ_i, which can only relate values (but not arbitrary expressions). Therefore we have to extend the definition of consistency in λ_i^+ to include such expressions. Intuitively, two expressions can be safely merged if their reduction result is the same, like $(2 + 1) : \mathsf{Int}$ or $3 \,,,\, (2 + 1)$. However, there are some difficulties regarding how to reason about expressions like the latter one. Consider two non-terminating programs, comparing them may never end. Instead we model consistency with a syntactic definition, which is less powerful in the sense that it does not allow $3 \,,,\, (2 + 1)$. But such a definition is enough to accept the terms generated by parallel application, which keeps the syntactic equivalence among the components of merges.

Records. Together with BCD subtyping, single-field records and record types are added into λ_i^+. There is a distributivity rule in subtyping for records as well: $\{l : A\} \,\&\, \{l : B\} \leqslant \{l : A \,\&\, B\}$. A merge of several records can be used as a single record, as long as the records have the same field name. This is similar to function application where functions in one merge share one input. In reduction, we treat record projection like function application. That is to say, the parallel application relation not only applies functions in a merge in parallel but also projects records in a merge at the same time.

$$(\{l = \mathsf{true}\} \,,,\, \{l = 2\}).l \hookrightarrow \mathsf{true} \,,,\, 2$$

Despite the similarity of projection and application, usually not every part in a multi-field record has the same label. Thus when projecting a merge we extract the fields from all matched components and ignore the rest.

Here we show the full reduction steps for the example with records. In this example, a function that expects a record of Bool takes a merge of two records. Only the one with the expected label and field is selected via casting. In parallel application, the projection is distributed into the merge, but only the valid results will be kept.

$$
\begin{array}{ll}
 & (\lambda x.\ (\{m = 2\} \,,, x)).m + 3 : \{l : \mathsf{Bool}\} \to \mathsf{Int})\ (\{m = 1\} \,,, \{l = \mathsf{true}\}) \\
\hookrightarrow & \{\text{ by rule } \textsc{Step-bcd-papp} \text{ and casting}\} \\
 & ((\{m = 2\} \,,, \{l = \mathsf{true}\})).m + 3) : \mathsf{Int} \\
\hookrightarrow & \{\text{ by rule } \textsc{Step-bcd-pproj} \text{ and parallel application }\} \\
 & (2 + 3) : \mathsf{Int} \\
\hookrightarrow & \{\text{ reduction for } +\ \} \\
 & 5 : \mathsf{Int} \\
\hookrightarrow & \{\text{ by rule } \textsc{Step-bcd-annov}\ \} \\
 & 5
\end{array}
$$

6.2 Syntax and Typing

The syntax of λ_i^+ is:

Types	A, B	::=	$\mathsf{Int} \mid \top \mid A \to B \mid A \,\&\, B \mid \{l : A\}$
Expressions	e	::=	$x \mid i \mid \top \mid e : A \mid e_1\ e_2 \mid \lambda x.e : A \to B \mid e_1 \,,, e_2 \mid \mathbf{fix}\ x : A.\ e$ $\mid \{l = e\} \mid e.l$
Values	v	::=	$i \mid \top \mid \lambda x.e : A \to B \mid v_1 \,,, v_2 \mid \{l = v\}$
Contexts	Γ	::=	$\cdot \mid \Gamma, x : A$
Typing modes	$\Leftrightarrow$	::=	$\Rightarrow \mid \Leftarrow$

Records and record types. $\{l = e\}$ stands for a single-field record whose label is l and its field is e. Projection $e.l$ selects the field(s) from e with label l. In a record type $\{l : A\}$, A is the type of the field. A merge of single-field records can be thought of as a multi-field record, and therefore can be used, for example, to model objects. Multi-field record types are desugared to intersections of single-field ones, and multi-field records are also desugared to merges:

$$
\begin{aligned}
\{l_1 : A_1; \ldots; l_n : A_n\} &\triangleq \{l_1 : A_1\} \,\&\, \cdots \,\&\, \{l_n : A_n\} \\
\{l_1 = e_1; \ldots; l_n = e_n\} &\triangleq \{l_1 = e_1\} \,,, \ldots ,, \{l_n = e_n\}
\end{aligned}
$$

Splittable types and subtyping. Figure 6.1 shows the subtyping-related definitions. It is based on the algorithmic formulation of subtyping present in Chapter 3, and uses the *splittable types* to characterize the distributivity of function types and record types over intersection. For a splittable type, its syntactical structure guarantees that it has an equivalent intersection type according to the subtyping. So it can be decomposed into two parts, while ordinary types cannot. Compared with the definitions present in Figure 3.2, for each relation, a rule for record types is added in a modular way. The record rule is often similar to the arrow type rule, especially if we interpret the type $\{l : A\}$ as $\{l\} \to A$, a function type that takes a first-class label.

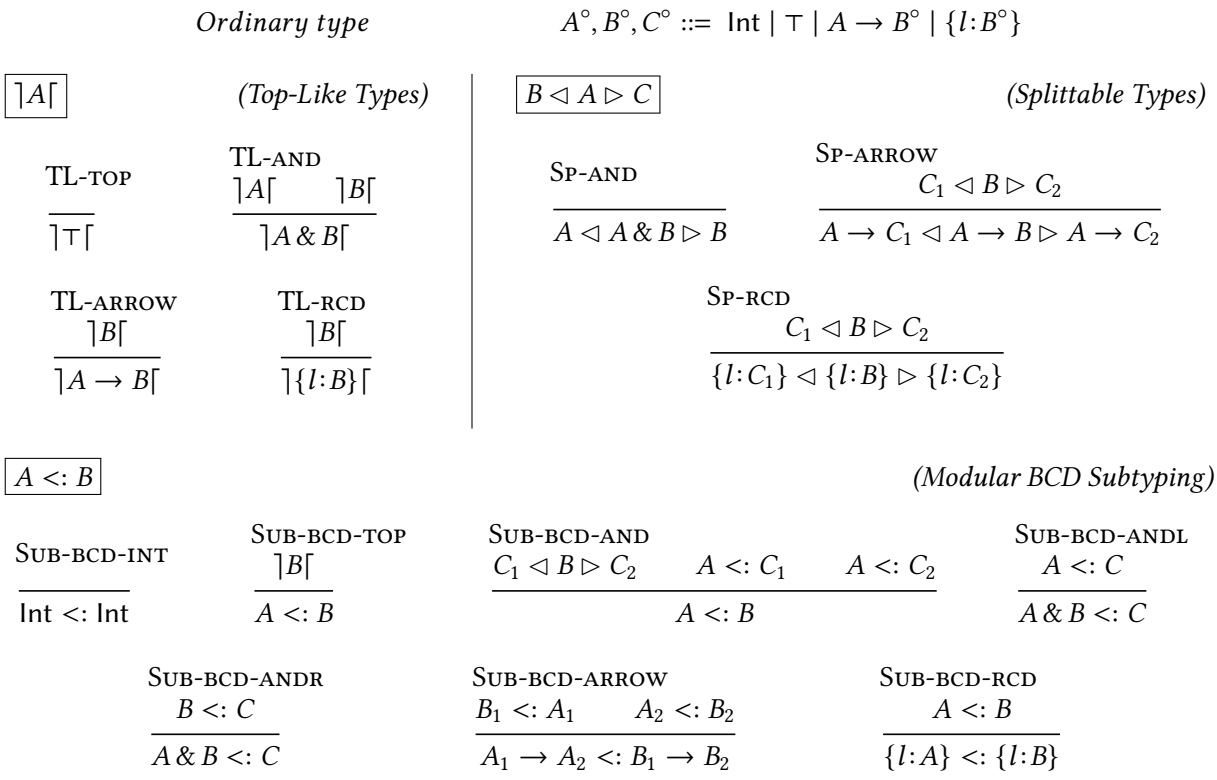

Figure 6.1: Subtyping rules.

Disjointness. The disjointness definition is presented on Figure 6.2. It might be a bit surprising that, except for the new record-related rules, the remaining rules are the same as λ_i's disjointness definition. The two systems both respect the specification of disjointness (Definition 5.1), from which we know that if type A is not disjoint with type B, then it is not disjoint to any subtypes of B. Since types in λ_i^+ can have more supertypes, its disjointness definition is expected to be stricter than λ_i. However, in λ_i, for arrow types, disjointness only cares about output types. In other words, the set of all output types in an intersection of arrow types decides the set of its disjoint types. For $(A \to B) \,\&\, (A \to C)$, if a type is disjoint to it, the type cannot contain B or C in the return type of any its components. The same criterion applies to types disjoint from $(A \to B \,\&\, C)$. Therefore, for $(A \to B) \,\&\, (A \to C)$, the additional supertype $A \to B \,\&\, C$ introduced by the distributivity rule in BCD subtyping, brings no extra *non-disjoint* type to it. Thus the disjointness definition does not change. What is more, the extended definition also has the following properties:

Lemma 6.1 (Disjointness properties)**.** Disjointness satisfies:

1. $A * B$ if and only if $A *_a B$.
2. if $A * (B_1 \to C)$ then $A * (B_2 \to C)$.

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

$$\frac{}{\top *_a A}\ \text{D-topL} \qquad \frac{}{A *_a \top}\ \text{D-topR} \qquad \frac{A_1 *_a B \qquad A_2 *_a B}{A_1 \,\&\, A_2 *_a B}\ \text{D-andL} \qquad \frac{A *_a B_1 \qquad A *_a B_2}{A *_a B_1 \,\&\, B_2}\ \text{D-andR} \qquad \frac{}{\mathsf{Int} *_a A_1 \to A_2}\ \text{D-IntArr}$$

$$\frac{}{A_1 \to A_2 *_a \mathsf{Int}}\ \text{D-arrInt} \qquad \frac{A_2 *_a B_2}{A_1 \to A_2 *_a B_1 \to B_2}\ \text{D-arrArr} \qquad \frac{}{\mathsf{Int} *_a \{l{:}A\}}\ \text{D-intRcd} \qquad \frac{}{\{l{:}A\} *_a \mathsf{Int}}\ \text{D-rcdInt}$$

$$\frac{}{A_1 \to A_2 *_a \{l{:}A\}}\ \text{D-arrRcd} \qquad \frac{}{\{l{:}A\} *_a A_1 \to A_2}\ \text{D-rcdArr} \qquad \frac{A *_a B}{\{l{:}A\} *_a \{l{:}B\}}\ \text{D-rcdEq} \qquad \frac{l_1 \neq l_2}{\{l_1{:}A\} *_a \{l_2{:}B\}}\ \text{D-rcdNeq}$$

Figure 6.2: Definition of disjointness in λ_i^+ (extends λ_i's definition in Figure 5.1).

3. if $A * B \,\&\, C$ then $A * B$ and $A * C$.

Pre-values and consistency. As shown in Section 6.1, merges like $e : A \,,, e : A$, could be produced during reduction, since merged functions both take the input. To type check such merges, we use *pre-values* to denote a sort of terms including values, annotated terms, and merges composed by them, and generalize consistency to pre-values. A pre-value's type, if it is not a merge or record, can be told directly from its form without analyzing its structure. The *principal type* of a term is the most specific one among all of its types, i.e. it is the subtype of every other type of the term. The top of Figure 6.3 shows the syntax-directed definition of principal types for pre-values. Its correctness is justified as follows.

Lemma 6.2 (Principal types)**.** For any pre-value u,

1. if $u : A$ and $\cdot \vdash u \implies B$, then $A = B$.
2. if $\cdot \vdash u \implies A$ then $u : A$.

Recall that the intuition of consistency is to allow two terms in a merge if they have disjoint types or their overlapped parts are equal. In λ_i, only values can be consistent, and the specification of consistency relies on casting, which is hard to extend to expressions. To extend consistency to pre-values, we now use an inductive relation to define consistency, where principal types are used to simplify the definition. Consistency is showed on the bottom of Figure 6.3. Notably, for values, the definition is sound and complete with respect to the specification (Definition 5.2).

Lemma 6.3 (Soundness and completeness of consistency definition)**.** For all well-typed value v_1 and v_2, $v_1 \approx v_2$ if and only if $v_1 \approx_{spec} v_2$.

Pre-values $u ::= v \mid e{:}A \mid u_1 \,,, u_2 \mid \{l = u\}$

$\boxed{u : A}$ *(Principal Type of Pre-Values)*

$$\frac{}{\top : \top}\ \text{PT-TOP} \qquad \frac{}{i : \mathsf{Int}}\ \text{PT-INT} \qquad \frac{}{(\lambda x.\, e{:}A \to B) : (A \to B)}\ \text{PT-LAM} \qquad \frac{}{(e{:}A) : A}\ \text{PT-ANNO}$$

$$\frac{u_1 : A \qquad u_2 : B}{(u_1 \,,, u_2) : (A \,\&\, B)}\ \text{PT-MERGE} \qquad \frac{u : A}{\{l = u\} : \{l{:}A\}}\ \text{PT-RCD}$$

$\boxed{u_1 \approx u_2}$ *(Consistency)*

$$\frac{}{i \approx i}\ \text{C-LIT} \qquad \frac{}{\lambda x.\, e{:}A \to B_1 \approx \lambda x.\, e{:}A \to B_2}\ \text{C-ABS} \qquad \frac{}{e{:}A \approx e{:}B}\ \text{C-ANNO} \qquad \frac{u_1 \approx u_2}{\{l = u_1\} \approx \{l = u_2\}}\ \text{C-RCD}$$

$$\frac{u_1 : A \qquad u_2 : B \qquad A *_a B}{u_1 \approx u_2}\ \text{C-DISJOINT} \qquad \frac{u_1 \approx u \qquad u_2 \approx u}{u_1 \,,, u_2 \approx u}\ \text{C-MERGEL} \qquad \frac{u \approx u_1 \qquad u \approx u_2}{u \approx u_1 \,,, u_2}\ \text{C-MERGER}$$

Figure 6.3: Definition of principal types and consistency in λ_i^+.

Typing and applicative distribution. Figure 6.4 presents the extension of typing and applicative distribution. The typing rules of application and projection rely on applicative distribution. It extends the initial definitions of λ_i in two dimensions. One is about the distributive rule of function type constructors over intersections, where rule AD-BCD-ANDARR now supports intersection of function-like types. Assuming that a term e_1 of type $(\mathsf{Int} \to \mathsf{Int}) \,\&\, (\mathsf{Bool} \to \mathsf{Bool})$ is applied to term e_2, via this relation, we can derive that e_2 should be checked against type $\mathsf{Int} \,\&\, \mathsf{Bool}$. The other dimension is to treat record types as one of the applicative forms. The additional relation $l \vdash A \rhd B$ computes the projection result of type A on label l. Rule ADR-AND combines both dimensions and enables the intersection of record-like types. For function application, it is required that the argument type matches with every function's parameter type. Differently, we employ a more flexible semantics on projection: records with incompatible labels can appear in the merge. But they produce empty results like the top type. An auxiliary function meet is used to drop the $\top$ type from the results so that the type of $\{l_1 = 1\,, l_2 = 2\,, l_3 = 3\}.l_1$ is computed as Int rather than $\mathsf{Int} \,\&\, \top \,\&\, \top$. Although $\top$ can behave like $\top \to \top$ or $\{l{:}\top\}$ when a function or record type is required, we do not extend this implicit conversion to other types. This is to avoid programs like projecting a function $(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}).l$ or applying an integer to an argument $(1\,\mathsf{true})$. Since projecting a top

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

$$\textsc{AD-bcd-arr}\ \frac{}{A \to B \rhd A \to B} \qquad \textsc{AD-bcd-topArr}\ \frac{}{\top \rhd \top \to \top}$$

$$\textsc{AD-bcd-andArr}\ \frac{A \rhd A_1 \to A_2 \quad B \rhd B_1 \to B_2}{A \,\&\, B \rhd A_1 \,\&\, B_1 \to A_2 \,\&\, B_2}$$

$\boxed{l \vdash A \rhd B}$ *(Applicative Distribution on Record Projection)*

$$\textsc{ADR-topRcd}\ \frac{}{l \vdash \top \rhd \top} \qquad \textsc{ADR-rcd}\ \frac{}{l \vdash \{l{:}A\} \rhd A}$$

$$\textsc{ADR-rcdNeq}\ \frac{l_1 \neq l_2}{l_1 \vdash \{l_2{:}A\} \rhd \top} \qquad \textsc{ADR-and}\ \frac{l \vdash A \rhd A' \qquad l \vdash B \rhd B'}{l \vdash A \,\&\, B \rhd \mathsf{meet}(A', B')}$$

$\boxed{\mathsf{meet}(A, B)}$ *(Definition of the* meet *function on types)*

$$\begin{array}{lcll} \mathsf{meet}(\top, A) & = & A & \\ \mathsf{meet}(A, \top) & = & A & \\ \mathsf{meet}(A, B) & = & A \,\&\, B & \text{when neither } A \text{ and } B \text{ is } \top \end{array}$$

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

$$\textsc{Typ-bcd-top}\ \frac{}{\Gamma \vdash \top \Rightarrow \top} \qquad \textsc{Typ-bcd-lit}\ \frac{}{\Gamma \vdash i \Rightarrow \mathsf{Int}} \qquad \textsc{Typ-bcd-var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A} \qquad \textsc{Typ-bcd-abs}\ \frac{\Gamma, x{:}A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.\, e{:}A \to B \Rightarrow A \to B}$$

$$\textsc{Typ-bcd-app}\ \frac{\Gamma \vdash e_1 \Rightarrow A \qquad A \rhd B \to C \qquad \Gamma \vdash e_2 \Leftarrow B}{\Gamma \vdash e_1\, e_2 \Rightarrow C} \qquad \textsc{Typ-bcd-proj}\ \frac{\Gamma \vdash e \Rightarrow A \qquad l \vdash A \rhd C}{\Gamma \vdash e.l \Rightarrow C}$$

$$\textsc{Typ-bcd-rcd}\ \frac{\Gamma \vdash e \Rightarrow A}{\Gamma \vdash \{l = e\} \Rightarrow \{l{:}A\}} \qquad \textsc{Typ-bcd-merge}\ \frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B \qquad A *_a B}{\Gamma \vdash e_1 \,,, e_2 \Rightarrow A \,\&\, B} \qquad \textsc{Typ-bcd-anno}\ \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash e{:}A \Rightarrow A}$$

$$\textsc{Typ-bcd-fix}\ \frac{\Gamma, x{:}A \vdash e \Leftarrow A}{\Gamma \vdash \mathbf{fix}\, x{:}A.\, e \Rightarrow A} \qquad \textsc{Typ-bcd-mergev}\ \frac{\cdot \vdash u_1 \Rightarrow A \qquad \cdot \vdash u_2 \Rightarrow B \qquad u_1 \approx u_2}{\Gamma \vdash u_1 \,,, u_2 \Rightarrow A \,\&\, B} \qquad \textsc{Typ-bcd-sub}\ \frac{\Gamma \vdash e \Rightarrow A \qquad A <: B}{\Gamma \vdash e \Leftarrow B}$$

Figure 6.4: Typing and applicative distributivity of λ_i^+ (extends Figure 5.2).

value ($\top.l$) is accepted, we also allow $\{l_1 = e\}.l_2$ when l_1 differs from l_2. When there is no label matched, the projection always has type $\top$.

Besides the changes due to records, rule Typ-bcd-mergev is generalized from values to *pre-values*. Thus, merges like $e{:}A \,,, e{:}A$ are well-typed in λ_i^+.

6.3 Operational Semantics

Casting. Compared with λ_i, the new casting reduction has one more rule for records, and two rules that change, as shown in Figure 6.5. We now need to make sure that rule Cast-bcd-arrow only applies to *ordinary* arrow types, as we will generate merges for splittable arrow types. The

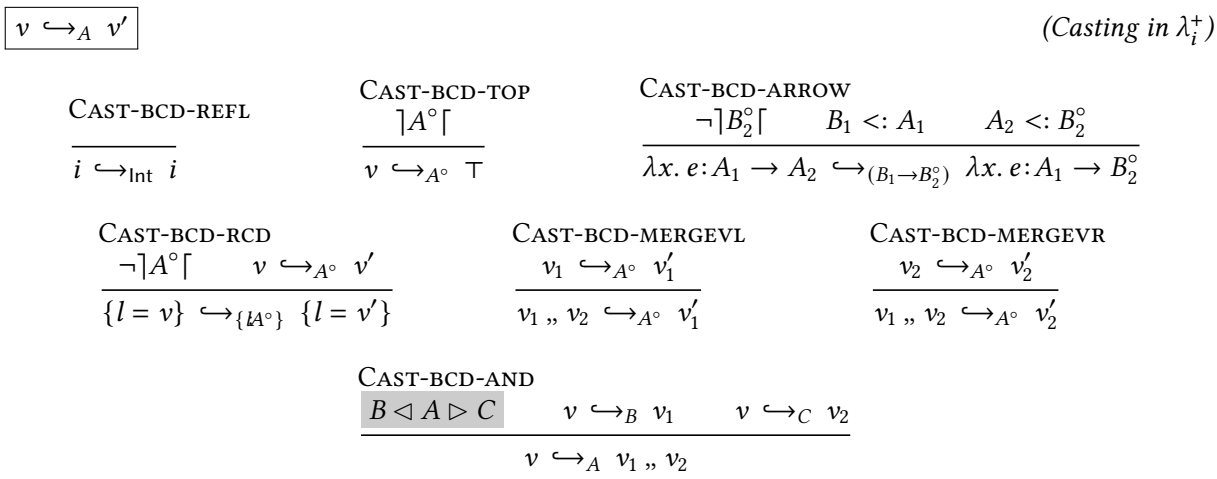

Figure 6.5: Casting of λ_i^+ (extends Figure 5.3).

condition is unnecessary in λ_i because every arrow type there is ordinary. Rule Cast-bcd-rcd mimics the arrow rule. Rule Cast-bcd-and works on splittable types, so now it needs to take care of more types than just intersections. Although we choose to merge the two results of the split types, there are some other alternative options. One possible design is to construct a lambda from the results. Therefore, we could prevent merges from being the inhabitants of arrow types. However, it is hard to maintain the commutativity of casting (Lemma 5.6) while manipulating the lambda body. In our reduction semantics, a merge of functions behaves the same as one function. So it is safe to not convert such merges to one lambda abstraction. We will review this issue in our later discussion about runtime subtyping and type soundness.

Reduction. In the reduction rules of λ_i^+, presented in Figure 6.6, rule Step-bcd-papp replaces the original beta-reduction rule and rule Step-top. Rule Step-bcd-rcd, rule Step-bcd-proj, and rule Step-bcd-pproj are added for records and record projection. The rules for merges are changed to reduce components in parallel.

Parallel application. To align record projection with the function application, we define *arguments* which abstract expressions, and record labels (at the top of Figure 6.6). The distributivity rule in BCD subtyping indicates that a merge of functions can be applied. Instead of converting the merge to a lambda abstraction, we chose to perform the application directly. An intuitive solution is to have a rule that distributes the input value, like

$$(v_1 \,,, v_2)\; v \;\hookrightarrow\; v_1\; v \,,, v_2\; v$$

Arguments $\quad arg ::= v \mid \{l\}$

$\boxed{\mathsf{meet}(v_1, v_2)}$ *(Definition of the* meet *function on values)*

$$\begin{array}{lcll} \mathsf{meet}(\top, v) & = & v & \\ \mathsf{meet}(v, \top) & = & v & \\ \mathsf{meet}(v_1, v_2) & = & v_1 \,,, v_2 & \text{when neither } v_1 \text{ and } v_2 \text{ is } \top \end{array}$$

$\boxed{v \bullet arg \hookrightarrow e}$ *(Parallel Application)*

$$\textsc{Papp-top}\ \frac{}{\top \bullet arg \hookrightarrow \top} \qquad \textsc{Papp-abs}\ \frac{v \hookrightarrow_A v'}{\lambda x.\, e : A \to B \bullet v \hookrightarrow (e[x \mapsto v']) : B} \qquad \textsc{Papp-mergeApp}\ \frac{v_1 \bullet v \hookrightarrow e_1 \qquad v_2 \bullet v \hookrightarrow e_2}{v_1 \,,, v_2 \bullet v \hookrightarrow e_1 \,,, e_2}$$

$$\textsc{Papp-proj}\ \frac{}{\{l = v\} \bullet \{l\} \hookrightarrow v} \qquad \textsc{Papp-rcdNeq}\ \frac{l_1 \neq l_2}{\{l_1 = v\} \bullet \{l_2\} \hookrightarrow \top} \qquad \textsc{Papp-absProj}\ \frac{}{\lambda x.\, e : A \to B \bullet \{l\} \hookrightarrow \top}$$

$$\textsc{Papp-mergeProj}\ \frac{v_1 \bullet \{l\} \hookrightarrow e_1 \qquad v_2 \bullet \{l\} \hookrightarrow e_2}{v_1 \,,, v_2 \bullet \{l\} \hookrightarrow \mathsf{meet}(e_1, e_2)}$$

$\boxed{e \hookrightarrow e'}$ *(Small-Step Semantics)*

$$\textsc{Step-bcd-papp}\ \frac{v_1 \bullet v_2 \hookrightarrow e}{v_1\ v_2 \hookrightarrow e} \qquad \textsc{Step-bcd-pproj}\ \frac{v \bullet \{l\} \hookrightarrow v'}{v.l \hookrightarrow v'} \qquad \textsc{Step-bcd-annov}\ \frac{v \hookrightarrow_A v'}{v : A \hookrightarrow v'} \qquad \textsc{Step-bcd-appl}\ \frac{e_1 \hookrightarrow e_1'}{e_1\ e_2 \hookrightarrow e_1'\ e_2} \qquad \textsc{Step-bcd-appr}\ \frac{e_2 \hookrightarrow e_2'}{v_1\ e_2 \hookrightarrow v_1\ e_2'}$$

$$\textsc{Step-bcd-merge}\ \frac{e_1 \hookrightarrow e_1' \qquad e_2 \hookrightarrow e_2'}{e_1 \,,, e_2 \hookrightarrow e_1' \,,, e_2'} \qquad \textsc{Step-bcd-mergel}\ \frac{e_1 \hookrightarrow e_1'}{e_1 \,,, v_2 \hookrightarrow e_1' \,,, v_2} \qquad \textsc{Step-bcd-merger}\ \frac{e_2 \hookrightarrow e_2'}{v_1 \,,, e_2 \hookrightarrow v_1 \,,, e_2'} \qquad \textsc{Step-bcd-anno}\ \frac{e \hookrightarrow e'}{e : A \hookrightarrow e' : A}$$

$$\textsc{Step-bcd-fix}\ \frac{}{\mathbf{fix}\ x : A.\, e \hookrightarrow e[x \mapsto \mathbf{fix}\ x : A.\, e] : A} \qquad \textsc{Step-bcd-rcd}\ \frac{e \hookrightarrow e'}{\{l = e\} \hookrightarrow \{l = e'\}} \qquad \textsc{Step-bcd-proj}\ \frac{e \hookrightarrow e'}{e.l \hookrightarrow e'.l}$$

Figure 6.6: Parallel application and reduction of λ_i^+ (extends Figure 5.5).

Assuming that v_1 and v_2 are consistent but not disjoint, to obtain preservation, $v_1\ v$ and $v_2\ v$ have to be consistent. To avoid the complexity of extending consistency to expressions including applications, we design *parallel application* (Figure 6.6) to distribute and substitute the input value in a big-step style, where a function application is divided into two parts v and arg, and steps to an expression e.

Consider a merge of three functions being applied to a value. Compared to adding the previous single rule to the small-step reduction, parallel reduction helps us to "jump" from $(f_1 \,,\, f_2 \,,\, f_3)\ v$ to a merge of annotated terms when reasoning about reduction. Every lambda gets the input directly without Intermediate reduction steps such as $((f_1 \,,\, f_2)\ v) \,,\, f_3\ v$. Record projection is handled in a similar style. In this case, v is a record value, and arg stands for a label instead.

$$\{l = 1\,,\, l = \mathsf{true}\,,\, l = 1\,,\, l = \top\,,\, m = 2\} \bullet \{l\} \hookrightarrow 1 \,,, \mathsf{true} \,,, 1$$

The above example shows how merged records are projected in parallel, and the whole term is kept consistent. We use the meet function to drop the top values as they may be produced by records with unmatched labels. Meanwhile, records that only contains a top value are also ignored.

Rule Papp-top shows that the top value can be used as a function that returns $\top$, or a record which contains $\top$ in its field. With it, rule Step-bcd-papp subsumes the rule Step-top in Figure 5.5.

Parallel reduction of merges. To maintain consistency of subterms in a merge (which may contain non-values), as required by the typing rule Typ-bcd-mergev, we reduce every component in a merge simultaneously through rule Step-bcd-merge. This rule is helpful to preserve consistency of pre-values during reduction, and therefore enables the type preservation theorem. As a counter-example, if parallel reduction is not employed, we might encounter the following reduction step:

$$(1 + 1) : \mathsf{Int} \,,, (1 + 1) : \mathsf{Int} \hookrightarrow 2 : \mathsf{Int} \,,, (1 + 1) : \mathsf{Int}$$

where the reduced term is ill-typed, since the subterms are not consistent any more. In contrast, rule Step-bcd-merge will keep the reduction of terms synchronized. When one of the subterms is already a value, rule Step-bcd-merge no longer applies. In that case, rule Step-bcd-mergel or rule Step-bcd-merger reduces the other subterm.

Note that for systems that have no side effects, like λ_i^+, the evaluation order of merges does not change the final result. This is to say, even if the reduction of merges is not conducted in parallel, a well-typed program will eventually step to a value that preserves its type. Potentially, with an extended definition of consistency, we can prove type soundness for the conventional

reduction on merges. The extended consistency should relate Kleene-equivalent terms, which have the same value form, like 2 and 1 + 1.

Overloading on return types. The λ_i^+ calculus support a form of overloading on return types. We can even merge multiple functions and apply them together to one input. Eventually, the return value would be a combination of the outputs of all functions, and can play the role of any single output. The following example shows how this mechanism works.

$$
\begin{array}{ll}
 & \mathsf{not}\ ((\lambda x.\ x + 1 : \mathsf{Int} \to \mathsf{Int} \,,,\, \lambda x.\ \mathsf{true} : \mathsf{Int} \to \mathsf{Bool})\ 1) \\
\hookrightarrow & \{\ \text{by } \textsc{Step-bcd-papp} \text{ and parallel application}\ \} \\
 & \mathsf{not}\ (2 : \mathsf{Int} \,,,\, \mathsf{true} : \mathsf{Bool}) \\
\hookrightarrow & \{\ \text{by } \textsc{Step-bcd-appr}, \textsc{Step-bcd-merge}, \textsc{Step-bcd-annov} \text{ and casting}\ \} \\
 & \mathsf{not}\ (2 \,,,\, \mathsf{true}) \\
\hookrightarrow & \{\ \text{assuming not has type } \mathsf{Bool} \to \mathsf{Bool}\ \} \\
 & \mathsf{false} : \mathsf{Bool} \\
\hookrightarrow & \{\ \text{by } \textsc{Step-bcd-annov} \text{ and casting}\ \} \\
 & \mathsf{false}
\end{array}
$$

6.4 Metatheory

6.4.1 Completeness of the type system with respect to the original λ_i^+'18 (or NeColus) calculus.

Besides the extra rule for consistent merges (rule Typ-bcd-mergev), λ_i^+ has two different rules for record projection and function application when compared with the type system of λ_i^+'18 (NeColus) [BOS18].

$\boxed{\Gamma \vdash_n e \Leftrightarrow A}$ *(λ_i^+'18 Typing (Selected))*

$$
\textsc{Nec-T-app}\ \frac{\Gamma \vdash_n e_1 \Rightarrow A_1 \to A_2 \qquad \Gamma \vdash_n e_2 \Leftarrow A_1}{\Gamma \vdash_n e_1\ e_2 \Rightarrow A_2}
\qquad\qquad
\textsc{Nec-T-proj}\ \frac{\Gamma \vdash_n e \Rightarrow \{l : A\}}{\Gamma \vdash_n e.l \Rightarrow A}
$$

To show that every well-typed term in λ_i^+'18 can be type-checked in λ_i^+, we prove the following lemmas:

Lemma 6.4 (λ_i^+ application subsumes λ_i^+'18's application)**.** For any expressions e_1 and e_2, if $\Gamma \vdash e_1 \Rightarrow A \to B$ and $\Gamma \vdash e_2 \Leftarrow A$, then $\Gamma \vdash e_1\ e_2 \Rightarrow B$.

Lemma 6.5 (λ_i^+ projection subsumes λ_i^+'18's projection)**.** For any expressions e and any label l, if $\Gamma \vdash e \Rightarrow \{l : A\}$, then $\Gamma \vdash e.l \Rightarrow C$.

$\boxed{A \ll B}$ *(Runtime Subtyping for λ_i^+)*

$$\frac{}{A \ll A}\ \textsc{Rsub-bcd-z} \qquad \frac{B_1 \lhd B \rhd B_2 \qquad A_1 \ll B_1 \qquad A_2 \ll B_2}{A_1 \,\&\, A_2 \ll B}\ \textsc{Rsub-bcd-split} \qquad \frac{B_1 <: A_1 \qquad A_2 \ll B_2}{A_1 \to A_2 \ll B_1 \to B_2}\ \textsc{Rsub-bcd-arr}$$

$$\frac{B_1 \lhd B \rhd B_2 \qquad \rceil B_1 \lceil \qquad A \ll B_2}{A \ll B}\ \textsc{Rsub-bcd-topl} \qquad \frac{\rceil A \lceil}{\top \ll A}\ \textsc{Rsub-bcd-top} \qquad \frac{B_1 \lhd B \rhd B_2 \qquad \rceil B_2 \lceil \qquad A \ll B_1}{A \ll B}\ \textsc{Rsub-bcd-topr}$$

Figure 6.7: Runtime subtyping of λ_i^+, extending the definition in Figure 5.4.

Then it is straightforward that λ_i^+'18 can be translated into λ_i^+. In our Coq formalization we designed an elaboration from λ_i^+'18 to λ_i^+ and proved the completeness of λ_i^+'s type system with respect to λ_i^+'18.

6.4.2 Properties of the TDOS.

The TDOS of λ_i^+ preserves determinism, progress, and subject-reduction. Most of the proof follows λ_i's structure. Runtime subtyping is extended, and the newly added parallel application requires some extra lemmas.

Runtime subtyping and preservation. We have seen that any term of a subtype can be cast to its supertype given explicit type annotations. But for a subset of the subtyping relation, we directly use values of the subtype as values of its supertype. This relation is runtime subtyping. Compared with the definition in λ_i, λ_i^+'s runtime subtyping has one rule changed for distributivity, and three rules added for records. As a consequence of the generalization of rule RSub-and to rule RSub-bcd-split, an intersection type can be a runtime subtype of an arrow type, or other splittable types. It corresponds the use of merges as a function or a record. Rules RSub-bcd-topl and RSub-bcd-topr are due to the elimination of the top value in projection. Since a top value may stand for a top-like type, these two rules allow dropping any top-like types in an intersection.

A key property of runtime subtyping is that it preserves the disjointness relation. We have know that if A is a subtype of B then any type disjoint with A is also disjoint with B. But the reverse direction only holds for runtime subtypes.

Lemma 6.6 (Runtime subtyping preserves disjointness)**.** If $A_1 \lesssim A_2$, then $A_1 * B$ if and only if $A_2 * B$.

Parallel application. For both function application and record projection, parallel reduction preserves the original type.

Lemma 6.7 (Type preservation of parallel application)**.**

- If $\cdot \vdash v_1\ v_2 \Rightarrow A$, and $v_1 \bullet v_2 \hookrightarrow e$ then $\cdot \vdash e \Rightarrow A$.
- If $\cdot \vdash v_1.l \Rightarrow A$, and $v_1 \bullet \{l\} \hookrightarrow e$ then $\cdot \vdash e \Rightarrow A$.

Furthermore we can prove the following determinism lemmas:

Lemma 6.8 (Determinism of parallel application)**.**

- If $\cdot \vdash v_1\ v_2 \Rightarrow A$, $v_1 \bullet v_2 \hookrightarrow e_1$, $v_1 \bullet v_2 \hookrightarrow e_2$ then $e_1 = e_2$.
- If $\cdot \vdash v_1.l \Rightarrow A$, $v_1 \bullet \{l\} \hookrightarrow e_1$, $v_1 \bullet \{l\} \hookrightarrow e_2$ then $e_1 = e_2$.

We can also prove the following progress lemmas for parallel application:

Lemma 6.9 (Progress of parallel application)**.**

- If $\cdot \vdash v_1\ v_2 \Rightarrow A$, then $\exists e$, $v_1 \bullet v_2 \hookrightarrow e$.
- If $\cdot \vdash v_1.l \Rightarrow A$, then $\exists e$, $v_1 \bullet \{l\} \hookrightarrow e$.

Type Soundness. Finally, based on all the lemmas above, the key properties of reduction can be derived, including type preservation with runtime subtyping:

Lemma 6.10 (Type preservation of $\hookrightarrow$ with respect to runtime subtyping)**.** If $\cdot \vdash e \Rightarrow A$, and $e \hookrightarrow e'$ then exists B, $\cdot \vdash e' \Rightarrow B$ and $B \lesssim A$.

And its corollary:

Theorem 6.1 (Type preservation of $\hookrightarrow$)**.** If $\cdot \vdash e \Rightarrow A$, and $e \hookrightarrow e'$ then $\cdot \vdash e' \Leftarrow A$.

Determinism and progress theorems are proved as well.

Theorem 6.2 (Determinism of $\hookrightarrow$)**.** If $\cdot \vdash e \Rightarrow A$, $e \hookrightarrow e_1$, $e \hookrightarrow e_2$, then $e_1 = e_2$.

Theorem 6.3 (Progress of $\hookrightarrow$)**.** If $\cdot \vdash e \Rightarrow A$, then e is a value or $\exists e'$, $e \hookrightarrow e'$.

Summary From λ_i to λ_i^+, we continue with the type-directed reduction in the call-by-value style. Subtyping is mainly interpreted as implicit conversions on values, and defined by the *casting* relation. *Runtime subtyping* is a subset of subtyping that has inclusive semantics. Both relations are extended for the addition of distributive rules, mainly to treat merges not only as values of intersections, but also as values of *splittable types*.

λ_i^+ supports both BCD subtyping and record concatenation. Compared with the elaboration approach, having a direct semantics avoids the translation process and a target calculus. This simplifies both informal and formal reasoning. For instance, establishing the coherence of elaboration in the original λ_i^+ [BOS18] requires much more sophistication than obtaining the determinism theorem in λ_i^+. Furthermore, the proof method for coherence in λ_i^+'18 cannot deal with non-terminating programs, whereas dealing with recursion is straightforward in λ_i and λ_i^+.

CHAPTER 7

The F_i^+ Calculus with Disjoint Polymophism

As we have seen in Section 2.5.2, the Compositional Programming language CP supports multiple inheritance and solves the Expression Problem in a modular way. CP is implemented on top of a polymorphic core language with disjoint intersection types called F_i^+.

This chapter presents a new formulation of F_i^+ based on the *type-directed operational semantics* (TDOS) that has been employed in the previous two chapters. We extend the TDOS approach can to languages with disjoint polymorphism and model the full F_i^+ calculus.

Our new formulation of F_i^+ differs from the original one in three aspects. Firstly, the semantics of the original F_i^+ [Bi+19]is given by elaborating to F_{co}, while our semantics for F_i^+ is a direct operational semantics. Secondly, our new formulation of F_i^+ supports recursion and impredicative polymorphism. Finally, we employ a call-by-name evaluation strategy, motivated by the need for lazy fixpoints to model the semantics of trait instantiation in Compositional Programming.

7.1 Motivations and Technical Innovations

In this section, we show how CP traits elaborate to F_i^+ expressions and discuss the practical issues that motivate us to reformulate F_i^+ as well as technical challenges and innovations.

7.1.1 Elaborating CP to F_i^+

The semantics of CP and its notion of traits is defined via an elaboration to the core calculus F_i^+. The elaboration of traits is inspired by Cook's *denotational semantics of inheritance* [Coo89]. In the denotational semantics of inheritance, the key idea is that mechanisms such as classes or traits, which support *self-references* (aka the `this` keyword in conventional OOP languages), can

be modeled via *open recursion*. In other words, the encoding of classes or traits is parametrized by a self-reference. This allows late binding of self-references at the point of instantiation and enables the modification and composition of traits before instantiation. Instantiation happens when `new` is used, just as in conventional OOP languages. When `new` is used, it essentially closes the recursion by binding the self-reference, which then becomes a recursive reference to the instantiated object. In the denotational semantics of inheritance, `new` is just a fixpoint operator. Here we focus on the elaboration of traits, and take a closer look at the connection between CP and F_i^+ expressions. We refer curious readers to the work by Zhang et al. [ZSO21] for the full formulation of the type-directed elaboration of CP.

Elaborating traits into F_i^+ To use a concrete example, we revisit the trait `repo` defined in Section 2.5.2. Both the creation and instantiation of traits are included in the definition of `repo`:

```
repo Exp = trait [self : ExpSig<Exp>] ⇒ {
  num = Add (Lit 4) (Lit 8);
  var = ...
};
```

The CP code above is elaborated to the corresponding F_i^+ code of the form:

```
repo = ΛExp. \_(self : ExpSig<Exp>).
  let $Lit = self.Lit in let $Add = self.Add in
  let $Let = self.Let in let $Var = self.Var in
    { num = fix self:Exp. $Add (fix self:Exp. $Lit 4 self)
                               (fix self:Exp. $Lit 8 self) self } ,,
    { var = ... };
```

The type parameter `Exp` in the `repo` trait is expressed by a System-F-style type lambda ($\Lambda X.\, e$). Note that CP employs a form of syntactic sugar for constructors to allow concise use of constructors and avoid explicit uses of `new`. The source code `Add (Lit 4) (Lit 8)` is first expanded into `new $Add (new $Lit 4) (new $Lit 8)`, which inserts `new` operators. Next we describe the elaboration process of creating and instantiating traits:

- **Creation of traits**: A `trait` is elaborated to a *generator* function whose parameter is a self-reference (like `self` above) and whose body is a record of methods;
- **Instantiation of traits**: The `new` construct is used to instantiate a trait. Uses of `new` are elaborated to a *fixpoint* which applies the elaborated trait function to a self-reference. In the definition of the field `num` there are three elaborations of `new`. For instance, the CP code `new $Lit 4` corresponds to the F_i^+ code `fix self:Exp. $Lit 4 self`.

It is clear now that our trait encoding is heavily dependent on recursion, due to the self-references employed by the encoding. However, the original F_i^+ does not support recursion, which reveals a gap between theory and practice.

7.1.2 The Gap Between Theory and Practice

Our primary motivation to reformulate F_i^+ is to bridge the gap between theory and practice. The original formulation of F_i^+ lacks recursion and impredicative polymorphism, and uses the traditional call-by-value (CBV) evaluation strategy. However, the recent work of CP assumes a different variant of F_i^+ that is equipped with fixpoints and the call-by-name (CBN) evaluation. It is worthwhile to probe into the causes of such differences.

Non-triviality of coherence Recursion is essential for general-purpose computation in programming. More importantly, our encoding of traits requires recursion. For example, `new e` is elaborated to `fix self. e self`. Unfortunately, in the original F_i^+ the proof technique for coherence is based on a logical relation called *canonicity* [BOS18], and it does not immediately scale to recursive programs and programs with impredicative polymorphism. A possible solution, known from the research of logical relations, is to move to a more sophisticated *step-indexed* form of logical relations [Ahm06]. However, this requires a major reformulation of the proofs and metatheory of the original F_i^+, and it is not clear whether additional challenges would be present in such an extension. Thus, the lack of the two features in the theory of the original F_i^+ remains a serious limitation since only terminating programs and predicative polymorphism are considered. In other words, we cannot encode traits as presented in Section 7.1.1 in the original F_i^+. To get around this issue and enable the encoding of traits, Zhang et al. [ZSO21] simply assumed an extension of F_i^+ with recursion and their proof of coherence for CP was done under the assumption that the original F_i^+ with recursion was coherent or deterministic.

Evaluation strategies The semantics of the F_{co} calculus is *call-by-value* (CBV) and, by inheritance, the elaboration semantics of the original F_i^+ has a CBV semantics as well, like most mainstream programming languages. But in CP, the fixpoint operators must be *lazy*; otherwise, self-references can easily trigger non-termination. CBN is a more natural evaluation strategy for object encodings such as Cook's denotational semantics of inheritance. As stated by Bruce et al. in their work on object encodings [BCP97]:

> *"Although we shall perform conversion steps in whatever order is convenient for the sake of examples, we could just as well impose a call-by-name reduction strategy. (Most of the examples would diverge under a call-by-value strategy. This can be repaired at the cost of some extra lambda abstractions and applications to delay evaluation at appropriate points.)"*

In our elaboration of traits, we adopt a similar approach to object encodings. For example, consider the following CP expression:

```
type A = { l1 : Int; l2 : Int };
new (trait [self : A] ⇒ { l1 = 1; l2 = self.l1 })
```

which is elaborated to the following (slightly simplified) F_i^+ expression:

$$\textbf{fix}\ self : A.\ \{l_1 = 1\} \,,,\, \{l_2 = self.l_1\}$$

The `trait` expression is elaborated to a function, and the `new` expression turns the function into a fixpoint. Unfortunately, this expression terminates under CBN but diverges under CBV. If evaluated under CBV, the variable *self* will be evaluated repeatedly, despite the fact that only $self.l_1$ is used:

$$\begin{aligned}
& \textbf{fix}\ self : A.\ \{l_1 = 1\} \,,,\, \{l_2 = self.l_1\} \\
\hookrightarrow\ & \{l_1 = 1\} \,,,\, \{l_2 = (\textbf{fix}\ self : A.\ \{l_1 = 1\} \,,,\, \{l_2 = self.l_1\}).l_1\} \\
\hookrightarrow\ & \{l_1 = 1\} \,,,\, \{l_2 = (\{l_1 = 1\} \,,,\, \{l_2 = (\textbf{fix}\ self : A.\ \{l_1 = 1\} \,,,\, \{l_2 = self.l_1\}).l_1\}).l_1\} \\
\hookrightarrow\ & \cdots
\end{aligned}$$

We may tackle the problem of non-termination by wrapping self-references in *thunks*, but CBN provides a simpler and more natural way. In our CBN formulation of F_i^+, $\{l = e\}$ is already a value (instead of $\{l = v\}$), so we do not need to further evaluate e:

$$\begin{aligned}
& \textbf{fix}\ self : A.\ \{l_1 = 1\} \,,,\, \{l_2 = self.l_1\} \\
\hookrightarrow\ & \{l_1 = 1\} \,,,\, \{l_2 = (\textbf{fix}\ self : A.\ \{l_1 = 1\} \,,,\, \{l_2 = self.l_1\}).l_1\}
\end{aligned}$$

The l_2 field is further evaluated only when a record projection is performed:

$$\begin{aligned}
& (\textbf{fix}\ self : A.\ \{l_1 = 1\} \,,,\, \{l_2 = self.l_1\}).l_2 \\
\hookrightarrow\ & (\{l_1 = 1\} \,,,\, \{l_2 = (\textbf{fix}\ self : A.\ \{l_1 = 1\} \,,,\, \{l_2 = self.l_1\}).l_1\}).l_2 \\
\hookrightarrow\ & (\textbf{fix}\ self : A.\ \{l_1 = 1\} \,,,\, \{l_2 = self.l_1\}).l_1 \\
\hookrightarrow\ & (\{l_1 = 1\} \,,,\, \{l_2 = (\textbf{fix}\ self : A.\ \{l_1 = 1\} \,,,\, \{l_2 = self.l_1\}).l_1\}).l_1 \\
\hookrightarrow\ & 1
\end{aligned}$$

This example illustrates how our new CBN formulation of F_i^+ avoids non-termination of trait instantiation.

Our Proof Flow Our work shows that the TDOS approach can be extended to languages with disjoint polymorphism and model the complete F_i^+ calculus with recursion and impredicative polymorphism. Moreover, With a TDOS, there is no need for a coherence proof. Instead, we can prove that the semantics is *deterministic*. The proof of determinism uses only simple reasoning techniques, such as straightforward induction, and is able to handle problematic features such as recursion and impredicative polymorphism. Thus, this removes the gap between

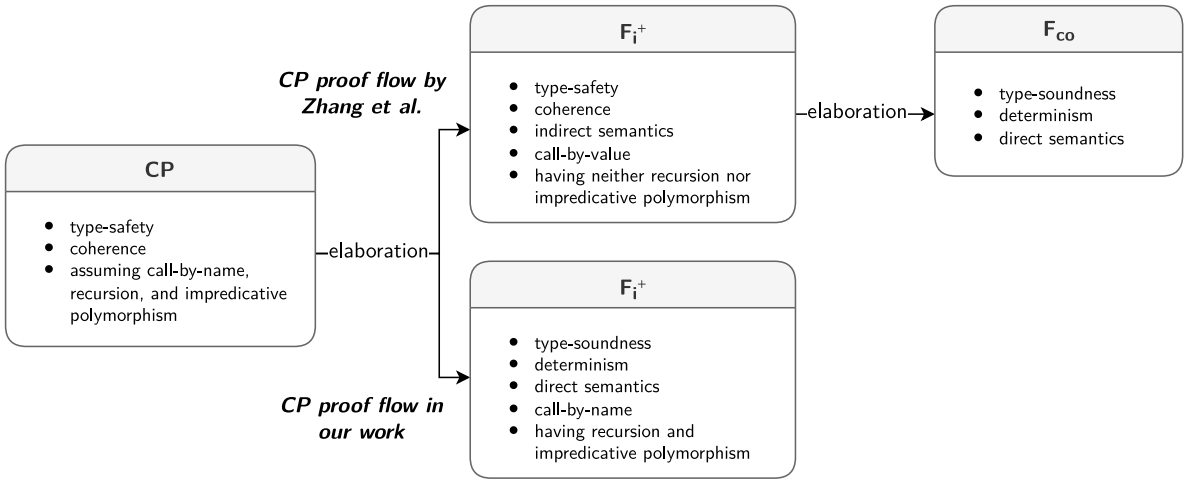

Figure 7.1: Contrasting the flow of results for CP using the original formulation, and our work.

theory and practice and validates the original proofs of correctness for the CP language. Figure 7.1 contrasts the differences in terms of proofs and implementation of CP using Zhang et al.'s original work and our own work. We formalized the TDOS variant of the F_i^+ calculus, together with its type-soundness and determinism proof in the Coq proof assistant. Moreover, we have a new implementation of CP based on our new reformulation of F_i^+, available at `https://plground.org`.

7.1.3 Technical Challenges and Innovations

While our reformulation of F_i^+ mostly follows the framework of λ_i^+, there are two main changes that bring some technical challenges: the addition of disjoint polymorphism to subtyping, and the use of call-by-name instead of call-by-value strategy in evaluation.

TDOS and function annotations In casting, values in a merge are selected based on type information. In the absence of runtime type-checking, we need to know the type of input value syntactically to match it with the target type. Thus, functions must be accompanied by type annotations. In the previous system λ_i^+ Chapter 6, functions have the form of $\lambda x.\, e : A \to C$. While the original argument type A is always kept during reduction, λ_i^+'s casting relation may generate a value that has a proper subtype of the requested type: $\lambda x.\, e : A \to C \hookrightarrow_{B_1 \to B_2} \lambda x.\, e : A \to B_2$. In F_i^+ we make casting more precise with a more liberal syntax in F_i^+. We allow bare abstractions $\lambda x : A.\, e$ while λ_i^+ does not. Our casting relation requires lambdas to be annotated $(\lambda x : A.\, e) : B$, but the full annotation B does not have to be a function type. For example, $(\lambda x : \mathsf{Int}.\, x \,,, \mathsf{true}) : (\mathsf{Int} \to \mathsf{Int}) \,\&\, (\mathsf{Int} \to \mathsf{Bool})$ still acts as a function, and is equivalent to $(\lambda x : \mathsf{Int}.\, x \,,, \mathsf{true}) : \mathsf{Int} \to \mathsf{Int} \,\&\, \mathsf{Bool}$. In beta reduction, instead of casting the argument, we wrap it by annotation for lazy evaluation.

Algorithmic subtyping with disjoint polymorphism F_i^+ extends the BCD-style distributive subtyping [BCD83] to disjoint polymorphism. A disjointness constraint can also be added to a type variable in a System F-style polymorphic type, such as $\forall\alpha * \mathsf{Int}.\ \alpha\ \&\ \mathsf{Int}$. $\forall\alpha * \mathsf{Int}.\ \alpha\ \&\ \mathsf{Int}$ represents the intersection of some type α and Int assuming α is disjoint to Int. Like arrows or records, such universal types distribute over intersections. Hence, $(\forall\alpha * \mathsf{Int}.\ \alpha)\ \&\ (\forall\alpha * \mathsf{Int}.\ \mathsf{Int})$ is a subtype of $\forall\alpha * \mathsf{Int}.\ \alpha\ \&\ \mathsf{Int}$. We extend the type-splitting approach introduced in Chapter 3 to disjoint polymorphic types to obtain an algorithmic formulation. Like intersections distribute over arrows and records, they distribute over universal quantifiers as well.

Enhanced subtyping and disjointness with more top-like types Unlike previous systems with disjoint polymorphism [AOS17; Bi+19], we add a context in subtyping judgments to track the disjointness assumption $\alpha *_a A$ whenever we open a universal type $\forall\alpha * A.\ B$, similar to the subtyping with F-bounded quantification. The extra information enhances subtyping: we know a type must be a supertype of $\top$, if it is disjoint with $\bot$. As an immediate result, $\forall\alpha * \bot.\ \alpha$ becomes a supertype to the greatest type $\top$. This also fixes the following broken property in the original F_i^+, as we now have more types that are *top-like*.

Definition 5.1 (Disjoint specification). $A * B \triangleq \forall C$ if $A \leqslant C$ and $B \leqslant C$ then $\rceil C \lceil$

To obtain a deterministic operational semantics, it is necessary for us to keep all the common supertypes of two disjoint types equivalent to $\top$ Meanwhile, we prove our subtyping and disjointness relations are decidable in Coq. Note that in the original F_i^+, the decidability of the two relations was proved manually, although the rest of the proof was mechanized.

7.2 The F_i^+ Calculus and Its Operational Semantics

This section introduces the F_i^+ calculus, including its static and dynamic semantics.

7.2.1 Syntax

The syntax of F_i^+ is as follows:

Types	$A, B, C ::= \alpha \mid \mathsf{Int} \mid \top \mid \bot \mid A\,\&\,B \mid A \to B \mid \forall\alpha * A.\ B \mid \{l{:}A\}$
Checkable terms	$p ::= \lambda x{:}A.\ e \mid \Lambda\alpha.\ e \mid \{l = e\}$
Expressions	$e ::= p \mid x \mid i \mid \top \mid e : A \mid e_1 \,,, e_2 \mid \mathbf{fix}\ x{:}A.\ e \mid e_1\ e_2 \mid e\,A \mid e.l$
Values	$v ::= p \mid p{:}A \mid i \mid \top \mid v_1 \,,, v_2$
Term contexts	$\Gamma ::= \cdot \mid \Gamma\,,\, x{:}A$
Type contexts	$\Delta ::= \cdot \mid \Delta\,,\, \alpha *_a A$

Types Two kinds of types are added: the uninhabited type $\bot$ and the universal quantified type with a disjointness restriction $\forall\alpha * A.\, B$, which expresses that the type variable α is bound inside B and disjoint to type A.

Expressions We introduce two new constructs for disjoint polymorphism: $\Lambda\alpha.\, e$ stands for type abstractions, and $e\, A$ is for type applications. As we will explain later with the typing rules, some expressions do not have an inferred type (or principal type), including lambda abstractions, type abstractions, and single-field records. We use metavariable p to represent these expressions, which with optional annotations, are values. Also, note that expressions inside record values do not have to be a value since our calculus employs call-by-name.

Contexts We have two contexts: Γ tracks the types of term variables; Δ tracks the disjointness information of type variables, which follows the original design of F_i^+. Although they can be combined, keeping type and term contexts apart brings clarity to the relations which only care about types, such as subtyping. We use $\Delta \vdash A$, $\vdash \Delta$, and $\Delta \vdash \Gamma$ judgments for the type well-formedness and the context well-formedness (defined in Figure 7.2). For multiple type well-formedness judgments, we combine them into one, i.e., $\Delta \vdash A\,, B$ means $\Delta \vdash A$ and $\Delta \vdash B$.

7.2.2 Subtyping

Figure 7.3 shows our subtyping relation, which extends BCD-style subtyping with disjoint polymorphism, records, and the bottom type. Compared with the original F_i^+, we add a context to track type variables and their disjointness information. The context not only ensures the well-formedness of types, but is also important to our new rule DS-FIP-TOPVAR. An equivalence relation (Definition 7.1) is defined on types that are subtype of each other. These equivalent types can be converted back and forth without loss of information.

Definition 7.1 (Type equivalence)**.** $\Delta \vdash A \equiv B \triangleq \Delta \vdash A \leqslant B$ and $\Delta \vdash B \leqslant A$.

For functions (rule DS-FIP-ARROW) and disjoint quantifications (rule DS-FIP-ALL), subtyping is covariant in positive positions and contravariant in negative positions. The intuition is that type abstractions of the more specific type (subtype) should have a *looser* disjointness constraint for the parameter type. $\forall\alpha * \top.\, A$ denotes that there is no constraint on α, since $\top$ is disjoint to all types. On the contrary, $\bot$ is the strictest constraint. It is useful in types like $\forall\alpha * \{l : \bot\}.\, A$, which expresses that α does not contain any informative field of label l. For intersection types, rules DS-FIP-ANDL, DS-FIP-ANDR, and DS-FIP-AND axiomatize that $A \,\&\, B$ is the greatest lower bound of A and B. As a typical characteristic of BCD-style subtyping, type constructors distribute over intersections, including arrows (rule DS-FIP-DISTARROW), records (rule DS-FIP-DISTRCD) and disjoint quantifications (rule DS-FIP-DISTALL).

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

$$\frac{}{\Delta \vdash \top}\ \text{TW-TOP} \qquad \frac{}{\Delta \vdash \bot}\ \text{TW-BOT} \qquad \frac{}{\Delta \vdash \mathsf{Int}}\ \text{TW-INT} \qquad \frac{\alpha *_a A \in \Delta}{\Delta \vdash \alpha}\ \text{TW-VAR} \qquad \frac{\Delta \vdash A}{\Delta \vdash \{l{:}A\}}\ \text{TW-RCD} \qquad \frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \to B}\ \text{TW-ARROW}$$

$$\frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \,\&\, B}\ \text{TW-AND} \qquad \frac{\Delta \vdash A \qquad \Delta, \alpha *_a A \vdash B}{\Delta \vdash \forall \alpha * A.\, B}\ \text{TW-ALL}$$

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

$$\frac{}{\vdash \cdot}\ \text{TCW-EMPTY} \qquad \frac{\vdash \Delta \qquad \Delta \vdash A}{\vdash \Delta, \alpha *_a A}\ \text{TCW-CONS}$$

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

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

Figure 7.2: Well-formedness rules in F_i^+.

Another feature of BCD subtyping, which is often overlooked, is the generalization of *top-like types*, i.e. supertypes of $\top$.

Definition 7.2 (Specification of top-like types)**.** $\Delta \vdash \rceil A \lceil \triangleq \Delta \vdash A \equiv \top$.

Initially, top-like types include $\top$ and intersections like $\top \,\&\, \top$. But the BCD subtyping adds $\top \to \top$ to it via rule DS-FIP-TOPARROW, as well as $A \to \top$ for any type A due to the contravariance of function parameters. Rule DS-FIP-TOPARROW can be viewed as a special case of rule DS-FIP-DISTARROW where intersections are replaced by $\top$ (one can consider it as an intersection of zero components). Like the original F_i^+, we extend this idea to universal types and record types (rules DS-FIP-TOPALL and DS-FIP-TOPRCD).

The most important change is the rule DS-FIP-TOPVAR. This rule means that a type variable is top-like if it is disjoint with the bottom type. Every type B is a common supertype of B itself and $\bot$. If B is disjoint with $\bot$, then it must be top-like. We proved that subtyping is decidable via an equivalent algorithmic formulation.

The discussion about algorithmic subtyping is in Section 7.3.1.

Lemma 7.1 (Decidability of subtyping)**.** $\Delta \vdash A \leqslant B$ is decidable.

$\boxed{\Delta \vdash A \leqslant B}$ *(Declarative Subtyping)*

$$\frac{\vdash \Delta \qquad \Delta \vdash A}{\Delta \vdash A \leqslant A}\ \textsc{DS-fip-refl} \qquad \frac{\Delta \vdash A \leqslant B \qquad \Delta \vdash B \leqslant C}{\Delta \vdash A \leqslant C}\ \textsc{DS-fip-trans} \qquad \frac{\vdash \Delta \qquad \Delta \vdash A}{\Delta \vdash A \leqslant \top}\ \textsc{DS-fip-top} \qquad \frac{\vdash \Delta \qquad \Delta \vdash A}{\Delta \vdash \bot \leqslant A}\ \textsc{DS-fip-bot}$$

$$\frac{\Delta \vdash A \leqslant B \qquad \Delta \vdash A \leqslant C}{\Delta \vdash A \leqslant B \,\&\, C}\ \textsc{DS-fip-and} \qquad \frac{\vdash \Delta \qquad \Delta \vdash A, B}{\Delta \vdash A \,\&\, B \leqslant A}\ \textsc{DS-fip-andl} \qquad \frac{\vdash \Delta \qquad \Delta \vdash A, B}{\Delta \vdash A \,\&\, B \leqslant B}\ \textsc{DS-fip-andr}$$

$$\frac{\Delta \vdash A_2 \leqslant A_1 \qquad \Delta \vdash B_1 \leqslant B_2}{\Delta \vdash A_1 \to B_1 \leqslant A_2 \to B_2}\ \textsc{DS-fip-arrow} \qquad \frac{\vdash \Delta \qquad \Delta \vdash A, B, C}{\Delta \vdash (A \to B) \,\&\, (A \to C) \leqslant A \to B \,\&\, C}\ \textsc{DS-fip-distArrow} \qquad \frac{\vdash \Delta}{\Delta \vdash \top \leqslant \top \to \top}\ \textsc{DS-fip-topArrow}$$

$$\frac{\Delta \vdash A \leqslant B}{\Delta \vdash \{l{:}A\} \leqslant \{l{:}B\}}\ \textsc{DS-fip-rcd} \qquad \frac{\vdash \Delta \qquad \Delta \vdash A, B}{\Delta \vdash \{l{:}A\} \,\&\, \{l{:}B\} \leqslant \{l{:}A \,\&\, B\}}\ \textsc{DS-fip-distRcd} \qquad \frac{\vdash \Delta}{\Delta \vdash \top \leqslant \{l{:}\top\}}\ \textsc{DS-fip-topRcd}$$

$$\frac{\Delta \vdash A_2 \leqslant A_1 \qquad \Delta, \alpha *_a A_2 \vdash B_1 \leqslant B_2}{\Delta \vdash \forall \alpha * A_1.\, B_1 \leqslant \forall \alpha * A_2.\, B_2}\ \textsc{DS-fip-all} \qquad \frac{\vdash \Delta}{\Delta \vdash \top \leqslant \forall \alpha * \top.\, \top}\ \textsc{DS-fip-topAll}$$

$$\frac{\vdash \Delta \qquad \Delta \vdash A \qquad \Delta, \alpha *_a A \vdash B_1, B_2}{\Delta \vdash (\forall \alpha * A.\, B_1) \,\&\, (\forall \alpha * A.\, B_2) \leqslant \forall \alpha * A.\, (B_1 \,\&\, B_2)}\ \textsc{DS-fip-distAll} \qquad \frac{\alpha *_a A \in \Delta \qquad \Delta \vdash A \leqslant \bot}{\Delta \vdash \top \leqslant \alpha}\ \textsc{DS-fip-topVar}$$

Figure 7.3: Declarative subtyping rules.

Disjointness The notion of disjointness (Definition 2.2), defined via subtyping, is used in the original F_i^+, as well as calculi with disjoint intersection types [OSA16]. We proved that our algorithmic definition of disjointness (written as $\Delta \vdash A * B$, in Section 7.3.2) is sound to a specification in terms of top-like types.

Lemma 7.2 (Disjointness soundness)**.** If $\Delta \vdash A * B$ then for all type C that $\Delta \vdash A \leqslant C$ and $\Delta \vdash B \leqslant C$ we have $\Delta \vdash \rceil C \lceil$.

Informally, two disjoint types do not have common supertypes, except for top-like types. This definition is motivated by the desire to prevent ambiguous upcasts on merges. That is, we wish to avoid casts that can extract *different* values of the same type from a merge. Thus in F_i^+ and other calculi with disjoint intersection types, we only allow merges of expressions whose only common supertypes are types that are (equivalent to) the top type. For instance, consider the merge $(1 \,,,\, \mathsf{true}) \,,,\, (2 \,,,\, \text{'c'})$. The first component of the merge $(1 \,,,\, \mathsf{true})$ has type $\mathsf{Int} \,\&\, \mathsf{Bool}$, while the second component $(2 \,,,\, \text{'c'})$ has type $\mathsf{Int} \,\&\, \mathsf{Char}$. This merge is problematic because Int is a supertype of the type of the merge $(\mathsf{Int} \,\&\, \mathsf{Bool}) \,\&\, (\mathsf{Int} \,\&\, \mathsf{Char})$, allowing us to extract two different integers by casting the two terms to Int. Fortunately, our disjointness restriction

Typing modes	$\Leftrightarrow ::= \Leftarrow \mid \Rightarrow$
Pre-values	$u ::= i \mid \top \mid e{:}A \mid u_1 \mathbin{,,} u_2$

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

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

$$\textsc{Typ-fip-abs}\ \frac{\Delta \vdash B_1 \leqslant A \qquad \Delta; \Gamma, x{:}A \vdash e \Leftarrow B_2}{\Delta; \Gamma \vdash \lambda x{:}A.\, e \Leftarrow B_1 \to B_2} \qquad \textsc{Typ-fip-tabs}\ \frac{\Delta \vdash \Gamma \qquad \Delta, \alpha *_a A; \Gamma \vdash e \Leftarrow B}{\Delta; \Gamma \vdash \Lambda\alpha.\, e \Leftarrow \forall \alpha * A.\, B}$$

$$\textsc{Typ-fip-rcd}\ \frac{\Delta; \Gamma \vdash e \Leftarrow A}{\Delta; \Gamma \vdash \{l = e\} \Leftarrow \{l{:}A\}} \qquad \textsc{Typ-fip-app}\ \frac{\Delta; \Gamma \vdash e_1 \Rightarrow A \qquad A \rhd B \to C \qquad \Delta; \Gamma \vdash e_2 \Leftarrow B}{\Delta; \Gamma \vdash e_1\, e_2 \Rightarrow C}$$

$$\textsc{Typ-fip-tapp}\ \frac{\Delta; \Gamma \vdash e \Rightarrow B \qquad B \rhd \forall \alpha * C_1.\, C_2 \qquad \Delta \vdash A * C_1}{\Delta; \Gamma \vdash e\, A \Rightarrow C_2[\alpha \mapsto A]} \qquad \textsc{Typ-fip-proj}\ \frac{\Delta; \Gamma \vdash e \Rightarrow A \qquad A \rhd \{l{:}C\}}{\Delta; \Gamma \vdash e.l \Rightarrow C}$$

$$\textsc{Typ-fip-merge}\ \frac{\Delta \vdash A * B \qquad \Delta; \Gamma \vdash e_1 \Rightarrow A \qquad \Delta; \Gamma \vdash e_2 \Rightarrow B}{\Delta; \Gamma \vdash e_1 \mathbin{,,} e_2 \Rightarrow A \,\&\, B}$$

$$\textsc{Typ-fip-mergev}\ \frac{\Delta \vdash \Gamma \qquad u_1 \approx u_2 \qquad \vdash \Delta \qquad \cdot;\cdot \vdash u_1 \Rightarrow A \qquad \cdot;\cdot \vdash u_2 \Rightarrow B}{\Delta; \Gamma \vdash u_1 \mathbin{,,} u_2 \Rightarrow A \,\&\, B} \qquad \textsc{Typ-fip-inter}\ \frac{\Delta; \Gamma \vdash e \Leftarrow A \qquad \Delta; \Gamma \vdash e \Leftarrow B}{\Delta; \Gamma \vdash e \Leftarrow A \,\&\, B}$$

$$\textsc{Typ-fip-fix}\ \frac{\Delta; \Gamma, x{:}A \vdash e \Leftarrow A}{\Delta; \Gamma \vdash \mathbf{fix}\, x{:}A.\, e \Rightarrow A} \qquad \textsc{Typ-fip-anno}\ \frac{\Delta; \Gamma \vdash e \Leftarrow A}{\Delta; \Gamma \vdash (e{:}A) \Rightarrow A} \qquad \textsc{Typ-fip-sub}\ \frac{\Delta; \Gamma \vdash e \Rightarrow A \qquad \Delta \vdash A \leqslant B}{\Delta; \Gamma \vdash e \Leftarrow B}$$

Figure 7.4: Bidirectional typing rules for F^+_i.

rejects such merges since the supertype Int is not top-like.

7.2.3 Bidirectional Typing

The type system of F^+_i, like the type systems we have seen in the previous two chapters, is bidirectional, with a subsumption rule triggered by type annotations.

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

$$\frac{A_1 \rhd B_1 \to C_1 \qquad A_2 \rhd B_2 \to C_2}{A_1 \,\&\, A_2 \rhd B_1 \,\&\, B_2 \to C_1 \,\&\, C_2}\ \textsc{AD-fip-andArrow} \qquad \frac{A_1 \rhd \{l{:}B_1\} \qquad A_2 \rhd \{l{:}B_2\}}{A_1 \,\&\, A_2 \rhd \{l{:}B_1 \,\&\, B_2\}}\ \textsc{AD-fip-andRcd}$$

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

Figure 7.5: Applicative distribution rules in F^+_i.

Typing As presented in Figure 7.4, there are two modes of typing: synthesis ($\Rightarrow$) and checking ($\Leftarrow$). We use $\Leftrightarrow$ as a metavariable for typing modes. $\Delta; \Gamma \vdash e \Leftrightarrow A$ indicates that under type context Δ and term context Γ, the expression e has type A in mode $\Leftrightarrow$. A bidirectional type system directly provides a type-checking algorithm. Δ, Γ, e are all inputs in both modes. Type synthesis generates a *unique* type as the output (also called the inferred type), while type checking takes a type as an input and examines the term.

Lemma 7.3 (Uniqueness of type synthesis)**.** If $\Delta; \Gamma \vdash e \Rightarrow A_1$ and $\Delta; \Gamma \vdash e \Rightarrow A_2$ then $A_1 = A_2$.

Checking abstractions, type abstractions, and records To check a function $\lambda x : A.\, e$ against $B_1 \to B_2$ by rule Typ-fip-abs, we track the type of the term variable as *the precise parameter type* A, and check if e can be checked against B_2. B_1 must be a subtype of A to guarantee the safety of the function application. The type-checking of type abstractions $\Lambda\alpha.\, e$ works by tracking the disjointness relation of the type variable with the context and checking e against the quantified type B. Typing of records works similarly. Additionally, there is a rule Typ-fip-inter, which checks an expression against an intersection type by separately checking the expression against the composing two types. With this design, we allow $\lambda x{:}\mathsf{Int}.\, x \mathbin{,,} \mathsf{true}$ to be checked against $(\mathsf{Int} \to \mathsf{Int}) \,\&\, (\mathsf{Int} \to \mathsf{Bool})$.

Application, record projection, and conversion of applicable types Like the λ^+_i calculus in Chapter 6, we allows a term of an intersection type to directly apply, as long as the intersection type can be converted into an applicable form. For example, $(\mathsf{Int} \to \mathsf{Int}) \,\&\, (\mathsf{Int} \to \mathsf{Bool})$ is converted into $(\mathsf{Int} \,\&\, \mathsf{Int}) \to (\mathsf{Int} \,\&\, \mathsf{Bool})$, which is a supertype of the former. Note that in the original F^+_i, this requires annotations since the expression being applied in an application must have an inferred arrow type. When inferring the type of the application $e_1\ e_2$, rule Typ-fip-app first *converts* the inferred type of e_1 into an arrow form $B \to C$ and then checks the argument e_2 against B. If the check succeeds, the whole expression has inferred type C.

In F_i^+, we have three applicable forms: *arrow types*, *record types*, *universal types*. Like rule Typ-fip-app, the typing of type application and record projection also allows the applied term to have an intersection type, and relies on *applicative distribution* to convert the type.

Applicative distribution $A \rhd B$ (defined in Figure 7.5) takes type A and generates a supertype B that has an applicable form. The first three rules bring all parts of the input intersection type together. For example, assuming that we apply several merged functions whose types are $A_1 \to B_1$, $A_2 \to B_2$, ..., $A_n \to B_n$, the combined function type is $(A_1 \,\&\, ... \,\&\, A_n) \to (B_1 \,\&\, ... \,\&\, B_n)$. It is equivalent to the input type only when A_1, A_2, ..., and A_n are all equivalent. Essentially, applicative distribution ($A \rhd B$) is a subset of subtyping ($A \leqslant B$). The supertype is selected to ensure that when a merge is applied to an argument, every component in the merge is satisfied. Unlike the design we showed in λ_i^+, even for record projection, incompatible conjuncts are not allowed in the applied type A. Although each one of the three first rules overlaps with the reflexivity rule, for any given type, at most one result has an applicable form.

Since merges are treated as a whole applicable term, programmers can extend functions via a *compositional* approach without modifying the original implementation. It also enables the modular extension of type abstractions and especially records, which play a core role in the trait encoding used in Compositional Programming.

Davies and Pfenning also employ a similar design in their bidirectional type system for refinement intersections [DP00]. Their type conversion procedure respects subtyping as well. Instead of combining function types, it makes use of $A \,\&\, B \leqslant A$ and $A \,\&\, B \leqslant B$ to enumerate components in intersections and uncover arrows.

Applicative distribution always determines:

Lemma 7.4 (Determinism of applicative distribution)**.** For any type A,

- if $A \rhd B_1 \to B_2$ and $A \rhd C_1 \to C_2$, then $B_1 = C_1$ and $B_2 = C_2$;
- if $A \rhd \forall \alpha * B_1.\, B_2$ and $A \rhd \forall \alpha * C_1.\, C_2$, then $B_1 = C_1$ and $B_2 = C_2$;
- if $A \rhd \{l{:}B\}$ and $A \rhd \{l{:}C\}$, then $B = C$.

Typing merges with disjointness and consistency Well-typed merges always have inferred types. There are two type synthesis rules for merges, both combining the inferred types of the two parts into an intersection. Typ-fip-merge requires the two subterms to have *disjoint* inferred types, like $1 \,,,\, \mathsf{true}$. Typ-fip-mergev relaxes the disjointness constraint to *consistency* checking (written as $u_1 \approx u_2$) to accept overlapping terms like $1 \,,,\, 1$. We will state the formal specification of consistency in Section 7.4.1 and show how it is involved in the proofs of determinism and type soundness. Informally, consistent merges cause no ambiguity in the runtime. For practical reasons, we only consider *pre-values* (defined at the top of Figure 7.4) in consistency checking, for which the inferred type can be told directly. The algorithms

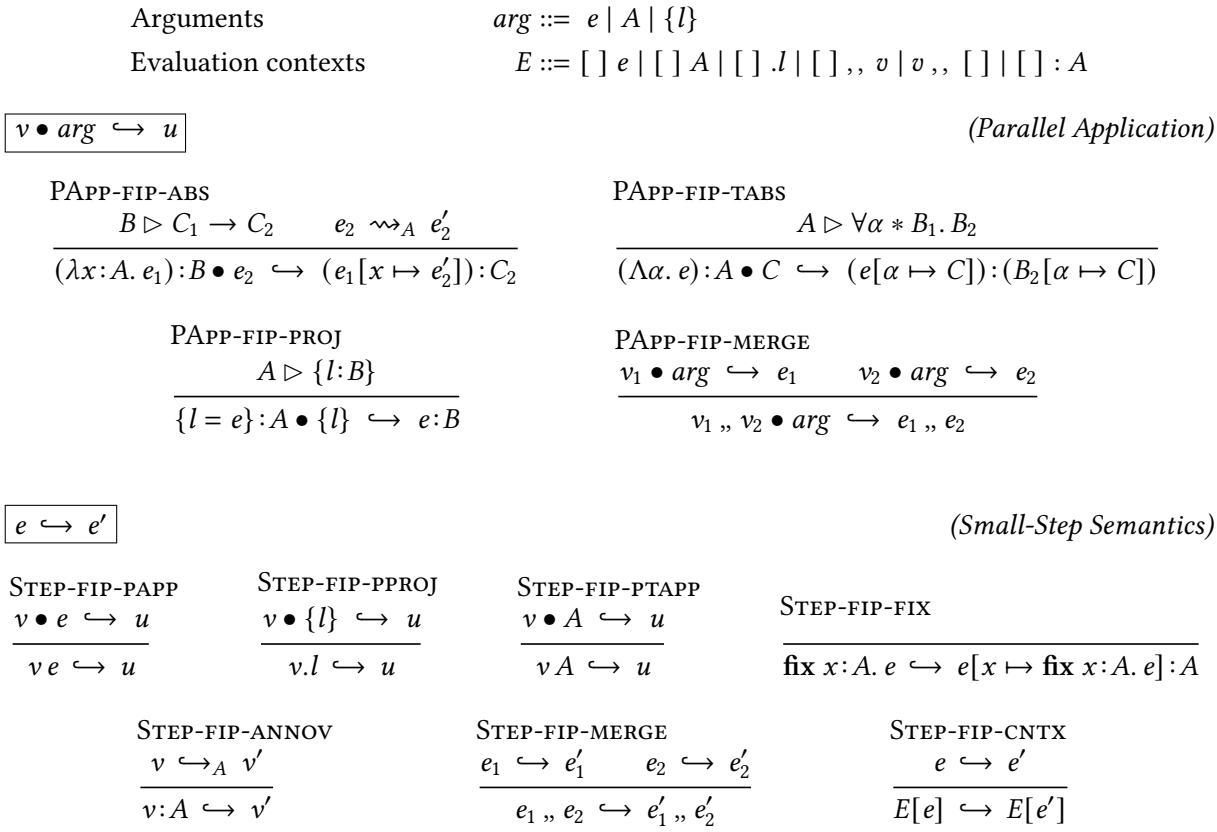

Figure 7.6: Small-step semantics rules.

for disjointness and consistency are presented in Section 7.3. In general, disjointness and consistency avoid introducing ambiguity of merges, and enable a deterministic semantics for F_i^+.

7.2.4 Small-Step Operational Semantics

We specify the *call-by-name* reduction of F_i^+ using a small-step operational semantics in Figure 7.6. Step-fip-papp, Step-fip-pproj, and Step-fip-ptapp are reduction rules for application and record projection. They trigger *parallel application* (defined in the middle of Figure 7.6) of merged values to the argument. Rule Step-fip-fix substitutes the fixpoint term variable with the fixpoint expression itself. Note that the result is annotated with A. With the explicit type annotation, the result of reduction preserves the type of the original fixpoint expression. Through rule Step-fip-annov, values are *cast* to their annotated type. Such values must also be pre-values. This is to filter out checkable terms p including bare abstractions or records without annotations, as $p{:}A$ is a form of value itself and thus should not step.

A merge of multiple terms may reduce in parallel, as shown in rule Step-fip-merge. Only when one side cannot step, the other side steps alone, as suggested by the evaluation context

$E\,,,\ v$ and $v\,,,\ E$. Rule Step-fip-cntx is the reduction rule of expressions within an *evaluation context*. Since the rule can be applied repeatedly, we only need evaluation contexts of depth one (shown at the top of Figure 7.6). Our operational semantics substitutes arguments *wrapped* by type annotations into function bodies, while it forbids the reduction of records since records are values.

Parallel application Parallel application is at the heart of what we call *nested composition* in CP. It provides the runtime behavior that is necessary to implement nested composition, and it reflects the subtyping distributivity rules at the term level. A merge of functions is treated as one function. The beta reduction of all functions in a merge happens in *parallel* to keep the consistency of merged terms. For type abstractions or records, things are similar. The parallel application handles these applicable merges uniformly via rule PApp-fip-merge. To align type application with the other two kinds of application, we extend *arguments* to types (at the top of Figure 7.6). In rule PApp-fip-abs, the argument expression is *wrapped* by the function argument type before we substitute it into the function body. Parallel application of type abstractions substitutes the type argument into the body and annotates the body with the substituted disjoint quantified type. Rule PApp-fip-proj projects record fields. Note these three rules have types to annotate the result, since in Typ-fip-abs, Typ-fip-tabs, and Typ-fip-rcd we only type the expression e inside in *checking* mode. With an explicit type annotation, the application preserves types.

Splittable types In Figure 7.7, we extend the type splitting algorithm of λ_i^+ to universal types in correspondence to the distributive subtyping rules (rule DS-fip-distArrow, rule DS-fip-distRcd, and rule DS-fip-distAll). It gives a decision procedure to check whether a type is splittable or ordinary.

Lemma 7.5 (Type splitting loses no information)**.** If $\vdash \Delta$ and $\Delta \vdash A$ and $B \lhd A \rhd C$ then $\Delta \vdash A \equiv B \,\&\, C$.

Expression wrapping Rules for expression wrapping ($e \rightsquigarrow_A u$) are listed in the middle of Figure 7.8. Basically, it splits the type A when possible, annotates a duplication of e by each ordinary part of A, and then composes all of them. The only exception is that it never uses top-like types to annotate terms, to avoid ill-typed results like $\{l = 1\} : \mathsf{Int} \to \top$, but rather generates a normal value whose inferred type is that top-like type, like $(\lambda x{:}\mathsf{Int}.\ \top) : \mathsf{Int} \to \top$, via the *top-like value generating* function $[\![A^\circ]\!]$, defined at the top of Figure 7.8.

Casting Casting (shown in Figure 7.8) is the core of the TDOS, and is triggered by the Step-fip-annov rule. Recalling that only values that are also pre-values will be cast, we can always tell the inferred type of the input value and cast it by any supertype of that inferred

Ordinary types $A^\circ, B^\circ, C^\circ ::= \alpha \mid \mathsf{Int} \mid \top \mid \bot \mid A \to B^\circ \mid \forall \alpha * A.\, B^\circ \mid \{l{:}A^\circ\}$

$\boxed{B \lhd A \rhd C}$ *(Splittable Types)*

$$\textsc{Sp-fip-arrow}\ \frac{C_1 \lhd B \rhd C_2}{A \to C_1 \lhd A \to B \rhd A \to C_2} \qquad \textsc{Sp-fip-rcd}\ \frac{C_1 \lhd B \rhd C_2}{\{l{:}C_1\} \lhd \{l{:}B\} \rhd \{l{:}C_2\}}$$

$$\textsc{Sp-fip-all}\ \frac{C_1 \lhd B \rhd C_2}{\forall \alpha * A.\, C_1 \lhd \forall \alpha * A.\, B \rhd \forall \alpha * A.\, C_2} \qquad \textsc{Sp-fip-and}\ \frac{}{A \lhd A \,\&\, B \rhd B}$$

Figure 7.7: Ordinary and splittable types in F^+_i.

type. The definition of casting uses the notion of splittable types. In rule Cast-fip-and, the value is cast under two parts of a splittable type separately, and the results are put together by the merge operator. The following example shows that a merge retains its form when cast under equivalent types.

$$\begin{aligned} & ((\lambda x{:}\mathsf{Int}.\, x){:}\mathsf{Int} \to \mathsf{Int}) \,,, ((\lambda x{:}\mathsf{Int}.\, \mathsf{true}){:}\mathsf{Int} \to \mathsf{Bool}) \\ \hookrightarrow_{(\mathsf{Int}\to\mathsf{Int})\,\&\,(\mathsf{Int}\to\mathsf{Bool})} \quad & ((\lambda x{:}\mathsf{Int}.\, x){:}\mathsf{Int} \to \mathsf{Int}) \,,, ((\lambda x{:}\mathsf{Int}.\, \mathsf{true}){:}\mathsf{Int} \to \mathsf{Bool}) \\ \hookrightarrow_{\mathsf{Int}\to\mathsf{Int}\,\&\,\mathsf{Bool}} \quad & ((\lambda x{:}\mathsf{Int}.\, x){:}\mathsf{Int} \to \mathsf{Int}) \,,, ((\lambda x{:}\mathsf{Int}.\, \mathsf{true}){:}\mathsf{Int} \to \mathsf{Bool}) \end{aligned}$$

In the latter case, the requested type is a *function* type, but the result has an intersection type. This change of type causes a major challenge for type preservation.

For ordinary types, rule Cast-fip-int casts an integer to itself under type Int. Under any ordinary top-like type, the cast result is the output of the *top-like value generator*. The casting of values with annotations works by changing the type annotation to the casting (not top-like) supertype. Rule Cast-fip-mergel and rule Cast-fip-merger make a selection between two merged values. The two rules overlap, but for a well-typed value, the casting result is unique.

Example We show an example to illustrate the behavior of our semantics:

$\boxed{[[A]]}$ *(Value Generator)*

$$[[\top]] = \top \qquad [[A_1 \to A_2^\circ]] = (\lambda x{:}\top.\,\top){:}A_1 \to A_2^\circ$$
$$[[\{l{:}A^\circ\}]] = \{l = \top\}{:}\{l{:}A^\circ\} \qquad [[\forall\alpha * A_1.\,A_2^\circ]] = (\Lambda\alpha.\,\top){:}\forall\alpha * A_1.\,A_2^\circ$$

$\boxed{e \rightsquigarrow_A u}$ *(Expression Wrapping)*

$$\textsc{EW-fip-top}\ \frac{\cdot \vdash \rceil A^\circ \lceil}{e \rightsquigarrow_{A^\circ} [[A^\circ]]} \qquad \textsc{EW-fip-anno}\ \frac{\cdot \vdash \neg\rceil B^\circ \lceil}{e \rightsquigarrow_{B^\circ} e{:}B^\circ} \qquad \textsc{EW-fip-and}\ \frac{B_1 \lhd A \rhd B_2 \qquad e \rightsquigarrow_{B_1} u_1 \qquad e \rightsquigarrow_{B_2} u_2}{e \rightsquigarrow_A u_1 \,,,\, u_2}$$

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

$$\textsc{Cast-fip-int}\ \frac{}{i \hookrightarrow_{\mathsf{Int}} i} \qquad \textsc{Cast-fip-top}\ \frac{\cdot \vdash \rceil A^\circ \lceil}{v \hookrightarrow_{A^\circ} [[A^\circ]]} \qquad \textsc{Cast-fip-mergel}\ \frac{v_1 \hookrightarrow_{A^\circ} v_1'}{v_1 \,,,\, v_2 \hookrightarrow_{A^\circ} v_1'} \qquad \textsc{Cast-fip-merger}\ \frac{v_2 \hookrightarrow_{A^\circ} v_2'}{v_1 \,,,\, v_2 \hookrightarrow_{A^\circ} v_2'}$$

$$\textsc{Cast-fip-anno}\ \frac{\cdot \vdash \neg\rceil B^\circ \lceil \qquad \cdot \vdash A \leqslant B^\circ}{e{:}A \hookrightarrow_{B^\circ} e{:}B^\circ} \qquad \textsc{Cast-fip-and}\ \frac{B_1 \lhd A \rhd B_2 \qquad v \hookrightarrow_{B_1} v_1 \qquad v \hookrightarrow_{B_2} v_2}{v \hookrightarrow_A v_1 \,,,\, v_2}$$

Figure 7.8: Type splitting, expression wrapping and value casting rules in F^+_i.

Let $f := \lambda x{:}\mathsf{Int} \,\&\, \top.\, x \,,,\, \mathsf{false}$ in
$((f{:}(\mathsf{Int} \,\&\, \top \to \mathsf{Int}) \,\&\, (\mathsf{Int} \,\&\, \top \to \mathsf{Bool})){:}\mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{Int} \,\&\, \mathsf{Bool})\ (1 \,,,\, \mathsf{true})$

$\hookrightarrow^*$ {by rules Step-fip-annov, Cast-fip-and, and Cast-fip-anno}
$(f{:}\mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{Int}) \,,,\, (f{:}\mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{Bool})\ (1 \,,,\, \mathsf{true})$

$\hookrightarrow^*$ {by rules Step-fip-papp, EW-fip-and, EW-fip-anno, and EW-fip-top}
$(((1 \,,,\, \mathsf{true}){:}\mathsf{Int} \,,,\, \top) \,,,\, \mathsf{false}){:}\mathsf{Int} \,,,\, (((1 \,,,\, \mathsf{true}){:}\mathsf{Int} \,,,\, \top) \,,,\, \mathsf{false}){:}\mathsf{Bool}$

$\hookrightarrow^*$ {by rules Step-fip-merge, Step-fip-annov, Cast-fip-int, Cast-fip-mergel, and Cast-fip-merger}
$1 \,,,\, \mathsf{false}$

This example shows that a function with a splittable type will be cast to a merge of two copies of itself with different type annotations, i.e., two split results. The application of a merge of functions works by distributing the argument to both functions. Finally, casting selects one side of the merge under the annotated type. From this example, we can see that without the precise parameter annotation of a lambda function (here $\mathsf{Int} \,\&\, \top$), there is no way to filter the argument $1 \,,,\, \mathsf{true}$, causing a conflict.

$\boxed{\lfloor A \rfloor}$ *(Bottom-like Types)*

$$\frac{}{\lfloor \bot \rfloor}\ \textsc{BL-bot} \qquad \frac{\lfloor A \rfloor}{\lfloor A \,\&\, B \rfloor}\ \textsc{BL-andl} \qquad \frac{\lfloor B \rfloor}{\lfloor A \,\&\, B \rfloor}\ \textsc{BL-andr}$$

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

$$\frac{\vdash \Delta}{\Delta \vdash \rceil \top \lceil}\ \textsc{TL-fip-top} \qquad \frac{\Delta \vdash \rceil A \lceil \qquad \Delta \vdash \rceil B \lceil}{\Delta \vdash \rceil A \,\&\, B \lceil}\ \textsc{TL-fip-and}$$

$$\frac{\Delta \vdash A \qquad \Delta \vdash \rceil B \lceil}{\Delta \vdash \rceil A \to B \lceil}\ \textsc{TL-fip-arrow} \qquad \frac{\Delta \vdash \rceil B \lceil}{\Delta \vdash \rceil \{l : B\} \lceil}\ \textsc{TL-fip-rcd} \qquad \frac{\Delta, \alpha *_a A \vdash \rceil B \lceil}{\Delta \vdash \rceil \forall \alpha * A.\, B \lceil}\ \textsc{TL-fip-all}$$

$$\frac{\vdash \Delta \qquad \alpha *_a A \in \Delta \qquad \lfloor A \rfloor}{\Delta \vdash \rceil \alpha \lceil}\ \textsc{TL-fip-var}$$

Figure 7.9: Bottom-like type rules and top-like type rules.

7.3 Algorithmics

In Section 7.2 we have presented several relations in a declarative form, including subtyping, disjointness, and consistency. For the purposes of implementation, it is important to formulate the corresponding algorithmic versions.

7.3.1 Algorithmic Subtyping

To obtain an equivalent algorithmic formulation of subtyping, we firstly define algorithms for *top-like types* and *bottom-like types* inductively in Figure 7.9, then we introduce our algorithmic subtyping and argue that it is equivalent to the declarative subtyping.

Top-like and bottom-like types Every top-like type is a supertype of all types (see Definition 7.2), which is equivalent to $\top$. Compared to the definition of top-like types in the original F_i^+ [Bi+19], we add a type context in the subtyping judgment to keep track of type variables that are disjoint to the bottom type. With type contexts, we can derive $\cdot \vdash \rceil \forall \alpha * \bot.\, \alpha \lceil$ by TL-FIP-VAR since only top-like types are disjoint to $\bot$. The corresponding declarative subtyping rule is the novel DS-FIP-TOPVAR. To eliminate the dependence of our top-like type algorithm on subtyping, we define *bottom-like types* as a separate relation and use $\lfloor A \rfloor$ when $\Delta \vdash A \leqslant \bot$ is needed.

Lemma 7.6 (Equivalence of bottom-like types)**.** If $\vdash \Delta$ and $\Delta \vdash A$ and $\lfloor A \rfloor$ if and only if $\Delta \vdash A \leqslant \bot$.

Then we obtain an algorithmic definition of top-like types (Figure 7.9) that is equivalent to Definition 7.2.

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

$$\textsc{S-fip-var}\ \frac{\vdash \Delta \qquad \Delta \vdash \alpha}{\Delta \vdash \alpha <: \alpha} \qquad \textsc{S-fip-int}\ \frac{\vdash \Delta}{\Delta \vdash \mathsf{Int} <: \mathsf{Int}} \qquad \textsc{S-fip-top}\ \frac{\Delta \vdash A \qquad \Delta \vdash \rceil B^{\circ} \lceil}{\Delta \vdash A <: B^{\circ}} \qquad \textsc{S-fip-bot}\ \frac{\vdash \Delta \qquad \Delta \vdash A^{\circ}}{\Delta \vdash \bot <: A^{\circ}}$$

$$\textsc{S-fip-andl}\ \frac{\Delta \vdash B \qquad \Delta \vdash A <: C^{\circ}}{\Delta \vdash A \,\&\, B <: C^{\circ}} \qquad \textsc{S-fip-andr}\ \frac{\Delta \vdash A \qquad \Delta \vdash B <: C^{\circ}}{\Delta \vdash A \,\&\, B <: C^{\circ}} \qquad \textsc{S-fip-arrow}\ \frac{\Delta \vdash A_2 <: A_1 \qquad \Delta \vdash B_1 <: B_2^{\circ}}{\Delta \vdash A_1 \to B_1 <: A_2 \to B_2^{\circ}}$$

$$\textsc{S-fip-all}\ \frac{\Delta \vdash B_1 <: A_1 \qquad \Delta, \alpha *_a B_1 \vdash A_2 <: B_2^{\circ}}{\Delta \vdash \forall \alpha * A_1.\, A_2 <: \forall \alpha * B_1.\, B_2^{\circ}} \qquad \textsc{S-fip-rcd}\ \frac{\Delta \vdash A <: B^{\circ}}{\Delta \vdash \{l{:}A\} <: \{l{:}B^{\circ}\}}$$

$$\textsc{S-fip-and}\ \frac{B_1 \lhd B \rhd B_2 \qquad \Delta \vdash A <: B_1 \qquad \Delta \vdash A <: B_2}{\Delta \vdash A <: B}$$

Figure 7.10: Algorithmic subtyping rules.

Lemma 7.7 (Top-like equivalence)**.** $\Delta \vdash \rceil A \lceil$ if and only if $\Delta \vdash \top \leqslant A$.

Algorithmic subtyping Our subtyping algorithm is shown in Figure 7.10. This algorithm is an extension of the algorithm used in λ_i^+ with splittable types (Chapter 3). The novel additions are the rules involving disjoint polymorphism, which λ_i^+ does not have. While rule S-FIP-AND requires the supertype B to be splittable, the remaining rules only apply to ordinary B. The basic idea is to split the intersection-like right-hand-side type by rule S-FIP-AND until types are in more atomic forms, i.e., ordinary, and then apply the remaining rules to decide whether the left-hand-side type is a subtype of each ordinary part. For a subtyping $A <: B$ checking to succeed where B is splittable, we need every sub-checking of split types to succeed, as described by the inversion lemma:

Lemma 7.8 (Inversion of the supertype in algorithmic subtyping)**.** If $B_1 \lhd B \rhd B_2$ then $\Delta \vdash A <: B$ if and only if $\Delta \vdash A <: B_1$ and $\Delta \vdash A <: B_2$.

The key for the distributive subtyping rules to work is encoded in how we split types. For example, only after we split $\mathsf{Int} \to \mathsf{Int} \,\&\, \mathsf{Bool}$ into $(\mathsf{Int} \to \mathsf{Int})$ and $(\mathsf{Int} \to \mathsf{Bool})$, it becomes straightforward to tell it is a supertype of $(\mathsf{Int} \to \mathsf{Int}) \,\&\, (\mathsf{Int} \to \mathsf{Bool})$. Assuming that we only have rule SP-FIP-AND (see Figure 7.8) that splits intersection types, the system degenerates to the conventional intersection subtyping rules with disjoint polymorphism (like F_i). In other words, the ordinary-type rules (all rules except for rule S-FIP-AND) are standard and the algorithm design is modular. They are mostly the same as the declarative formulation with the additional ordinary-type condition, except that rules S-FIP-ANDL and S-FIP-ANDR are embedded with

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

$$\textsc{Dax-symm}\ \frac{B *_{ax} A}{A *_{ax} B} \qquad \textsc{Dax-intArrow}\ \frac{}{\mathsf{Int} *_{ax} A_1 \to A_2} \qquad \textsc{Dax-intRcd}\ \frac{}{\mathsf{Int} *_{ax} \{l{:}A\}} \qquad \textsc{Dax-intAll}\ \frac{}{\mathsf{Int} *_{ax} \forall\alpha * A.\, B} \qquad \textsc{Dax-arrowRcd}\ \frac{}{A_1 \to A_2 *_{ax} \{l{:}A\}}$$

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

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

$$\textsc{D-fip-ax}\ \frac{\vdash \Delta \qquad \Delta \vdash A, B \qquad A *_{ax} B}{\Delta \vdash A * B} \qquad \textsc{D-fip-topl}\ \frac{\Delta \vdash B \qquad \Delta \vdash \rceil A \lceil}{\Delta \vdash A * B} \qquad \textsc{D-fip-topr}\ \frac{\Delta \vdash A \qquad \Delta \vdash \rceil B \lceil}{\Delta \vdash A * B}$$

$$\textsc{D-fip-arrow}\ \frac{\Delta \vdash A_1, A_2 \quad \Delta \vdash B_1 * B_2}{\Delta \vdash A_1 \to B_1 * A_2 \to B_2} \qquad \textsc{D-fip-rcdEq}\ \frac{\Delta \vdash A * B}{\Delta \vdash \{l{:}A\} * \{l{:}B\}} \qquad \textsc{D-fip-all}\ \frac{\Delta \vdash A_1, A_2 \qquad \Delta, \alpha *_a A_1 \,\&\, A_2 \vdash B_1 * B_2}{\Delta \vdash \forall\alpha * A_1.\, B_1 * \forall\alpha * A_2.\, B_2}$$

$$\textsc{D-fip-varl}\ \frac{\alpha *_a A \in \Delta \quad \Delta \vdash A <: B}{\Delta \vdash \alpha * B} \qquad \textsc{D-fip-varr}\ \frac{\alpha *_a A \in \Delta \quad \Delta \vdash A <: B}{\Delta \vdash B * \alpha}$$

$$\textsc{D-fip-andl}\ \frac{A_1 \lhd A \rhd A_2 \qquad \Delta \vdash A_1 * B \qquad \Delta \vdash A_2 * B}{\Delta \vdash A * B}$$

$$\textsc{D-fip-andr}\ \frac{B_1 \lhd B \rhd B_2 \qquad \Delta \vdash A * B_1 \qquad \Delta \vdash A * B_2}{\Delta \vdash A * B}$$

Figure 7.11: Algorithmic definition of disjointness in F^{+}_{i}.

transitivity. For two universal types $\forall\alpha * A_1.\, A_2$ and $\forall\alpha * B_1.\, B_2$, the subtyping of A_1 and B_1 is contravariant, and the subtyping of A_2 and B_2 is covariant. When deciding the subtyping of A_2 and B_2, we add $\alpha *_a B_1$ into the context to track the disjointness of the type variable. For the special case of α disjoint to bottom-like types as rule DS-FIP-TOPVAR, we have the context to decide α to be top-like by our top-like type algorithm in rule S-FIP-TOP. Our algorithmic subtyping is equivalent to the declarative subtyping and decidable:

Lemma 7.9 (Equivalence of subtyping)**.** $\Delta \vdash A <: B$ if and only if $\Delta \vdash A \leqslant B$.

Lemma 7.10 (Decidability of algorithmic subtyping)**.** $\Delta \vdash A <: B$ is decidable.

$\boxed{u : A}$ *(Principal Type of Pre-Values)*

$$\frac{}{\top : \top}\ \text{PT-FIP-TOP} \qquad \frac{}{i : \mathsf{Int}}\ \text{PT-FIP-INT} \qquad \frac{}{(e{:}A) : A}\ \text{PT-FIP-ANNO} \qquad \frac{u_1 : A \qquad u_2 : B}{(u_1 \,,, u_2) : (A \,\&\, B)}\ \text{PT-FIP-MERGE}$$

$\boxed{u_1 \approx u_2}$ *(Consistency)*

$$\frac{}{i \approx i}\ \text{C-FIP-LIT} \qquad \frac{}{e{:}A \approx e{:}B}\ \text{C-FIP-ANNO} \qquad \frac{\cdot \vdash A * B \qquad u_1 : A \qquad u_2 : B}{u_1 \approx u_2}\ \text{C-FIP-DISJOINT}$$

$$\frac{u_1 \approx u \qquad u_2 \approx u}{u_1 \,,, u_2 \approx u}\ \text{C-FIP-MERGEL} \qquad \frac{u \approx u_1 \qquad u \approx u_2}{u \approx u_1 \,,, u_2}\ \text{C-FIP-MERGER}$$

Figure 7.12: Pre-value consistency rules in F_i^+.

7.3.2 Disjointness

As shown in Figure 7.11, the disjointness definition is almost the same as the original F_i^+ [Bi+19]. The disjointness judgment $\Delta \vdash A * B$ ensures that under the context Δ, any common supertype of A and B is a top-like type. Disjointness helps ensure the determinism of F_i^+ by forbidding merging terms of types that are not disjoint with each other (rule Typ-fip-merge). Our algorithm of disjointness is sound with respect to our specification (Lemma 7.2).

Compared to the original F_i^+, there are two main novelties. Firstly, now the context may tell us if a type variable is top-like, i.e., is disjoint to bottom-like types, and a top-like type variable should be disjoint to any types. Secondly, in rules D-fip-andl and D-fip-andr, we split types instead of allowing only intersection types for the convenience of the proof. We have an alternative disjointness definition in the original style proved to be equivalent in our Coq formalization. We can prove the following properties:

Theorem 7.1 (Covariance of disjointness)**.** If $\Delta \vdash A * B$ and $\Delta \vdash B <: C$ then $\Delta \vdash A * C$.

Theorem 7.2 (Substitution of disjointness)**.** If $\Delta \,,\, \alpha *_a C \,,\, \Delta' \vdash A * B$ and $\Delta \vdash C' * C$ then $\Delta \,,\, \Delta'[\alpha \mapsto C'] \vdash A[\alpha \mapsto C'] * B[\alpha \mapsto C']$.

Theorem 7.1 is a generalization of rules D-fip-varl and D-fip-varr. In short disjointness is covariant: supertypes of disjoint types are still disjoint. Theorem 7.2 ensures the correct behavior of the type instantiation of the type application. Disjoint types are still disjoint with each other after instantiation.

7.3.3 Consistency

Rule Typ-fip-mergev is aimed to type merges of *pre-values* produced by *type casting* and *expression wrapping*. The intention is not to accept more programs, so this rule is not supposed to be exposed to users. The first question is: why are we considering the consistency of pre-values rather than values or expressions? The answer is that our system allows an expression (or a value) to be duplicated, and merged with type annotations in our dynamic semantics by rule Cast-fip-and and rule EW-fip-and. For example,

$$
\begin{aligned}
& ((\lambda x{:}\mathsf{Int}.\, x \,,,\, \mathsf{false}{:}\mathsf{Int} \to \mathsf{Int} \,\&\, \mathsf{Bool}){:}\mathsf{Int} \to \mathsf{Int} \,\&\, \mathsf{Bool})\ 1 \\
\hookrightarrow\ & (\lambda x{:}\mathsf{Int}.\, x \,,,\, \mathsf{false}{:}\mathsf{Int} \to \mathsf{Int} \,,,\, \lambda x{:}\mathsf{Int}.\, x \,,,\, \mathsf{false}{:}\mathsf{Int} \to \mathsf{Bool})\ 1 \\
\hookrightarrow\ & (1 \,,,\, \mathsf{false}){:}\mathsf{Int} \,,,\, (1 \,,,\, \mathsf{false}){:}\mathsf{Bool} \\
\hookrightarrow\ & 1 \,,,\, \mathsf{false}
\end{aligned}
$$

Moreover, it is also allowed to produce a result of duplicated terms with the same type annotation, like $1{:}\mathsf{Int} \,,,\, 1{:}\mathsf{Int}$. Such values cannot pass disjointness checking but are harmless at runtime. To keep type preservation for the application results like the merge after the first and second steps, consistency judgments on annotated terms are necessary. In consistency checking (Figure 7.12), we take two pre-values and analyze them structurally. A merged pre-value $u_1 \,,,\, u_2$ is consistent to another pre-value u if pre-values composing the merge are consistent with u. Any two basic components from each term must be either disjoint (rule C-fip-disjoint) or only differ in the annotation (except for the argument annotation of lambda abstractions in rule C-fip-anno). For instance, functions with different annotations but the same body are consistent (as the one after the first step in the example above), and $1 \,,,\, \mathsf{true}$ is consistent with $1 \,,,\, \text{`a'}$. Note that we use $u{:}A$ to represent the principal type of u is A, which is a syntactical approach to compute the types from pre-values (shown in Figure 7.12) This syntactic approach is complete with respect to our type system.

Lemma 7.11 (Pre-values have principal types)**.** $\exists\, \mathrm{A},\ u{:}A$.

Lemma 7.12 (Completeness of principal types)**.** If $\Delta; \cdot \vdash u \implies A$ then $u : A$.

Also note that any disjoint pre-values are also consistent. So rule Typ-fip-mergev is a strict relaxation of rule Typ-fip-merge on pre-values.

7.4 Type Soundness and Determinism

In this section, we show that the operational semantics of F_i^+ is type-sound and deterministic. In F_i^+, determinism also plays a key role in the proof of type soundness.

7.4.1 Determinism

A common problem with determinism for calculi with a merge operator is the ambiguity of selection between merged values. In our system, ambiguity is removed by employing disjointness and consistency constraints on merges via typing.

Lemma 7.13 (Consistent values cause no casting ambiguity)**.** If $v_1 \approx v_2$ then for all type A that $v_1 \hookrightarrow_A v_1'$ and $v_2 \hookrightarrow_A v_2'$ then $v_1' = v_2'$.

Two values in a merge have no conflicts as long as casting both values under any type leads to the same result. This specification allows v_1 and v_2 to contain identical expressions (may differ in annotations), and terms with disjoint types as such terms can only be cast under top-like types, and the cast result is only decided by that top-like type.

Lemma 7.14 (Top-like casting is term irrelevant)**.** If $\cdot \vdash \rceil A \lceil$ and $v_1 \hookrightarrow_A v_1'$ and $v_2 \hookrightarrow_A v_2'$ then $v_1' = v_2'$.

This is because casting only happens when the given type is a supertype of the cast value's type, and disjoint types only share top-like types as common supertypes (Lemma 7.2).

Lemma 7.15 (Upcast only)**.** If $\cdot;\cdot \vdash v \Rightarrow B$ and $v \hookrightarrow_A v'$ then $\cdot \vdash B \leqslant A$.

With consistency, casting all well-typed values leads to a unique result. The remaining reduction rules, including expression wrapping and parallel application, are trivially deterministic.

Lemma 7.16 (Determinism of casting)**.** If $\cdot;\cdot \vdash v \Rightarrow B$ and $v \hookrightarrow_A v_1$ and $v \hookrightarrow_A v_2$, then $v_1 = v_2$.

Theorem 7.3 (Determinism of reduction)**.** If $\cdot;\cdot \vdash e \Rightarrow A$ and $e \hookrightarrow e_1$ and $e \hookrightarrow e_2$ then $e_1 = e_2$.

7.4.2 Progress

Annotated values trigger casting, for which the progress lemma can be directly proved, as we know, v must have an inferred type that is a subtype of B.

Lemma 7.17 (Progress of casting)**.** If $\cdot;\cdot \vdash v \Rightarrow A$ and $\cdot;\cdot \vdash v \Leftarrow B$ then there exists a v' such that $v \hookrightarrow_B v'$.

The progress lemma for expression wrapping is more relaxed. It does not enforce that e is checked against the wrapping type A because that is the typical situation where we need to use the relation.

Lemma 7.18 (Progress of expression wrapping)**.** If $\cdot;\cdot \vdash e \Leftarrow A$ and $\cdot \vdash B$ then there exists an expression e' that $e \rightsquigarrow_B e'$.

Parallel applications deal with function application, type application, and record projection. We use the term *general application* of a value v to an argument *arg* next to denote all of these for simplicity.

Lemma 7.19 (Progress of parallel application)**.** If $\cdot;\cdot \vdash v \bullet \text{arg} \Rightarrow A$ then there exists a pre-value u such that $v \bullet arg \hookrightarrow u$.

Finally, the progress property of reduction can be proved.

Theorem 7.4 (Progress of reduction)**.** If $\cdot;\cdot \vdash e \Leftrightarrow A$ then either e is a value or there exists an expression e' such that $e \hookrightarrow e'$.

7.4.3 Preservation

Proving progress is straightforward, but preservation is much more challenging. When typing merges, we need to satisfy the extra side conditions in rules Typ-fip-merge and Typ-fip-mergev: disjointness and consistency. While the former only depends on types, the latter needs special care.

Consistency Like λ_i^+, casting may duplicate terms. For example, $1 \hookrightarrow_{\mathsf{Int}\,\&\,\mathsf{Int}} 1 \,,,\, 1$ by rule Cast-fip-and. We have to ensure any two merged casting results are consistent:

Lemma 7.20 (Value consistency after casting)**.** If $\cdot;\cdot \vdash v \Rightarrow C$ and $v \hookrightarrow_A v_1$ and $v \hookrightarrow_B v_2$ then $v_1 \approx v_2$.

Then we need to make sure that consistency is preserved during reduction. Recall that consistency is defined on pre-values. Consider any three components from consistent merges, since merges are reduced in parallel (rule Step-fip-merge), the sub-expression e in merges like $e:A \,,,\, e:B \,,,\, e:C$ remains the same until it is cast by A, B, and C respectively. Guarded by the determinism theorem of reduction (Theorem 7.3) and the consistency lemma of casting (Lemma 7.20), we prove consistent pre-values are always consistent after possible reduction.

Lemma 7.21 (Reduction keeps consistency)**.** If $\cdot;\cdot \vdash u_1 \Rightarrow A$ and $\cdot;\cdot \vdash u_2 \Rightarrow B$ and $u_1 \approx u_2$ then

- if u_1 is a value and $u_2 \hookrightarrow u_2'$ then $u_1 \approx u_2'$;
- if u_2 is a value and $u_1 \hookrightarrow u_1'$ then $u_1' \approx u_2$;
- if $u_1 \hookrightarrow u_1'$ and $u_2 \hookrightarrow u_2'$ then $u_1' \approx u_2'$.

Besides, when parallel application substitutes arguments into merges of applicable terms or projects the wished field, consistency is preserved as well. This requirement enforces us to define consistency not only on values but also on pre-values since the application transforms a value merge into a pre-value merge.

$\boxed{A \lesssim B}$ *(Runtime Subtyping in F_i^+)*

$$\frac{}{A \lesssim A}\ \text{RS-FIP-REFL} \qquad \frac{B_1 \lhd B \rhd B_2 \qquad A_1 \lesssim B_1 \qquad A_2 \lesssim B_2}{A_1 \,\&\, A_2 \lesssim B}\ \text{RS-FIP-AND}$$

Figure 7.13: Runtime subtyping in F_i^+.

Lemma 7.22 (Parallel application keeps consistency)**.** If $\cdot;\cdot \vdash v_1 \Rightarrow A$ and $\cdot;\cdot \vdash v_2 \Rightarrow B$ and $v_1 \approx v_2$ and $v_1 \bullet \mathit{arg} \hookrightarrow u_1$ and $v_2 \bullet \mathit{arg} \hookrightarrow u_2$ then $u_1 \approx u_2$ when

- *arg* is a well-typed expression;
- or *arg* is a label;
- or *arg* is a type C; we know $A \rhd \forall \alpha * A_1.\, A_2$ and $B \rhd \forall \alpha * B_1.\, B_2$; and $\cdot \vdash C * A_1 \,\&\, B_1$.

Runtime subtyping In F_i^+, types are not always precisely preserved by all reduction steps. Specifically, when we cast a value $v \hookrightarrow_A v'$ (in rule Step-fip-annov) or wrap a term $e \rightsquigarrow_A u$ (in rule PApp-fip-abs), the context expects v' or u to have type A, but this is not always true. In our casting rules shown at the bottom of Figure 7.8, most values will be reduced to results with the exact type that we want, except for rule Cast-fip-and. The inferred type of the result is always an intersection, which may differ from the original splittable type. The *runtime subtyping* relation (Figure 7.13) describes the change of types during reduction accurately, If $A \lesssim B$, we say A is a runtime subtype of B. Compared to λ_i^+, the relation is simplified thanks to the increase of type annotation in values. The following lemma shows that while the two types in a runtime subtyping relation may be syntactically different, they are equivalent under an empty type context (i.e. $\cdot \vdash A \leqslant B$ and $\cdot \vdash B \leqslant A$).

Theorem 7.5 (Runtime subtypes are equivalent)**.** If $A \lesssim B$ then $\cdot \vdash A \equiv B$.

With runtime subtyping, we define the preservation property of casting, expression wrapping, and parallel application as follows.

Lemma 7.23 (Casting preserves typing)**.** If $\cdot;\cdot \vdash v \Rightarrow A$ and $v \hookrightarrow_B v'$ then there exists a type C such that $\cdot;\cdot \vdash v' \Rightarrow C$ and $C \lesssim B$.

Lemma 7.24 (Expression wrapping preserves typing)**.** If $\cdot;\cdot \vdash e \Leftarrow B$ and $\cdot \vdash B \leqslant A$ and $e \rightsquigarrow_A u$ then there exists a type C such that $\cdot;\cdot \vdash u \Rightarrow C$ and $C \lesssim A$.

Lemma 7.25 (Parallel application preserves typing)**.** If $\cdot;\cdot \vdash v \bullet \mathrm{arg} \Rightarrow A$ and $v \bullet \mathit{arg} \hookrightarrow u$ then there exists a type B such that $\cdot;\cdot \vdash u \Rightarrow B$ and $B \lesssim A$.

Of course, we can prove that the result of casting always has a subtype (or an equivalent type) of the requested type instead of a runtime subtype. But it would be insufficient for type preservation of reduction. In summary, if casting or wrapping generates a term of type B when the requested type is A, we need B to satisfy:

- B is a subtype of A because we want a preservation theorem that respects subtyping.
- For any type C, $A *_a C$ implies $B *_a C$. This is for the disjointness and consistency checking in rules Typ-fip-merge and Typ-fip-mergev. Note that $B \leqslant A$ is not enough for this property.
- If A converts into an applicable type C, then B converts into an applicable type too.

Although the type equivalence satisfies the first two conditions, it breaks the last one. For example, $\top$ is equivalent to $\top \to \top$, but one may not convert $\top$ to an applicable type by the applicative distribution. So it is infeasible to replace the runtime subtyping with the type equivalence.

Restricted check subsumption Conventionally, a term of type B can be checked by any type C that is a supertype of B in a bidirectional type system. However, this property does not hold in our system when C is a top-like type. For example, although $\top \to \top$ is a supertype of any types, values like $(\lambda x\!:\!\mathsf{Int}.\, x + 1)\!:\!\top \to \top$ and $(\{l = 1\})\!:\!\top \to \top$ are forbidden. Obviously, when these terms are applied, for example, to $\top$, we cannot perform beta reduction.

$$((\lambda x\!:\!\mathsf{Int}.\, x + 1)\!:\!\top \to \top)\ \top \hookrightarrow ((x + 1)[x \mapsto \top\!:\!\mathsf{Int}])\!:\!\top \qquad \textbf{ill-typed}$$

We want to make sure in a valid application $((\lambda x : A.\, e_1) : B)\ e_2$, the argument e_2 always satisfies the parameter annotation A, so it can be substituted in. What is more, we do not want the reduction result depends on B. Therefore, when two values that only differ in annotation B apply to the same argument, their reduction results will only differ in the outermost annotation too. Then the consistency is kept during parallel application (Lemma 7.22).

Therefore, our check subsumption lemma is in a restricted form.

Lemma 7.26 (Restricted check subsumption)**.** If $\Delta; \Gamma \vdash e \Leftarrow A$ and $\Delta \vdash A <: B$ and B° and $\neg\Delta \vdash \rceil B \lceil$ then $\Delta; \Gamma \vdash e \Leftarrow B$.

Replacing the checked type by its supertype is safe as long as it does not have any top-like part. Nevertheless, a function annotated with an intersection, like $(\lambda x\!:\!\mathsf{Int}.\, x \,,,\, \mathsf{true})\!:\!(\mathsf{Int} \to \mathsf{Int}) \,\&\, (\mathsf{Int} \to \mathsf{Bool})$, is still well-typed.

Motivation of expression wrapping Before we reach the type preservation lemma of expression wrapping (Lemma 7.27), it may not be obvious why we have to design such a new relation to handle the argument before substituting it into a lambda body. We need expression wrapping because other alternatives do not work. First, in CBN evaluation, the argument does not necessarily have to be a value. And we cannot tell the shape of the argument passed in. This is why we do not simply use the casting relation like λ_i^+. Another direct option is to annotate the argument by the parameter type of the function. But it leads to ill-typed results because of the restriction in our check-subsumption lemma.

$$(\lambda x{:}\top.\, e{:}(\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int})\,(\lambda x{:}\mathsf{Int}.\, x) \;\hookrightarrow\; (e[x \mapsto (\lambda x{:}\mathsf{Int}.\, x){:}\top]){:}\mathsf{Int}$$

To overcome the incomplete check subsumption, we could annotate the argument twice, like the following example. However, it causes severe problems in consistency.

$$((\lambda x{:}A.\, e){:}B_1 \to B_2 \,,\, (\lambda x{:}A.\, e){:}C_1 \to C_2)\; e' \;\hookrightarrow\; (e[x \mapsto e'{:}B_1{:}A]){:}B_2 \,,\, (e[x \mapsto e'{:}C_1{:}A]){:}C_2$$

The pair of functions are consistent since they only differ in outermost annotation. But after the application reduces, they turn into two annotated expressions, and the inside do not have to be identical, and therefore may not be consistent.

Because terms like $(\lambda x{:}A.\, e){:}B_1 \to B_2$ is consistent with $(\lambda x{:}A.\, e){:}C_1 \to C_2$, but after application on e', it will turned into $(e[x \mapsto e'{:}B_1{:}A]){:}B_2$ and $(e[x \mapsto e'{:}C_1{:}A]){:}C_2$ which do not have to be not consistent.

Next, we show the preservation of expression wrapping and parallel application.

Lemma 7.27 (Expression wrapping preserves typing)**.** If $\cdot;\cdot \vdash e \Leftarrow B$ and $\cdot \vdash B \leqslant A$ and $e \rightsquigarrow_A u$ then $\exists\, C$ that $\cdot;\cdot \vdash u \Rightarrow C$ and $C \lesssim A$.

Lemma 7.28 (Parallel application preserves typing)**.** If $\cdot;\cdot \vdash v \bullet \mathrm{arg} \Rightarrow A$ and $v \bullet arg \hookrightarrow u$ then $\exists\, B$ that $\cdot;\cdot \vdash u \Rightarrow B$ and $B \lesssim A$.

Narrowing and substitution lemmas Before proving type preservation, we have to prove narrowing and substitution lemmas, including both type and term substitutions. The narrowing lemma for typing is mainly to deal with proof cases regarding universal types. To check an expression against a supertype, we need this property to tighten the disjointness constraint of the type variable in the context of a typing judgment, since a supertype of a universal type has a more tight disjointness constraint.

Lemma 7.29 (Typing narrowing)**.** If $\Delta\,,\, \alpha *_a A\,,\, \Delta';\Gamma \vdash e \Leftrightarrow C$ and $\Delta \vdash B \leqslant A$ then $\Delta\,,\, \alpha *_a B\,,\, \Delta';\Gamma \vdash e \Leftrightarrow C$.

The type substitution lemma for typing is necessary for the instantiation of type variables.

We always use types satisfying the disjointness constraint of a universal type to substitute. We need to make sure that the typing judgment still holds after type substitution:

Lemma 7.30 (Type substitution preserves typing)**.** If $\Delta\,, \alpha *_a A\,, \Delta'; \Gamma \vdash e \Leftrightarrow C$ and $\Delta \vdash A * B$ then $\Delta\,, \Delta'[\alpha \mapsto B]; \Gamma[\alpha \mapsto B] \vdash e[\alpha \mapsto B] \Leftrightarrow C[\alpha \mapsto B]$.

In beta reduction or the reduction of fixpoints, we want the term substitution of the parameter inside the function body to always preserve types when the argument has a *runtime subtype* of the function input type. This is different from traditional term substitution lemmas. Our expression wrapping does not necessarily produce a wrapped result with the identical type to the wrapping type (Lemma 7.27). In addition, if we substitute a term with a runtime subtype into an expression, the substituted result does not *infer* the original type: it should infer an runtime subtype, though it can still be *checked* by the original checking type.

Lemma 7.31 (Term substitution preserves type synthesis)**.** If $\Delta; \Gamma\,, x : A\,, \Gamma' \vdash e \Rightarrow C$ and $\Delta; \Gamma \vdash e' \Rightarrow B$ and $B \lesssim A$ then there exists a type C' that $\Delta; \Gamma\,, \Gamma' \vdash e[x \mapsto e'] \Rightarrow C'$ and $C' \lesssim C$.

Lemma 7.32 (Term substitution preserves type checking)**.** If $\Delta; \Gamma\,, x : A\,, \Gamma' \vdash e \Leftarrow C$ and $\Delta; \Gamma \vdash e' \Rightarrow B$ and $B \lesssim A$, then $\Delta; \Gamma\,, \Gamma' \vdash e[x \mapsto e'] \Leftarrow C$.

Finally, with the lemmas above and runtime subtyping, we have the type preservation property of F_i^+. That is, after one or multiple steps of reduction ($\hookrightarrow^*$), the inferred type of the reduced expression is a runtime subtype. Therefore, for checked expressions, the initial type-checking always succeeds.

Theorem 7.6 (Type preservation with runtime subtyping)**.** If $\cdot; \cdot \vdash e \Leftrightarrow A$ and $e \hookrightarrow^* e'$ then there exists a type B such that $\cdot; \cdot \vdash e' \Leftrightarrow B$ and $B \lesssim A$.

Corollary 7.1 (Type preservation)**.** If $\cdot; \cdot \vdash e \Leftrightarrow A$ and $e \hookrightarrow^* e'$ then $\cdot; \cdot \vdash e' \Leftarrow A$.

Summary In this chapter, we presented a new formulation of the F_i^+ calculus and showed how it serves as a direct foundation for Compositional Programming. In contrast with the original F_i^+, we adopt a direct semantics based on the TDOS approach and embrace call-by-name evaluation. As a result, the metatheory of F_i^+ is significantly simplified, especially due to the fact that a coherence proof based on logical relations and contextual equivalence is not needed. In addition, our formulation of F_i^+ enables recursion and impredicative polymorphism, validating the original trait encoding by Zhang et al. [ZSO21]. We proved the type-soundness and determinism of F_i^+ using the Coq proof assistant. Our research explores further possibilities of the TDOS approach and shows some novel notions that could inspire the design of other calculi with similar features.

CHAPTER 8

Discussion and Related work

8.1 Discussion

This section provides some discussion on design choices, a comparison with elaboration semantics as well as implementation considerations and possible extensions for future work.

8.1.1 TDOS versus an Elaboration Semantics

This dissertation proposes the use of type-directed operational semantics for modelling languages with a merge operator. Since a general form of the merge operator has a type-directed semantics, previous work has favored an elaboration semantics. Traditionally, for languages with type-directed semantics, elaboration has been a common choice. That is the case, for instance, for many previous calculi with the merge operator [Dun14; OSA16; AOS17; BOS18; Bi+19], *gradual typing* [ST06], or *type classes* [WB89; Kae88]. In the elaboration approach, the idea is that the source language can be translated via a type-directed translation into a conventional target calculus, whose semantics is not type-directed. In the case of languages with the merge operator, the target calculus is typically a conventional calculus (such as the STLC) extended with pairs. The implicit upcasts that extract components from merges are modelled by explicit projections with pairs. One very appealing benefit of the elaboration semantics is that it gives a simple way to obtain an implementation. Since elaboration targets conventional languages/calculi, that means that it can simply reuse existing and efficient implementations of languages. For example, in the case of the merge operator, a key motivation of Dunfield was that the elaboration could just target ML-like languages, which have several efficient implementations available. From the implementation point of view, the TDOS would be more useful for guiding the design of a dedicated virtual machine, compiling directly to bytecode/assembly, or having an interpreter. However, obtaining an efficient implementation for the latter options would require significantly more effort than with an elaboration approach.

Nevertheless, the reason for designing a semantics for a language or calculus is not merely implementation. In fact, the implementation aspects are not our primary motivation for TDOS. Although, as we will discuss in Section 8.1.2, TDOS can provide some insights for obtaining efficient implementations as well. Furthermore we do not view TDOS as being mutually exclusive with elaboration: both approaches have interesting aspects and they can serve different purposes. Our main motivation for the TDOS is reasoning. A semantics is supposed to describe the meaning of the language constructs in the language. Having a clear and high-level presentation of the semantics is then useful for language implementers to understand the behaviour of the language. Furthermore, it is also useful to provide programmers with a mental model of how programs are executed, and to study the properties of the language. Next, we give more details on the advantages of a direct semantics over the elaboration semantics in terms of reasoning and proof methods employed in previous work on disjoint intersection types.

Shorter, more direct reasoning. Programmers want to understand the meaning of their programs. A formal semantics can help with this. With our TDOS we can essentially employ a style similar to equational reasoning in functional programming to directly reason about programs written in λ_i. For example, it takes a few reasoning steps to work out the result of $(\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int})\ (2 \,,, \text{`c'})$:

$$
\begin{array}{lll}
 & (\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int})\ (2 \,,, \text{`c'}) & \\
\hookrightarrow & (2 + 1) : \mathsf{Int} & \text{by } \textsc{Step-beta} \text{ and casting} \\
\hookrightarrow & 3 : \mathsf{Int} & \text{by } \textsc{Step-anno} \text{ and arithmetic} \\
\hookrightarrow & 3 & \text{by } \textsc{Step-annov} \text{ and casting}
\end{array}
$$

Here reasoning is easily justifiable from the small-step reduction rules and type-directed reduction. Building tools (such as debuggers), that automate such kind of reasoning should be easy using the TDOS rules.

However, with an elaboration semantics, the (precise) reasoning steps to determine the final result are more complex. Firstly the expression has to be translated into the target language before reducing to a similar target term. Figure 8.1 shows this elaboration process in λ_i, where an expression in the source language is translated into an expression in a target language with products. The source term $(\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int})\ (2 \,,, \text{`c'})$ is elaborated into the target term $(\lambda x.\, x + 1)\ (\mathsf{fst}\ (2 \,, \text{`c'}))$. As we can see the actual derivation is rather long, so we skip the full steps. Also, for simplicity's sake, here we assume the subtyping judgement produces the most straightforward coercion fst. This elaboration step and the introduction of coercions into the program make it harder for programmers to precisely understand the semantics of a program. Moreover while the coercions inserted in this small expression may not look too bad, in larger programs the addition of coercions can be a lot more severe, hampering the understanding of

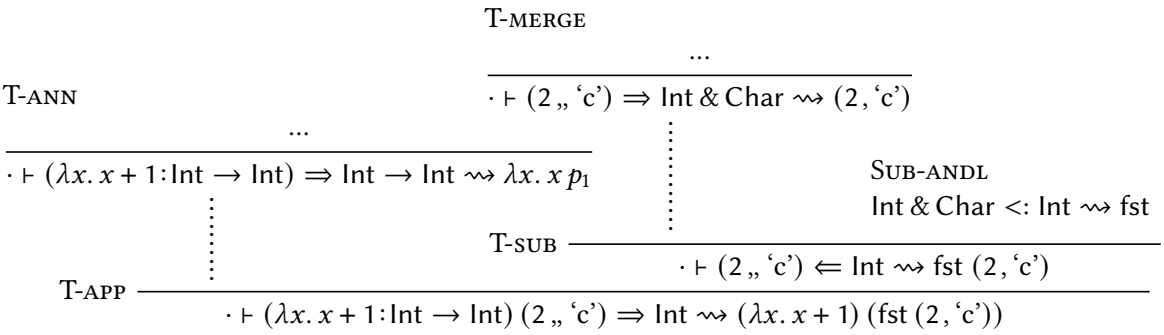

Figure 8.1: Elaboration of $(\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int})\ (2 \,,, \text{`c'})$ to a calculus with products.

the program.
After elaboration we can then use the target language semantics, to determine a target language value.

$$
\begin{array}{lll}
 & (\lambda x.\, x + 1)\ (\mathsf{fst}\ (2 \,, \text{`c'})) & \\
\hookrightarrow & (\lambda x.\, x + 1)\ 2 & \text{reduction for application and pairs} \\
\hookrightarrow & 2 + 1 & \text{by the beta reduction rule} \\
\hookrightarrow & 3 & \text{by arithmetic}
\end{array}
$$

A final issue is that sometimes it is not even possible to translate back the value of the target language into an equivalent "value" on the source. For instance in the λ_i^+'18 calculus [BOS18] $1 : \mathsf{Int}\ \&\ \mathsf{Int}$ results in $(1, 1)$, which is a pair in the target language. But the corresponding source value $1 ,, 1$ is not typable in λ_i^+'18. In essence, with an elaboration, programmers must understand not only the source language, but also the elaboration process as well as the semantics of the target language, if they want to precisely understand the semantics of a program. Since the main point of semantics is to give clear and simple rules to understand the meaning of programs, a direct semantics is a better option for providing such understanding.

Simpler proofs of unambiguity. For calculi with an elaboration semantics, unrestricted intersections make it harder to prove coherence. Our λ_i calculus, on the other hand, has a deterministic semantics, which implies unambiguity directly. For instance, $(1 : \mathsf{Int}\ \&\ \mathsf{Int}) : \mathsf{Int}$ only steps to 1 in λ_i. But it can be elaborated into two target expressions in the λ_i^+'18 calculus corresponding to two typing derivations:

$$
\begin{array}{l}
(1 : \mathsf{Int}\ \&\ \mathsf{Int}) : \mathsf{Int} \rightsquigarrow \mathsf{fst}\ (1 \,,\, 1) \\
(1 : \mathsf{Int}\ \&\ \mathsf{Int}) : \mathsf{Int} \rightsquigarrow \mathsf{snd}\ (1 \,,\, 1)
\end{array}
$$

Thus the coherence proof needs deeper knowledge about the semantics: the two different terms are known to both reduce to 1 eventually. Therefore they are related by the logical

relation employed in λ_i^+'18 for coherence. Things get more complicated for functions. The following example shows two possible elaborations of the same function. Relating them requires reasoning inside the binders and a notion of contextual equivalence.

$$\lambda x.\, x + 1 : \mathsf{Int} \,\&\, \mathsf{Int} \to \mathsf{Int} \rightsquigarrow \lambda x.\, \mathsf{fst}\, x + 1$$
$$\lambda x.\, x + 1 : \mathsf{Int} \,\&\, \mathsf{Int} \to \mathsf{Int} \rightsquigarrow \lambda x.\, \mathsf{snd}\, x + 1$$

Furthermore, the two target expressions above are *clearly not equivalent* in the general case. For instance, if we apply them to $(1, 2)$ we get different results. However, the target expressions will always behave equivalently when applied to arguments *elaborated from the λ_i^+'18 source calculus.* λ_i^+'18, forbids terms like $(1 \,,, 2)$ and thus cannot produce a target value $(1, 2)$. Because of elaboration and also this deeper form of reasoning required to show the equivalence of semantics, calculi defined by elaboration require a lot more infrastructure for the source and target calculi and the elaboration between them, while in a direct semantics only one calculus is involved and the reasoning required to prove determinism is quite simple.

Not limited to terminating programs. The (basic) forms of logical relations employed by λ_i^+'18 and F_i^+'19 cannot deal with non-terminating programs. In principle, recursion could be supported by using a *step-indexed logical relation* [Ahm06], but this is left for future work. Our calculi smoothly handle unrestricted intersections and recursion, using TDOS to reach determinism with a significantly simpler proof method. It also makes other features that lead to non-terminating programs, such as *recursive types*, feasible.

8.1.2 Implementation considerations

The TDOS for λ_i, λ_i^+, and F_i^+ is implementable directly, since the relations developed in this dissertation are all essentially algorithmic. However, a direct implementation will not be very efficient, for multiple reasons. Next we discuss some considerations for the design of efficient implementations of languages with the merge operator.

Merge lookups. From the efficiency point of view, one particularly bad aspect of the TDOS is that run-time lookup on merges (triggered by casting) does not exploit statically known information. For instance, if we have a program such as:

$$\mathsf{let}\ x : A \,\&\, \mathsf{Int} \,\&\, A \ =\ e \,,, 2 \,,, e\ \mathsf{in}\ x + 1$$

extracting the number 2 from the merge currently requires blindly going through the elements in the merge at runtime until the integer value is found. The approach in the elaboration semantics is much better here, since during type-checking we know statically where the integer value can be found. Therefore, when generating code in the target language we can simply

look up the value directly at that position, thus avoiding having to search for the value in the merge at runtime. It should be possible to create optimizations for the TDOS by employing similar ideas to the elaboration. That is, using the type system to statically determine where to find values. For this to work with the TDOS, we would need to extend the TDOS with explicit projections, so that code with implicit projections could be replaced with explicit projections. For instance, if the integer 2 is at position 5 in a merge, the code $x + 1$ could be replaced by $x[5] + 1$, where $x[5]$ denotes the projection of the 5th element in the merge. We plan to investigate them for future work.

Overhead from annotations. One other source of concern for efficiency is the overhead caused by type annotations, and their use at runtime. Type annotations in a TDOS play a very similar role to casts in cast languages [WF09; SW10], which are used, for instance, as targets for gradually typed languages. The problem of how to design efficient cast languages is an important topic in the gradual typing literature, and the overhead caused by casts have been a notorious challenge in that area. In particular, a main source of overhead comes from casting functions. In calculi such as the blame calculus, function values need to accumulate casts to avoid raising blame too early. Thus, function values can have an arbitrary number of casts, which also leads to space efficiency concerns. Luckily, for λ_i and λ_i^+, we do not need to accumulate annotations around function values. Indeed, as shown in Section 7.2, function values are of the form $\lambda x.\, e : A \to B$ in λ_i and λ_i^+. That is, they only contain a single annotation. In contrast to cast calculi, the approach used in λ_i and λ_i^+, when reduction encounters multiple annotations around a function is to simply replace the type annotation of the value. This process happens in the casting rule Cast-arrow for functions:

$$\textsc{Cast-arrow}\quad \frac{\neg\rceil A_2 \lceil \qquad A_1 <: B_1 \qquad B_2 <: A_2}{\lambda x.\, e : B_1 \to B_2 \hookrightarrow_{(A_1 \to A_2)} \lambda x.\, e : B_1 \to A_2}$$

Moreover, as discussed in Section 5.3.2, the premise $A_1 <: B_1$ is not needed at runtime, and can be avoided in an implementation. Thus, while the overhead caused by type annotations is still a concern, we believe that λ_i and λ_i^+ avoid some of the thornier issues that gradually typed languages have to deal with. Moreover, annotation replacement could actually have some advantages over an elaboration approach. Multiple annotations around functions, would result in multiple coercions being applied to the function in an elaboration approach. In contrast annotation replacement avoids such coercions and immediately collapses annotations, so that, ultimately, we only need to type-reduce the argument and the result of the function one time. Nevertheless, a proper assessment of the performance impact of annotations at runtime is outside of the scope of this work, and left for future work.

Parallel application. Parallel application in λ_i^+ may raise some concern if available in unrestricted ways to programmers. In λ_i^+ parallel application arises as a consequence of the distributivity of intersections over functions. That is, the following is a valid subtyping statement in λ_i^+ (and BCD subtyping):

$$(A_1 \to B_1)\&(A_2 \to B_2) <: A_1\&A_2 \to B_1\&B_2$$

In essence, an intersection of two functions can be viewed as a single function with the intersections of the inputs and outputs. Thus, parallel application arises naturally in the semantics of the language in order to support such form of conversions at runtime.

The semantics for the application of merged functions is to apply all of the functions to the argument. This turns what looks like one function call into two or more function calls and could have a significant impact on the time complexity of a program. A program that looks linear-time could in fact be exponential, due to parallel application. Therefore it is important to consider the consequences of parallel application and how they can be mitigated. One option would be to redesign the calculus so that parallel application is avoided. This would probably require weakening the subtyping relation to avoid type conversions like the above. However, parallel application is an important aspect of nested composition, and dropping parallel application from the calculus would prevent important applications of nested composition. Instead we believe that a better approach is to restrict the use of parallel application in the source language that targets λ_i^+. In the following section we will come back to this topic.

8.2 Formal Relations to Existing Calculi

In this section, we compare the basic system λ_i (introduced in Chapter 5) with two closely related work in the literature: the original λ_i [OSA16] calculus and Dunfield's system [Dun14].

8.2.1 Completeness of λ_i with Respect to the Original Type System

To disambiguate, we use λ_i'16 to denote the original calculus and λ_i for our variant. We prove that the type system of the new variant is at least as expressive as the λ_i'16 calculus[1]. The syntax of λ_i'16 (minus pairs and product types) is almost the same as λ_i, except that there are no fixpoints and the lambdas do not have any type annotations. Thus lambdas can only be typed in checked mode. Figure 8.2 presents an excerpt of the type system. The type system has a type well-formedness definition and a slightly different disjointness relation compared to our variant of λ_i. Also note that the rule for the merge of values (rule Typ-mergev) is absent

[1]Note that the original λ_i includes pairs and product types. In the Coq formalization we have a variant with pairs and product types as well. It has all the previous properties proved in this section. For simplicity and consistency of presentation, we use the variant without pairs and product types here.

$\boxed{\Gamma \models A}$ *(Type Well-formedness)*

$$\textsc{Wf-top}\ \frac{}{\Gamma \models \top} \qquad \textsc{Wf-int}\ \frac{}{\Gamma \models \mathsf{Int}} \qquad \textsc{Wf-arr}\ \frac{\Gamma \models A \qquad \Gamma \models B}{\Gamma \models A \to B} \qquad \textsc{Wf-and}\ \frac{\Gamma \models A \qquad \Gamma \models B \qquad A * B}{\Gamma \models A \,\&\, B}$$

$\boxed{\Gamma \models e \Leftrightarrow A \hookrightarrow e'}$ *(Bidirectional Typing)*

$$\textsc{IBTyp-top}\ \frac{}{\Gamma \models \top \Rightarrow \top \hookrightarrow \top} \qquad \textsc{IBTyp-lit}\ \frac{}{\Gamma \models i \Rightarrow \mathsf{Int} \hookrightarrow i} \qquad \textsc{IBTyp-var}\ \frac{x : A \in \Gamma}{\Gamma \models x \Rightarrow A \hookrightarrow x}$$

$$\textsc{IBTyp-sub}\ \frac{\Gamma \models e \Rightarrow A \hookrightarrow e' \qquad A <: B}{\Gamma \models e \Leftarrow B \hookrightarrow e'}$$

$$\textsc{IBTyp-app}\ \frac{\Gamma \models e_1 \Rightarrow A \to B \hookrightarrow e_1' \qquad \Gamma \models e_2 \Leftarrow A \hookrightarrow e_2'}{\Gamma \models e_1\ e_2 \Rightarrow B \hookrightarrow e_1'\ e_2'} \qquad \textsc{IBTyp-merge}\ \frac{\Gamma \models e_1 \Rightarrow A \hookrightarrow e_1' \qquad \Gamma \models e_2 \Rightarrow B \hookrightarrow e_2' \qquad A * B}{\Gamma \models e_1 \,,, e_2 \Rightarrow A \,\&\, B \hookrightarrow e_1' \,,, e_2'}$$

$$\textsc{IBTyp-anno}\ \frac{\Gamma \models e \Leftarrow A \hookrightarrow e'}{\Gamma \models e{:}A \Rightarrow A \hookrightarrow e'{:}A}$$

$$\textsc{IBTyp-fix}\ \frac{\Gamma \models A \qquad \Gamma, x{:}A \models e \Leftarrow A \hookrightarrow e'}{\Gamma \models \mathsf{fix}\, x.\, e \Leftarrow A \hookrightarrow \mathbf{fix}\, x{:}A.\, e'} \qquad \textsc{IBTyp-lam}\ \frac{\Gamma \models A \qquad \Gamma, x{:}A \models e \Leftarrow B \hookrightarrow e'}{\Gamma \models \lambda x.\, e \Leftarrow A \to B \hookrightarrow (\lambda x.\, e' : A \to B)}$$

Figure 8.2: The type system of the original λ_i (extended with fixpoints). Three rules that relate with product types are ignored.

because the disjointness restriction in well-formedness prevents duplicated values.

Some details need to be explained before presenting the completeness theorem. Firstly, subtyping in our variant of λ_i is stronger due to top-like types. Secondly, top-like types are disjoint to any type in our variant, while the disjointness in the original λ_i'16 is restricted to types that are not top-like. We extended the bidirectional type system of the original λ_i'16 with recursion and designed an elaboration from the extended system to λ_i. We proved a theorem which shows the type system of λ_i can type check any well-typed terms in λ_i'16, with type annotations inserted based on the typing derivation:

Theorem 8.1 (Completeness of typing with respect to the extended original λ_i)**.** If $\Gamma \models e \Leftrightarrow A \hookrightarrow e'$, then $\Gamma \vdash e' \Leftrightarrow A$.

The well-typed expression e in λ_i'16 is mapped to e' in λ_i which is proved to be well-typed. It means that λ_i'16 can be used as a surface language where many of the explicit annotations of λ_i are inferred automatically. Moreover, the extension of fixpoints further shows that some type inference with recursion is feasible. As the elaboration is also complete to the original type system, the λ_i'16 calculus can be translated into λ_i without loss of expressivity or flexibility.

8.2.2 Soundness of λ_i with respect to Dunfield's Operational Semantics

Dunfield's non-deterministic operational semantics [Dun14] motivates our TDOS. Here, we show the soundness of the operational semantics of λ_i with respect to a slightly extended version of Dunfield's semantics. The need for extending Dunfield's original semantics is mostly due to the generalization of the rule S-TOP in subtyping. In the conference paper [HO20] we also discuss a variant of λ_i, which uses the original subtyping, and show that such a variant requires no changes to Dunfield's semantics.

Dunfield's original reduction rules are presented in Section 2.2. We extend her operational semantics with two rules rules DSTEP-TOP and DSTEP-TOPARR.

$\boxed{e \rightsquigarrow e'}$ *(The Extension of Dunfield's Operational Semantics)*

$$\frac{e_1 \rightsquigarrow e_1'}{e_1\ e_2 \rightsquigarrow e_1'\ e_2}\ \text{DSTEP-APPL} \qquad \frac{e_2 \rightsquigarrow e_2'}{v_1\ e_2 \rightsquigarrow v_1\ e_2'}\ \text{DSTEP-APPR} \qquad \frac{}{(\lambda x.\ e)\ v \rightsquigarrow e[x \mapsto v]}\ \text{DSTEP-BETA} \qquad \frac{}{\mathsf{fix}\ x.\ e \rightsquigarrow e[x \mapsto \mathsf{fix}\ x.\ e]}\ \text{DSTEP-FIX}$$

$$\frac{}{e_1 \,,, e_2 \rightsquigarrow e_1}\ \text{DSTEP-UNMERGEL} \qquad \frac{}{e_1 \,,, e_2 \rightsquigarrow e_2}\ \text{DSTEP-UNMERGER} \qquad \frac{e_1 \rightsquigarrow e_1'}{e_1 \,,, e_2 \rightsquigarrow e_1' \,,, e_2}\ \text{DSTEP-MERGEL} \qquad \frac{e_2 \rightsquigarrow e_2'}{e_1 \,,, e_2 \rightsquigarrow e_1 \,,, e_2'}\ \text{DSTEP-MERGER}$$

$$\frac{}{e \rightsquigarrow e \,,, e}\ \text{DSTEP-SPLIT} \qquad \frac{}{v \rightsquigarrow \top}\ \text{DSTEP-TOP} \qquad \frac{}{\top\ v \rightsquigarrow \top}\ \text{DSTEP-TOPARR}$$

Rule DSTEP-TOPARR states that the value $\top$ can be used as a lambda which returns $\top$, suggested by the newly added top-like types for arrow types returning $\top$. Rule DSTEP-TOP states that any value can be reduced to $\top$, corresponding to $A <: \top$. Dunfield avoids having a rule DSTEP-TOP by performing a simplifying elaboration step in advance:

$$\frac{}{\Gamma \vdash v : \top \hookrightarrow \top}\ \text{DUNFIELD-TYPING-T}$$

With such a rule, values of type $\top$ are directly translated into $\top$, and do not need any further reduction in the target language. We do not have such an elaboration step. Instead we extend the original semantics with the two rules above.

Type erasure. Differently from Dunfield's calculus, λ_i uses type annotations in its syntax to obtain a direct operational semantics. $|e|$ erases annotations in term e. By erasing all annotations, terms in λ_i can be converted to terms in Dunfield's calculus (and also the original

λ_i). Note that for every value v in λ_i, $|\, v \,|$ is a value as well.

$$
\begin{array}{rcl}
|\, i \,| & = & i \\
|\, \top \,| & = & \top \\
|\, \lambda x.\, e : A \to B \,| & = & \lambda x.\, |\, e \,| \\
|\, \mathbf{fix}\ x : A.\, e \,| & = & \mathsf{fix}\, x.\, |\, e \,| \\
|\, e \,| : A & = & |\, e \,| \\
|\, e_1 \,|\, e_2 & = & |\, e_1 \,|\, |\, e_2 \,| \\
|\, e_1 \,|\, ,, e_2 & = & |\, e_1 \,|\, ,, |\, e_2 \,|
\end{array}
$$

Soundness. Given Dunfield's extended semantics, we can show a theorem that each step in the TDOS of λ_i corresponds to zero, one, or multiple steps in Dunfield's semantics.

Theorem 8.2 (Soundness of $\hookrightarrow$ with respect to Dunfield's semantics)**.** If $e \hookrightarrow e'$, then $|\, e \,| \rightsquigarrow^* |\, e' \,|$.

A necessary auxiliary lemma for this theorem is the soundness of casting.

Lemma 8.1 (Soundness of casting with respect to Dunfield's semantics)**.** If $v \hookrightarrow_A v'$, then $|\, v \,| \rightsquigarrow^* |\, v' \,|$.

This lemma shows that although the type information guides the reduction of values, it does not add additional behavior to values. For example, a merge can step to its left part (or the right part) with rule Cast-mergevl (or rule Cast-mergevr), corresponding to rule DStep-unmergel (or rule DStep-unmerger). Rule Cast-and can be understood as a combination of splitting (rule DStep-split $v \rightsquigarrow v \,,,\, v$) and further reduction on each component separately.

8.3 Calculi with the Merge Operator

This section discusses various lines of related work, including calculi with a merge operator, record calculi and some work on OOP languages.

For easy reference and distinction, sometimes we attach the publication year to the work to the corresponding calculus name. For instance, F_i^+'19 means the original formulation of F_i^+ by Bi et al. [Bi+19].

8.3.1 Calculi with the merge operator and a direct semantics

Intersection types with a merge operator are a key feature of the Forsythe language of Reynolds [Rey88]. He also studied a core calculus with similarities to λ_i [Rey91]. However, merges in Forsythe are restricted and use a syntactic criterion to determine what merges are allowed. A merge is permitted only when the second term is a lambda abstraction or a single

field record, which makes the structure of merges always biased. To prevent potential ambiguity, the latter overrides the former when they overlap. If formalized as a tree, the right child of every node is a leaf. The only place for primitive types is the leftmost component. Forsythe follows the standard call-by-name small-step reduction, during which types are ignored. The reduction rules deal with merges by continuously checking if the second component can be used in the context (abstractions for application, records for projection). This simple approach, however, is unable to reduce merges when (multiple) primitive types are required. Reynolds admitted this issue in his later work [Rey97]. We use types to select values from a merge and the disjointness restriction guarantees the determinism. Therefore the order of a value in a merge is not a deciding factor on whether the value is used.

The calculus $\lambda\&$ proposed by Castagna, Ghelli, and Longo has a restricted version of the merge operator for functions only [CGL95]. The merge operator is indexed by a list of types of its components. Its operational semantics uses the runtime types of values to select the "best approximate" branch of an overloaded function. $\lambda\&$ requires runtime type checking on values, while in TDOS, all type information is already present in type annotations. Another obvious difference is that λ_i supports merges of any type (not just functions), which are useful for applications other than overloading of functions, including: multifield *extensible records with subtyping* [OSA16]; encodings of *objects* and *traits* [BO18]; *dynamic mixins* [AOS17]; or simple forms of *family polymorphism* [BOS18].

Several other calculi with intersection types and overloading of functions have been proposed [CX11; Cas+15; Cas+14], but these calculi do not support a merge operator, and thus avoid the ambiguity problems caused by the construct.

8.3.2 Calculi with a Merge Operator and an Elaboration Semantics

Instead of a direct semantics, many recent works [Dun14; OSA16; AOS17; BOS18; Bi+19] on intersection types employ an elaboration semantics, translating merges in the source language to products (or pairs) in a target language. With an elaboration semantics the subtyping derivations are coercive [Luo99]: they produce coercion functions that explicitly convert terms of one type to another in the target language. This idea was first proposed by Dunfield, where she shows how to elaborate a calculus with intersection and union types and a merge operator to a standard call-by-value lambda calculus with products and sums [Dun14]. Dunfield also proposed a direct semantics, which served as inspiration for our work. However, her direct semantics is non-deterministic and lacks subject reduction (as discussed in detail in Section 2.2). Unlike Forsythe and $\lambda\&$, Dunfield's calculus has unrestricted merges and allows a merge to work as an argument. Her calculus is flexible and expressive and can deal with several programs that are not allowed in Forsythe and $\lambda\&$.

To remove the ambiguity issues in Dunfield's work, the original λ_i calculus [OSA16] forbids overlapping in intersections using the disjointness restriction for all well-formed intersections.

	$\lambda_{\&}$	λ_i'16	F_i	λ_i^+'18	F_i^+'19	λ_i	λ_i^+	F_i^+
Disjointness	○	●	●	●	●	●	●	●
Unrestricted Intersections	●	○	○	●	●	●	●	●
Determinism / Coherence	No	Coh.	Coh.	Coh.	Coh.	Det.	Det.	Det.
Recursion	●	○	○	○	○	●	●	●
Direct / Elaboration Semantics	Dir.	Ela.	Ela.	Ela.	Ela.	Dir.	Dir.	Dir.
Subject Reduction	○	-	-	-	-	●	●	●
Distributive Subtyping	○	○	○	●	●	○	●	●
Disjoint Polymorphism	○	○	●	○	●	○	○	●
Evaluation Strategy	CBV	CBV	CBV	CBV	CBV	CBV	CBV	CBN

Figure 8.3: Summary of intersection calculi with the merge operator.
(● = yes, ○ = no, - = not applicable)

Since it does not support unrestricted intersections, the proof of coherence in λ_i'16 is relatively simple. Likewise, in the following work on the F_i calculus [AOS17], which extends λ_i with disjoint polymorphism, *all* intersections must be *disjoint*. However, the disjointness restriction causes difficulties because it breaks *stability of type substitutions*. Stability is a desirable property in a polymorphic type system that ensures that if a polymorphic type is well-formed then any instantiation of that type is also well-formed. Unfortunately, with disjoint intersections only, this property is not true in general. Thus F_i can only prove a restricted version of stability, which makes its metatheory non-trivial.

Disjointness of all well-formed intersections is only a sufficient (but not necessary) restriction to ensure an unambiguous semantics. The λ_i^+'18 calculus [BOS18] relaxes the restriction without introducing ambiguity. λ_i^+'18 employs the disjointness restriction *only* on merges, but otherwise allows *unrestricted intersections*. It allows $1 : \mathsf{Int} \,\&\, \mathsf{Int}$, but the same term is rejected in the original λ_i. Unfortunately, this comes at a cost: it is much harder to prove the coherence of elaboration. Both λ_i^+'18 and F_i^+'19 [Bi+19] deal with this problem by establishing coherence using contextual equivalence and a *logical relation* [Tai67; Plo73; Sta85] to prove it. The proof method, however, cannot deal with non-terminating programs. In fact, none of the existing calculi with disjoint intersection types supports recursion, which is a severe restriction.

We retain the essence of the power of Dunfield's calculus (modulo the disjointness restrictions to rule out ambiguity), and gain benefits from the direct semantics.

Figure 8.3 summarizes the key differences between our work and prior work, focusing on the most recent work on disjoint intersection types. The last three calculi are present in the thesis.

8.4 Record Calculi with Record Concatenation and Subtyping

As we have seen, in calculi with disjoint intersection types and records, the merge operator concatenates records in a symmetric way. However, designing a record concatenation operator, no matter symmetric or asymmetric, is a difficult problem in calculi with subtyping, as identified by Cardelli and Mitchell [CM91]. In both cases, a record can "hide" some fields via subsumption to bypass the restriction on types. This issue has been discussed in Section 2.3. As far as we know, no existing record calculus in the literature supports nested composition. However, there are numerous designs in the literature with concatenation or subtyping or both.

Asymmetric concatenation without subtyping. Record concatenation is used by Wand to model multiple inheritance [Wan89]. He has a biased operator, which overrides the first term by the second if they have conflicting fields. He makes every record types explicitly state whether a field is present in it or absent with respect to a fixed set of labels, which could be infinite. It is similar to the algorithm implemented by Rémy to type-check records in an ML extension, which keeps track of if a field is absent or not [Rém89].

Symmetric concatenation without subtyping. Harper and Pierce design a record calculus with symmetric concatenation [HP91]. Their system $\lambda^{||}$ keeps the extension and restriction operators in terms and types and generalizes them to work on two records (or record types). A compatibility check is enforced on types, via the typing of record concatenation and type well-formedness definition. Their type quantification only takes care of negative information. For example, $\Lambda a\#l.a$ stands for any type that does not have a field of name l. The bottom type serves as the key to resolving the semantic difference between type disjointness and compatibility. A compatibility constraint with `{l: t}` in $\lambda^{||}$ is equivalent to a disjointness constraint with `{l: Bot}` in F_i^+. Although they avoid subtyping, it is still necessary to reason about type equivalence with the type operations in their calculus. Their *compatibility* constraint plays a similar role disjoint quantification in our system. Xie et al. demonstrated that disjoint polymorphism subsumes the form of row polymorphism present in $\lambda^{||}$ [Xie+20].

A similar disjointness constraint for symmetric record concatenation is employed for the language Ur presented by Chlipala, which has a more comprehensive reasoning on type equivalence [Chl10]. Ur is a dependently typed language with first-class labels, designed for statically typed meta-programming with type inference. It encodes disjoint assertions in guarded types. Ur uses type-level computation, including a type-level map operation, to allow flexible and generic programs to be written using records. The semantics is given by elaboration, and a translation of Ur programs into terms of the Calculus of Inductive Constructions ensures type soundness. Like Harper and Pierce's work, Ur has no subtyping as it introduces ambiguity.

The absence of subtyping avoids the "hidden fields" problem.

Record subtyping without concatenation. Cardelli and Mitchell propose to use extension and restriction as primitive operators instead of concatenation [CM91]. They introduce type operators and negative restrictions in record types, so that in their calculus, via bounded quantification, programmers can declare a polymorphic function that takes any records lacking certain fields. Like merges in λ_i^+, extension in their system is conflict-free, which is ensured by static type-checking. Compared to restriction, we can use type annotations to drop fields in a record in λ_i^+. A difference is that the two operators deal with a record and a field, while our system can handle two records in a merge. As identified by them, with subtyping, a record can "hide" some fields via subsumption to bypass the restriction on types, which makes it hard to capture the absent fields in record calculi. Their solution is to add type operations that can encode negative information in subtyping. Corresponding to the term operators, they have two type operators of the same name: extension on a type requires that the given field is absent from the type; restriction on a type explicitly excludes the field from it, if it has such a field. For example, $\{\}\backslash l$ stands for an empty record with a restriction that cannot have field l. Any subtype cannot have l either. Besides special subtyping rules, they define type equivalence rules to reason about type operations on records.

Subtyping-constraint-based calculi. Rémy [Rém95] and following work by Pottier [Pot00] handle both symmetric and asymmetric concatenation in a constraint-based type system. To deal with record concatenation, type operators or conditional constraints are used to express two branches: either a field exists or is absent, mirroring the reduction of programs. In subtyping, the type of records are distinguished into two forms: rigid record types and flexible record types. A rigid record type of a term reflects all fields in it. Rigid records have no subtyping, but they can be used in a concatenation with another record. Every rigid record type corresponds to a flexible record type, which has subtypes and supertypes. However flexible records cannot be used with concatenation. In λ_i and λ_i^+ all records are flexible and they can be used with concatenation. Moreover, the subtyping in their systems is not expressive enough to support nested composition.

Record calculi as extensions of system $F_{<:}$. The $F_{<:\rho}$ calculus proposed by Cardelli [Car94] extends System $F_{<:}$ by extensible records, and combines row quantification used in the previously discussed Harper and Pierce's work [HP91] with bounded quantification. $F_{<:\rho}$ does not have record concatenation as a primitive operator. Instead, it has row extension and restriction. A translation to $F_{<:}$ is provided. Poll [Pol97] solves the polymorphic record update problem in System F with a restricted formulation of subtyping: it only supports width-subtyping on record types. It has a record-update operator instead of concatenation. One record-update

operation only alters a field in a record. The subtype checking in its typing rule makes sure the record contains that field of the expected type.

The $F_{\#}$ calculus by Zwanenburg supports intersection types [Zwa95] (in its later version [Zwa97] intersection types are eliminated) and record concatenation in a $F_{<:}$-like system. Similar to λ_i^+, multi-field records are obtained by concatenating single-field records, and there is a distributivity rule for records in subtyping as well. They use a "with" construct for record concatenation which is similar to the merge operator. Like rule Typ-merge, the typing of "with" introduces intersections, and it has a compatibility pre-condiction for the two terms' types (written as $A\#B$). Only record types or Top can be compatible. The concatenation operator is asymmetric. When two concatenated records have the same label, the right one overwrites the left. Correspondingly, two compatible types can have common fields as long as for those shared fields, the right one has a subtype of the left's , e.g. $\{l : \mathsf{Int}\}\#\{l : \mathsf{Int}\} \,\&\, \{l : \mathsf{Char}\}$. In contrast, disjointness is symmetric, and a type (unless it is top-like) cannot be disjoint with its subtypes, to ensure the two sides of a merge coexist safely. To prevent the issue of subsumption "hiding" fields of different types the compatibility checking, they require explicit annotations on merged records. These annotations are used during elaboration to a target calculus, therefore affecting the program behavior, like in our calculus. The semantics of $F_{\#}$ is given by elaborating into system F with pairs and records. In this sense, it predates Dunfield's work. Concatenated records are translated into pairs, where a special "overwriter" function, generated by the compatibility derivation, is applied to update the overlapped fields in the first record by the second one. In Zwanenburg's work coherence is left for future work.

8.5 Languages and Calculi with a Type-Dependent Semantics

Typed operational semantics. Goguen's typed operational semantics combines typing and reduction in one relation [Gog94]. It is designed for studying meta-theoretic properties, especially about the reduction of well-typed terms, and is not aimed to describe type-dependent semantics. Typed operational semantics has been applied to several systems, include *simply typed lambda calculi* [Gog95], calculi with *dependent types* [Gog94; FL09] and *higher-order subtyping* [CG03]. Note that the semantics of these systems does not depend on typing, and the untyped (type-erased) reduction relations are still presented to describe how to evaluate programs. As in typed operational semantics, reduction has access to typing derivations, it is possible to model the semantics of merge operator in this framework. But the resulting systems might be similar to the prior work on disjoint intersection types with elaboration semantics, and it is unclear how to cope with non-terminating programs.

Type classes Type classes [WB89; Kae88] are an approach to parametric overloading used in languages like Haskell. The commonly adopted compilation strategy for it is the dictionary passing style elaboration [WB89; Hal+96; CKJ05; Cha+05]. Other mechanisms inspired by type classes, such as Scala's *implicits* [OMO10], Agda's *instance arguments* [DP11] or Ocaml's *modular implicits* [WBY14] have an elaboration semantics as well. In one of the pioneering works of type classes, Kaes gives two formulations for a direct operational semantics [Kae88]. One of them decides the concrete type of the instance of overloaded functions at *run-time*, by analyzing all arguments after evaluating them. In both Kaes' work and the following work by Odersky, Wadler, and Wehr [OWW95], the run-time semantics has some restrictions with respect to type classes. For example, overloading on return types (needed for example for the *read* function in Haskell) is not supported. Interestingly, the semantics of λ_i allows overloading on return types, which is used whenever two functions coexist on a merge. For a detailed example, please refer to Section 6.3.

Gradual typing. Gradual typing [ST06] is another example of a type-dependent mechanism, since the success or not of an (implicit) cast may depend on the particular type used for the implicit cast. Thus the semantics of a gradually typed language is type-dependent. Like other type-dependent mechanisms the semantics of gradually typed source languages is usually given by a (type-dependent) elaboration semantics into a cast calculus, such as the *Blame calculus* [WF09] or the *Threesome calculus* [SW10].

Multiple dispatching. Multiple dispatching [Cli+00; CC99; Mus+08; Par+19] generalizes object-oriented dynamic dispatch to determine the overloaded method to invoke based on the runtime type of all its arguments. Similarly to TDOS, much of the type information is recovered from type annotations in multiple dispatching mechanisms, but, unlike TDOS, they only use input types to determine the semantics.

8.6 OOP

8.6.1 Dealing with Conflicts in OOP

There is a rich literature in OOP with various solutions for dealing with (method) conflicts. As we have mentioned throughout this dissertation, the approach to deal with conflicts by employing disjointness and a symmetric merge operator is closely related to the trait model [Sch+03] in OOP. In the trait model, the idea is that the composition of traits should only be accepted if there are no conflicts: i.e. conflicts should result in errors. To resolve errors the trait model typically allows operations for renaming and/or removing methods from an inherited trait. We can resolve conflicts in the same way, using subtyping to hide (remove) some values in a merge. Bi and Oliveira [BO18] have shown how to build a source language with first-class

traits on top of a calculus with a symmetric merge operator. An alternative approach would be to have an asymmetric merge operator, such as the one from Dunfield [Dun14]. In such a case, conflicts would be automatically resolved by simply employing the order of composition. Such a semantics is closely related to the traditional semantics for mixins [BC90; FKF98]. However, this approach is prone to subtle errors arising from the implicit overriding of values/methods triggered by the automatic (and implicit) resolution of conflicts.

Other approaches to conflicts in class-based languages. Several other approaches to deal with conflicts have been studied in the context of OOP. C++ supports a very expressive model of multiple inheritance that accepts two classes A and B with conflicting method implementations to be inherited by a third class C. To deal with method invocations that have multiple conflicting implementations, programmers in C++ can upcast C to either A or B and then the ambiguity is removed because the static type (A or B) is used to decide which of the methods to invoke. A simple formal model of such a mechanism in C++ (and some extensions) was described by Wang et al. [Wan+18]. A closely related approach was proposed by Flatt, Krishnamurthi, and Felleisen [FKF98] in the context of mixins for a Java-like setting. The idea there is also that if a class inherits from multiple mixins with conflicting methods, then invocations of conflicted methods can still be disambiguated by first upcasting the class to the mixin with the method implementation that the programmer wishes to invoke. This approach is more flexible than the traditional biased approach of mixin inheritance that employs the order of mixins in composition to automatically override methods with conflicts. Furthermore, it avoids the issues of unintentional errors due to accidental overriding of methods. In such approaches *nominal types* play an important role, since the nominal types enable distinguishing possible overlapping (or even equal) structural types. Our work is done in the context of languages with structural types and thus it is closer to the work on record calculi. However the addition of nominal types would be interesting to investigate and could allow for improved mechanisms for conflict resolution.

Delegation-based languages. More generally, the very dynamic nature of merges is closely related to *delegation*-based OOP languages [Lie86; Kni99; FM95; US88; Ost02; BW00]. Delegation, originally introduced by Lieberman [Lie86], is a form of *dynamic inheritance*. With dynamic inheritance the inherited implementation is not statically known. This is in contrast to static inheritance (which is widely adopted by mainstream OOP languages), where the inherited class must be known statically: in mainstream OOP languages, when using `class A extends B`, `B` is some concrete (statically known) class. Delegation can itself be further classified into two difference forms: *static delegation* and *dynamic delegation* [Kni99]. In static delegation, the inherited implementation may be statically unknown, but it cannot be changed after it is "bound" to the object. In dynamic delegation, the inherited implementation is stored in a

mutable reference in the object and can be changed at any time. The CP language, for instance, adopts a static delegation model, since the self-reference is immutable.

The Self language [US88] was the first OOP language to employ delegation. JavaScript has a closely related model and is directly inspired by Self. In Self and JavaScript the idea is that an object has a property that holds a link to another object, which is called the *prototype*. The prototype object can have a prototype of its own, and so on. This offers a very dynamic model where prototype objects can be changed at runtime, by simply mutating the prototype reference. In other words, Self and JavaScript follow a dynamic delegation approach. Due to their dynamically typed nature, there is no support for statically detecting conflicts in JavaScript or Self.

Research on type systems for delegation-based languages is much less explored compared with class-based languages. Fisher and Mitchell [FM95] were among the first to study type systems for delegation. Kniesel [Kni99] further explored this space and has investigated ways to deal with issues arising from conflicts. In his model, he avoids most accidental overriding conflicts, by adopting a rule that a method `m` in one of the parents can only (implicitly) override another method implementation with the same name in another parent, if the two parents have a common parent class. This way, unrelated methods with the same name cannot be accidentally overridden. The work by Ostermann [Ost02] adapts the idea of family polymorphism to a delegation-based setting, and offers some of the advantages of nested composition that is enabled in $\lambda_i^{\dagger}$. There are a few other statically typed approaches to delegation in the context of OOP programming [BW00; SBD11]. Overall the main difference is that we are looking at foundational calculi using intersection types and a merge operator, instead of looking at the semantics of high-level OOP languages. One application of the calculi is indeed to model high-level programming abstractions for OOP languages, but that is not the focus of this work. In contrast the aforementioned related work on delegation is specifically focused on the semantics of higher-level OOP languages and language mechanisms.

8.7 Subtyping Algorithms

This section focus on subtyping algorithms with distributivity rules.

8.7.1 BCD Subtyping Algorithms

Pierce developed an algorithm for a form of subtyping close to BCD subtyping using a queue of types. His algorithmic decision procedure $le(\sigma, \overline{\tau}, \tau)$ is equivalent to the declarative judgment $\sigma \leq \overline{\tau} \to \tau$, where $\overline{\tau}$ is the queue, containing known argument types of the right-hand-side function type. When τ is a function type $\tau_1 \to \tau_2$, its argument type τ_1 is added to the queue. When σ is an intersection type $\sigma_1 \,\&\, \sigma_2$, the queue is duplicated on both sub-branches in order to reflect the distributivity rule, by distributing the argument types to both components of an

intersection type. The rules for function types, top types and intersection types then take care of argument types in the queue. Bi, Oliveira, and Schrijvers adapted Pierce's algorithm to BCD subtyping and extended it with record types without major difficulties.

The decidability of BCD subtyping is shown in several other works [KT95; RU11; Sta15] through manual proofs, and there are also proofs formalized in Coq [Lau12; Bes+16]. Bessai, Rehof, and Düdder developed a fast algorithm verified in Coq [BRD19]. Their algorithm is presented as a relational abstract machine specification, with a long proof due to the mismatch between the styles of the declarative system and algorithmic system. In contrast, our algorithm is defined in a simple relational form, keeping the modularity of existing rules, resulting in a novel, simple and concise formulation of the metatheory for the algorithm. Of course, the two lines of work have quite distinct goals: while we emphasize the modularity and simplicity of the metatheory, Bessai et al. are interested in a fast algorithm, which justifies the additional complexity in the metatheory of their approach.

Siek [Sie19], inspired by Laurent [Lau19], proposed a new subtyping system and proved the transitivity lemma directly. Siek keeps the judgment form $A <: B$ (like us), but most subtyping rules require changes, and are less modular than our rules. Siek's transitivity proof involves a size measure, while we avoid any size measure by using an alternative relation of types (proper types), which exploits the properties of our splittable type relation. Both works formalize the transitivity property, as well as soundness and completeness to BCD subtyping in proof assistants.

Algorithms for minimal relevant logic subtyping. There are some algorithms designed for the provability problem of the minimal relevant logic [GGR95; Vig00]. Such algorithms build a deduction system. But their formalization is quite different from conventional subtyping relations. Here we focus on algorithms designed for subtyping systems that are similar to the minimal relevant logic in expressiveness power. Unlike Pierce's decision procedure or modular BCD subtyping, most algorithms for minimal relevant logic subtyping require a pre-processing step on types. Typically this pre-processing step is some reduction of types into a normal form.

Muehlboeck and Tate [MT18] proposed a composable algorithmic framework called integrated subtyping. Integrated subtyping is able to generate decision procedures for various systems with union and intersection types including BCD subtyping, and the one arising from minimal relevant logic. Their strategy is to transform types on the left-hand side to a normal form, which is the disjunctive normal form in the setting without distributivity over arrows. A function called intersector is used in the conversion, and it varies according to the subtyping rules. To decide $A <: B$, they rewrite A with (a generalized version of) rule OS-distArr as much as possible. In contrast, we split B to make the types match.

Subtyping in the Delta-calculus [Sto19] extends BCD subtyping with union types, and has rules similar to minimal relevant logic. This work provides an algorithm for deciding subtyping,

which first rewrites types into some standard normal forms. After rewriting, the left-hand type is a union of intersections, while the right-hand type is an intersection of unions. The basic components in both types include type variables and arrow types rephrased into a normal form, which corresponds to our intersection-ordinary arrow types. Because all arrow types are naturally union-ordinary, these components are ordinary in both modes, and therefore the left-hand side type is an intersection of intersection-ordinary types, while the right-hand side type is a union of union-ordinary types. However, this method may not be able to scale if union arrow distributivity rules (rule S-distArrR-union and rule S-distArrL-union, discussed in Section 4.2.2) are to be added in the future. As discussed at the end of Section 4.3, the next step of the algorithm is similar to ours, except that it works on normalized types.

Frisch, Castagna, and Benzaken [FCB08] extended *semantic subtyping* to intersection and union types for the ℂDuce project. Type constructors act as their corresponding set-theoretic operators. They provided an algorithm for the induced subtyping relation. Firstly they write types into a disjunctive normal form. Then they check whether the result of the left-hand type minus the right-hand type is an empty type by enumeration. Improved versions of the algorithm were implemented and run in the current implementation of ℂDuce [Fri04]. Pearce followed their path and introduced a subtyping algorithm with a constructive proof for soundness and completeness [Pea13].

8.8 Other Problems in Calculi with Intersection and Union Types

Parallel reduction for typing unions. As discussed in Section 6.3 and Section 7.2.4, λ_i^+ and F_i^+ applies a merge of multiple functions to the input together, and reduces every component in the resulting merge simultaneously. Although their system does not have merges, Barbanera, Dezani-Ciancaglini, and De'Liguoro also employed a similar evaluation strategy called parallel beta-reduction in a calculus with intersection and union types. They consider beta-redexes rather than components in a merge. Instead of reducing the leftmost beta-redex, all occurrences of the same redex are reduced together. Likewise, their motivation is a typing rule that does not preserve types under conventional beta-reduction. This typing rule eliminates union types. If e has type $A \mid B$, the rule considers two cases: every occurrence of e has type A or has type B, and type-checks the expression twice under the two assumptions. Since beta-reduction can change the syntax form of such subterms, it has to be done in parallel. This kind of reduction strategy was proposed by Knuth, and formalized in "The lambda calculus - its syntax and semantics"[Chapter 13], and is known as Gross-Knuth reduction. As merges can serve as an elimination construct of unions, there might exists a deeper connection for this similarity in evaluation strategy.

CHAPTER 9

Conclusions and Future work

In this thesis, we designed algorithmic formulations for subtyping relations that support intersection types, union types, and distributivity rules. Most previous work has addressed similar problems using a pre-processing step to transform types into a normal form, before comparing types for subtyping. We presented a new algorithm that directly compares source types for subtyping, without a pre-processing phase. Splittable types are key to our algorithm. Starting from the subtyping of intersection types, we illustrate that splittable types can scale up to systems with union types and additional distributivity rules.

Then we showed how a type-directed operational semantics allows us to address the ambiguity problems of calculi with a merge operator utilizing the subtyping algorithms. Therefore, with a TDOS approach, we answered the question of how to give a deterministic direct operational semantics for both the general merge operator in a setting with intersection types, as well as, calculi with record concatenation and subtyping. Both of these problems are well-known to be challenging in the literature, while at the same time having important practical applications. Compared with the elaboration approach, having a direct semantics avoids the translation process and a target calculus. This simplifies both informal and formal reasoning. For instance, establishing the coherence of elaboration in the original λ_i^+ [BOS18] requires much more sophistication than obtaining the determinism theorem in λ_i^+. Furthermore, the proof method for coherence in the original λ_i^+ cannot deal with non-terminating programs, whereas dealing with recursion is straightforward in λ_i and λ_i^+. Besides recursion, our formulation of F_i^+ also enables impredicative polymorphism, validating the original trait encoding by Zhang et al. [ZSO21]. The TDOS approach exploits type annotations to guide reduction. The key component of TDOS is a casting reduction, which allows values to be further reduced depending on their type.

9.1 Future Work

9.1.1 Mutable References

All three calculi present in this thesis are purely functional and programs have no side effects. As a consequence, the CP language does not support imperative objects. Objects can be composed to create new objects, but their values cannot be changed after initialization. Although immutable data is safer and sufficient to simulate mutable data, in a practical programming language, being able to alter the content of objects is convenient to users and sometimes can improve the performance of code, as writing and reading data from memory cells is natural to the current computer infrastructure. As a step towards integration of computational effects and disjoint intersection types, one direction of our future work is to support mutable data.

Davies and Pfenning designed a calculus supporting refinement intersection types and mutable references [DP00]. Their system allows a reference term to be typed by an intersection type, for example, (Int ref) & (Pos ref) for ref 1. And being typed by an intersection type means the term can have either of the conjuncts, Int ref and Pos ref. Knowing the reference has type Int ref, we can alter its content by 0, which breaks type safety if we later dereference the term and expect it to be a Pos. To make the type system sound, Davies and Pfenning introduced a value restriction for intersection types so terms like ref 1 cannot be typed by (Int ref) & (Pos ref). They also dropped the distributivity law (rule OS-distArr) from subtyping because the value restriction can be bypassed through it: function applications can have intersection types even if the function body is not a value.

Blaauwbroek proposed a calculus with intersection and union types, and an unrestricted merge operator [Bla17]. It models computational effects via pass by sharing, where variables are mutable and there are no reference types or explicit referencing and dereferencing. For assignments, the left part must be a variable, and no subtyping is allowed for the variable. Therefore, the previous unsoundness problem is avoided in Blaauwbroek's calculus.

Note that the unsoundness problem is due to the subtyping around references, like the information hiding problem in record concatenation. It is possible to overcome it with our type system design, where each expression have a synthesized type which is its principal type.

Next we will try adding reference types to the λ_i^+ calculus, and discuss some potential challenges. Besides the intention to enhance the calculus, we also want to explore the interaction of the distributivity laws in subtyping and mutable references under the setting of disjoint intersection types.

Syntax We use A ref to denote the reference type of type A, and add three new terms constructs: ref e allocates a reference with the initial value e. !e takes a reference e and reads the corresponding value. Assignment expression $e_1 := e_2$ changes the content of reference e_1 by e_2. We adopt a call-by-value reduction so that e_2 will be assigned to e_1 after both evaluating

to values and the whole expression reduces to $\top$. r denotes the cells or store locations, which are the values of allocation expressions.

$$
\begin{array}{lll}
\text{Type} & A, B ::= \mathsf{Int} \mid \{l : A\} \mid A \to B \mid A \,\&\, B \mid \top \mid & A\,\mathsf{ref} \\
\text{Expr} & e ::= x \mid i \mid \top \mid e : A \mid \{l = e\} \mid \lambda x.\, e : A \to B \mid \mathbf{fix}\; x : A.\, e \mid e_1\; e_2 \mid e_1 \,,, e_2 & \\
 & \quad \mid \mathsf{ref}\; e \mid\; !e \mid\; e_1 := e_2 \mid\; r & \\
\text{Value} & v ::= \top \mid i \mid \lambda x.\, e : A \to B \mid v_1 \,,, v_2 \mid \{l = v\} \mid\; r &
\end{array}
$$

We also add stores and cell contexts for reduction and typing respectively.

$$
\begin{array}{ll}
\text{Stores} & \mu ::= \cdot \mid \mu, r = v \\
\text{Cell Contexts} & \Sigma ::= \cdot \mid \Sigma, r : A
\end{array}
$$

Subtyping The subtyping of reference types is invariant. There are two contexts where a reference can be used: reading and writing. Reading is covariant. Assuming $A\,\mathsf{ref}$ is expected, in this case a reference of a subtype of A is sufficient. Writing is contravariant. Since we are assigning values to the reference, the type is a constraint to the value, and a subtype is harder to satisfy than its supertype. So the conversion of types is only justified in the subtype to supertype direction.

$$
\text{OS-}\textsc{ref}\ \frac{A \leqslant B \qquad B \leqslant A}{A\,\mathsf{ref} \leqslant B\,\mathsf{ref}}
\qquad\qquad
\text{OS-}\textsc{distRef}\ \frac{}{(A\,\mathsf{ref}) \,\&\, (B\,\mathsf{ref}) \leqslant (A \,\&\, B)\,\mathsf{ref}}
$$

Introducing the distributivity rule of references over intersections allows us to read and write cells in a composed way: a merge of two reference terms can be dereferenced together and be filled in one assignment.

But this rule, unlike other distributivity rules we have seen in previous chapters, does not imply the equivalence of $(A \,\&\, B)\,\mathsf{ref}$ and $(A\,\mathsf{ref}) \,\&\, (B\,\mathsf{ref})$. Because $(A \,\&\, B)\,\mathsf{ref}$ is not a subtype of $(A\,\mathsf{ref}) \,\&\, (B\,\mathsf{ref})$. (Assuming it is, we will have $(A \,\&\, B)\,\mathsf{ref} \leqslant A\,\mathsf{ref}$.) Thus if we split the type $(A \,\&\, B)\,\mathsf{ref}$ we will lose information, which brings challenges to the subtyping algorithm design. A possible solution is to separate the two roles of the reference type: like Reynolds suggested in the design of Forsythe, the type of integer variables can be viewed as an intersection of integer expressions, which can be evaluated, and integer acceptors, which can be assigned [Rey97].

Another alternative design is to interpret assignment as an updating operation. Then the reference type constructor becomes covariant and we can split a reference type just like how we split record types.

$$\textsc{Sp-ref}\ \frac{C_1 \lhd B \rhd C_2}{C_1\ \mathsf{ref} \lhd B\,\mathsf{ref} \rhd C_2\ \mathsf{ref}}$$

In record calculi, we can update some fields of a record without touching the remaining fields. Generalized to merges, it is imaginable that if a cell already contains $1 \,„\, \mathsf{true}$, assigning 2 to it can replace the integer field and the whole value becomes $2 \,„\, \mathsf{true}$. It is a bit subtle to tell whether a function can be updated by another one but with enough type information it is decidable [XHO23]. However, this kind of design breaks the read-after-write property.

Top-like Types and Disjointness One may expect the following new rule for top-like types and a congruence rule for reference types to be disjoint.

$$\textsc{TL-ref}\ \frac{⌉A⌈}{⌉A\,\mathsf{ref}⌈} \qquad \textsc{D-refCov}\ \frac{A *_a B}{A\,\mathsf{ref} *_a B\,\mathsf{ref}}$$

But as $A\,\mathsf{ref} \leqslant \top\,\mathsf{ref}$ does not hold generally, it is unnatural to force some reference types to be top-like. So the definition of top-like types remains unchanged.

Also, only if reference type constructors are covariant, we can justify the congruence rule. Under the current invariant subtyping rule, two reference types can share non-top-like supertypes when they are equivalent. In other cases, they are disjoint.

$$\textsc{D-refInv}\ \frac{\neg(A \leqslant B \wedge B \leqslant A)}{A\,\mathsf{ref} *_a B\,\mathsf{ref}}$$

Typing The cell context Σ is added to the bidirectional typing judgment $\Gamma; \Sigma \vdash e \Leftrightarrow A$. Like the typing context, we assume it contains no identical cell names. All the four rules are in synthesis mode to maintain the typing properties in λ_i^+: all well-typed terms have a unique synthesized type.

$$\textsc{mut-Typ-cell}\ \frac{r : A \in \Sigma}{\Gamma; \Sigma \vdash r \Rightarrow A\,\mathsf{ref}} \qquad \textsc{mut-Typ-ref}\ \frac{\Gamma; \Sigma \vdash e \Rightarrow A}{\Gamma; \Sigma \vdash \mathsf{ref}\ e \Rightarrow A\,\mathsf{ref}}$$

$$\textsc{mut-Typ-set}\ \frac{\Gamma; \Sigma \vdash e_1 \Rightarrow A \qquad A \rhd B\,\mathsf{ref} \qquad \Gamma; \Sigma \vdash e_2 \Leftarrow B}{\Gamma; \Sigma \vdash e_1 := e_2 \Rightarrow \top} \qquad \textsc{mut-Typ-get}\ \frac{\Gamma; \Sigma \vdash e \Rightarrow A \qquad A \rhd B\,\mathsf{ref}}{\Gamma; \Sigma \vdash !e \Rightarrow B}$$

The typing of assignment and dereference makes use of the runtime subtyping relation, allowing A to be an intersection type. This is to cope with the distributivity rule rule OS-distRef. In case the type constructor of reference is not distributive, the rule AD-andRef should be dropped so that only terms of a reference type can be dereferenced or assigned.

$$\textsc{mut-AD-ref}\ \frac{}{A\,\mathsf{ref} \rhd A\,\mathsf{ref}} \qquad \textsc{mut-AD-andRef}\ \frac{A \rhd A'\,\mathsf{ref} \qquad B \rhd B'\,\mathsf{ref}}{A \,\&\, B \rhd (A' \,\&\, B')\,\mathsf{ref}}$$

Reduction Semantics We use store μ to keep track of the contents of cells during reduction. The following two congruence rules demonstrate how to add the store trivially for rules that do not make use of it.

$$\textsc{mut-Step-anno}\ \frac{\mu|e \hookrightarrow \mu'|e'}{\mu|e : A \hookrightarrow \mu'|e' : A} \qquad \textsc{mut-Step-refCtx}\ \frac{\mu|e \hookrightarrow \mu'|e'}{\mu|\mathsf{ref}\ e \hookrightarrow \mu'|\mathsf{ref}\ e'}$$

These conventional rules define how to read from and write to memory. An assignment expression changes the store and evaluates to the top value.

$$\textsc{mut-Step-get}\ \frac{r = v \in \mu}{\mu_1|!r \hookrightarrow \mu|v} \qquad \textsc{mut-Step-ref}\ \frac{}{\mu|\mathsf{ref}\ v \hookrightarrow \mu, r = v|r}$$

$$\textsc{mut-Step-set}\ \frac{}{\mu_1, r = v', \mu_2|r := v \hookrightarrow \mu_1, r = v, \mu_2|\top}$$

Thanks to the distributivity rule, a merge of references can act like a reference. Rule Step-getMrg has no side-effect so it is safe to keep using the same store μ. For assignments, we

want to accumulate all the changes to the store. The disjointness and consistency restriction in typing should guarantee that the evaluation order of all the assignments of the subterms does not affect the result.

$$\textsc{mut-Step-getMrg}\ \frac{\mu|!v_1 \hookrightarrow \mu|v_1' \qquad \mu|!v_2 \hookrightarrow \mu|v_2'}{\mu|!(v_1 \,,, v_2) \hookrightarrow \mu|v_1' \,,, v_2'} \qquad \textsc{mut-Step-setMrg}\ \frac{\mu_1|v_1 := v \hookrightarrow \mu_2|\top \qquad \mu_2|v_2 := v \hookrightarrow \mu_3|\top}{\mu_1|(v_1 \,,, v_2) := v \hookrightarrow \mu_3|\top}$$

In the parallel reduction rule, both subterms in the merge take the same store and reduce. An immediate problem is how to combine the two changed stores. Especially, the same cell could be written in both expressions. Could the consistency restriction on merges guarantee that such changes never conflict?

$$\textsc{mut-Step-merge}\ \frac{\mu|e_1 \hookrightarrow \mu_1|e_1' \qquad \mu|e_2 \hookrightarrow \mu_2|e_2'}{\mu|e_1 \,,, e_2 \hookrightarrow \mu_3|e_1' \,,, e_2'}$$

Casting Consider casting a cell by a reference type. Due to the invariance in subtyping, the target type has to be equivalent to the original type of the cell. Therefore it is unnecessary to coerce the stored value.

The problem is mainly around casting merges. Assuming there is a merge containing two reference cells, even though they are disjoint, we need to know the exact types they have to distinguish them. Either we look up the content of cells in the store, or we record the type information, for example, as part of the syntax of cells. The following rules present the two designs respectively. The second rule assumes that the value form of cells is annotated like $r : A$. Such annotation is compared and updated during casting.

$$\textsc{mut-CstAlt-ref}\ \frac{r = v \in \mu \qquad v : B \qquad A \leqslant B \qquad B \leqslant A}{\mu|r \hookrightarrow_{(A\,\mathrm{ref})} r} \qquad \textsc{mut-Cst-ref}\ \frac{A \leqslant B \qquad B \leqslant A}{r : B \hookrightarrow_{(A\,\mathrm{ref})} r : A}$$

9.1.2 Unannotated Lambda Functions

Type checking verifies programs and prevents runtime errors. While types make programs safer, it could be tedious for users to write type annotations explicitly. Type inference provides automatic reasoning and infers the type of some expressions to reduce the burden of users. To improve the type inference of our calculi and the CP language, one approach is via global type inference, which generates constraints from a program that contains no type annotation and

then solves the constraints. Related work on intersection types including algebraic subtyping [DM17; Par20] and type inference for disjoint intersection types [Ber19].

But here we want to consider another approach, that is to further utilize bidirectional type checking for local type inference, specifically, to accept unannotated lambda functions. In the previous calculi, we use bidirectional typechecking to obtain algorithmic type systems. Given a program, our type checking algorithm can determine whether to accept it in a decidable manner. This technique propagates information and is often used to type check applied functions that have no type annotation. But this widely followed design cannot be directly applied to our calculi. At the moment all the functions in CP are required to have annotations, which are used to cast the argument before substituting it in the function body.

$$\textsc{Typ-abs}\ \frac{\Gamma\,,\, x{:}A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.\, e{:}A \to B \Rightarrow A \to B} \qquad \textsc{Typ-var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A}$$

$$\textsc{Typ-app}\ \frac{\Gamma \vdash e_1 \Rightarrow C \qquad C \rhd A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\, e_2 \Rightarrow B}$$

In the above two rules, we can see the function body is typed with the assumption $x{:}A$, which implies x has synthesized type A. But the typing of application only checks the argument by A. (Here we only consider the case when the first premise of rule Typ-app is a direct result of rule Typ-abs). Therefore, the synthesized type of the argument may be smaller than A, and the typing derivation may not be preserved after direct substitution when it contains disjointness checking. For example, although $(\lambda x.\, x \,,,\, 1 : \mathsf{Bool} \to \mathsf{Bool} \,\&\, \mathsf{Int})\,(\mathsf{true} \,,,\, 2)$ is well-typed, changing the argument annotation for the lambda function leads to the failure of typing $\lambda x.\, x \,,,\, 1 : \mathsf{Bool} \,\&\, \mathsf{Int} \to \mathsf{Bool} \,\&\, \mathsf{Int}$.

To allow direct substitution in beta-reduction, we need to have hypotheses about checked types rather than synthesized types. So the typing context now contains two different forms of assumptions. The old assumption is reformatted as $x \Rightarrow A$. This new assumption $x \Leftarrow A$ stands for that a variable x has a checked type A, which matches with the checking premise in rule Typ-app precisely.

$$\textsc{inf-Typ-varInf}\ \frac{x \Rightarrow A \in \Gamma}{\Gamma \vdash x \Rightarrow A} \qquad \textsc{inf-Typ-varChk}\ \frac{x \Leftarrow A \in \Gamma \qquad A \leqslant B}{\Gamma \vdash x \Leftarrow B} \qquad \textsc{inf-Typ-abs2}\ \frac{\Gamma\,,\, x \Leftarrow A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.\, e \Leftarrow A \to B}$$

We keep the property that for any expression, if it can be checked against type A, it can

also be checked against any supertype of A. Previously in λ_i^+ every well-typed expression has a unique synthesized type. So having the subsumption rule is sufficient to guard this property. But now it requires more care, like adding subtyping in rule INF-TYP-VARCHK and an extra introduction rule for intersection types because rule INF-TYP-ABS2 only deals with function types.

$$\frac{\Gamma \vdash e \Rightarrow A \qquad A \leqslant B}{\Gamma \vdash e \Leftarrow B}\ \textsc{inf-Typ-sub} \qquad \frac{\Gamma \vdash e \Leftarrow A \qquad \Gamma \vdash e \Leftarrow B}{\Gamma \vdash e \Leftarrow A \,\&\, B}\ \textsc{inf-Typ-inter}$$

9.1.3 Lazy merges

In our call-by-name variant of F_i^+, although the evaluation of function arguments is delayed, subterms in merges are still reduced eagerly. For example, in

$$((1 + 1) \,,,\, (\mathsf{not}\ \mathsf{true}) \,,,\, \text{`c'}) : \mathsf{Char} \hookrightarrow (2 \,,,\, \mathsf{false} \,,,\, \text{`c'}) : \mathsf{Char} \hookrightarrow \text{`c'}$$

the integer part and the boolean part do not contribute to the final result, but they are computed for the whole merge to reach a value form. For the three values in $2 \,,,\, \mathsf{false} \,,,\, \text{`c'}$, we can tell their types directly and compare them with the cast type Char.

To obtain a truly *lazy* merge, we need to recognize the type of subterms without evaluation. A direct solution is to stop evaluating merges when all the subterms in it are wrapped by a type annotation, like

$$(1 + 1) : \mathsf{Int} \,,,\, (\mathsf{not}\ \mathsf{true}) : \mathsf{Bool} \,,,\, \text{`c'} : \mathsf{Char}$$

Recall the definition of pre-values in F_i^+:

$$\text{Pre-values} \qquad u ::= i \mid \top \mid e : A \mid u_1 \,,,\, u_2$$

As pre-values correspond to all the expressions that have a directly recognizable principal type, we can use $u_1 \,,,\, u_2$ as the value form of merges.

The other possible design is to annotate the merge operator, which introduces more annotations, but could help speed up the casting process.

$$(1 + 1)_{\mathsf{Int}} \,,,_{\mathsf{Bool}\,\&\,\mathsf{Char}}\, ((\mathsf{not}\ \mathsf{true})_{\mathsf{Bool}} \,,,_{\mathsf{Char}}\, \text{`c'})$$

Furthermore, we can separate the merging construct that stores all the expressions and the corresponding type information which serves as an index.

$$((1 + 1) \,,,\, (\mathsf{not}\ \mathsf{true}) \,,,\, \text{`c'})\ \mathsf{as}\ \mathsf{Int} \,\&\, \mathsf{Bool} \,\&\, \mathsf{Char}$$

References

[Ahm06] Amal Ahmed. "Step-indexed syntactic logical relations for recursive and quantified types". In: *European Symposium on Programming (ESOP)*. 2006.

[AOS17] João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. "Disjoint Polymorphism". In: *European Symposium on Programming (ESOP)*. 2017.

[Bak+00] Steffen van Bakel et al. *The Minimal Relevant Logic and the Call-by-Value Lambda Calculus*. Tech. rep. TR-ARP-05-2000. The Australian National University, 2000.

[Bar84] Henk Barendregt. "The lambda calculus - its syntax and semantics". In: *Studies in logic and the foundations of mathematics*. Vol. 103. 1984.

[BC90] Gilad Bracha and William R. Cook. "Mixin-based inheritance". In: *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 1990. DOI: 10.1145/97945.97982.

[BCD83] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. "A Filter Lambda Model and the Completeness of Type Assignment". In: *The Journal of Symbolic Logic* 48.4 (1983), pp. 931–940.

[BCP97] Kim B Bruce, Luca Cardelli, and Benjamin C Pierce. "Comparing object encodings". In: *International Symposium on Theoretical Aspects of Computer Software*. Springer. 1997, pp. 415–438.

[BDD95] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo De'Liguoro. "Intersection and union types: syntax and semantics". In: *Information and Computation* 119.2 (1995), pp. 202–230.

[Ber19] Birthe van den Berg. *Type Inference for Disjoint Intersection Types*. 2019.

[Bes+16] Jan Bessai et al. "Extracting a formally verified Subtyping Algorithm for Intersection Types from Ideals and Filters". In: *TYPES*. 2016.

[Bi+19] Xuan Bi et al. "Distributive Disjoint Polymorphism for Compositional Programming". In: *European Symposium on Programming (ESOP)*. 2019.

[Bla17] Lasse Blaauwbroek. “On the Interaction Between Unrestricted Union and Intersection Types and Computational Effects”. MA thesis. Eindhoven University of Technology, 2017.

[BO18] Xuan Bi and Bruno C. d. S. Oliveira. “Typed First-Class Traits”. In: *European Conference on Object-Oriented Programming (ECOOP)*. 2018.

[BOS18] Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. “The Essence of Nested Composition”. In: *European Conference on Object-Oriented Programming (ECOOP)*. 2018.

[BRD19] Jan Bessai, Jakob Rehof, and Boris Düdder. “Fast verified BCD subtyping”. In: *Models, Mindsets, Meta: The What, the How, and the Why Not?* Vol. 11200. Lecture Notes in Computer Science. Springer, Cham, 2019, pp. 356–371. DOI: 10.1007/978-3-030-22348-9_21.

[BW00] Martin Büchi and Wolfgang Weck. “Generic wrappers”. In: *European Conference on Object-Oriented Programming (ECOOP)*. 2000.

[Car94] Luca Cardelli. *Extensible Records in a Pure Calculus of Subtyping*. Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design. Foundations of Computing Series. The MIT Press, Jan. 1994, pp. 373–425.

[Cas+14] Giuseppe Castagna et al. “Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation”. In: *Symposium on Principles of Programming Languages (POPL)*. 2014. DOI: 10.1145/2535838.2535840.

[Cas+15] Giuseppe Castagna et al. “Polymorphic Functions with Set-Theoretic Types: Part 2: Local Type Inference and Type Reconstruction”. In: *Symposium on Principles of Programming Languages (POPL)*. 2015. DOI: 10.1145/2676726.2676991.

[CC99] Craig Chambers and Weimin Chen. “Efficient Multiple and Predicated Dispatching”. In: *Object-oriented Programming: Systems, Languages and Applications (OOPSLA)*. 1999. DOI: 10.1145/320384.320407.

[CD78] Mario Coppo and Mariangiola Dezani-Ciancaglini. “A new type assignment for λ-terms”. In: *Archiv für mathematische Logik und Grundlagenforschung* 19.1 (1978), pp. 139–156.

[CD80] Mario Coppo and Mariangiola Dezani-Ciancaglini. “An extension of the basic functionality theory for the λ-calculus”. In: *Notre Dame Journal of Formal Logic* 21.4 (1980), pp. 685–693. DOI: 10.1305/ndjfl/1093883253.

[CDS79] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Patrick Sallé. “Functional Characterization of Some Semantic Equalities inside Lambda-Calculus”. In: *Proceedings of the 6th Colloquium, on Automata, Languages and Programming*. Berlin, Heidelberg: Springer-Verlag, 1979, pp. 133–146. ISBN: 3540095101.

[CDV80] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. “Principal Type Schemes and Lambda-calculus Semantics”. In: (1980), pp. 535–560.

[CDV81] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. “Functional Characters of Solvable Terms”. In: *Math. Log. Q.* 27.2-6 (1981), pp. 45–58. DOI: 10.1002/malq.19810270205.

[CF58] Haskell Brooks Curry and Robert M. Feys. *Combinatory Logic Vol. 1.* Amsterdam, Netherlands: North-Holland Publishing Company, 1958.

[CG03] Adriana B. Compagnoni and Healfdene Goguen. “Typed operational semantics for higher-order subtyping”. In: *Inf. Comput.* 184.2 (2003), pp. 242–297. DOI: 10.1016/S0890-5401(03)00062-2.

[CGL95] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. “A calculus for overloaded functions with subtyping”. In: *Information and Computation* 117.1 (1995), pp. 115–135.

[Cha+05] Manuel M. T. Chakravarty et al. “Associated types with class”. In: *Symposium on Principles of Programming Languages (POPL)*. 2005, pp. 1–13. DOI: 10.1145/1040305.1040306.

[Chl10] Adam Chlipala. “Ur: statically-typed metaprogramming with type-level record computation”. In: *ACM Sigplan Notices* 45.6 (2010), pp. 122–133.

[CKJ05] Manuel M. T. Chakravarty, Gabriele Keller, and Simon L. Peyton Jones. “Associated type synonyms”. In: *International Conference on Functional Programming (ICFP)*. 2005. DOI: 10.1145/1086365.1086397.

[Cli+00] Curtis Clifton et al. “MultiJava: modular open classes and symmetric multiple dispatch for Java”. In: *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. Ed. by Mary Beth Rosson and Doug Lea. 2000. DOI: 10.1145/353171.353181.

[CM91] Luca Cardelli and John Mitchell. “Operations on Records”. In: *Mathematical Structures in Computer Science* 1 (1991), pp. 3–48.

[Coo89] William R. Cook. “A Denotational Semantics of Inheritance”. PhD thesis. Brown University, 1989.

[Coq21] The Coq Development Team. *The Coq Reference Manual, version 8.13.2*. Available electronically at https://coq.inria.fr/distrib/current/refman/. Apr. 2021.

[CP] Arthur Charguéraud and François Pottier. *TLC: a non-constructive library for Coq*. https://www.charglueraud.org/softs/tlc/.

[CW85] Luca Cardelli and Peter Wegner. “On Understanding Types, Data Abstraction, and Polymorphism”. In: *ACM Comput. Surv.* 17.4 (1985), pp. 471–522. DOI: 10.1145/6041.6042.

[CX11] Giuseppe Castagna and Zhiwu Xu. “Set-theoretic foundation of parametric polymorphism and subtyping”. In: *International Conference on Functional Programming (ICFP)*. 2011. DOI: 10.1145/2034773.2034788.

[DM17] Stephen Dolan and Alan Mycroft. “Polymorphism, subtyping, and type inference in MLsub”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 2017, pp. 60–72.

[DP00] Rowan Davies and Frank Pfenning. “Intersection types and computational effects”. In: *International Conference on Functional Programming (ICFP)*. 2000.

[DP11] Dominique Devriese and Frank Piessens. “On the bright side of type classes: instance arguments in Agda”. In: *International Conference on Functional Programming (ICFP)*. 2011. DOI: 10.1145/2034773.2034796.

[Dun14] Jana Dunfield. “Elaborating intersection and union types”. In: *Journal of Functional Programming* 24.2-3 (2014), pp. 133–165. DOI: 10.1017/S0956796813000270.

[EP00] Martin Erwig and Simon Peyton Jones. “Pattern Guards and Transformational Patterns”. In: *Haskell Workshop 2000*. Sept. 2000.

[FCB08] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. “Semantic Subtyping: Dealing Set-Theoretically with Function, Union, Intersection, and Negation Types”. In: *J. ACM* 55.4 (Sept. 2008). ISSN: 0004-5411. DOI: 10.1145/1391289.1391293.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. “Classes and Mixins”. In: *Symposium on Principles of Programming Languages (POPL)*. 1998. DOI: 10.1145/268946.268961.

[FL09] Yangyue Feng and Zhaohui Luo. “Typed Operational Semantics for Dependent Record Types”. In: *Proceedings Types for Proofs and Programs, Revised Selected Papers, TYPES*. EPTCS. 2009. DOI: 10.4204/EPTCS.53.3.

[FM95] Kathleen Fisher and John Mitchell. “A delegation-based object calculus with subtyping”. In: *Fundamentals of Computation Theory*. 1995.

[Fri04] Alain Frisch. “Théorie, conception et réalisation d’un langage de programmation adapté à XML”. PhD thesis. PhD thesis, Université Paris 7, 2004.

[GGR95] Paul Gochet, Pascal Gribomont, and Didier Rossetto. “ALGORITHMS FOR RELEVANT LOGIC: Leo Apostel in memoriam”. In: *Logique et Analyse* 38.150/152 (1995), pp. 329–346. ISSN: 00245836, 22955836.

[Gir72] Jean-Yves Girard. “Interprétation fonctionnelle et élimination des coupures de l’arithmétique d’ordre supérieur”. PhD thesis. Université Paris 7, 1972.

[Gog94] Healfdene Goguen. “A typed operational semantics for type theory”. PhD thesis. University of Edinburgh, UK, 1994.

[Gog95] Healfdene Goguen. “Typed Operational Semantics”. In: *Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA*. 1995. DOI: 10.1007/BFb0014053.

[Hal+96] Cordelia V. Hall et al. “Type Classes in Haskell”. In: *ACM Trans. Program. Lang. Syst.* 18.2 (1996), pp. 109–138. DOI: 10.1145/227699.227700.

[HO20] Xuejing Huang and Bruno C. d. S. Oliveira. “A Type-Directed Operational Semantics for a Calculus with a Merge Operator”. In: *34th European Conference on Object-Oriented Programming, ECOOP*. 2020.

[How80] William A. Howard. “The formulae-as-types notion of construction”. In: *To H.B. Curry: Essays on Combinatory Logic, λ-calculus and Formalism*. Ed. by J. Hindley and J. Seldin. Academic Press, 1980, pp. 479–490.

[HP91] Robert Harper and Benjamin Pierce. “A Record Calculus Based on Symmetric Concatenation”. In: *Symposium on Principles of Programming Languages (POPL)*. 1991.

[Kae88] Stefan Kaes. “Parametric Overloading in Polymorphic Programming Languages”. In: *European Symposium on Programming (ESOP)*. Ed. by Harald Ganzinger. 1988. DOI: 10.1007/3-540-19027-9_9.

[Kni99] Günter Kniesel. “Type-safe delegation for run-time component adaptation”. In: *European Conference on Object-Oriented Programming (ECOOP)*. 1999.

[Knu71] Donald E Knuth. “Examples of formal semantics”. In: *Symposium on Semantics of Algorithmic Languages*. Vol. 188. LNM. Springer. 1971.

[KT95] Toshihiko Kurata and Masako Takahashi. “Decidable Properties of Intersection Type Systems”. In: *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications*. TLCA ’95. Berlin, Heidelberg: Springer-Verlag, 1995, pp. 297–311. ISBN: 354059048X.

[Lau12] Olivier Laurent. “Intersection types with subtyping by means of cut elimination”. In: *Fundamenta Informaticae* 121.1-4 (2012), pp. 203–226.

[Lau19] Olivier Laurent. “Intersection Subtyping with Constructors”. In: *Proceedings Twelfth Workshop on Developments in Computational Models and Ninth Workshop on Intersection Types and Related Systems (DCM 2018 and ITRS 2018)*. Ed. by Michele Pagani and Sandra Alves. Vol. 293. Electronic Proceedings in Theoretical Computer Science. Apr. 2019, pp. 73–84.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. “Monad transformers and modular interpreters”. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on principles of programming languages*. 1995, pp. 333–343.

[Lie86] Henry Lieberman. “Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems”. In: *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 1986.

[Luo99] Zhaohui Luo. “Coercive Subtyping”. In: *J. Log. Comput.* 9.1 (1999), pp. 105–130. DOI: 10.1093/logcom/9.1.105.

[Mil78] Robin Milner. “A Theory of Type Polymorphism in Programming”. In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4.

[MT18] Fabian Muehlboeck and Ross Tate. “Empowering union and intersection types with integrated subtyping”. In: *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2018.

[Mus+08] Radu Muschevici et al. “Multiple dispatch in practice”. In: *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Ed. by Gail E. Harris. ACM, 2008, pp. 563–582. DOI: 10.1145/1449764.1449808.

[OCR20] Bruno C. d. S. Oliveira, Shaobo Cui, and Baber Rehman. “The Duality of Subtyping”. In: *European Conference on Object-Oriented Programming (ECOOP)*. 2020. DOI: 10.4230/LIPIcs.ECOOP.2020.29.

[Ode+04] Martin Odersky et al. *An overview of the Scala programming language*. Tech. rep. École Polytechnique Fédérale de Lausanne, 2004.

[OMO10] Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. “Type classes as objects and implicits”. In: *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 2010. DOI: 10.1145/1869459.1869489.

[OSA16] Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. “Disjoint intersection types”. In: *International Conference on Functional Programming (ICFP)*. 2016. DOI: 10.1145/2951913.2951945.

[Ost02] Klaus Ostermann. “Dynamically composable collaborations with delegation layers”. In: *European Conference on Object-Oriented Programming (ECOOP)*. 2002.

[OWW95] Martin Odersky, Philip Wadler, and Martin Wehr. “A Second Look at Overloading”. In: *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995*. ACM, 1995, pp. 135–146. DOI: 10.1145/224164.224195.

[Par+19] Gyunghee Park et al. “Polymorphic Symmetric Multiple Dispatch with Variance”. In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290324.

[Par20] Lionel Parreaux. “The simple essence of algebraic subtyping: principal type inference with subtyping made easy (functional pearl)”. In: *Proceedings of the ACM on Programming Languages* 4.ICFP (2020), pp. 1–28.

[Pea13] David J. Pearce. “Sound and Complete Flow Typing with Unions, Intersections and Negations”. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni. 2013. ISBN: 978-3-642-35873-9.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN: 978-0-262-16209-8.

[Pie89] Benjamin C. Pierce. *A decision procedure for the subtype relation on intersection types with bounded variables*. Tech. rep. School of Computer Science, Carnegie-Mellon University, Sept. 1989.

[Pie91] Benjamin C. Pierce. “Programming with Intersection Types and Bounded Polymorphism”. PhD thesis. Carnegie Mellon University, Dec. 1991.

[Plo73] Gordon Plotkin. *Lambda-definability and logical relations*. Edinburgh University, 1973.

[Pol97] Erik Poll. “System F with width-subtyping and record updating”. In: *International Symposium on Theoretical Aspects of Computer Software*. 1997, pp. 439–457.

[Pot00] François Pottier. “A 3-part type inference engine”. In: *European Symposium on Programming*. Springer. 2000, pp. 320–335.

[Pot80] Garrel Pottinger. “A type assignment for the strongly normalizable λ-terms”. In: (1980). Ed. by J. Hindley and J. Seldin, pp. 561–577.

[PS97] Benjamin Pierce and Martin Steffen. “Higher-order subtyping”. In: *Theoretical Computer Science* 176.1 (1997), pp. 235–282.

[PT98] Benjamin C. Pierce and David N. Turner. “Local Type Inference”. In: *Proceedings of ACM Symposium on Principles of Programming Languages*. Jan. 1998, pp. 252–265.

[PZ04] Jens Palsberg and Tian Zhao. “Type inference for record concatenation and subtyping”. In: *Information and Computation* 189.1 (2004), pp. 54–86. DOI: https://doi.org/10.1016/j.ic.2003.10.001.

[Rém89] Didier Rémy. “Type checking records and variants in a natural extension of ML”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, pp. 77–88.

[Rém95] Didier Rémy. “A case study of typechecking with constrained types: Typing record concatenation”. Presented at the workshop on Advances in types for computer science at the Newton Institute, Cambridge, UK. 1995.

[Rey74] John C Reynolds. “Towards a theory of type structure”. In: *Programming Symposium*. Springer. 1974, pp. 408–425.

[Rey88] John C Reynolds. *Preliminary design of the programming language Forsythe*. Tech. rep. CMU-CS-88-159. Carnegie Mellon University, 1988.

[Rey91] John C. Reynolds. “The Coherence of Languages with Intersection Types”. In: *Theoretical Aspects of Computer Software, International Conference TACS*. 1991. DOI: 10.1007/3-540-54415-1_70.

[Rey97] John C Reynolds. “Design of the Programming Language Forsythe”. In: *ALGOL-like languages*. Springer, 1997, pp. 173–233.

[RM72] Richard Routley and Robert K. Meyer. “The Semantics of Entailment: III”. In: *Journal of Philosophical Logic* 1.2 (1972), pp. 192–208.

[RU11] Jakob Rehof and Paweł Urzyczyn. “Finite combinatory logic with intersection types”. In: *International Conference on Typed Lambda Calculi and Applications*. 2011.

[SBD11] Ina Schaefer, Lorenzo Bettini, and Ferruccio Damiani. “Compositional Type-Checking for Delta-oriented Programming”. In: *Proceedings of the tenth international conference on Aspect-oriented software development, AOSD ’11*. 2011.

[Sch+03] Nathanael Schärli et al. “Traits: Composable units of behaviour”. In: *European Conference on Object-Oriented Programming (ECOOP)*. 2003.

[SDO22] Yaozhu Sun, Utkarsh Dhandhania, and Bruno C. d. S. Oliveira. “Compositional Embeddings of Domain-Specific Languages”. In: *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2022.

[Sew+07] Peter Sewell et al. “Ott: Effective Tool Support for the Working Semanticist”. In: *SIGPLAN Not.* 42.9 (Oct. 2007), pp. 1–12. ISSN: 0362-1340. DOI: 10.1145/1291220.1291155.

[Sie19] Jeremy G. Siek. “Transitivity of Subtyping for Intersection Types”. In: *CoRR* abs / 1906.09709 (2019). arXiv: 1906.09709.

[SO11] Tom Schrijvers and Bruno C. d. S. Oliveira. “Monads, zippers and views: virtualizing the monad stack”. In: *Proceedings of the 16th ACM SIGPLAN international conference on functional programming*. 2011, pp. 32–44.

[ST06] Jeremy G. Siek and Walid Taha. “Gradual typing for functional languages”. In: *Scheme and Functional Programming Workshop*. 2006.

[ST07] Jeremy G. Siek and Walid Taha. “Gradual Typing for Objects”. In: *European Conference on Object-Oriented Programming (ECOOP)*. 2007.

[Sta15] Rick Statman. “A Finite Model Property for Intersection Types”. In: *Electronic Proceedings in Theoretical Computer Science* 177 (2015), pp. 1–9.

[Sta85] Richard Statman. “Logical relations and the typed λ-calculus”. In: *Information and Control* 65.2-3 (1985), pp. 85–97. DOI: 10.1016/S0019-9958(85)80001-2.

[Sto19] Claude Stolze. “Combining union, intersection and dependent types in an explicitly typed lambda-calculus”. PhD thesis. Université Côte d’Azur, 2019.

[SW10] Jeremy G. Siek and Philip Wadler. “Threesomes, with and without blame”. In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. Ed. by Manuel V. Hermenegildo and Jens Palsberg. ACM, 2010, pp. 365–376. DOI: 10.1145/1706299.1706342.

[Tai67] William W. Tait. “Intensional Interpretations of Functionals of Finite Type I”. In: *J. Symb. Log.* 32.2 (1967), pp. 198–212. DOI: 10.2307/2271658.

[US88] David Ungar and Randall B Smith. “SELF: the power of simplicity (object-oriented language)”. In: *Thirty-Third IEEE Computer Society International Conference, Digest of Papers*. 1988.

[Vig00] Luca Viganò. “An O(n log n)-Space Decision Procedure for the Relevance Logic B+”. In: *Studia Logica: An International Journal for Symbolic Logic* 66.3 (2000), pp. 385–407. ISSN: 00393215, 15728730.

[Wad98] Philip Wadler. “The Expression Problem”. In: *Posted on the Java Genericity mailing list* (1998).

[Wan+18] Yanlin Wang et al. “FHJ: A Formal Model for Hierarchical Dispatching and Overriding”. In: *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*. Ed. by Todd D. Millstein. Vol. 109. LIPIcs. 2018.

[Wan89] Mitchell Wand. “Type inference for record concatenation and multiple inheritance”. In: *Proceedings. Fourth Annual Symposium on Logic in Computer Science*. 1989, pp. 92–97. DOI: 10.1109/LICS.1989.39162.

[WB89] Philip Wadler and Stephen Blott. “How to Make ad-hoc Polymorphism Less ad-hoc”. In: *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 1989, pp. 60–76. DOI: 10.1145/75277.75283.

[WBY14] Leo White, Frédéric Bour, and Jeremy Yallop. “Modular implicits”. In: *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014*. Ed. by Oleg Kiselyov and Jacques Garrigue. Vol. 198. EPTCS. 2014, pp. 22–63. DOI: 10.4204/EPTCS.198.2.

[WF09] Philip Wadler and Robert Bruce Findler. “Well-Typed Programs Can’t Be Blamed”. In: *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by Giuseppe Castagna. Vol. 5502. Lecture Notes in Computer Science. Springer, 2009, pp. 1–16. DOI: 10.1007/978-3-642-00590-9_1.

[XHO23] Han Xu, Xuejing Huang, and Bruno C. d. S. Oliveira. “Making a Type Difference: Subtraction on Intersection Types as Generalized Record Operations”. In: *Symposium on Principles of Programming Languages (POPL)*. 2023. DOI: 10.1145/3571224.

[Xie+20] Ningning Xie et al. “Row and Bounded Polymorphism via Disjoint Polymorphism”. In: *European Conference on Object-Oriented Programming (ECOOP)*. 2020.

[ZSO21] Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. “Compositional Programming”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43.3 (2021), pp. 1–61.

[Zwa95] Jan Zwanenburg. *Record Concatenation with Intersection Types*. Tech. rep. 95/34. Eindhoven University of Technology, 1995.

[Zwa97] Jan Zwanenburg. *A Type System for Record Concatenation and Subtyping*. Tech. rep. Eindhoven University of Technology, July 1997.