Compositional Programming in Action

by

Yaozhu Sun
孫耀珠

香港大學

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

21 January 2025

Abstract of thesis entitled

"**Compositional Programming in Action**"

Submitted by

Yaozhu Sun

for the degree of Doctor of Philosophy
at the University of Hong Kong
on 21 January 2025

Compositionality is an important principle in software engineering, meaning that a complex system can be built by composing simpler parts. However, it is non-trivial to achieve compositionality in practice. For example, the expression problem poses a challenge for reconciling modularity and extensibility. CP, a new statically typed programming language, naturally solves such challenges by providing language-level support for compositional programming.

This thesis studies the practical aspects of CP. We first give a crash course in CP and then showcase why compositional programming matters with two applications. The first one is about embedded domain-specific languages. We show that CP enables a new embedding technique that combines the advantages of shallow and deep embeddings and surpasses other techniques like tagless-final embeddings by supporting modular dependencies. The second application is about dynamic inheritance in object-oriented programming. CP innovatively embraces a trait model with merging and prevents implicit overriding. By comparing with type-unsafe code in TypeScript, we show that CP supports dynamic multiple inheritance and family polymorphism without sacrificing type safety.

After that, we present the design and implementation of the CP compiler. With novel language features for compositionality, the efficient compilation of CP code is non-trivial, especially when separate compilation is desired. The key idea is to compile merges to type-indexed records, which outperforms previous theoretic work using nested pairs. To ensure the type safety of dynamic inheritance, CP's type system employs coercive subtyping, leading to a significant slowdown in compiled code. We mitigate the issue by several optimizations, including eliminating coercions for equivalent types. We evaluate the impact of these optimizations using benchmarks and show that the optimized compiler

targeting JavaScript can be orders of magnitude faster than a naive compilation scheme, obtaining performance comparable to class-based JavaScript programs.

Finally, besides ubiquitous intersection types in CP, we explore the extension with union types, which provides a solid foundation for named and optional arguments. Our approach avoids a critical type-safety issue found in popular static type checkers for Python and Ruby, particularly in handling first-class named arguments in the presence of subtyping. A survey of named and optional arguments in existing languages shows that CP's design achieves a good balance of simplicity and effectiveness.

Both the compilation scheme for CP and the encoding of named and optional arguments are formalized in Coq and proven to be type-safe.

An abstract of 384 words

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.

.....

Yaozhu Sun

21 January 2025

Acknowledgments

First of all, I would like to thank my PhD supervisor, Prof. Bruno C. d. S. Oliveira, for his continuous guidance and support throughout my studies. Bruno is a very patient and responsive supervisor, who is always willing to listen to every detail of my research and provide valuable feedback in our weekly meetings. I knew very little about type systems before I started my studies at HKU. It was Bruno who introduced me to the world of intersection types and gradually guided me to the research area of my thesis. Without his help, I can hardly imagine how I could have published several papers and completed my PhD studies. Bruno is also easy-going in daily life, so I never hesitate to share happy moments or difficulties with him. I could not expect a better supervisor than Bruno.

My past supervisor during my undergraduate exchange to Tokyo Institute of Technology, Prof. Hidehiko Masuhara, was the first professor who showed me the fun of programming languages research. I am grateful to him and Dr. Matthias Springer for their very first guidance in my research career. I really miss the enjoyable one year in Tokyo. Fortunately, Prof. Taro Sekiyama recently afforded me a postdoctoral position at National Institute of Informatics, allowing me to return to Tokyo and continue my research on type systems. I cannot wait to reunite with my friends in Tokyo and enjoy the life there again. Furthermore, I would also like to thank Prof. Jonathan Aldrich from Carnegie Mellon University, who approved to be my external examiner.

The first two or three years of my PhD studies were tough because of the COVID-19 pandemic. I was stuck in Hong Kong and could not travel to any conferences abroad. I only went home once during the pandemic due to mainland China's strict quarantine policies. Nevertheless, I was lucky to have a group of friends who accompanied me and made my life in Hong Kong more colorful. These friends include my past roommates: Xu Xue, Chen Cui, Zhengjie Shu, Guangqin Song, and Yunsong Lu. Sincere thanks also go to my friends-and-collaborators: Weixin Zhang, Xuejing Huang, Andong Fan, Han Xu, Utkarsh Dhandhania, and Wenjia Ye. Other members or alumni of HKU programming languages group deserve my thanks as well: Xuan Bi, Yanlin Wang, Ningning Xie, Jinxu Zhao, Yaoda Zhou, Baber Rehman, Mingqi Xue, Shengyi Jiang, Jinhao Tan, Litao Zhou, Qianyong Wan, Bowen Su, Yicong Luo, and Ziyu Li.

I would like to extend my gratitude to Chuchu Gui from AKB48 Team SH, among other idols, who has been a constant source of positive energy for me during my hard times. I really appreciate the joy of growing up along with her.

Last but not least, my greatest thanks are due to my parents. My parents have always been supportive of my pursuit of a PhD degree. When I was hesitating between continuing my studies in academia and finding a job in industry, they encouraged me to follow my heart and worry less about money. My life and my current life are both given by my parents.

Contents

List of Figures

List of Tables

List of Theorems

Part I

Prologue

1 Introduction

This thesis focuses on the practical aspects of *compositional programming*, which is a statically typed programming paradigm that emphasizes modularity and extensibility, proposed by Weixin Zhang, Yaozhu Sun (the author), and Bruno C. d. S. Oliveira [**?**]. In this chapter, we first give a brief overview of motivations and contributions of this thesis. Then, we outline the organization of this thesis.

1.1 Motivations

Compositionality is an important principle in software engineering, meaning that a complex system can be built by composing simpler parts. However, it is non-trivial to achieve compositionality in practice. During the evolution of a software system, we often need to add new features, including extending the current system with both new data types and new operations. The *expression problem* [**?**] illustrates the difficulty of such two-dimensional extensibility in a modular way.

Embedded domain-specific languages. The difficulty of compositionality is not only a theoretic problem, but also a practical issue, for example, in the development of domain-specific languages (DSLs). A common approach to defining DSLs is via a direct embedding into a host language. This approach is used in several programming languages, such as Haskell, Scala, and Racket. In those languages, various DSLs – including pretty printers [**??**], parser combinators [**?**], and property-based testing frameworks [**?**] – are defined as embedded DSLs. There are a few techniques for such embeddings, including the well-known *shallow* and *deep* embeddings [**?**].

Unfortunately, shallow and deep embeddings come with various trade-offs in existing programming languages. Such trade-offs have been widely discussed in the literature [**???**]. On the one hand, the strengths of shallow embeddings are in providing *linguistic reuse* [**?**], exploiting meta-language optimizations, and allowing the addition of new DSL constructs easily. On the other hand, deep embeddings shine in enabling the definition of complex semantic interpretations and optimizations over the abstract syntax tree (AST) of the DSL,

and they enable adding new semantic interpretations easily. Regarding such trade-offs, **?** made the following striking comment:

> *"The holy grail of embedded language implementation is to be able to combine the advantages of shallow and deep in a single implementation."*

While progress has been made in embedded language implementation, the holy grail is still not fully achieved in existing programming languages. Owing to the trade-offs between shallow and deep embeddings, many realistic embedded DSLs end up using a mix of both approaches in practice or more advanced forms of embeddings. For instance, there have been several approaches [**???**] promoting the use of shallow embeddings as the frontend of the DSL to enable linguistic reuse, while deep embeddings are used as the backend for added flexibility in defining semantic interpretations. While such approaches manage to alleviate some of the trade-offs, they require translations between the two embeddings, a substantial amount of code, and some advanced coding techniques. Alternatively, there are more advanced embedding techniques, which are inspired by work on extensible Church encodings of algebraic data types [**???**]. Such techniques include *tagless-final embeddings* [**??**], *polymorphic embeddings* [**?**], and *object algebras* [**?**], and they are able to eliminate some of the trade-offs too. In particular, those approaches eliminate the trade-offs with respect to extensibility, facilitating both the addition of new DSL constructs and semantic interpretations. However, being quite close to shallow embeddings, those approaches lack some important capabilities, such as the ability to define complex interpretations and the use of (nested) pattern matching to express semantic interpretations and transformations easily and modularly.

Type-safe dynamic inheritance. In general-purpose programming languages, compositionality often requires mechanisms of code reuse. Object-oriented programming languages usually employs inheritance, which allows code reuse by defining new classes that inherit from existing ones. However, traditional inheritance mechanisms have limitations. For example, multiple inheritance can lead to the *diamond problem*, where a class inherits from two classes that have a common ancestor. This can cause ambiguity when resolving method calls. Traits [**?**] are a mechanism that addresses the diamond problem by detecting ambiguous compositions and requiring the programmer to resolve the conflicts explicitly. However, the detection of conflicts becomes more challenging when dealing with dynamic inheritance, where classes (or traits) can be composed at run time. This feature can typically be found in languages with first-class classes [**??**]. Most statically typed languages only provide static inheritance to achieve type safety at the cost of flexibility.

Ideally, programming languages with dynamic inheritance and first-class classes should have three properties:

1. **Flexibility.** The language should be flexible so that highly dynamic patterns of inheritance are allowed. Thus, it should be possible to support dynamic forms of mixins or traits, as well as nested classes or even virtual classes and family polymorphism.

2. **Reasonable efficiency and separate compilation.** For practical implementations, it is desirable to have a compilation model that is reasonably efficient and supports good software engineering properties, such as separate compilation.

3. **Type safety.** The language should be type-safe, so that type errors can be prevented statically.

Both JavaScript and TypeScript support the 1st and 2nd points well. With first-class classes, we can model dynamic inheritance, mixins, nested classes, and even virtual classes [**???**] and family polymorphism [**???**]. Therefore, the inheritance model provided by JavaScript and TypeScript is expressive and flexible. Furthermore, there has been a lot of work on optimizing JavaScript implementations, so JavaScript and TypeScript's inheritance and class model are reasonably efficient.

Unfortunately, for the 3rd point, TypeScript's support for type-checking first-class classes has a few type-soundness holes. Some of these holes, such as the use of bivariant subtyping, are known and documented. First-class classes bring new issues, such as a type-safety issue that we later call the *inexact superclass problem*. The inexact superclass problem can be avoided by moving into a model based on static inheritance, which is the option widely adopted by most mainstream languages. However, this trades flexibility for type safety. Ideally, we want to avoid this trade-off. Retaining flexibility and type safety while addressing the inexact superclass problem is non-trivial. In particular, it seems to be hard with the overriding semantics of JavaScript, which simply overrides properties that have the same name. Thus, to achieve the three goals together, a new compilation scheme seems desirable.

Solving the three-fold challenge: efficient compilation for CP. Previous work on compositional programming has addressed the 1st and 3rd points. However, no previous work has studied practical implementability questions, such as how to have a reasonably efficient compilation model with separate compilation. Although there is an implementation of the CP language, this implementation is based on an interpreter. Moreover, the

semantics for compositional programming languages is based on *coercive subtyping* [**?**], which raises immediate questions in terms of efficiency, since upcasts have a computational cost. A naive implementation that inserts coercions every time upcasting is needed has a prohibitive cost, which can be *orders of magnitude* slower than JavaScript programs. To fully solve the aforementioned three-fold challenge, we need to investigate how to compile compositional programming much more efficiently while also supporting separate compilation.

Compositional programming with union types. The modularity and extensibility of compositional programming are supported by intersection types from a type-theoretic perspective. It is tempting to explore how compositional programming can be extended with union types, the dual of intersection types. Our preliminary investigation shows that union types enable optional arguments in CP, while named arguments are already supported by intersection types. Named and optional arguments are prevalent features in many existing programming languages [**???**], enhancing code readability and flexibility. Despite widespread use, their formalization has not been extensively studied in the literature. Especially in languages with subtyping, such as Python and Ruby, first-class named arguments can lead to type-safety issues. It is worthwhile to conduct a formal study of type-safe foundations for named and optional arguments.

1.2 Contributions

?? mainly serves as an appetizer for the whole thesis, presenting two applications of compositional programming that illustrate the importance of the paradigm:

- **Compositional embeddings.** Compositional programming, together with the CP language, enables a new form of embedding, which we call a compositional embedding. We reveal that compositional embeddings have most of the advantages of shallow and deep embeddings. We also make a detailed comparison between compositional embeddings and various other techniques used in embedded language implementations, including hybrid, polymorphic, and tagless-final embeddings.

- **Dynamic inheritance via merging.** We identify a type-safety issue in TypeScript which we call the inexact superclass problem. We model family-polymorphic dynamic multiple inheritance as nested trait composition via merging in CP, which is free from the inexact superclass problem and semantic ambiguity by avoiding im-

plicit overriding. Disjointness plays a crucial role in ensuring type safety in the presence of dynamic inheritance.

The main body of this thesis significantly improves the implementation of the CP language in several aspects:[1]

- **In-browser interpreter and the EXT DSL.** We reimplement CP as an in-browser interpreter. On top of this, we implement a DSL for document authoring called EXT as a realistic instance of compositional embeddings. EXT is extremely flexible and customizable by users, with many features implemented in a modular way. We have built several applications with EXT, three of which are discussed in more detail in this thesis. The largest application is Minipedia, and the other two applications illustrate computational graphics like fractals and a document extension for charts.[2]
- **Compiler from CP to JavaScript.** We implement a compiler for CP that targets JavaScript, supporting modular type checking and separate compilation. We propose an efficient compilation scheme that translates merges into extensible records, where types are used as record labels to perform lookup on merges. In our concrete implementation, records are modeled as JavaScript objects, and record extension is modeled by JavaScript's support for object extension.
- **Several optimizations and an empirical evaluation.** We discuss several optimizations that we employ in the CP compiler and conduct an empirical evaluation to measure their impact. Besides, we benchmark the JavaScript code generated by our compiler together with handwritten JavaScript code.[3]
- **Extension with named and optional arguments.** We further extend CP with union types and add support for named and optional arguments. Named arguments in CP are first-class values and avoid the type-safety issue in Python and Ruby. We show a simple example of fractals that benefits from named and optional arguments.

To ensure the correctness of the CP compiler and its extension, we provide the following formalization mechanized using the Coq proof assistant:

- **Type-safety proofs for the compilation scheme.** We formalize the compilation scheme as an elaboration from the λ_{i}^{+} calculus [??] to a calculus with extensible records called λ_r and prove the type safety thereof.[4]

[1] The latest version of CP is available at https://github.com/yzyzsun/CP-next.
[2] The online editor for EXT and its applications are available at https://plground.org.
[3] The benchmark suite is available at https://github.com/yzyzsun/CP-next/tree/toplas.
[4] The Coq formalization is available at https://github.com/yzyzsun/CP-next/tree/main/theories.

- **Type-safety proofs for named and optional arguments.** We formalize the encoding of named and optional arguments as an elaboration from a minimal functional language called Uaena to a core calculus with intersection and union types called λ_{iu} [**?**] and prove the type safety thereof.[5]

1.3 Organization

Part I is the prologue. Chapter 1 motivates this thesis and outlines its organization.

Part II provides background information. Chapter 2 introduces intersection and union types, merges, disjointness, and traits. **??** gives a crash course in the CP language, which implements the compositional programming paradigm.

?? explains why compositional programming matters. We illustrate the reasons with two applications of compositional programming in this part: **??** proposes a new embedding of domain-specific languages; **??** presents a type-safe approach to dynamic inheritance via merging in CP.

?? focuses on the compilation of compositional programming. **??** describes the key ideas in our compilation scheme and its implementation in the CP compiler. **??** formalizes a simplified version of the compilation scheme along some of the key ideas. **??** explains implementation details, including the JavaScript code that is generated and some core optimizations in the CP compiler. **??** provides an empirical evaluation.

?? further extends the CP language with union types. **??** shows that this extension enables a type-safe encoding of named and optional arguments.

?? is the epilogue. **??** discusses related work, while **??** concludes this thesis and outlines future work.

[5] The Coq formalization is available at `https://github.com/yzyzsun/lambda-iu`.

Prior publications. The main content of this thesis is based on three of my papers. **??** is based on a conference paper:

- Yaozhu Sun, Utkarsh Dhandhania, and Bruno C. d. S. Oliveira. 2022. **Compositional Embeddings of Domain-Specific Languages**. In OOPSLA *(ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications)*.

?? and the whole **??** are based on an unpublished paper:

- Yaozhu Sun, Xuejing Huang, and Bruno C. d. S. Oliveira. 2025. **Type-Safe Compilation of Dynamic Inheritance via Merging**. In submission to *ACM Transactions on Programming Languages and Systems*.

?? is based on another conference paper:

- Yaozhu Sun and Bruno C. d. S. Oliveira. 2025. **Named Arguments as Intersections, Optional Arguments as Unions**. In ESOP *(European Symposium on Programming)*.

In addition, **??** in the background part also adapts some material about modular dependencies from my co-authored journal paper:

- Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2021. **Compositional Programming**. *ACM Transactions on Programming Languages and Systems*.

Part II

Background

2 Disjoint Intersection Types and First-Class Traits

This part provides a brief introduction to the background of this thesis. We first introduce intersection and union types, merging, disjointness, and traits in this chapter, laying the type-theoretic foundation for CP. Next in **??**, we will give a crash course in the CP language in a friendly manner.

2.1 Intersection Types

Generics or parametric polymorphism provides a mechanism for code reuse by assigning infinite possibilities of types to a single definition. This kind of polymorphism is *uniform* in that all possibilities behave identically despite different types instantiated.

In contrast, intersection types allows explicitly enumerating all possible types that a single definition can have. In recently developed theories [**??**], intersection types are closely related to overloading or *ad-hoc* polymorphism. For example, the type of the addition operator `(+)` for both integers and floating-point numbers can be written as:

$$(\mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}) \quad \& \quad (\mathbf{Double} \times \mathbf{Double} \to \mathbf{Double})$$

Note that the two versions of addition do not necessarily have the same behavior. Semantic subtyping [**?**] provides a set-theoretic foundation for overloaded functions typed as intersections and is employed in the Elixir type system [**?**].

However, this connection is not the original motivation for intersection types (the connection did not even hold!) if we look back at the history from 1970s [**?**]. Instead, functions with intersection types had uniform behavior and could be regarded as a finite portion of parametrically polymorphic functions. This notion is called *coherent* overloading or *finitary* polymorphism [**?**], which is a limited form of ad-hoc polymorphism. Nevertheless, intersection types are useful for some functions that cannot be typed in the simply typed λ-calculus (à la Curry), such as the Ω-combinator $(\lambda x.\, x\, x)$. The Ω-combinator can be typed

$$\textsc{T-AndIntro}\ \frac{e : A \qquad e : B}{e : A \,\&\, B} \qquad \textsc{T-AndElimL}\ \frac{e : A \,\&\, B}{e : A} \qquad \textsc{T-AndElimR}\ \frac{e : A \,\&\, B}{e : B}$$

Figure 2.1: Typing rules for intersection types.

using the two elimination rules for intersection types in Figure 2.1 (T-Abs and T-App are standard rules for functions and thus omitted):

$$\dfrac{\dfrac{\dfrac{\textsc{T-AndElimL}\ \dfrac{\cdots}{x : (A \to B) \,\&\, A \ \vdash\ x : A \to B} \qquad \textsc{T-AndElimR}\ \dfrac{\cdots}{x : (A \to B) \,\&\, A \ \vdash\ x : A}}{x : (A \to B) \,\&\, A \ \vdash\ x\, x : B}\ \textsc{T-App}}{\cdot \vdash \lambda x.\, x\, x : (A \to B) \,\&\, A \to B}\ \textsc{T-Abs}$$

Intersection types in this setting are typically used to characterize the termination properties of λ-terms [?]. Because of the correspondence with termination, type inference with intersection types is *not* decidable; neither is type inhabitation [?].

In light of the analogy with set intersection, it is tempting to add subtyping to an intersection type system. Figure 2.2 shows three subtyping rules for intersection types, which are naturally induced by set inclusion. Besides the subtyping rules, we also need to add the standard subsumption rule to the typing rules in Figure 2.1:

$$\textsc{T-Sub}\ \frac{e : A \qquad A <: B}{e : B}$$

With T-Sub, the previous elimination rules T-AndElimL and T-AndElimR can now be derived from the subtyping rules S-AndL and S-AndR.

$$\textsc{S-AndL}\ A \,\&\, B \ <:\ A \qquad \textsc{S-AndR}\ A \,\&\, B \ <:\ B \qquad \textsc{S-And}\ \frac{A <: B \qquad A <: C}{A \ <:\ B \,\&\, C}$$

Figure 2.2: Subtyping rules for intersection types.

Distributive subtyping. It is well known that λ-calculi have a close relationship with logical systems [?]: the Curry-Howard isomorphism states that propositions are types and proofs are terms. An interesting property in logic is that implication is left-distributive over conjunction, as shown in the following equivalence:

$$A \to B \wedge C \quad \Longleftrightarrow \quad (A \to B) \wedge (A \to C)$$

Since implication corresponds to function types and conjunction corresponds to intersection types in our setting, the distributive property can be interpreted as two subtyping rules:

$$A \to B \,\&\, C \quad <: \quad (A \to B) \,\&\, (A \to C) \tag{2.1}$$

$$(A \to B) \,\&\, (A \to C) \quad <: \quad A \to B \,\&\, C \tag{2.2}$$

Formula 2.1 can be derived from the aforementioned subtyping rules for intersection types:

$$\dfrac{\dfrac{A <: A \qquad \dfrac{}{B \,\&\, C \;<:\; B}\,\text{S-AndL}}{A \to B \,\&\, C \;<:\; A \to B}\,\text{S-Fun} \qquad \dfrac{\dfrac{}{B \,\&\, C \;<:\; C}\,\text{S-AndR} \qquad A <: A}{A \to B \,\&\, C \;<:\; A \to C}\,\text{S-Fun}}{A \to B \,\&\, C \;<:\; (A \to B) \,\&\, (A \to C)}\,\text{S-And}$$

However, Formula 2.2 is not derivable so far. Instead, it is a key rule of distributive subtyping, originating from the BCD type assignment system [?]. An interesting consequence of

adding Formula 2.2 is that $(A \to B) \,\&\, (C \to D)$ is a subtype of $A \,\&\, C \to B \,\&\, D$. This can be derived as follows via transitivity:

$$\dfrac{\dfrac{\dfrac{\dfrac{A \,\&\, C <: A \qquad B <: B}{A \to B \;<:\; A \,\&\, C \to B}\,\text{S-Fun}}{(A \to B) \,\&\, (C \to D) <: A \,\&\, C \to B}\,\text{S-AndL} \qquad \dfrac{\dfrac{A \,\&\, C <: C \qquad D <: D}{C \to D \;<:\; A \,\&\, C \to D}\,\text{S-Fun}}{(A \to B) \,\&\, (C \to D) <: A \,\&\, C \to D}\,\text{S-AndR}}{\text{(I)}\ (A \to B) \,\&\, (C \to D) \;<:\; (A \,\&\, C \to B) \,\&\, (A \,\&\, C \to D)}\,\text{S-And}$$

$$\dfrac{\begin{array}{l}\text{(I)}\quad (A \to B) \,\&\, (C \to D) \;<:\; (A \,\&\, C \to B) \,\&\, (A \,\&\, C \to D) \\ \text{(II)}\quad (A \,\&\, C \to B) \,\&\, (A \,\&\, C \to D) \;<:\; A \,\&\, C \to B \,\&\, D\end{array}}{(A \to B) \,\&\, (C \to D) \;<:\; A \,\&\, C \to B \,\&\, D}\,\text{S-Trans}$$

Judgment (I) is derivable as shown above, and Judgment (II) directly follows Formula 2.2. In short, distributive subtyping allows using an intersection of two function types as if it is a function type with the two parameter types intersected and the two return types intersected. This form of distributivity can be extended to record types and reveals the essence of nested composition [?]. As a result, family polymorphism [?] can be achieved, providing an elegant solution to the expression problem [??]. A detailed discussion about family polymorphism in CP can be found in ??.

Interaction with mutable references. Although distributive intersection subtyping is powerful, it poses a significant challenge for type safety when mutable references are involved. In fact, intersection subtyping alone without distributivity is already problematic. ? illustrated the problem with the following example, assuming that `Pos` (positive numbers) is a subtype of `Nat` (natural numbers):

```
let x = ref 48 : Ref Nat & Ref Pos in
let y = (x := 0) in    -- x is used as Ref Nat
let z = !x in z : Pos  -- x is used as Ref Pos
```

The code is well-typed but could cause a runtime error, because `z` is expected to be positive while it is actually a non-positive natural number `0`. The solution proposed by ? is a variant of *value restriction* [?], which requires that the introduction rule of intersection types (i.e. T-AndIntro in Figure 2.1) only applies to values. By this means, `ref 1` cannot be typed as `Ref Nat & Ref Pos` since it is not a value. However, this solution does not work well with distributive subtyping. ? showed a similar example to the previous one:

```
let x = (\() → ref 48) () : Ref Nat & Ref Pos in
let y = (x := 0) in    -- x is used as Ref Nat
let z = !x in z : Pos  -- x is used as Ref Pos
```

Since the anonymous function `(\() → ref 48)` is a value, it can be typed as an intersection type `(() → Ref Nat) & (() → Ref Pos)`. Besides, this function can be used as `() → Ref Nat & Ref Pos` via distributive subtyping. Applying it to `()` yields almost the same code as the previous example. To address this type-safety issue, **?** have to drop the distributivity rule.

Recently, **?** proposed a simpler solution to this problem based on *bidirectional typing* [**?**]. Bidirectional type system divides traditional type assignment ($e : A$) into two modes: type checking ($e \Leftarrow A$) and type inference ($e \Rightarrow a$). The key idea of **?**'s solution is that the type of the value stored in a reference can only be inferred but not checked:

$$\frac{e : A}{\mathbf{ref}\ e : \mathbf{Ref}\ A}\ \text{T-Ref-Before} \qquad \frac{e \Rightarrow A}{\mathbf{ref}\ e \Rightarrow \mathbf{Ref}\ A}\ \text{T-Ref-After}$$

With the rule T-Ref-After, `ref 48` can only have type `Ref Pos` because `48` is inferred to have type `Pos`. Moreover, `ref 48` cannot be checked against `Ref Nat` since reference types are invariant. As a result, the previous two examples cannot type-check in this bidirectional type system. A final note is that, to have a reference of type `Ref Nat` with the initial value `48`, one can write `ref (48 : Nat)` instead.

2.2 Merging and Disjointness

In last section, we have mentioned that intersection types correspond to logical conjunction when introducing distributive subtyping. However, this correspondence does not apply to the original intersection type systems, including the BCD system [**?**]. **?** gave a counterexample showing that some uninhabited intersection types are provable in most logics. The root cause is that the introduction rule (T-AndIntro in Figure 2.1) requires the two premises to have the same term e, which is not the case in logical conjunction. **?** proposed to add a merge construct $(e_1 \,,\, e_2)$, where the comma[1] $(,)$ is called the merge operator, to close the gap between intersection types and logical conjunction. Two typing

[1] We use a single comma in this thesis instead of double commas used by **?**. Although it is styled as $,$ in formulas, it is written as a plain comma `,` in CP code.

rules (T-MergeL and T-MergeR) are added by **?**, and together with T-AndIntro, we can show that $e_1 \,,\, e_2$ has type $A \,\&\, B$ if e_1 has type A and e_2 has type B:

$$\text{T-MergeL}\ \frac{e_1 : A}{e_1 \,,\, e_2 : A} \qquad \text{T-MergeR}\ \frac{e_2 : B}{e_1 \,,\, e_2 : B}$$

$$\text{T-AndIntro}\ \frac{\text{T-MergeL}\ \dfrac{e_1 : A}{e_1 \,,\, e_2 : A} \qquad \text{T-MergeR}\ \dfrac{e_2 : B}{e_1 \,,\, e_2 : B}}{e_1 \,,\, e_2 : A \,\&\, B}$$

The merge operator can be traced back to Forsythe, an ALGOL-like language designed by **?**. Merging in Forsythe is biased to avoid semantic ambiguity. For example, if f is a function, $e \,,\, f$ will override all function components in e. Consequently, merging in Forsythe cannot be used to encode function overloading, though intersections of function types are supported and a limited form of *coherent* overloading can be achieved. A significant outcome of merging is the ability to encode multi-field records, where width and permutation subtyping are naturally supported via intersection subtyping.

Contrary to Forsythe, merging in **?**'s system has the nice property of *commutativity*: the order of merging does not matter. Since there is no bias in favor of either side, semantic ambiguity becomes an important problem. For example, applying $(\lambda x.\ x + 1) \,,\, (\lambda x.\ x + 2)$ to 0 can yield either 1 or 2. To address this issue, **?** require that the two terms to be merged must have *disjoint* types. A typical typing rule for disjoint merges is as follows:

$$\text{T-Merge-Disjoint}\ \frac{e_1 \Rightarrow A \qquad e_2 \Rightarrow B \qquad A * B}{e_1 \,,\, e_2 \Rightarrow A \,\&\, B}$$

The premise $A * B$ denotes that A and B are disjoint. The aforementioned example is now *not* well-typed because $\mathbf{Int} \to \mathbf{Int}$ is *not* disjoint with $\mathbf{Int} \to \mathbf{Int}$ itself. More generally, $A \to \mathbf{Int}$ is not disjoint with $B \to \mathbf{Int}$ no matter what A and B are, because they have an overlapping part $A \,\&\, B \to \mathbf{Int}$ (or technically speaking, a common subtype). For example, consider this overloaded function:

$$(\lambda x.\ x + 1) \,,\, (\lambda x.\ \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ 0) : (\mathbf{Int} \to \mathbf{Int}) \,\&\, (\mathbf{Bool} \to \mathbf{Int})$$

Applying it to $1 \,,\, \mathbf{false}$ can yield 2 if we choose the left-hand side or 0 if we choose the right-hand side. In contrast, *overloading by return type* is allowed in **?**'s system with disjoint intersection types. For example, consider this exotically overloaded function:

$$(\lambda x.\ x + 1) \,,\, (\lambda x.\ x > 0) : (\mathbf{Int} \to \mathbf{Int}) \,\&\, (\mathbf{Int} \to \mathbf{Bool})$$

Applying it to an integer never causes ambiguity.

A more exciting application of disjoint merges is to model record concatenation. The merge $r_1 \,,\, r_2$ is concatenating two records if r_1 and r_2 are non-overlapping records. The inherent difficulty of unambiguous record concatenation with subtyping has been mentioned by **?**. **?** further provide a type-sound and coherent foundation for polymorphic record concatenation. With parametric polymorphism, disjointness can also be used as a constraint on quantified types, exhibiting similar expressiveness to bounded polymorphism [**?**]. A detailed discussion about the interaction among record concatenation, subtyping, and polymorphism can be found in **??**.

2.3 Union Types

Union types are the dual concept of intersection types. While a value of the intersection type $A \,\&\, B$ can be assigned both A and B, a value of the union type $A \mid B$ can be assigned either A or B. Although tagged unions are more common in functional languages, as seen in algebraic data types, we focus on *un*tagged unions in this thesis.

In classic λ-calculi with union types [**?**], typical typing rules for union types are as follows:

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash e : A \mid B}\ \text{T-OrIntroL} \qquad \frac{\Gamma \vdash e : B}{\Gamma \vdash e : A \mid B}\ \text{T-OrIntroR} \qquad \frac{\Gamma \vdash e' : A \mid B \qquad \Gamma, x : A \vdash e : C \qquad \Gamma, x : B \vdash e : C}{\Gamma \vdash [e'/x]e : C}\ \text{T-OrElim}$$

The union introduction rules (T-OrIntroL and T-OrIntroR) are straightforward and dual to the intersection elimination rules (T-AndElimL and T-AndElimR). In contrast, the union elimination rule (T-OrElim) is more intriguing. It basically states that, if $[e'/x]e$ can be typed as C no matter whether we assume e' evaluates to a value of type A or B, then it is safe to type $[e'/x]e$ as C when e' has type $A \mid B$. An intermediate variable x is used because a typing context can only associate a type with a variable rather than an expression.

A direct application of union types is to model nullable types [**?**], which can be represented as $A \mid \mathbf{Null}$. To make union or nullable types more useful, *occurrence typing* [**?**],

also known as *flow-sensitive typing*, is usually employed. Without occurrence typing, the following code cannot type-check:

```
double (x: Int|Null) = if x == null then 0 else 2*x;
```

The reason is that `2*x` is well-typed only if `x` has type **`Int`**, but `x` actually has type **`Int|Null`** as declared. With occurrence typing, the type of `x` is refined to **`Int`** in the **`else`**-clause (and is refined to **`Null`** in the **`then`**-clause), so the code is well-typed. **?** generalize the **`null`**-test to *type-cases* and develop a theoretic framework to refine the type of expressions occurring in type-cases. CP follows a simpler treatment proposed by **?**, where a type-case is written as **`switch`** `e0` **`as`** `x` **`case`** `A` ⇒ `e1` **`case`** `B` ⇒ `e2`. For example, the previous function can be rewritten as:

```
double (x: Int|Null) = switch x as x case Null ⇒ 0 case Int ⇒ 2*x;
```

Now union types have an explicit elimination form, T-OrElim can be replaced by T-Switch:

T-Switch
$$\frac{\Gamma \vdash e_0 : A \mid B \qquad \Gamma, x : A \vdash e_1 : C \qquad \Gamma, x : B \vdash e_2 : C}{\Gamma \vdash \textbf{switch}\ e_0\ \textbf{as}\ x\ \textbf{case}\ A \Rightarrow e_1\ \textbf{case}\ B \Rightarrow e_2 : C}$$

Distributive subtyping. Again, if we consider the logical counterparts of intersection and union types, a noticeable property is that implication is right-*anti*distributive over disjunction:

$$A \vee B \to C \quad \Longleftrightarrow \quad (A \to C) \wedge (B \to C)$$

Note that the disjunction on the left-hand side turns into a conjunction on the right-hand side. This property comes from the duality between conjunction and disjunction in De Morgan's laws. In fact, this duality has been exploited in the design of Forsythe [**?**], where $\lambda x : \textbf{Int} \mid \textbf{Double}.\ x > 0$ has type $(\textbf{Int} \to \textbf{Bool}) \,\&\, (\textbf{Double} \to \textbf{Bool})$. Although union types are not supported by Forsythe, the notation of the alternative operator ($\mid$) implies that the parameter type is essentially a union. In other words, the following subtyping relation holds (assuming $A = \textbf{Int}$, $B = \textbf{Double}$, and $C = \textbf{Bool}$):

$$A \mid B \to C \quad <: \quad (A \to C) \,\&\, (B \to C) \tag{2.3}$$

Similar to Formula 2.1, Formula 2.3 can also be derived from standard subtyping rules for intersection and union types:

$$\dfrac{\dfrac{\text{S-OrL}\ \dfrac{}{A <: A \mid B} \qquad C <: C}{\text{S-Fun}\ A \mid B \to C <: A \to C} \qquad \dfrac{\text{S-OrR}\ \dfrac{}{B <: A \mid B} \qquad C <: C}{\text{S-Fun}\ A \mid B \to C <: B \to C}}{\text{S-And}\ A \mid B \to C <: (A \to C) \,\&\, (B \to C)}$$

However, its converse (Formula 2.4) is not derivable and is added as an axiom, for example, by **?** and **?**:

$$(A \to C) \,\&\, (B \to C) \quad <: \quad A \mid B \to C \tag{2.4}$$

A final note is that intersections and unions can usually distribute over each other:

$$(A \mid B) \,\&\, C \quad <: \quad (A \,\&\, C) \mid (B \,\&\, C) \tag{2.5}$$

$$(A \mid C) \,\&\, (B \mid C) \quad <: \quad (A \,\&\, B) \mid C \tag{2.6}$$

The converses of Formula 2.5 and Formula 2.6 are derivable so they need not be axioms. The algorithm for subtyping becomes non-trivial with the four distributivity axioms. The one employed in CP is proposed by **?**, having the advantage of not requiring types to be normalized and being relatively simple to implement.

2.4 Traits and Mechanisms for Code Reuse

Traits are an overloaded term in the literature, referring to various mechanisms for code reuse. The earliest use of the term "traits" in the context of object-oriented programming can be traced back to Mesa [**?**], an ALGOL-like language developed by Xerox PARC. That traits model of subclassing is used to handle multiple inheritance in Xerox Star workstation software. Later, **?** reused the term in the context of Self, a dynamically typed prototype-based language, which is also developed by Xerox PARC. There is no class in a prototype-based language, and objects inherits directly from other objects. Traits in Self are parent objects that are shared among multiple objects, playing a similar role to traditional classes.

? reconceptualized traits as a reuse mechanism for class-based languages. In their model, traits are purely units of reuse, while classes are generators of objects. Not only can a trait *provide* methods, but it can also *require* methods that are used but not implemented.

A class can be composed of multiple traits while resolving conflicts and implementing the required methods in the meantime. An important property of traits is that the composition order is irrelevant, in contrast to order-sensitive mixins [**?**]. Besides the sum operator (+) for trait composition, exclusion (−), aliasing (⟶), and overriding (▷) are also supported in **?**'s traits model to resolve conflicts.

The term "traits" has recently been popularized by Scala and Rust, though it refers to different concepts from all the mechanisms mentioned above. Scala's traits correspond to Java 8's interfaces, which may contain abstract members and default implementations thereof. The semantics of trait composition in Scala is more like mixins despite the name. Rust's traits are similar to type classes in Haskell, providing a less ad-hoc mechanism for ad-hoc polymorphism. In short, traits in Scala and Rust are significantly different from traits in CP, which follows the principle of the traits mechanism proposed by **?**

First-class traits. CP-flavored traits are first-class [**?**] in the sense that they can be passed around like other values. For example, we can write a function that takes a trait (not an instance of a trait) and returns a new trait, or assigns a trait to a variable. First-class traits usually imply dynamic trait inheritance, where a trait can inherit from a trait expression, whose result is determined dynamically. The dynamic nature of first-class traits poses a challenge for static typing. The aforementioned notions of merging and disjointness play a crucial role in ensuring that the dynamic trait inheritance is type-safe. A type-safety issue of dynamic inheritance can be found in **??**, and CP's solution is later presented in **??**.