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

February 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
in February 2025

Compositionality is an important principle in software engineering, meaning that a complex system can be built by composing simpler parts. However, achieving compositionality in practice remains challenging, as exemplified by the expression problem, which highlights the tension between modularity and extensibility. CP, a new statically typed programming language, naturally solves such challenges through language-level support for compositional programming.

This thesis mainly studies the practical aspects of CP. We start with a crash course in CP and showcase why compositional programming matters with two applications. First, regarding object-oriented programming, CP innovates with a trait-based model with merging that prevents implicit overriding in inheritance. By comparing with type-unsafe code in TypeScript, we show that CP achieves dynamic multiple inheritance and family polymorphism without sacrificing type safety. Second, regarding domain-specific languages, CP enables a novel embedding technique that combines the advantages of shallow and deep embeddings and surpasses other techniques like tagless-final embeddings by supporting modular dependencies.

Next, we detail 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. Our key innovation is to compile merges to type-indexed records, which outperforms prior theoretic work based on nested pairs. To maintain type safety in trait inheritance, CP's type system employs coercive subtyping, which incurs a performance penalty in compiled code. We mitigate the issue with 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 on par with class-based JavaScript programs.

Finally, we extend CP with union types, complementing ubiquitous intersection types, in order to provide a solid foundation for named and optional arguments. Our approach resolves 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 detailed comparative analysis 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 372 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

February 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 accepted 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 [2021]. In this chapter, we first give an overview of the motivation and contributions of this thesis. Then, we outline the organization of this thesis.

1.1 Motivation

Modern software systems are becoming increasingly complex, with codebases spanning millions to billions of lines of code. Such complexity escalates development costs and maintenance efforts, making it harder to evolve software systems. Brooks [1987] claimed in his famous essay *No Silver Bullet*:

> *"The complexity of software is an essential property, not an accidental one."*

However, the situation is far less pessimistic nowadays. We have seen many advances in programming languages, tools, and methodologies that help manage complexity. Even back in 1990s, Cox [1995] criticized Brooks's claim and emphasized the importance of *reusable software components*. Modularity, the ability to decompose a system into independent reusable components, is a cornerstone of managing complexity in software engineering. Like LEGO bricks, these components are standardized, self-contained units that snap together via precise interfaces, requiring no modification. The principle of modularity has been widely adopted in modern software systems. For example, the microservices architecture [Richardson 2018], employed by Amazon, Netflix, Alibaba, and many other companies, organizes a system as a collection of fine-grained services.

The benefits of modularity are threefold. First, it reduces coupling between components, minimizing the impact of changes in one component on others. Second, it enhances code reuse, allowing components to be repurposed across different projects. Third, team collaboration is facilitated by modular design, as different team members can work on different components independently.

Modularity promises "plug-and-play" composition, where components can interact with each other through well-defined interfaces. However, achieving such compositionality in practice remains challenging. The fundamental challenges include:

1. **Flexibility versus safety.** Dynamic languages like JavaScript allow flexible composition but lack static guarantees, while static languages like Java pursue type safety at the cost of flexibility.

2. **Compositional conflicts.** The composition of two components may lead to ambiguity, inconsistency, or unintended behavior. These conflicts are typically due to name clashes, interface mismatches, or semantic contradictions.

3. **Modular dependencies.** A component may depend on another component, and the dependencies between components form a complex graph, which may include cycles. Thus, it is challenging to manage dependencies modularly.

4. **Expression problem** [Wadler 1998]**.** Traditional paradigms, such as object-oriented programming (OOP) and functional programming (FP), struggle to support two-dimensional extensibility – adding new data types and new operations – in a modular way.

The need to address these challenges simultaneously calls for a new programming paradigm that emphasizes compositionality. The compositional programming paradigm and the CP language [Zhang et al. 2021] are an attempt to address these challenges.

Type-safe dynamic inheritance. To better illustrate the challenges of compositionality, let us first consider their manifestation in the realm of OOP. Modularity in OOP is often achieved via classes, and classes can be extended or composed via inheritance. In other words, inheritance provides a mechanism for 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*, as shown in Figure 1.1a, where a class inherits from two classes that have a common ancestor. The method call `new D().m()` is ambiguous, as it is unclear whether it should return `"B"` or `"C"`.

Traits [Ducasse et al. 2006] 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 [Lee et al. 2015; Takikawa

```
class A { m() { return "A"; } }
class B extends A { m() { return "B"; } }
class C extends A { m() { return "C"; } }
class D extends B, C {}
new D().m() // Should it return "B" or "C"?
```

(a) Multiple inheritance.

```
function compose(X, Y) {
  return class extends X, Y {};
}
D = compose(B, C);
new D().m()
```

(b) Dynamic multiple inheritance.

Figure 1.1: The diamond problem in OOP.

et al. 2012]. Figure 1.1b shows an example of the diamond problem with dynamic multiple inheritance. Since we know nothing about the classes `X` and `Y` until the function `compose` is called, it is difficult to detect the ambiguity in advance. Similar code can cause more severe issues that break type safety. As a result, most statically typed languages only provide static inheritance to achieve type safety at the cost of flexibility.

The diamond problem manifests the challenges of compositionality in OOP, especially the aforementioned 1st (*flexibility versus safety*) and 2nd (*compositional conflicts*) points. Ideally, an OOP language should be flexible so that highly dynamic patterns of inheritance are allowed. At the same time, it should be type-safe and can identify conflicts statically, so that type errors and semantic ambiguities are prevented at compile time. It is interesting to see whether and how the CP language can address these challenges.

Embedded domain-specific languages. The difficulty of compositionality also arises in the context of domain-specific languages (DSLs). A DSL is a programming language tailored to a specific domain, such as database queries, web development, and document authoring. DSLs can be external or internal [Ghosh 2010]. External DSLs are standalone languages with their own syntax and semantics, such as SQL, HTML, and CSS. Internal DSLs are typically embedded in a host language, such as parser combinators [Leijen and Meijer 2001] and fluent interfaces [Fowler 2005a]. There are several well-known techniques to do such embeddings, including *deep* and *shallow* embeddings [Boulton et al. 1992].

If we take a minimal DSL with numbers and addition as an example, deep and shallow embeddings are implemented in Figure 1.2. In a deep embedding, the abstract syntax tree of the DSL is represented as a data type (e.g. `Exp`), and the semantics of the DSL are defined by an interpreter (e.g. `eval`). In a shallow embedding, there is no explicit data type for the DSL; the DSL is defined directly in terms of its semantics (e.g. `lit` and `add` for evaluation).

The two approaches have different trade-offs. Deep embeddings support definitions by pattern matching, including optimizations and transformations over the abstract syn-

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

eval :: Exp → Int
eval (Lit n) = n
eval (Add l r) = eval l + eval r
```

(a) Deep embedding.

```
type Exp = Int

lit :: Int → Exp
lit n = n

add :: Exp → Exp → Exp
add l r = l + r
```

(b) Shallow embedding.

Figure 1.2: Deep and shallow embeddings of a minimal DSL.

tax, and allows for multiple interpretations. Shallow embeddings have the ability to reuse host-language optimizations in the DSL and easily add new DSL constructs. Owing to these trade-offs, many embedded DSLs end up using a mix of approaches in practice, requiring a substantial amount of code, as well as some advanced coding techniques. What is worse, if one interpretation of the DSL depends on another, there is no easy way in existing embedding techniques to decouple them and maintain modularity. These challenges manifest the 3$^{\text{rd}}$ (*modular dependencies*) and 4$^{\text{th}}$ (*the expression problem*) points mentioned earlier and motivate us to explore a new embedding technique that not only combines the advantages of deep and shallow embeddings but also supports modular dependencies.

The 5$^{\text{th}}$ challenge: efficient compilation for CP. While previous work on compositional programming enables a solution to the four challenges of compositionality, a critical gap remains in terms of practical implementability – especially, how to achieve efficient compilation with separate compilation. To date, the only implementation of the CP language is based on an interpreter, which, while functional, does not provide the performance necessary for practical applications.

It is non-trivial to compile CP efficiently and separately because CP supports several novel features that are not present in mainstream programming languages. As mentioned earlier, *dynamic inheritance* and *compositional embeddings* are two key features of compositional programming. They already pose significant challenges to type-safe compilation, let alone efficiency and separate compilation. Another challenging feature is *nested composition* [Bi et al. 2018]. This feature enables a form of family polymorphism [Ernst 2001], providing an elegant solution to the expression problem [Ernst 2004], so it is crucial to modularity and extensibility in CP. Finally, though *parametric polymorphism* is a common feature in mainstream languages, it raises additional challenges in the context of compilation for CP. The reason why this feature is challenging is a bit more technical: the compilation scheme for CP relies on the concrete type of each term, while paramet-

ric polymorphism allows abstracting over types and delays type instantiation until run time. All these features are worth further investigation to understand how to compile CP efficiently.

Moreover, the semantics underlying compositional programming languages rely on *coercive subtyping* [Luo et al. 2013], which introduces inherent efficiency concerns, since upcasts lead to computational overhead. 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. Therefore, how to reduce the overhead of coercive subtyping is also an inevitable part of the 5$^{\text{th}}$ challenge.

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 [Flatt and Barzilay 2009; Garrigue 2001; Rytz and Odersky 2010], 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

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

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

- **Compositional embeddings.** Compositional programming, together with the CP language, enables a new form of embedding, which we call a compositional em-

bedding. 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.

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:

[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.

- **Type-safety proofs for the compilation scheme.** We formalize the compilation scheme as an elaboration from the λ_{i}^{+} calculus [Bi et al. 2018; Huang et al. 2021] to a calculus with extensible records called λ_r and prove the type safety thereof.[4]

- **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} [Rehman 2023] 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. Chapter 3 gives a crash course in the CP language, which implements the compositional programming paradigm.

Part III explains why compositional programming matters. We illustrate the reasons with two applications of compositional programming in this part: Chapter 4 presents a type-safe approach to dynamic inheritance via merging in CP; Chapter 5 proposes a new embedding technique for domain-specific languages.

Part IV focuses on the compilation of compositional programming. Chapter 6 describes the key ideas in our compilation scheme and its implementation in the CP compiler. Chapter 7 formalizes a simplified version of the compilation scheme along some of the key ideas. Chapter 8 explains implementation details, including the JavaScript code that is generated and some core optimizations in the CP compiler. Chapter 9 provides an empirical evaluation.

Part V further extends the CP language with union types. Chapter 10 shows that this extension enables a type-safe encoding of named and optional arguments.

Part VI is the epilogue. Chapter 11 discusses related work, while Chapter 12 concludes this thesis and outlines future work.

[4] The Coq formalization is available at `https://github.com/yzyzsun/CP-next/tree/main/theories`.
[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. Chapter 5 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)*.

Chapter 10 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)*.

Chapter 4 and the whole Part IV 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*.

In addition, Chapter 3 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 Chapter 3, 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 allow explicitly enumerating all possible types that a single definition can have. In recently developed theories [Castagna 2023; Dunfield 2014], 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 [Frisch et al. 2008] provides a set-theoretic foundation for overloaded functions typed as intersections and is employed in the Elixir type system [Castagna et al. 2023].

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 [Bono and Dezani-Ciancaglini 2020]. 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 [Pierce 1991], 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

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

Figure 2.1: Typing rules for intersection types.

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

Figure 2.2: Subtyping rules for intersection types.

Ω-combinator ($\lambda x.\ x\ x$). The Ω-combinator can be typed 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):[1]

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

Intersection types in this setting are typically used to characterize the termination properties of λ-terms [Dezani-Ciancaglini et al. 2005]. Because of the correspondence with termination, type inference with intersection types is *not* decidable; neither is type inhabitation [Urzyczyn 1999].

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:

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

[1] In this thesis, we assume that & has precedence over →. For example, $(A \to B) \,\&\, A \to B$ is parsed as $((A \to B) \,\&\, A) \to B$.

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.

Distributive subtyping. It is well known that λ-calculi have a close relationship with logical systems [Wadler 2015]: 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:

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

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 [Barendregt et al. 1983]. 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:

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

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

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 [Bi et al. 2018]. As a result, family polymorphism [Ernst 2001] can be achieved, providing an elegant solution to the expression problem [Ernst 2004; Wadler 1998]. A detailed discussion about family polymorphism in CP can be found in Section 4.3.3.

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. Davies and Pfenning [2000] 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 Davies and Pfenning is a variant of *value restriction* [Wright 1995], 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. Davies and Pfenning 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, Davies and Pfenning have to drop the distributivity rule.

Recently, Ye et al. [2024] proposed a simpler solution to this problem based on *bidirectional typing* [Dunfield and Krishnaswami 2021; Pierce and Turner 2000]. 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 Ye et al.'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. By this means, the type-safety issue is resolved without enforcing the value restriction on intersection introduction or sacrificing distributive intersection subtyping. 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 [Barendregt et al. 1983]. Hindley [1983] 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. Dunfield [2014] proposed to add a merge construct $(e_1 \,,\, e_2)$, where the comma[2] $(,)$ is called the merge operator, to close the gap between intersection types and logical conjunction. Two typing rules (T-MergeL and T-MergeR) are added by Dunfield, 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:

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

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

The merge operator can be traced back to Forsythe, an ALGOL-like language designed by Reynolds [1997]. Merging in Forsythe is biased to avoid semantic ambiguity. For example,

[2] We use a single comma in this thesis instead of double commas used by Dunfield [2014]. Although it is styled as $,$ in formulas, it is written as a plain comma `,` in CP code.

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 Dunfield'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, Oliveira et al. [2016] require that the two terms to be merged must have *disjoint* types. A typical typing rule for disjoint merges is as follows:

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

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 Oliveira et al.'s system with disjoint intersection types. For example, consider this exotically overloaded function:

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

Applying it to an integer never causes ambiguity when the expected type is known. For example, $f\ 1 + 2$ yields 4, while $\mathbf{not}\ (f\ 1)$ yields $\mathbf{false}$.

Another application of disjoint merges is to model record concatenation: if specializing the operands to records, the merge operator is essentially concatenating two records. As shown previously in T-Merge-Disjoint, the two records must have disjoint types to avoid ambiguity. For example, `{ x: Int }` and `{ x: Int }` itself are not disjoint, so the merge `r,s` is rejected in the following code:

```
let merge (r: { x: Int }) (s: { x: Int }) = r,s in  -- Type Error!
merge { x = 1 } { x = 3 }  --> { x = ? }
```

Interaction between merging and subtyping. If we further consider subtyping, the merge operator is still problematic, and disjointness alone is not sufficient to prevent ambiguity. For example, consider the following code:

```
let merge (r: { x: Int }) (s: { y: Int }) = r,s in
merge { x = 1; y = 2 } { x = 3; y = 4 }  --> { x = ?; y = ? }
```

Note that we change the type of `s` from `{ x: Int }` to `{ y: Int }`. Although the type of `s` is now disjoint with that of `r`, we can pass terms of their subtypes to `merge`. In this case, `r` has an extra field `y` and `s` has an extra `x`. Now the issue of ambiguity occurs again.

If we look at the function `merge` statically, we would expect that the field `x` is from `r` and `y` from `s`. Therefore, the most reasonable result for the code above is `{ x = 1; y = 4 }`. However, there is no naive way to implement the merge operator to achieve this result. *Neither* left-biased *nor* right-biased overriding is able to handle this case. Furthermore, selecting other fields at run time can lead to type unsoundness. For example, consider a variant of the previous merge:

```
merge { x = 1; y = "Hi" } { x = "Bye"; y = 4 }  --> { x = ?; y = ? }
```

Statically, the function is expected to compute a value of type `{ x: Int; y: Int }`, but fields of type `String` could be selected. The interaction between record concatenation and subtyping is inherently difficult and was the reason preventing Cardelli and Mitchell [1991] from choosing concatenation as the primitive operator in their calculus.

The solution found in the line of work by Oliveira et al. [2016] is to employ a *coercive* semantics of subtyping, where a subtyping relationship $A <: B$ implies a coercion function of type $A \to B$. This solution picks the field `x` from `s` and `y` from `r`, by being aware of the static types when selecting components. In the previous example, during the function application, `r` is coerced to a single-field record `{ x = 1 }`, corresponding to the parameter type `{ x: Int }`. A similar coercion is inserted for `s` as well, coercing it to `{ y = 4 }`. Then the merge operator simply concatenates `{ x = 1 }` and `{ y = 4 }`, which has no ambiguity. Thus, a combination of disjointness and a coercive approach to subtyping is able to eliminate the ambiguity introduced by an unrestricted merge operator.

Disjoint polymorphism and disjointness constraints. In the previous example, some type information about the records being merged is lost. But we may wish to preserve

other fields in the records that do not create ambiguity. This can be achieved by merging polymorphic terms, whose static types are not fully known. For example, consider a variant of the previous example:

```
let mergeSub (A <: { x: Int }) (B <: { y: Int }) (r: A) (s: B) = r,s in
mergeSub @{ x: Int; y: Int } @{ x: Int; y: Int }  -- explicit type application
         { x = 1;  y = 2  }  { x = 3;  y = 4  }
```

The code is written in pseudo-CP, where `<:` denotes the upper bound of a type parameter. In this example, `A` and `B` are declared to be subtypes of `{ x: Int }` and `{ y: Int }` respectively. Since CP does not yet support implicit polymorphism, both type parameters are instantiated explicitly on the second line. Like in Haskell, `@` is the prefix of type arguments in CP. With bounded quantification [Cardelli and Wegner 1985], we cannot guarantee the disjointness of `A` and `B`, so the issue of ambiguity comes back again. This issue can be solved by *disjoint quantification* [Alpuim et al. 2017]:

```
let mergeDis (A * { y: Int }) (B * A & { x: Int })  -- B * A and B * { x: Int }
             (r: A & { x: Int }) (s: B & { y: Int }) = r,s in
mergeDis @{y: Int} @{x: Int} { x = 1; y = 2 } { x = 3; y = 4 }  -- Type Error!
mergeDis @Top @Top { x = 1 } { y = 4 }  --> { x = 1; y = 4 }
```

Note that the type of `r` is now `A & { x: Int }` instead of `A`. This is how we usually translate subtype-bounded quantification to disjoint quantification [Xie et al. 2020]. The type parameter `A` is declared to be disjoint with `{ y: Int }` to avoid the overlap, and `B` is disjoint with `{ x: Int }` similarly. Another important constraint here is the disjointness of `A` and `B`, ensuring that other fields will never conflict as well. For example, consider a third field of type `{ z: Int }`:

```
mergeDis @{z: Int} @{z: Int} { x = 1; z = 5 } { y = 4; z = 6 }  -- Type Error!
mergeDis @Top @{z: Int} { x = 1 } { y = 4; z = 6 }  --> { x = 1; y = 4; z = 6 }
```

The first line of code fails to type-check because `A` and `B` are not disjoint and both contain a field of type `{ z: Int }`. The second line resolves the conflict, and we can access all three fields after merging. As we shall see in Section 4.3.2, the ability to express absence of certain fields is important for CP to safely handle dynamic inheritance.

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 [Barbanera et al. 1995], typical typing rules for union types are as follows:

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

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 [Nieto et al. 2020], which can be represented as $A \mid \mathbf{Null}$. To make union or nullable types more useful, *occurrence typing* [Castagna et al. 2022], 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. Castagna et al. 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 Rehman [2023] and represents a type-case 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:

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

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 [Reynolds 1997], where $\lambda x : \mathbf{Int} \mid \mathbf{Double}.\ x > 0$ has type $(\mathbf{Int} \to \mathbf{Bool}) \,\&\, (\mathbf{Double} \to \mathbf{Bool})$. 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 $A = \mathbf{Int}$, $B = \mathbf{Double}$, and $C = \mathbf{Bool}$):

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

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

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

However, its converse (Formula 2.4) is not derivable and is added as an axiom, for example, by Barbanera et al. [1995] and Rehman [2023]:

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

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 Huang and Oliveira [2021], having the advantage of not requiring types to be normalized and being relatively simple to implement.

2.4 Traits and Dependency Injection

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 [Curry et al. 1982], 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, Ungar et al. [1991] 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.

Ducasse et al. [2006] 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 [Bracha and Cook 1990]. Besides the sum operator (+) for trait composition, exclusion (−), aliasing (⟶), and overriding (▷) are also supported in Ducasse et al.'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 Ducasse et al.

Dependency injection. Dependency injection is a design pattern that addresses the challenge of modular dependencies by decoupling component creation from component usage. Instead of hard-coding dependencies within a component, dependencies are provided externally. This pattern is widely used in object-oriented programming to enhance modularity and testability. As previously mentioned, traits can *require* methods that are used but not implemented. In CP, a trait type is denoted by `Trait<Required ⇒ Provided>`, where `Required` represents the dependencies of a trait. The trait type serves as an explicit contract between the trait and its dependencies or dependents.

Before a trait is instantiated, trait composition or inheritance injects the required methods into the trait, following the principle of inversion of control in dependency injection. CP even supports dynamic composition of traits, allowing dependencies to be resolved

at run time, potentially guided by user input or configuration. This powerful feature is closely related to first-class traits, which are discussed next.

First-class traits. CP-flavored traits are first-class [Bi and Oliveira 2018] in the sense that they can be passed around like other values. For example, it is possible to write a function that takes a trait (rather than an instance of a trait) and returns a new trait, or assign a trait to a variable. First-class traits usually imply dynamic trait inheritance, where a trait may inherit from a trait expression, whose result is determined dynamically.

However, the dynamic nature of first-class traits poses a challenge for static typing. Ensuring type safety in this context relied heavily on the notions of merging and disjointness, which are essential for handling dynamic inheritance safely. Section 4.2.2 explores type-safety issues related to dynamic inheritance, and Section 4.3.2 presents CP's solution to this challenge.

3 A Crash Course in the CP Language

CP, short for *compositional programming*, is a statically typed programming language that emphasizes modularity and extensibility. It incorporates several novel features in line with recent research on compositional programming at the University of Hong Kong. These features include:

- Disjoint intersection types [Oliveira et al. 2016] and disjoint polymorphism [Alpuim et al. 2017; Xie et al. 2020];
- Distributive intersection subtyping [Bi et al. 2018, 2019] and its interaction with mutable references [Ye et al. 2024];
- Compositional interfaces, method patterns, and modular dependencies [Zhang et al. 2021];
- Iso-recursive types with nominal unfolding [Zhou et al. 2022];
- Generalized record operations with type difference [Xu et al. 2023];
- Compositional embeddings of domain-specific languages (Chapter 5);
- Dynamic inheritance (Chapter 4) of first-class traits [Bi and Oliveira 2018];
- Named and optional arguments (Chapter 10) via a blend of intersection and union types [Rehman 2023].

Before diving into the novel features, we first introduce the basics of CP.

3.1 Overview

A CP program consists of a sequence of definitions, either for types or for terms, and ends with a main expression to be evaluated. For example, the following program defines a type alias `I` and a term variable `i` of type `I`, and finally evaluates `i + 1`:

```
type I = Int;  -- type definition
i: I = 1;      -- term definition
i + 1          -- main expression
```

Every definition must end with a semicolon. Anything after `--` (or within `{- block -}`) is a comment. Note that a type must start with an uppercase letter, while a term variable must start with a lowercase letter. The type annotation (`: I`) is not required for term definitions as long as the type can be inferred.

Table 3.1: List of term operators by precedence in CP.

Operators	Operands	Arity	Description
`-`	**`Int`** or **`Double`**	Unary	Neg.
√	**`Double`**	Unary	Sqrt.
#	Arrays	Unary	Length
!	References	Unary	Deref.
! !	Arrays (lhs), **`Int`** (rhs)	Binary	Index
`* / %`	**`Int`** or **`Double`**	Binary	Mul. & Div. & Mod.
`+ -`	**`Int`** or **`Double`**	Binary	Add. & Sub.
`++`	**`String`** or Arrays	Binary	Concat.
`== != < <= > >=`	**`Int`** or **`Double`** or **`String`** or **`Bool`** or References	Binary	Comparison
&&	**`Bool`**	Binary	Conjunction
`||`	**`Bool`**	Binary	Disjunction
^	Traits (lhs)	Binary	Forwarding
,	*Disjoint*	Binary	Merging
`\-`	*Subtractable*	Binary	Diff.
`:=`	References (lhs)	Binary	Assignment
>>	Unit (lhs)	Binary	Sequencing
`if then else`	**`Bool`** (1st operand)	Ternary	Conditional

lhs: left-hand side; rhs: right-hand side.

3.2 Primitive Types

CP supports five primitive types and their literal forms:

- **`Int`**: integers, e.g. `48`, `-2`, `0o77` (octal), and `0xFF` (hexadecimal);
- **`Double`**: floating-point numbers, e.g. `48.0` and `-1e8` (scientific notation);
- **`String`**: strings, e.g. `"foo \n bar"`;
- **`Bool`**: booleans, either **`true`** or **`false`**;
- `()`: unit. The unit type has only one value, also denoted by `()`. It is used as **`null`** in Chapter 10 and as the return value of variable assignment.

Primitive operations on these types can be found in Table 3.1, which also lists other operations to be introduced later.

There are also two special types originating from subtyping: **`Top`** for the top type and **`Bot`** for the bottom type. They are the maximal and minimal types in the type lattice respectively. The top type is a supertype of any type, and the bottom type is a subtype of any type. In other words, all terms can be assigned the top type, while the bottom type is uninhabited theoretically. However, for practical convenience, **`undefined`** can be assigned any type, including the bottom type.

3.3 Compound Types

Compound types in CP include references, arrays, records, intersections, and unions.

References. A reference is similar to a pointer in an imperative language. A reference type is denoted by **`Ref`** `T`, where `T` is the type of the value stored in the reference cell. Note that reference types are *invariant*, meaning that **`Ref Int`** is neither a subtype nor a supertype of **`Ref Top`**, in order to ensure type safety.

A reference cell can be created using the **`ref`** keyword. For example, **`ref`** `48` creates a cell of type **`Ref Int`** with the initial value `48`. The value in a reference cell can be accessed using the `!` operator and updated using the `:=` operator. For example, `r := !r - 2` updates the value in `r` by subtracting 2. The assignment is an expression that returns `()`, i.e. the unit value. Finally, `e1 >> e2` denotes sequential composition, which means that `e2` is returned after `e1` (required to have the unit type) is executed.

Arrays. An array type is denoted by `[T]`, where `T` is the element type. Since arrays are *immutable* in CP, array types are treated *covariantly*. For example, `[Int]` is a subtype of `[Top]` because `Int` is a subtype of `Top`.

A mutable array can be represented as an array of references, i.e. `[Ref T]`. Note that `[Ref Int]` is not a subtype of `[Ref Top]` due to the invariance of reference types.

An array literal can be conveniently written as `[e1; e2]`. Note that we use semicolons to separate elements instead of commas because `e1,e2` is parsed as a merge. Two arrays can be concatenated using the `++` operator. The length of an array can be obtained like `#arr`, and the `i`-th element can be accessed like `arr!!i`.

Records. A record type has the form `{ l1: T1; l2: T2 }`, where `l1` and `l2` are labels, and `T1` and `T2` are field types. A record literal has the form `{ l1 = e1; l2 = e2 }`, where `e1` and `e2` are field expressions. Since there is no tuple type in CP, a pair can be represented as a record with two fields. Record fields can be accessed using the dot notation like `rcd.l1`.

An empty record `{}` is also allowed, which is inferred to have the top type `Top`. Note that `{}` is different from the unit value `()`, which has the unit type `()`.

Two records can be concatenated using the merge operator like `e1,e2`. Many other operations on records are supported via merging and type difference, as shown in Table 3.2.

Table 3.2: List of operations on records or traits in CP.

Subtraction by Label	`e \ l`	$\triangleq$	`e \\ { l: Bot }`
Subtraction by Type	`e \\ T`	$\triangleq$	`e : T0\T` if `e` has type `T0`
Subtraction by Term	`e1 \- e2`	$\triangleq$	`e1 : T1\T2` if e_i has type T_i
Field Updating	`{ e1 with l = e2 }`	$\triangleq$	`e1\l , { l = e2 }`
Label Renaming	`e [l1 <- l2]`	$\triangleq$	`e\l1 , { l2 = e.l1 }`
Leftist Merge	`e1 +, e2`	$\triangleq$	`e1 , (e2 \- e1)`
Rightist Merge	`e1 ,+ e2`	$\triangleq$	`(e1 \- e2) , e2`

Subtracting type `T1` by type `T2` is denoted by `T1\T2`. Taking the difference of record types as an example, `{ l1: Int; l2: Bool } \ { l1: Int }` is equal to `{ l2: Bool }`.

Intersections. An intersection of types `T1` and `T2` is denoted by `T1&T2`. Intersection types have an explicit introduction form in CP, which is called a merge and written as `e1,e2`, where `e1` is an expression of type `T1` and `e2` is of type `T2`. Intersection types have no explicit elimination form, but they can be implicitly eliminated via subtyping. For example, `e1,e2` can be used as if it has type `T1` or `T2`, because `T1` and `T2` are subtypes of `T1&T2`.

A multi-field record is encoded as an intersection of single-field records in CP. For example, a record type with two fields `{ l1: Int; l2: Bool }` is encoded as `{ l1: Int } & { l2: Bool }`. Similarly, at the term level, a two-field record `{ l1 = 48; l2 = true }` is encoded as `{ l1 = 48 } , { l2 = true }` using the merge operator.

Unions. A union of types `T1` and `T2` is denoted by `T1|T2`. Union types have an explicit elimination form in CP, written as `switch e0 as x case T1 ⇒ e1 case T2 ⇒ e2`. Here `x` is a variable bound to the value of `e0`, and it is refined to type `T1` in `e1` and to `T2` in `e2`. Union types have no explicit introduction form, but they can be implicitly introduced via subtyping, e.g. `48 : Int|()` and `() : Int|()`.

3.4 Functions

Functions are first-class values in CP, just like in other functional languages. A function type is written as `T1 → T2`, where `T1` is the parameter type and `T2` is the return type. A function literal is written as `\(x: T1) → e`, where `x` is the parameter variable and `e` is the body expression. CP employs curried functions to achieve multiple arguments. For example, a function of type `Int → Int → Int` can be regarded as a function that takes two integers. For example, an addition function can be defined as follows:

```
add (x: Int) (y: Int) = x + y;
-- is equivalent to:
add = \(x: Int) → \(y: Int) → x + y;
```

Higher-order functions like function composition can be defined as follows:

```
compose (f: Int → Int) (g: Int → Int) (x: Int) = f (g x);
compose (\(x: Int) → x + 1) (\(x: Int) → x * 2) 3  --> 7
```

Unfortunately, CP does not support type inference for parameter types, so they must be explicitly annotated as shown above. However, the return type can be inferred for non-recursive functions. For recursive functions, the return type must also be annotated. For example, a factorial function can be defined as follows:

```
fact (n: Int): Int = if n == 0 then 1 else n * fact (n - 1);
-- is equivalent to:
fact = fix fact: Int → Int.
        \(n: Int) → if n == 0 then 1 else n * fact (n - 1);
```

Recursive functions can be made anonymous using the **fix** operator as shown above.

If a local recursive function is desired rather than a top-level definition, the **letrec** expression can be used. For example, a tail-recursive version of factorial can be defined as follows:

```
fact = letrec fact' (acc: Int) (n: Int): Int =
         if n == 0 then acc else fact' (n * acc) (n - 1) in
       fact' 1;
```

Note that changing **letrec** into **let** will cause an error complaining that `fact'` is undefined, because **let** does not bind the variable being defined in its own definition. Nevertheless, this behavior of **let** can sometimes be useful for refining an existing definition:

```
letrec fact (acc: Int) (n: Int): Int =
  if n == 0 then acc else fact (n * acc) (n - 1) in
let fact = fact 1 in
...
```

Note that **let** `fact = fact 1` is not a recursive definition. Instead, `fact 1` is partially applying the previously defined `fact` to `1`, and the previous `fact` is shadowed by the new `fact`.

Named and optional arguments. CP natively supports named and optional arguments. The syntax of function definitions with named arguments are similar to record patterns in ML-like languages, though CP does not support pattern matching in general. Below is an exponential function with two named arguments:

```
exp { x: Int; base = 10 }: Int =
  if x == 0 then 1 else base * exp { x = x - 1; base = base };
exp { base = 2; x = 15 }  --> 2^15 = 32768
exp { x = 4 }  --> 10^4 = 10000
```

The function can be called with named arguments in any order. Here `base` is an optional argument with a default value `10`. Therefore, the second call `exp { x = 4 }` is equivalent to `exp { x = 4; base = 10 }` since `base` is absent.

3.5 Parametric Polymorphism

Unlike implicit polymorphism in ML-like languages, CP employs explicit polymorphism in System-F style. A polymorphic type is written as **`forall`** `A. T`, where `A` is the type variable (must start with an uppercase letter). A polymorphic term is written as `/\A. e`, and instantiating a polymorphic term `poly` with type **`Int`** is written as `poly` **`@Int`**.

For example, a polymorphic identity function can be defined as follows:

```
id A (x: A) = x;
-- is equivalent to:
id = /\A. \(x: A) → x;
```

Calling `id` **`@Int`** `48` will return `48` as is. A more intriguing example is a polymorphic function with *disjoint quantification*:

```
concat R (r: R) = r , { l = 48 };  -- Type Error!
concat' (R * { l: Int }) (r: R) = r , { l = 48 };  -- OK!
```

When concatenating `r` with a new field `{ l = 48 }`, we need to check whether it conflicts with any existing field in `r`. However, we do not know anything about `r` if it has a totally abstract type `R`. That is why the first definition above is rejected by the type checker, preventing potential field conflicts. The second definition type-checks because the quantification `(R * { l: Int })` indicates that the quantified type variable `R` is disjoint (will not overlap) with `{ l: Int }`, so `r` will never have a field that conflicts with `{ l = 48 }`. The disjointness constraint will be checked at the point of instantiation:

```
concat' @{ l: Int; l': Int } { l = 46; l' = 0 }  -- Type Error!
concat' @{ l': Int } { l' = 0 }  --> { l' = 0; l = 48 }
```

The first instantiation is rejected because `{ l: Int; l': Int }` is not disjoint (overlaps) with `{ l: Int }`. The second one type-checks since the two labels `l` and `l'` are distinct.

3.6 Traits

CP-flavored traits are composable reuse units and are similar to classes (not interfaces!) in traditional object-oriented languages. A trait implements an interface by providing concrete implementations or inheriting from other traits. A trait expression is anonymous

and has the basic form `trait ⇒ e`, where `e` is the body expression, usually a record literal. A simple example is shown below:

```
type Point = { x: Int; y: Int };        -- interface (a type definition)

point (x: Int) (y: Int) =  -- trait with parameters (a term definition)
  trait implements Point ⇒ { x = x; y = y };  -- has type Trait<Point>
```

Note that traits in CP are *structurally* typed, so the trait has type `Trait<Point>` even without declaring `implements Point`. The `implements`-clause is mainly a subtyping check, ensuring that the trait has implemented all the fields required by the interface, so we recommend always writing it.

Inheritance. The `Point` interface above can be extended using intersection types, and an extended trait can be defined by inheriting from `point x y`:

```
type ColorPoint = Point & { color: String };

colorPoint (x: Int) (y: Int) (c: String) =
  trait implements ColorPoint inherits point x y ⇒ { color = c };
```

Instantiating a trait is done by the `new` operator (the precedence of `new` is lower than function application):

```
blackPoint = colorPoint 1 2 "black";  -- has type Trait<ColorPoint>
new blackPoint  --> { x = 1; y = 2; color = "black" } : ColorPoint
```

Overriding. Trait inheritance in CP prevents unexpected overriding by requiring the `override` keyword. One can access fields in the super-trait using the `super` keyword. For example, below we define a point by reflecting `blackPoint`:

```
reflected = trait inherits blackPoint ⇒ {
  override x = -super.x;
  override y = -super.y;
};
```

The two fields `x` and `y` are overridden while `color` is inherited. Removing the `override` keyword will cause a type error complaining that disjointness is violated.

Multiple inheritance. Following the trait approach to multiple inheritance, there cannot be field conflicts between multiple super-traits. In other words, one must explicitly resolve the conflict, for example, by using the restriction operators, renaming, or biased merging in Table 3.2. Below we show a fine-grained resolution with restriction operators:

```
mixedPoint = trait [self] inherits blackPoint\color , reflected\\Point ⇒ {
  override x = super.x + (reflected^self).x;
};
new mixedPoint  --> { x = 0; y = 2; color = "black" }
```

Here `blackPoint\color` removes the `color` field, while `reflected\\Point` removes all fields in type `Point` (i.e. `x` and `y`). In the trait body, we intentionally override `x` to show that the excluded `x` field in `reflected` can still be accessed using the forwarding operator `^`. Basically, `t^self` is the way to access all fields in another trait `t`. We will explain what `self` means soon.

Dynamic inheritance. Traits in CP are first-class and allow dynamic inheritance, where a trait can inherit from a trait expression, whose result is determined dynamically. For example, we can decide whether the inherited trait is reflected at run time:

```
reflectedOrNot (b: Bool) =
  trait inherits if b then reflected else blackPoint ⇒ { amazing = true };
```

When combined with parametric polymorphism, dynamic inheritance may require certain disjointness constraints to ensure that there is no field conflict:

```
mixin (T * { amazing: Bool }) (base: Trait<T>) =
  trait [self: T] inherits base ⇒ { amazing = true };
```

3.7 Recursive Types

Recursive types are needed if a method refers to its enclosing trait. For example, let us consider adding a method to the `Point` type, which compares two points for equality:

```
type EqPoint = { x: Int; y: Int; eq: EqPoint → Bool };
```

Now `EqPoint` has to refer to itself in the type of the `eq` method. However, the definition above does not work because `EqPoint` is not yet defined when the type of `eq` is being checked. To solve this problem, the **`interface`** keyword can be used:

```
interface EqPoint { x: Int; y: Int; eq: EqPoint → Bool };
-- is equivalent to:
type EqPoint = fix EqPoint. { x: Int; y: Int; eq: EqPoint → Bool };
```

The form of recursive types implemented in CP is called *iso*-recursive, meaning that `EqPoint` is *not* equivalent to `{ x: Int; y: Int; eq: EqPoint → Bool }`. Instead, they are isomorphic up to folding/unfolding. For example, **`unfold`** `@EqPoint p` converts `p` from type `EqPoint` to the record type, while **`fold`** `@EqPoint { ... }` does the opposite:

```
getX (p: EqPoint) = (unfold @EqPoint p).x;
getY (p: EqPoint) = (unfold @EqPoint p).y;
eqPoint (x: Int) (y: Int) = fold @EqPoint {
  x = x;  y = y;
  eq (p: EqPoint)  =  getX p == x && getY p == y
};
```

Implicit folding/unfolding. Nevertheless, CP simplifies typical uses of recursive types by implicitly inserting **`fold`** and **`unfold`** when necessary. For example, **`fold`** is automatically inserted for a trait if the declared interface is a recursive type, and **`unfold`** is inserted for record projection. The previous example can be rewritten as follows:

```
eqPoint (x: Int) (y: Int) = trait implements EqPoint ⇒ {
  x = x;  y = y;
  eq p  =  p.x == x && p.y == y;
};
```

Note that we no longer need to specify the type of the method parameter `p` in `eq p`, because the type can be inferred from the declared interface `EqPoint`.

Due to type-theoretic difficulties, a trait implementing a recursive type cannot be inherited or merged with other traits in CP. There is ongoing research on how to address this problem.

3.8 Self-References

A more intriguing example is traits with self-type annotations:

```
type Even = { isEven: Int → Bool };
type Odd  = { isOdd:  Int → Bool };

even = trait [self: Odd] implements Even ⇒ {
  isEven n  =  if n == 0 then true else self.isOdd (n - 1);
};  -- has type Trait<Odd ⇒ Even>

odd = trait [self: Even] implements Odd ⇒ {
  isOdd n  =  if n == 0 then false else self.isEven (n - 1);
};  -- has type Trait<Even ⇒ Odd>
```

Here the annotations `[self: Odd]` and `[self: Even]` allow the trait bodies to refer to late-bound self-references by `self`, which is declared to have types `Odd` and `Even` respectively.

By this means, `even` and `odd` are not coupled with each other, allowing them to be composed with other implementations. Self-type annotations play a crucial role in modular dependencies in CP.

The trait type becomes a bit more complicated with a self-type annotation, written as `Trait<In ⇒ Out>`, where `In` is the self-type and `Out` is the inferred type for the trait body (a subtype of, but not necessarily the same as, the declared interface).

If we review the type `Trait<Point>` in last section, we will find that it is not the most precise type for the trait `point x y`. `Trait<Point>` is a shorthand for `Trait<Point ⇒ Point>` in the latest version of CP, which is a supertype of `Trait<Top ⇒ Point>`. The latter is more precise because it indicates that the self-reference is not used in the trait body (`Top` contains no information).

If we try to instantiate the traits separately by `new even` or `new odd`, a type error will be triggered saying that the required interface is not satisfied (the input type is not supertype of the output type). However, we can instantiate them together since they satisfy each other's requirement:

```
new even,odd  --> { isEven = ...; isOdd = ... }
```

Here `new` is effectively obtaining a fixpoint of the merge of the two trait, allowing late binding of the self-reference. The merge `even,odd` works because `Trait<Odd⇒Even> & Trait<Even⇒Odd>` can be used as `Trait<Odd&Even>`.

3.9 Compositional Interfaces and Method Patterns

Although CP does not support algebraic data types or pattern matching in a strict sense, similar functionalities can be achieved using compositional interfaces and method patterns.

Compositional interfaces refer to a kind of special interfaces that contain constructors. A compositional interface is parametrized by a *sort*, which serves as the return type for the constructors. For example, an abstract syntax tree for a language with integer literals and addition can be defined as follows, with a Haskell counterpart (using the notation of generalized algebraic data types) on the right for comparison:

```
type ExpSig<Exp> = {              -- data Exp where
  Lit: Int → Exp;                 --   Lit :: Int → Exp
  Add: Exp → Exp → Exp;           --   Add :: Exp → Exp → Exp
};
```

Here `Exp` is a sort, which is delimited by angle brackets to distinguish it from a normal type parameter. `ExpSig` contains two constructors: `Lit` takes an integer and `Add` takes two subexpressions, both returning the sort `Exp`. Note that constructors must start with an uppercase letter to distinguish them from normal fields.

`ExpSig` can be implemented by providing a method `eval`:

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

This syntax is similar to pattern matching in Haskell on the right and is thus called *method patterns*. The sort is instantiated with a concrete type `Eval` in `ExpSig<Eval>`, which represents a method `eval` defined for both constructors `Lit` and `Add`.

Different from algebraic data types, which are closed in Haskell, compositional interfaces and traits are open to extension in CP. As we have seen, interfaces can be extended via intersection types, and traits can be inherited or extended via merging.

Furthermore, a wildcard pattern is also supported in CP. For example, the trait below evaluates all `Lit` and `Add` expressions to `0`:

```
evalDummy = trait implements ExpSig<Eval> ⇒ { _.eval = 0 };  -- eval _ = 0
```

3.10 Modular Dependencies

A merit of compositional interfaces is that they can modularly handle dependencies between different methods for the constructors. Consider a funny pretty-printer below:

```
type Print = { print: String };
printExp = trait implements ExpSig<Eval ⇒ Print> ⇒ {
  (Lit     n).print = toString n;
  (Add e1 e2).print = if e2.eval == 0 then e1.print
                      else e1.print ++ " + " ++ e2.print;
};
```

Here `(Add e1 e2).print` depends on `e2.eval`, but there is no implementation of `eval` in `printExp`. To make such a dependency work, instead of `ExpSig<Print>`, we declare the

interface as `ExpSig<Eval ⇒ Print>`. This notation means that `e1` and `e2` (and other subexpressions in constructors, if any) has type `Eval&Print`. They only have type `Print` with `ExpSig<Print>`.

It is useful to compare this example with a simpler approach based on inheritance:

```
printChild = trait implements ExpSig<Eval&Print> inherits evalExp ⇒ {
  (Lit     n).print = toString n;
  (Add e1 e2).print = if e2.eval == 0 then e1.print
                      else e1.print ++ " + " ++ e2.print;
};
```

The body of `printChild` is exactly the same as that of `printExp`. The difference is that `printChild` implements `ExpSig<Eval&Print>` by inheriting from `evalExp`, essentially containing both `eval` and `print`. Although both approaches work, `printChild` is tightly coupled with `evalExp`, while `printExp` can be composed with any other trait that implements `eval`. By the way, the former is similar to directly writing the following code in Haskell:

```
print (Add e1 e2) = if eval e2 == 0 then ...
```

The definition of `print` is tightly coupled with a concrete function `eval`. We could use type classes to decouple dependencies in Haskell, as seen in tagless-final embeddings, but this would require planning in advance for `eval` and `print`. In CP, the natural way to implement `eval` and `print` is already modular and extensible, requiring no pre-planning.

Combination with self-references. As introduced previously, self-type annotations are also a powerful mechanism for modular dependencies. In fact, we can combine the two mechanisms together to achieve more flexible and modular designs. For example, we can define a funnier pretty-printer like this:

```
printSelf = trait implements ExpSig<Eval ⇒ Print> ⇒ {
         (Lit     n).print = toString n;
  [self]@(Add e1 e2).print = if self.eval == 0 then "0"
                             else e1.print ++ " + " ++ e2.print;
};
```

Now `(Add e1 e2).print` depends on `self.eval` rather than `e2.eval`. Therefore, we add a self-type annotation in `[self]@(Add e1 e2)`. The type of `self` is inferred to be `Eval&Print` according to the interface `ExpSig<Eval ⇒ Print>`, but it can be explicitly declared as `[self: Eval]@(Add e1 e2)` if needed.

Virtual constructors. Before concluding this section, we briefly introduce how to use the traits to construct an expression and pretty-print it. A lightweight way is: first merging

`evalExp` into `printSelf` to form a self-contained trait, and then using the **open** keyword to bring into scope both constructors in `factory`:

```
factory = new evalExp,printSelf : ExpSig<Eval&Print>;
let e1 = open factory in Add (Lit 4) (Lit 8) in e1.print  --> "4 + 8"
let e2 = open factory in Add (Lit 0) (Lit 0) in e2.print  --> "0"
```

Note that we do not need to add **new** before `Add` and `Lit`, because **new** is implicitly inserted for constructors (syntactically starting with an uppercase letter). In case **new** is not desired, `$Add` and `$Lit` can be used instead. In other words, `Lit 0` is equivalent to **new** `$Lit 0`.

Another way is to define a `repo` trait with a self-type annotation:

```
repo Exp = trait [self: ExpSig<Exp>] ⇒ open self in {
  e1 = Add (Lit 4) (Lit 8);
  e2 = Add (Lit 0) (Lit 0);
};
```

This trait allows us to use constructors *virtually* to construct expressions without committing to a concrete implementation of `ExpSig`. Then we can, for example, apply `repo` to type `Eval&Print` and merge it with `evalExp` and `printSelf`:

```
exp = new evalExp , printSelf , repo @(Eval&Print);
exp.e1.print  --> "4 + 8"
exp.e2.print  --> "0"
```

3.11 Implementations

There are two implementations of the CP language:

1. A reference interpreter bundled with the original CP paper [Zhang et al. 2021], which is no longer maintained and is archived at https://github.com/wxzh/CP.
2. A new implementation including both an interpreter and a compiler to JavaScript, which is under active development at https://github.com/yzyzsun/CP-next.

Both implementations first desugar CP source code into a core calculus called F_{i}^{+}, and then interpret or compile the F_{i}^{+} code. The main difference between the two interpreters is that the original one further elaborates F_{i}^{+} to a simpler calculus called F_{co} [Bi et al. 2019] before evaluation, while the new interpreter directly evaluates F_{i}^{+} code according to type-directed operational semantics [Fan et al. 2022; Huang et al. 2021]. The internals of the new compiler will be introduced in Part IV.

Part III

Why Compositional Programming Matters

4 Type-Safe Dynamic Inheritance

In this part, we illustrate why compositional programming matters with two case studies. The first one, which is covered in this chapter, focuses on the type safety of dynamic inheritance. While dynamic inheritance is supported by many dynamically typed languages, such as JavaScript, it poses significant challenges for statically typed languages to implement it in a type-safe manner. Section 4.1 starts with a manifesto of our proposal based on a trait model with merging. Then in Section 4.2, we show some concrete examples in TypeScript, which extends JavaScript with static typing, revealing type-safety issues. Finally in Section 4.3, we show how CP solves these issues.

4.1 First-Class Traits with Merging

Many programming language constructs are first-class. First-class functions are a key construct of functional programming. Similarly, objects are first-class in object-oriented programming (OOP). First-class constructs enable the corresponding values to be abstracted by variables, passed as arguments, or returned by functions or methods.

While classes are pervasive in most OOP languages, first-class classes are *much less* studied, and they are *rarely* supported in mainstream statically typed OOP languages. Languages such as Java, C#, and Swift, just to name a few, do not support first-class classes. In these languages, no variables can abstract over classes, and thus a class cannot pick which class to inherit from at run time. Nevertheless, some dynamically typed languages treat classes as first-class constructs and allow dynamic inheritance. Taking JavaScript[1] as an example, a base class can be passed as an argument, and the inheritance hierarchy is determined at run time, after the application of the function happens:

```
function Mixin(Base) {
  return class extends Base {
    greet() { alert('Hello, world!') }
  };
}
```

[1] Although object-orientation in JavaScript is originally prototype-based, newer standards (ECMAScript 6+) also support classes on top of prototypes.

```
class A {                    class B extends A {
  m() { return 48 }            m() { return "Hi" }
}                            }
```

Figure 4.1: JavaScript allows unconstrained overriding, whereas TypeScript's type system attempts to prevent type-unsafe overriding and statically rejects the example.

First-class classes offer powerful and flexible abstraction mechanisms for programmers. For instance, *mixins* [Bracha and Cook 1990], which are class-like abstractions that can be mixed into other classes to add new features, are encodable via first-class classes and dynamic inheritance. In our example, the `Mixin` function creates a class that inherits from `Base` and adds a `greet` method. At run time, we can apply `Mixin` to different base classes that need `greet`. Dynamic inheritance rejects the common assumption that inheritance hierarchies are fixed at compile time, providing a greater degree of flexibility compared to static inheritance. Furthermore, first-class classes provide natural support for *nested classes*: classes defined within another class, or even inside methods or functions as in the `Mixin` example. Nested classes can access definitions and methods in the surrounding lexical scope. In JavaScript, nested classes are supported via first-class classes. Some other OOP languages, such as Java, support nested classes without supporting first-class classes.

To ensure type-safe inheritance, an important concern is how to deal with *overriding* and, more generally, method conflicts. JavaScript deals with method conflicts by employing *implicit* overriding. That is, a method in a subclass will override a method in the superclass if the superclass contains a method with the *same name*. Otherwise, a new method is defined in the subclass if no method with the same name exists in the superclass. In JavaScript or other dynamically typed languages, overriding is completely unconstrained, allowing the overriding method to return a different type. An example is shown in Figure 4.1. Such overriding is not type-safe if an object of the subclass `B` is to be used in the place where the superclass `A` is expected, since the method `m` in `A` is expected to return a number instead of a string.

Since TypeScript is a superset of JavaScript, it adopts the same implicit overriding approach. However, like most statically typed OOP languages, TypeScript places restrictions on overriding to ensure type safety. In TypeScript, overriding methods must have types compatible with the overridden ones, in order to allow for the safe use of a subclass in the place of a superclass. Another possibility is to allow subclasses not to be subtypes of the superclass [Cook et al. 1990], which is sometimes seen in structurally typed OOP languages. In this case, a subclass may not always be used in place of a superclass, and a type system can prevent the use of subclasses that are not subtypes. Nevertheless, this does not

imply that overriding can be fully unconstrained, as it is still possible to have type-safety issues even when inheritance does not imply subtyping.

First-class classes and dynamic inheritance make type-safe overriding much harder. Few statically typed languages attempt to support such features, and some of the ones that do have type-unsound designs. For instance, in addition to supporting conventional static inheritance idioms, TypeScript also supports dynamic inheritance, but its type system cannot always ensure type-safe overriding. With dynamic inheritance, the exact type of the superclass is unknown statically, so it is hard to guarantee that no method is accidentally overridden with an incompatible type at run time. We will illustrate this point using examples in TypeScript later in this chapter.

Implicit overriding is not the only way to deal with method conflicts. Another possibility is to detect and prevent conflicts, *disallowing* any form of implicit overriding. For instance, the *trait* model [Ducasse et al. 2006] adopts an approach where implicit overriding is disallowed. With traits overriding is still possible, but it must be *explicitly* triggered by the programmer, instead of being implicitly done by the compiler. For instance, when composing two traits with conflicts, the composition will be *rejected*. To resolve conflicts, a programmer can, for example, decide to take one of the implementations for the method, or provide a new method implementation instead.

Yet another possibility to deal with conflicts is what we call *merging*. Merging is not a new idea and has been used to a certain degree in existing programming language designs. For instance, merging is central in programming language designs with *virtual classes* [Clarke et al. 2007; Ernst et al. 2006; Madsen and Møller-Pedersen 1989] and *family polymorphism* [Ernst 2001; Saito et al. 2008; Zhang and Myers 2017]. Virtual classes are a form of nested classes. However, the main feature of virtual classes is that, when a virtual class conflicts with another virtual class with the same name, the old class is *not overridden*. Instead, the behaviors of the two classes are *merged*: the new class will contain all the methods of the old class as well as the new methods. So, unlike overriding, merging does not replace existing behaviors. Instead, it preserves existing behaviors and adds some new ones.[2]

The idea of merging can be extended to deal with conventional methods as well. For example, in a language that adopts merging, code similar to that in Figure 4.1 can be accepted. Class `B` would contain *two* versions of the method `m`: one returning a number and the other returning a string. In other words, merging would act as a kind of *overloading*

[2]Strictly speaking, most designs with virtual classes will combine merging with overriding, in the case that the two virtual classes have conflicting methods. In our discussion, when we describe merging, we assume that the sets of methods in the two virtual classes are disjoint and, consequently, no overriding takes place.

in this case, enabling two methods with the same name but different types to coexist in the same class. Invocations of `m` could be disambiguated by the surrounding context or, if needed, by the programmer. Of course, in the merging model, the combination of two methods with the same name and *related types* would still be problematic, as it would not be clear how to choose and disambiguate between the two method implementations.

A solution to this problem is to adopt a trait-like model with merging. This is the model adopted by the CP language and is the focus of this chapter. In a trait-like model with merging, method conflicts are still forbidden, but methods with the same name and *disjoint types* (i.e. the types are unrelated) do not create a conflict. In other words, if we compose two traits, each having a method with the same name and related types, then we get an *error* due to a method conflict. However, if the methods have the same name but disjoint types, then the composition is accepted, and the resulting object will retain the two method implementations. By allowing merging in the disjoint case, we can express the forms of composition that are required for virtual classes. In such cases, virtual classes are modeled as fields, and two virtual classes with the same name but disjoint interfaces (i.e. the types of the virtual class methods) can be merged.

Such a model offers important advantages over designs that adopt implicit overriding instead. A first advantage is that we can obtain flexible, powerful, and type-safe inheritance models and avoid many of the restrictions imposed by languages with static inheritance. With merging it is possible to have a model of inheritance that allows *dynamic inheritance* and forms of *multiple inheritance* and *family polymorphism* all at once. We are aware of some designs with dynamic inheritance and first-class classes [Lee et al. 2015; Takikawa et al. 2012], but without family polymorphism. We are also aware of some designs with both multiple inheritance and family polymorphism [Aracic et al. 2006; Clarke et al. 2007; Nystrom et al. 2006], but without dynamic inheritance. Except for the CP language, the only statically typed language we know that supports all three features is `gbeta` [Ernst 2000]. However, `gbeta` cannot statically guarantee that every use of dynamic inheritance is type-safe, although Ernst [2002] proves a subset of use cases to be type-safe. Moreover, separate compilation can only be supported with an inefficient linear search through super-mixins for inherited attributes. To the best of our knowledge, CP is the only language that supports all three features in a completely type-safe manner, without compromising on modular type checking or separate compilation. We believe that this absence in the design space is because it is hard to have flexible and type-safe designs that support all these features at once.

Because our design is based on the trait model, we also inherit its advantages. In particular, since merging extends behavior rather than replace behavior, it is less prone to

problems such as the *fragile base class problem* [Mikhajlov and Sekerinski 1998]. As we shall see in next section, in the presence of dynamic inheritance, designs based on implicit overriding, such as TypeScript, exacerbate the fragile base class problem: not only can overriding break invariants of the superclass, but it can also break type safety! Designs based on a trait-model with merging preserve the behavior of inherited classes and avoid the issues due to (implicit) overriding.

4.2 Dynamic Inheritance, Overriding, and Type Safety

There are only a few statically typed languages that support first-class classes and dynamic inheritance, among which are `gbeta` [Ernst 2000], TypeScript [Microsoft 2012], Typed Racket [Takikawa et al. 2012], and Wyvern [Lee et al. 2015]. Here we take the most popular one, TypeScript, as the main example to illustrate the challenges of type-safe dynamic inheritance and reveal significant limitations of TypeScript's type system. We will also briefly mention JavaScript and Java to further illustrate concepts related to first-class classes and dynamic inheritance. Discussions about `gbeta`, Typed Racket, and Wyvern can be found in Section 11.2.

4.2.1 Class Inheritance and Structural Typing

Classes are the reusable building blocks in most OOP languages. They are reused by inheritance, a mechanism to create a new class (called a *subclass*) based on an existing class (called a *superclass*). Inheritance enables the reuse of implementations of methods or properties that are already provided in the superclass. Furthermore, it is possible to override methods of the superclass with new implementations that are more suitable for the subclass.

To make sure that instances of the subclass can be used in any context where its superclass is expected, there is usually a requirement that the subclass has a *subtype* with respect to its superclass. While inheritance is related to implementations, subtyping is a relation between types. In many programming languages, class definitions `class B extends A {...}` introduce both relations between `A` and `B`: class `B` inherits the implementation from class `A`, and it also introduces a subtyping relation between the types of the two classes. For example, type `B` is required to be a subtype of type `A` in TypeScript:

```
class A {}
```

```
class B extends A {
  m(): number { return 0; }
}
```

Owing to the same reason, when a method in the superclass is overridden, the new method in the subclass must have a subtype. For example, the method `f` in `C` is overridden by the one in `D` below:

```
class C {
  f(x: B): number { return x.m(); }
}
```

```
class D extends C {
  f(x: A): number { return 48; }
}
```

The overriding is type-safe because the latter method has a subtype of the former's. According to the standard subtyping rule for functions, the parameter type is *contravariant*, and the return type is *covariant*. Since `A` is a supertype of `B`, the function type `(x: A) ⇒ number` is a subtype of `(x: B) ⇒ number`.

Bivariant subtyping in TypeScript. Perhaps surprisingly, the following code also type-checks:

```
class E {
  f(x: A): number { return 48; }
}
```

```
class F extends E {
  f(x: B): number { return x.m(); }
}
```

The parameter type of method `f` becomes a subtype of that in the superclass, but it still passes the subtyping check. In other words, TypeScript does *not* follow the standard type-theoretic treatment of function subtyping. Instead, TypeScript allows *bivariant* subtyping for method parameters, where the type of method parameters being overridden can either be a subtype or a supertype of the corresponding type in the superclass method. Bivariant subtyping is a well-known source of type unsoundness. It would lead to a runtime error that could have been prevented statically:

```
const o: E = new F;
o.f(new A)  // Runtime Error!
```

TypeScript developers are aware of this, but they motivate the use of bivariant subtyping by large numbers of use cases in the libraries that require this functionality.[3] In essence, TypeScript trades type soundness for flexibility and thus supports a more flexible model of inheritance in some cases.

A type-safe alternative model for structural typing. TypeScript's class model adopts the approach that subclasses always generate subtypes of the superclass. Thus, it retains the familiar model that is common in mainstream nominally typed languages like Java, C#, or Scala, which can be seen as an advantage for attracting programmers from those languages.

[3] https://www.typescriptlang.org/tsconfig#strictFunctionTypes

However, unlike these mainstream programming languages, TypeScript is *structurally* typed. With structural typing, there is a well-known alternative that would enable the overriding in class `F` to be type-safe. As observed by Cook et al. [1990], inheritance is not subtyping. In the context of a language of classes, this means that sometimes subclasses may not be subtypes of the superclass. In particular, the parameter of a *binary method* [Bruce et al. 1995] is supposed to be an object of the class being defined. In this case, the subclass will covariantly refine the type of the method parameters, and thus detaching inheritance from subtyping can be helpful. Since there is no subtyping relation between subclasses and superclasses in an inheritance-is-not-subtyping approach, the standard contravariant subtyping rule, instead of bivariant subtyping, can be used for function parameters, thus preventing type-safety issues that arise from bivariant subtyping.

If TypeScript adopted an inheritance-is-not-subtyping approach instead, then the code for `F` could still type-check, but the subclass `F` would not be a subtype of its superclass `E`. Therefore, the runtime error would be prevented because the line would be rejected with a type error:

```
// Invalid upcast in an inheritance-is-not-subtyping approach!
const o: E = new F;
```

While type-safe, the inheritance-is-not-subtyping approach departs from the conventional model adopted by mainstream languages. So, it could be harder for programmers (especially those used to other mainstream OOP languages) to understand that sometimes subclasses cannot be subtypes. This is perhaps a reason (among others) for TypeScript not adopting this approach. Nevertheless, we adopt a model based on inheritance-is-not-subtyping because it allows a more *flexible* but still *type-safe* form of inheritance.

4.2.2 Unsafe Overriding with Dynamic Inheritance

TypeScript differs from other mainstream OOP languages in that it also supports dynamic inheritance. Dynamic inheritance brings new type-safety considerations with respect to overriding. These issues are not due to the use of bivariant subtyping and appear to be unknown or undocumented by the TypeScript implementers. Nevertheless, in order to obtain a type-safe design, we must be able to address the type-safety issues that may arise from dynamic inheritance. Thus, the purpose of this subsection is to identify such a problem in TypeScript. We call this problem the *inexact superclass problem*, because it arises from a mismatch between the statically expected type of the superclass and the actual (exact) type of the superclass. In Section 4.3.2, we will show how this problem can be addressed in a type-safe manner.

```
type Constructor = new (...args: any[]) ⇒ {};

function Mixin<TBase extends Constructor>(Base: TBase) {
  return class extends Base {
    // If m() exists in Base, that one will be overridden by this definition.
    m(): number { return 48; }
  };
}

class A {
  m(): string { return "foobar"; }
  n(): string { return this.m().toUpperCase(); }
}

// Below we use A as Base, which contains m() with a different type.
const B = Mixin(A);
(new B).n()  // Runtime Error!
```

Figure 4.2: *Inexact Superclass Problem:* Dynamic inheritance is type-unsafe in TypeScript.

Dynamic inheritance in TypeScript. While JavaScript accepts the unsafe overriding in Figure 4.1, TypeScript detects the type mismatch between the two methods and rejects the code. For top-level classes and static inheritance, TypeScript's type system is quite standard and rejects many unsafe examples. However, the checks that TypeScript does are insufficient for *dynamic inheritance*, which is recommended by the TypeScript documentation to implement mixins.[4] We illustrate the issue in the program in Figure 4.2.

Our example follows the guidelines in the TypeScript documentation to type mixins and first-class classes. First of all, a type `Constructor` is declared to represent a class. Since its return type is an empty object type, the type of every class is a subtype of `Constructor`. In other words, every class can be used as `Base`. The function `Mixin` takes a base class of type `TBase` and returns a new class that extends (or overrides) the base class with the method `m`. Then we obtain class `B` by applying `Mixin` to class `A`. Note that `A` already has a method `m` with a different type, and the other method `n` relies on `m` returning a *string*. However, the subclass returned by `Mixin` overrides `m` with a method that returns a *number* instead. Finally, we instantiate the class `B` and call the method `n`. A runtime error occurs because the method `m` is unexpectedly overridden. In essence, we cannot predict the exact type of the superclass at compile time, so we cannot prevent the unsafe overriding as statically typed languages do for second-class classes and static inheritance.

[4] https://www.typescriptlang.org/docs/handbook/mixins.html

Constrained mixins. The TypeScript documentation also mentions *constrained mixins*, which provide finer control on the base class. In a constrained mixin, the base class is known to have some methods, which is useful for the subclass to safely rely on those methods being present in the superclass. Constrained mixins are modeled with a generic version of `Constructor`:

```
type GConstructor<T = {}> = new (...args: any[]) ⇒ T;
```

The generic parameter `T` represents the interface of the base class and defaults to an empty object type. For example, we can define another mixin that relies on a method called `pow`:

```
type Exponentiatable = GConstructor<{ pow: (x: number, y: number) ⇒ number }>;

function AnotherMixin<TBase extends Exponentiatable>(Base: TBase) {
  return class extends Base {
    cube(x: number) { return this.pow(x, 3); }
  };
}
```

In `AnotherMixin`, the method `cube` relies on `this.pow`, which is declared to be present by the interface `Exponentiatable`. Similarly, in the definition of `Mixin` in Figure 4.2, we could declare that `TBase` extends some type like `GConstructor<AInterface>`. Although the base class is constrained by the interface now, it still does not help with the issue of unsafe overriding. The problem here is that the base class may contain more methods than the expected interface. For instance, the base class could contain another method called `cube` that would return a *string*, and would be called in the base class by some other method `rubik`. Then we could still run into the same problem, if `rubik` is called from an object that combines both classes. There is no way in TypeScript to express the constraint that `cube` is *absent* in the base class. Such absence constraints are key to preventing unsafe overriding in dynamic inheritance while retaining flexibility.

From static to dynamic inheritance. The crucial point in our examples is that dynamic inheritance has the flexibility to pass a class with a *subtype* of the expected type for the base class in `Mixin`. Languages with static inheritance and second-class classes, like Java or C#, do not have this flexibility. Subclassing is usually modeled with a construct like `class B extends A {...}`. In languages with first-class classes, `A` can be an arbitrary expression; but in languages with static inheritance, it can only be a concrete class name. In Java, for instance, a class `A` is associated both to a type `A`, which is the *exact* interface (or type) of the class, and a corresponding implementation of type `A`. In other words, a class declaration has two roles in these languages: declaring an interface and providing

```
class ANSIString {
  constructor(str) {
    this.length = str.length;
    this.chars = str.split('');
  }

  Iterator() {
    const outer = this;
    return class {
      index = 0;
      hasNext() { return this.index < outer.length; }
      next() { return outer.chars[this.index++]; }
    };
  }

  print() {
    const it = new (this.Iterator());  // Iterator is dynamically bound.
    while (it.hasNext()) alert(it.next());
  }
}
```

Figure 4.3: A string iterator in JavaScript using nested classes.

an implementation with exactly that interface. Thus, we can never inherit from an implementation that has a subtype of the superclass type. This avoids the inexact superclass problem that we have to face with dynamic inheritance in our TypeScript example, at the cost of flexibility.

4.2.3 Nested Classes via First-Class Classes

Both JavaScript and TypeScript support first-class classes: a class can be defined in various places including within another class, or even a method. Thus, *nested classes* come (almost) for free once a language supports first-class classes. In contrast, some other OOP languages, such as Java, do not support first-class classes, but they still add support for nested classes as a separate feature.

Nested classes are useful for encapsulation, and usually, they can make use of the definitions from the outer class. For example, Figure 4.3 shows how to model a string-specific iterator as a nested class in JavaScript. The constructor for the `Iterator` class is modeled by a *factory* method. The method `print` relies on the nested iterator class to iterate over the characters in the string.

Why not a class field? In JavaScript, the use of the factory method is important to provide access to `this` of the outer class. If we declare a class field directly with `Iterator = class {...}`, we would not be able to access the properties and methods of `ANSIString` within `Iterator`. JavaScript does not provide a direct way to refer to the outer `this` from the nested class. That is why we have to capture the reference to the outer `this` in a variable `outer` before using it in the nested class. Then, the properties declared in the outer class, `length` for example, can be accessed via `outer.length`.

The second reason for using a factory method is to make access to `super.Iterator` possible in a subclass of `ANSIString`. In JavaScript, a class field defined by the superclass is not accessible in the subclass via `super`. Declaring `Iterator` as a factory method bypasses the restriction. Although we do not use `super.Iterator` in the current example, Section 4.2.4 will show some use cases.

Overriding nested classes. In JavaScript, the inheritance behavior for nested classes is consistent with that for methods: they both employ an overriding semantics. This is partly because nested classes must always be accessed via a property or a method, and then we just use the default overriding semantics for them. The ability to override nested classes allows some useful forms of family polymorphism, as we shall discuss in Section 4.2.4. However, it is also a problem for type safety since we can override a class with another class that has an entirely (or partially) different set of methods. For example, the following code is allowed in JavaScript:

```
class UTF8String extends ANSIString {
  Iterator() { return class {
    forEach(callback) { /* ... */ }
  }; }
}
(new UTF8String("Hi")).print();  // Runtime Error!
```

The class `Iterator` nested in `ANSIString` contains two methods `hasNext` and `next`, while the one nested in `UTF8String` only contains a different method `forEach`. After overriding, `print` triggers a runtime error since it depends on the aforementioned two methods. Therefore, code relying on nested classes having a certain interface can be completely broken by an override that replaces the class with some other incompatible class.

Nested classes in TypeScript. TypeScript also attempts to prevent type-unsafe overriding for nested classes. Similar code will be rejected by TypeScript because of the type

```
class ANSIString {
    int length;
    char[] chars;
    ANSIString(String str) {
        length = str.length();
        chars = str.toCharArray();
    }
    class Iterator {
        int index;
        boolean hasNext() { return index < length; }
        char next() { return chars[index++]; }
    }
    void print() {
        Iterator it = new Iterator();  // Iterator is statically bound.
        while (it.hasNext()) System.out.print(it.next());
    }
}
class UTF8String extends ANSIString {
    // We trivially call super's constructor.
    UTF8String(String str) { super(str); }

    class Iterator {  // This class shadows ANSIString.Iterator.
        void forEach(Consumer<? super Character> action) { /* ... */ }
    }
}
```

Figure 4.4: Nested classes in Java, with a shadowing semantics.

incompatibility between the two nested classes. However, with dynamic inheritance, the type system still suffers from similar issues to those shown in Figure 4.2:

```
function Mixin<TBase extends Constructor>(Base: TBase) {
  return class extends Base {
    Iterator() { return class {
      forEach(callback: (_: string) ⇒ void) { /* ... */ }
    }; }
  };
}
const UTF8String = Mixin(ANSIString);
(new UTF8String("Hi")).print();  // Runtime Error!
```

Therefore, TypeScript's support for nested classes is also affected by the inexact superclass problem. Thus, nested classes can have type-soundness issues as well.

Nested classes with shadowing in Java. Finally, let us make a small digression to see how nested classes are treated in Java. Figure 4.4 illustrates a variant of our example in Java. Similarly to the example in JavaScript, we create a new class `UTF8String` that inherits from `ANSIString` and define a different set of methods in the nested class `Iterator`. The code type-checks in Java and is still type-safe. The key to the type safety is that, unlike methods, nested classes are not implicitly overridden in Java. Instead, the `Iterator` in `UTF8String` *shadows* the one in `ANSIString`. In other words, `new Iterator()` in `print` is statically bound and is always instantiating `ANSIString.Iterator`. Nested classes are not dynamically dispatched in Java, which is inconsistent with the inheritance behavior for methods. The shadowing approach has the advantage of type safety, but this comes at the cost of flexibility, since the ability to override and dynamically bind nested classes is useful, as we shall see next.

4.2.4 Virtual Classes and Family Polymorphism

The ability to override or refine nested classes provides a considerable amount of flexibility, and is a key idea behind concepts such as *virtual classes* [Clarke et al. 2007; Ernst et al. 2006; Madsen and Møller-Pedersen 1989] and *family polymorphism* [Ernst 2001; Saito et al. 2008; Zhang and Myers 2017]. Thus, as we shall argue in this subsection, both JavaScript and TypeScript support virtual classes to a large extent, which can be useful for writing highly modular and reusable code.

Virtual classes. As we have seen before, a method in a superclass can be overridden in a subclass to refine its behavior. A call to the method is dynamically dispatched according to the runtime type of the object. Such a late-bound method is called a *virtual method*. In the same way, the power of dynamic dispatching can be extended to nested classes. *Virtual classes* are nested classes that can be overridden (or rather refined) in subclasses, and the reference to the virtual class is determined by the runtime type of the object of the outer class. Virtual classes were originally introduced in the BETA programming language [Madsen et al. 1993], and they are also essentially supported in JavaScript and TypeScript via first-class classes and the overriding semantics.

Family polymorphism. Virtual classes enable *family polymorphism*, which naturally solves the long-standing dilemma of modularity and extensibility – *the expression problem* [Wadler 1998] – in a Scandinavian style [Ernst 2004]. In the expression problem, the challenge is to provide various operations (evaluation, pretty-printing, etc.) over various

```
type Eval = { eval: () ⇒ number };

class FamilyEval {
  Lit(n: number) {
    return class {
      eval() { return n; }
    };
  }
  Add(l: Eval, r: Eval) {
    return class {
      eval() { return l.eval() +
                      r.eval(); }
    };
  }
}
```

(a) Initial family.

```
type Print = { print: () ⇒ string };

class FamilyPrint extends FamilyEval {
  Lit(n: number) {
    return class extends super.Lit(n) {
      print() { return n.toString(); }
    };
  }
  Add(l: Eval&Print, r: Eval&Print) {
    return class extends super.Add(l, r) {
      print() { return l.print() + " + " +
                      r.print(); }
    };
  }
}
```

(b) Adding a new operation.

```
class FamilyNeg extends FamilyPrint {
  Neg(e: Eval&Print) {
    return class {
      eval()  { return -e.eval(); }
      print() { return "-(" + e.print() + ")"; }
    };
  }
}
```

(c) Adding a new expression.

Figure 4.5: Expression Problem in TypeScript.

expressions (numbers, addition, negation, etc.) in a modular fashion. A satisfactory solution should allow modular, type-safe extension to both expressions and operations.

We start the example with numeric literals and addition, as well as the evaluation operation in Figure 4.5a. `Lit`, for numeric literals, and `Add`, for addition, form the initial class family `FamilyEval`. Since TypeScript is structurally typed, we do not need to declare an abstract class or interface `Exp` together with `Lit` and `Add`. Instead, we can directly use type `Eval` to annotate the parameters of `Add`.

To add a new operation, say pretty-printing, we can create a new class family `FamilyPrint` that inherits from `FamilyEval`. Figure 4.5b shows the code for the new family. In the new family, `Lit` and `Add` also inherit from **`super`**`.Lit` and **`super`**`.Add`, and a new method `print` is added to both of them. The new operation is represented by type `Print`, and the parameters of `Add` are refined to have type `Eval&Print`. As mentioned in Section 4.2.1, TypeScript allows bivariant subtyping for parameters of class members, so the unusual refinement of `Add` type-checks here. Note that the overriding of nested classes is a special case here: `Lit`

and `Add` are simply extended with new methods, with no existing methods being overridden. In other words, the nested classes are being *merged*, instead of overriding existing functionality.

Similarly, we create a new family `FamilyNeg` for a new expression, say negation in Figure 4.5c. Finally, we instantiate `FamilyNeg` and build an expression using all three constructors (`Lit`, `Add`, and `Neg`). Both operations (`eval` and `print`) are available for the expression:

```
const fam = new FamilyNeg();
const e = new (fam.Add(new (fam.Lit(48)), new (fam.Neg(new (fam.Lit(2))))));
e.print() + " = " + e.eval()  // "48 + -(2) = 46"
```

In this way, we can solve the expression problem in TypeScript (modulo the type-safety requirement). Although TypeScript does not fully ensure type safety, its support for a rather minimal encoding of virtual classes allows a lot of flexibility and reuse, which can be quite useful in practice.

One final remark is that the solution in TypeScript is still not completely satisfactory because the order of extensions is fixed by the inheritance hierarchy (from `FamilyEval` to `FamilyPrint` to `FamilyNeg`). In other words, the extension of `Neg` is coupled to the extension of `Print`, and we cannot use the extension of `Neg` independently. This issue was not mentioned by Wadler in the original expression problem, but it was later identified by Zenger and Odersky [2005] as *independent extensibility*. In TypeScript, a possibility to address the coupling issue is to adopt the mixin pattern, making class families such as `FamilyPrint` and `FamilyNeg` functions parametrized by the family superclass. For simplicity of presentation, we have just employed static inheritance here. We will also address this issue in CP's solution in Section 4.3.3, which provides a simple and natural approach to avoid coupling and, additionally, is type-safe.

4.3 Dynamic Inheritance in CP

CP is a statically typed language that supports dynamic inheritance via merging and still guarantees type safety. In this section, we first show how inheritance is modeled as merging in CP. We then demonstrate how CP resolves potential conflicts in dynamic inheritance and addresses the inexact superclass problem. Finally, we present a form of *dynamic* family polymorphism in CP.

4.3.1 From Merging to Inheritance

Let us first discuss how inheritance is modeled in CP. According to the denotational semantics of inheritance [Cook and Palsberg 1989], an object is essentially a record, and a

class (or a trait in CP) is essentially a function over records. Therefore, class `A` in Figure 4.2 can be encoded as:

```
type Rcd = { m: String; n: String };

-- class A
mkA = \(this: Rcd) → { m = "foobar"; n = toUpperCase this.m };
```

The function parameter `this` is a self-reference. With the self-reference, we can refer to other fields like `this.m` in the `n` field. In this model, the instantiation of a class is obtained by taking a fixpoint of the function. Furthermore, class inheritance can be encoded as record concatenation:

```
-- class B extends A
mkB = \(this: Rcd) → let super = mkA this in super , { m = 48 };
```

We first provide the new self-reference to `mkA` to obtain `super`. Then we merge `super` with the body of class `B` to obtain the final object. After instantiating class `B` with a fixpoint, we can access the `n` field:

```
o = fix this: Rcd. mkB this;  --> { m = "foobar"; n = "FOOBAR"; m = 48 }
o.n  --> "FOOBAR"
```

Here we get the expected result instead of a runtime error. The key point is that we allow duplicate labels as long as the fields have disjoint types. Because of the merging semantics of CP, `o` will have two `m` fields: one of type **`Int`** and the other of type **`String`**. Thus, unlike TypeScript, no implicit (and type-unsafe) overriding happens in this case. Instead, both `{ m = "foobar" }` and `{ m = 48 }` are kept in the record `o`, and `toUpperCase this.m` will automatically pick the former one. Internally, `o.m` has the intersection type **`String`**`&`**`Int`**, which means it contains a merge of a string and an integer. Such behavior is a kind of *overloading by return type*, which is supported in some languages such as Swift and Haskell (via type classes) [Marntirosian et al. 2020].

Traits in CP follow the aforementioned model of inheritance. Therefore, the example above can be rewritten in the form of traits:

```
mkA = trait [this: Rcd] ⇒ { m = "foobar"; n = toUpperCase this.m };
mkB = trait [this: Rcd] inherits mkA ⇒ { m = 48 };
```

The self-type annotation `[this: Rcd]` corresponds to the function parameter `this` in the previous code. If there is no use of `this` in any field, the self-type annotation can be omitted. The instantiation of a trait is more conveniently done by the **`new`** keyword:

```
o = new mkB;  o.n  --> "FOOBAR"
```

```
mixin (TBase * { m: Int }) (base: Trait<TBase>) =
  trait [this: TBase] inherits base ⇒ { m = 48 };

mkA = trait [this: { m: String; n: String }] ⇒ {
  m = "foobar";
  n = toUpperCase this.m;
};

o = new mixin @{ m: String; n: String } mkA;
o.n  --> "FOOBAR"
```

Figure 4.6: Solving the inexact superclass problem in CP.

4.3.2 Dynamic Inheritance in CP

Now let us go back to the safety issue demonstrated in Figure 4.2 and see how it can be solved in CP. The code for a CP solution is shown in Figure 4.6. Here the function `mixin` has two parameters: `TBase` is a type parameter, which is disjoint with `{ m: Int }`; and `base` is a term parameter, which is a trait that implements `TBase`. Like first-class classes in TypeScript, we can dynamically create a trait that inherits from `base` in CP. The difference here is that we can declare the absence of `{ m: Int }` in the trait `base` to make sure that there is no conflict. As mentioned in Section 4.3.1, CP does a fine-grained disjointness check that considers, not only the label name, but also the field type. Therefore, `{ m: String }` is disjoint with `{ m: Int }`, and there is no conflict in the dynamic inheritance. Since both versions of `m` fields are available in `o`, the `n` field can still rely on the original `m` field that contains a string. Together with disjointness constraints, type safety is guaranteed in CP without sacrificing the flexibility of dynamic inheritance.

Finally, note that if we apply `mixin` to a different trait that contains a `m` field of type **`Int`**:

```
mkA' = trait ⇒ { m = 0; n = 0 };
o = new mixin @{ m: Int; n: Int } mkA';  -- Type Error!
```

We will get a type error because `{ m: Int; n: Int }` is *not* disjoint with `{ m: Int }`. In other words, the field `m` in `mkA'` conflicts with `m` in `mixin`.

4.3.3 Family Polymorphism in CP

Here we revisit the example of family polymorphism in Section 4.2.4 and show how it can be implemented in CP. As before, we start with the evaluation of numeric literals and addition. The CP code is shown in Figure 4.7a. The compositional interface `AddSig` serves as the specification of expressions, while type `Eval` represents the evaluation operation.

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

type Eval = { eval: Int };

familyEval =
  trait implements AddSig<Eval> ⇒ {
    (Lit   n).eval = n;
    (Add l r).eval = l.eval + r.eval;
  };
```

(a) Initial family.

```
type Print = { print: String };

familyPrint =
  trait implements AddSig<Print> ⇒ {
    (Lit   n).print = toString n;
    (Add l r).print = l.print ++ " + "
                   ++ r.print;
  };
```

(b) Adding a new operation.

```
type NegSig<Exp> = { Neg: Exp → Exp };

familyNeg =
  trait implements NegSig<Eval&Print> ⇒ {
    (Neg e).eval  = -e.eval;
    (Neg e).print = "-(" ++ e.print ++ ")";
  };
```

(c) Adding a new expression.

Figure 4.7: Expression Problem in CP.

Note that `<Exp>` is a special type parameter called a *sort* in CP. A sort is kept abstract until it is instantiated with a concrete type like in `AddSig<Eval>`. The interface `AddSig<Eval>` is implemented by trait `familyEval`, where syntactic sugar called *method patterns* is used to keep code compact. The desugared code is:

```
familyEval = trait implements AddSig<Eval> ⇒ {
  Lit = \n   → trait ⇒ { eval = n };
  Add = \l r → trait ⇒ { eval = l.eval + r.eval };
};
```

Although the syntactic sugar makes it seem that `eval` is defined by pattern matching of constructors, `(Lit n)` and `(Add l r)` are actually nested traits, which are virtual and can be refined in CP.

The solution to the expression problem in CP is quite straightforward. To extend operations, we instantiate the sort with another type and implement it with another trait. For example, Figure 4.7b shows how to add support for pretty-printing. In the other dimension, we add negation to numeric literals and addition. We define a new compositional interface and implement both operations with a trait in Figure 4.7c. This time we instantiate the sort of `NegSig` with the intersection type `Eval&Print`.

Finally, we can compose the two-dimensional extensions together by the merge operator easily:

```
fam = new familyEval , familyPrint , familyNeg
    : AddSig<Eval&Print> & NegSig<Eval&Print>;
```

Nested composition and distributive subtyping. The merge of the three traits seems simple from a syntactic perspective. However, it requires a more sophisticated mechanism under the hood. Let us look at the desugared code for the merge between `familyEval` and `familyPrint`:

```
trait implements AddSig<Eval> ⇒ {                -- familyEval
  Lit = \n   → trait ⇒ { eval = n };
  Add = \l r → trait ⇒ { eval = l.eval + r.eval };
} ,
trait implements AddSig<Print> ⇒ {               -- familyPrint
  Lit = \n   → trait ⇒ { print = toString n };
  Add = \l r → trait ⇒ { print = l.print ++ " + " ++ r.print };
}
```

Our expectation is that the result of merging should contain, for example, a single constructor `Lit` that supports both the `eval` and `print` operations. Therefore, the result should be equivalent to:

```
trait implements AddSig<Eval&Print> ⇒ {
  Lit = \n   → trait ⇒ { eval = n;
                        print = toString n };
  Add = \l r → trait ⇒ { eval = l.eval + r.eval;
                        print = l.print ++ " + " ++ r.print };
}
```

To achieve this, CP employs *nested composition* [Bi et al. 2018] and *distributive subtyping* [Barendregt et al. 1983], where traits, records, and functions distribute over intersections. In other words, merging applies to the whole trait hierarchy, including nested traits. This example showcases family polymorphism by the refinement of nested traits (i.e. CP's version of virtual classes).

With these features available in CP, we can access the three constructors (`Lit`, `Add`, and `Neg`) as well as the two operations (`eval` and `print`), similarly to the previous TypeScript code:

```
e = new fam.Add (new fam.Lit 48) (new fam.Neg (new fam.Lit 2));
e.print ++ " = " ++ toString e.eval  --> "48 + -(2) = 46"
```

Dynamic family polymorphism. Since merging generalizes dynamic inheritance, we can rewrite `familyNeg`, for instance, using a mixin style:

```
familyNeg (TBase * NegSig<Eval&Print>) (base: Trait<TBase>) =
  trait [this: TBase] implements NegSig<Eval&Print> inherits base ⇒ {
    (Neg e).eval  = -e.eval;
    (Neg e).print = "-(" ++ e.print ++ ")";
  };
fam = new familyNeg @AddSig<Eval&Print> (familyEval,familyPrint)
    : AddSig<Eval&Print> & NegSig<Eval&Print>;
```

By applying `familyNeg` to `(familyEval,familyPrint)`, we dynamically create a trait that inherits from the latter. Of course, we can choose other traits as a base trait at run time, which is supported by dynamic inheritance in CP.

Note that in Section 4.2.4, `FamilyEval`, `FamilyPrint`, and `FamilyNeg` have a statically fixed inheritance hierarchy. As a result, the negation expression cannot be separated from the other two expressions because `FamilyNeg` is a subclass of `FamilyPrint`. In contrast, the inheritance hierarchy can be dynamically determined in CP, so `familyEval`, `familyPrint`, and `familyNeg` can all be individually used or composed with any other traits. In fact, CP's solution solves a dynamic variant of the expression problem, which can be seen as the combination of the expression product line [Lopez-Herrejon et al. 2005] and dynamic software product lines [Hallsteinsen et al. 2008].

4.3.4 Discussion

In this and the previous section, we have seen that both CP and JavaScript/TypeScript support a powerful and expressive form of dynamic inheritance. However, there are some important differences worth noting:

- **CP is type-safe.** While the three languages provide a high degree of flexibility, CP is the only language which combines flexibility and type safety.
- **No implicit overriding in CP.** Unlike JavaScript/TypeScript, where implicit overriding is common, CP adopts a trait model, so implicit overriding can *never* happen.
- **Dealing with conflicts using disjoint types.** In JavaScript/TypeScript, method overriding is based on *names*. So even when the method or field in the superclass has a different (or disjoint) type, overriding happens when the subclass has a method with the same name. As we have seen, this is the source of type unsoundness in the inexact superclass problem. In CP, methods with disjoint types can coexist in

the same object. Thus, for the same situation, CP will not override but inherit the method from the superclass.

These differences are important to obtain flexibility while preserving type safety. However, these differences also mean that the dynamic semantics of CP needs to be different from that of JavaScript/TypeScript. In particular, the dynamic semantics of CP has to be aware of types, since types play a role in determining whether conflicts exist or not, and in unambiguously performing method lookup. This creates important challenges in obtaining an efficient implementation, which have not been addressed in previous work.

5 Compositional Embeddings of Domain-Specific Languages

This chapter presents the second case study: compositional embeddings of domain-specific languages (DSLs). We concentrate on how to embed DSLs in a modular way and compose them to build larger DSLs. We start with an introduction to the problem that we aim to solve in Section 5.1. Then we demonstrate existing embedding techniques in Section 5.2 and show in Section 5.3 that *compositional embeddings* combine the advantages of shallow and deep embeddings and surpasses other techniques like tagless-final embeddings by supporting modular dependencies. We validate our approach by showcasing a compositionally embedded DSL for document authoring in Section 5.4 and further evaluating it in Section 5.5.

5.1 Introduction

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 [Hughes 1995; Wadler 2003], parser combinators [Leijen and Meijer 2001], and property-based testing frameworks [Claessen and Hughes 2000] – are defined as embedded DSLs. There are a few techniques for such embeddings, including the well-known *shallow* and *deep* embeddings [Boulton et al. 1992].

Unfortunately, shallow and deep embeddings come with various trade-offs in existing programming languages. Such trade-offs have been widely discussed in the literature [Gibbons and Wu 2014; Rompf et al. 2012; Scherr and Chiba 2014]. On the one hand, the strengths of shallow embeddings are in providing *linguistic reuse* [Krishnamurthi 2001], 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, Svenningsson and Axelsson [2015] 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 [Jovanović et al. 2014; Rompf et al. 2012; Svenningsson and Axelsson 2015] 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 [Hinze 2006; Oliveira 2009; Oliveira et al. 2006]. Such techniques include *tagless-final embeddings* [Carette et al. 2009; Kiselyov 2010], *polymorphic embeddings* [Hofer et al. 2008], and *object algebras* [Oliveira and Cook 2012], 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.

In this chapter, we show that the compositional programming paradigm and the CP language provide improved programming language support for embedded DSLs. CP provides an alternative way to define data structures, which is modularly extensible, in contrast to algebraic data types in functional programming or class hierarchies in object-oriented programming. CP does not suffer from the infamous *expression problem* [Wadler 1998] and comes with several mechanisms to express *modular dependencies*, allowing powerful forms of dependency injection and complex semantic interpretations.

With those programming language features, we obtain a new form of embedding called a *compositional embedding*, with nearly all of the advantages of both shallow and deep embeddings. On the one hand, compositional embeddings enable various forms of linguistic reuse that are characteristic of shallow embeddings, including the ability to reuse host-

language optimizations in the DSL and easily add new DSL constructs. On the other hand, similarly to deep embeddings, compositional embeddings support definitions by pattern matching or dynamic dispatching, including optimizations and transformations over the abstract syntax of the DSL, as well as the ability to add new interpretations. In short, we believe that compositional embeddings come very close to the holy grail of embedded language implementation desired by Svenningsson and Axelsson [2015].

We also illustrate an instance of compositional embeddings with a DSL for document authoring called EXT (*EXtensible Typesetting*). The DSL is highly flexible and extensible, allowing users to create various non-trivial extensions. For instance, EXT supports extensions that enable the production of wiki-like documents, LATEX documents, SVG charts, or computational graphics like fractals.

5.2 Embeddings of DSLs

In this section, we give some background on existing approaches to embedded DSLs and evaluate their strengths and drawbacks. We focus on *shallow* and *deep* embeddings [Boulton et al. 1992], which are the two main alternative forms of embeddings. We also discuss some other forms of embeddings near the end of this section. To illustrate all these embeddings, we present programs in Haskell, which is well known for its good support for embedded DSLs and has a syntax close to the CP language.

5.2.1 A Simple Region DSL

Inspired by Hudak [1998] and Hofer et al. [2008], we consider a simple region DSL for plane geometry. To illustrate the challenges in developing such a DSL, we consider five separate steps in the development, which illustrate various desired features, such as linguistic reuse and the ease of adding features for software evolution.

1. **Initial system.** We start with a small region language with five constructs: `circle` for creating a circular region with a given radius, `outside` for a complement to a certain region, two set operators `union` and `intersect`, and finally `translate` for moving a region by a given vector. The initial interpretation is to simply calculate the syntactic size of a region.

2. **Linguistic reuse.** We create a region with structure sharing to assess the difficulty of reusing host-language optimizations.

```haskell
data Vector = Vector { x :: Double, y :: Double } deriving Show

type Region = Int

circle :: Double → Region
circle _  =  1

outside :: Region → Region
outside a  =  a + 1

union :: Region → Region → Region
union a b  =  a + b + 1

intersect :: Region → Region → Region
intersect a b  =  a + b + 1

translate :: Vector → Region → Region
translate _ a  =  a + 1
```

(a) A shallow embedding.

```haskell
data Region where
  Circle    :: Double → Region
  Outside   :: Region → Region
  Union     :: Region → Region → Region
  Intersect :: Region → Region → Region
  Translate :: Vector → Region → Region


size :: Region → Int
size (Circle      _) = 1
size (Outside     a) = size a + 1
size (Union     a b) = size a + size b + 1
size (Intersect a b) = size a + size b + 1
size (Translate _ a) = size a + 1
```

(b) A deep embedding.

Figure 5.1: The initial system for the region DSL.

3. **Extensibility with new constructs.** We extend the region language with three more constructs: `univ` for the universal region that contains all points, `empty` for the empty region that contains no points, and `scale` for resizing a region by two scale factors in a vector.

4. **Extensibility with new operations.** We add a new interpretation that checks whether a given point resides in a region.

5. **Complex interpretations, transformations, and optimizations.** Last but not least, we discuss three kinds of more complex interpretations that illustrate dependent interpretations (i.e. interpretations that need to call other interpretations in their implementations) and transformations over the AST of the region language.

5.2.2 Initial System

Figure 5.1 shows the definitions of the initial DSL, including the language interface and a simple interpretation for shallow and deep embeddings.

Language interfaces. In the shallow embedding, `Region` denotes the semantic domain. All the five language constructs are implemented as functions (called denotation functions), which implement the respective semantics of the language constructs. Thus, the

language interface is essentially the signature of the denotation functions and is entangled with the definition of a particular semantic domain. In the deep embedding, `Region` is defined as an algebraic data type. Algebraic data types only capture the abstract syntax, thus enabling the language interface to be separated from any concrete semantics.

Semantic interpretations. There can be many semantic interpretations, which associate language constructs with their meanings. The initial semantics that we choose here is an abstract interpretation, which calculates the syntactic size of a region (i.e. the number of AST nodes). We opt to have an interpretation that is slightly artificial, but simple, so that we can focus on the key aspects of compositional embeddings. In the shallow embedding, the semantics is defined together with the language interface since we must specify some concrete type (**`Int`** in this case) for `Region`. In contrast, to create a new semantic interpretation in the deep embedding, we define a function `size` using pattern matching over the `Region` data type.

We can see that the two embeddings work quite differently: the shallow embedding encodes semantics in region constructors, and the interpretation function from `Region` to **`Int`** is merely the identity function; the deep embedding does nothing concerning constructors but hands over the work to the interpretation function `size`.

5.2.3 Linguistic Reuse and Meta-Language Optimizations

An important concern for embedded DSLs is *linguistic reuse* [Krishnamurthi 2001]. One of the key selling points of embedded DSLs is that they can reuse much of the infrastructure of the host language and inherit various optimizations that are available in the host language. However, there are important differences between shallow and deep embeddings with respect to linguistic reuse. As widely observed in previous work [Jovanović et al. 2014; Rompf et al. 2012; Svenningsson and Axelsson 2015], shallow embeddings make linguistic reuse easier. To illustrate this, we can create a region that contains a series of repeated subregions with horizontally aligned circles:

```
circles = go 20 (2 ** 18)
  where go :: Int → Double → Region
        go 0 offset = circle 1
        go n offset = let shared = go (n - 1) (offset / 2)
                      in union (translate Vector { x = -offset, y = 0 } shared)
                               (translate Vector { x =  offset, y = 0 } shared)
```

The region `circles` is defined via an auxiliary recursive function `go`. In the shallow embedding, `shared` is of type **`Int`**, and using `shared` twice in the **`let`**-body will avoid interpreting

the region twice. The code above also works for the deep embedding, assuming some smart constructors such as `circle = Circle`. However, in the deep embedding, `shared` is an AST of type `Region`. In this case, the AST will be duplicated in the **`let`**-body, and later when we interpret the region, we need to traverse the same AST twice, duplicating the size calculation. Since `go` is recursive, it will lead to a lot of sharing for the shallow approach and a lot of repetition for the deep approach. As a result, the evaluation of `circles` is instantaneous in the shallow approach, whereas in the deep approach, it basically does not terminate in a reasonable amount of time. In short, in shallow embeddings, we are able to preserve sharing by naturally exploiting the sharing optimizations present in the host language, but sharing is lost in deep embeddings.

We should remark that this particular issue regarding sharing (which is just an instance of linguistic reuse) is well known [Gill 2009; Kiselyov 2011; Oliveira and Löh 2013]. There have been several proposed techniques that enable preserving sharing in deeply embedded DSLs. However, all those techniques add extra complexity to the DSL encoding, and they may even make the DSL harder to use. This is in contrast to the shallow approach, where no extra work is required by the programmer and host-language features are naturally reused.

5.2.4 Adding More Language Constructs

As part of evolving the DSL with additional features, we may decide to add more constructs for regions. To evaluate how easy it is to add more language constructs, we add three more language constructs for the universal or empty region as well as a scaling operator. Shallow embeddings make such extensions very easy. We only need to add three new denotation functions:

```
univ :: Region      empty :: Region     scale :: Vector → Region → Region
univ = 1            empty = 1           scale _ a  =  a + 1
```

Notwithstanding the modular extension in shallow embeddings, it is awkward to add language constructs in deep embeddings. We have to modify the algebraic data type `Region` and all related interpretation functions to add new cases. This is not modular because we must change the existing code. Even if we have access to previous definitions, it is inevitable to recompile all the code that depends on `Region`.

In essence, once we start adding new features, we are quickly faced with an instance of the *expression problem* [Wadler 1998]. Shallow embeddings make adding constructs easy, whereas adding new constructs in deep embeddings is more difficult and non-modular. As we shall see next, when adding new semantic interpretations, we have a dual problem.

5.2.5 Adding a New Interpretation

The syntactic size is too abstract to describe a region, so let us add a new interpretation that checks whether a region contains a given point. For example, a circular region of radius r contains (x, y) if and only if $x^2 + y^2 \leq r^2$. Although adding constructs is awkward in deep embeddings, adding a new interpretation function is trivial:

```
contains :: Region → Vector → Bool
Circle      r `contains` p  =  p.x ** 2 + p.y ** 2 ≤ r ** 2
Outside     a `contains` p  =  not (a `contains` p)
Union     a b `contains` p  =  a `contains` p || b `contains` p
Intersect a b `contains` p  =  a `contains` p && b `contains` p
Translate Vector {..} a `contains` p  =
                    a `contains` Vector { x = p.x - x, y = p.y - y }
```

The code is mostly straightforward.[1] The only minor remark is that we use a record wildcard (`Vector {..}`) in the case of `Translate` to bring record fields (`x` and `y`) into scope.

However, there is no easy modular way to support multiple interpretations in shallow embeddings. We need to completely change the definition of `Region` and remap all the language constructs to a new semantic domain. The issue of semantic extension in shallow embeddings is dual to that of language extension in deep embeddings. To address the tension between the two dimensions, some alternative embeddings are proposed, such as tagless-final embeddings [Carette et al. 2009; Kiselyov 2010] and polymorphic embeddings [Hofer et al. 2008], though they still have some significant drawbacks, which will be revealed shortly.

5.2.6 Dependencies, Transformations, and Complex Interpretations

One area where deeply embedded DSLs shine is in enabling more complex kinds of interpretations. These interpretations may, for example, enable transformations or rewritings on the AST, which are helpful for writing domain-specific optimizations, among other things. For writing such complex forms of interpretations, we often require multiple *dependent* functions defined by pattern matching, and sometimes we may even need nested pattern matching. Such interpretations are very challenging for shallow DSLs and are often the reason why DSL writers prefer deep embeddings.

Dependent interpretations. Let us start with a dependent interpretation that shows a text representation of a region using a deep embedding:

[1] Two GHC language extensions are required here: *OverloadedRecordDot* and *RecordWildCards*.

```
text :: Region → String
text   (Circle      r)  =  "circular region of radius " ++ show r
text   (Outside     a)  =  "outside a region of size " ++ show (size a)
text s@(Union     _ _)  =
    "union of two regions of size " ++ show (size s) ++ " in total"
text s@(Intersect _ _)  =
    "intersection of two regions of size " ++ show (size s) ++ " in total"
text s@(Translate _ _)  =  "translated region of size " ++ show (size s)
```

Note that the definition of `text` *depends* on the previously defined `size` function. Such a definition is challenging in shallow embeddings. A workaround sometimes used in shallow embeddings is to use tuples to define multiple interpretations simultaneously. For example, we can define `size` and `text` together as:

```
type Region = (Int, String) -- (size, text)

circle      r  =  (1, "circular region of radius " ++ show r)
outside     a  =  (fst a + 1, "outside a region of size " ++ show (fst a))
union     a b  =
    (size, "union of two regions of size " ++ show size ++ " in total")
  where size = fst a + fst b + 1
intersect a b  =
    (size, "intersection of two regions of size " ++ show size ++ " in total")
  where size = fst a + fst b + 1
translate _ a  =  (size, "translated region of size " ++ show size)
  where size = fst a + 1
```

Mutually dependent interpretations. A similar but more interesting example is two interpretations for checking universality and emptiness:

```
isUniv :: Region → Bool
isUniv            Univ  =  True
isUniv (Outside     a)  =  isEmpty a
isUniv (Union     a b)  =  isUniv a || isUniv b
isUniv (Intersect a b)  =  isUniv a && isUniv b
isUniv (Translate _ a)  =  isUniv a
isUniv (Scale     _ a)  =  isUniv a
isUniv               _  =  False

isEmpty :: Region → Bool
isEmpty           Empty  =  True
isEmpty (Outside     a)  =  isUniv a
```

```
isEmpty (Union     a b)  =  isEmpty a && isEmpty b
isEmpty (Intersect a b)  =  isEmpty a || isEmpty b
isEmpty (Translate _ a)  =  isEmpty a
isEmpty (Scale     _ a)  =  isEmpty a
isEmpty               _  =  False
```

Unlike in the previous example, where only `text` depends on `size`, the two definitions are *mutually recursive*, depending on each other. If we want to rewrite them using shallow embeddings, they also have to be encoded as a pair to call each other via **`fst`** and **`snd`**:

```
type Region = (Bool, Bool) -- (isUniv, isEmpty)

univ           =  (True,           False)
empty          =  (False,          True)
circle      _  =  (False,          False)
outside     a  =  (snd a,          fst a)
union     a b  =  (fst a || fst b, snd a && snd b)
intersect a b  =  (fst a && fst b, snd a || snd b)
translate _ a  =  (fst a,          snd a)
scale     _ a  =  (fst a,          snd a)
```

Generally speaking, using tuples to deal with dependencies is non-modular and awkward to use in that interpretations become tightly coupled. Adding more interpretations with dependencies, for example, would require a complete rewrite of the code: dependent interpretations have to be defined together with the interpretations that they depend on. More advanced encodings, such as tagless-final embeddings [Kiselyov 2010] or polymorphic embeddings [Hofer et al. 2008], provide ways to modularize *independent* interpretations into reusable units, but they cannot scale well to *dependent* interpretations and often have to resort to similar techniques using tuples like the code above. In Appendix A, we encode the two examples above in tagless-final embeddings and illustrate that they suffer from the same problems as shallow embeddings with respect to dependencies, both mutual and non-mutual.

Nested pattern matching. Another strength of deep embeddings is their ability to perform nested pattern matching, which is rather useful to inspect smaller constructs. Here we take an optimization that eliminates double negation for example. Nested pattern matching is used to check if an `Outside` is directly wrapped around an outer `Outside`. If so, both of them are eliminated; otherwise, `eliminate` is recursively called with inner constructs:

```
eliminate :: Region → Region
```

```
eliminate (Outside (Outside a))  =  eliminate a
eliminate (Outside           a)  =  Outside (eliminate a)
eliminate (Union           a b)  =  Union (eliminate a) (eliminate b)
eliminate (Intersect       a b)  =  Intersect (eliminate a) (eliminate b)
eliminate (Translate       v a)  =  Translate v (eliminate a)
eliminate (Scale           v a)  =  Scale v (eliminate a)
eliminate                    a   =  a
```

The common use of nested pattern matching poses a second challenge to domain-specific optimizations. Kiselyov [2010] discusses the "*seemingly impossible pattern-matching*" problem with nested patterns, which appear not directly encodable in tagless-final embeddings. Instead, he proposes that such algorithms can often be implemented in a tagless-final interpreter as context-dependent interpretations by essentially changing the algorithm. An extra parameter is added so the semantic type looks like `Ctx → repr`. However, in the general case, it is not obvious how to convert an algorithm with nested pattern matching into one based on contexts.

5.3 Compositional Embeddings

After looking back at the existing embedding techniques, this section shows that compositional programming enables a new form of DSL embedding called a *compositional embedding*. Compositional programming is a statically typed modular programming paradigm implemented by the CP language. For the details of the semantics of CP, we refer the reader to work by Zhang et al. [2021]. We have reimplemented CP as an in-browser interpreter, which can be directly run on a serverless web page. There are some small syntactic differences between our implementation and the original work, which we point out when necessary. Additionally, our implementation uses the call-by-need evaluation strategy, which optimizes the original call-by-name one.

Here, we only introduce the necessary language features of CP to illustrate compositional embeddings. With compositional embeddings, we can get many of the benefits of both shallow and deep embeddings without most of the drawbacks. We revisit the challenges illustrated in Section 5.2 using our approach in this section. A detailed comparison between compositional embeddings and various existing forms of embeddings, including shallow, deep, hybrid, and polymorphic embeddings, is presented at the end of this section.

5.3.1 Initial System, Revisited: Compositional Interfaces and Traits

As before, we start with an initial language interface inspired by Hudak [1998] and an abstract interpretation that calculates the syntactic size of a region:

```
type Vector = { x: Double; y: Double };

type HudakSig<Region> = {
  Circle:    Double → Region;
  Outside:   Region → Region;
  Union:     Region → Region → Region;
  Intersect: Region → Region → Region;
  Translate: Vector → Region → Region;
};

type Size = { size: Int };
sz = trait implements HudakSig<Size> ⇒ {
  (Circle      _).size = 1;
  (Outside     a).size = a.size + 1;
  (Union     a b).size = a.size + b.size + 1;
  (Intersect a b).size = a.size + b.size + 1;
  (Translate _ a).size = a.size + 1;
};
```

Since CP is also statically typed, a *compositional interface* (`HudakSig`) serves as the specification of the DSL syntax, playing a similar role to algebraic data types in the deep embedding. The type parameter `Region` is called the *sort* of the interface `HudakSig`. Sorts can be instantiated with different concrete types that represent different semantic domains. Every constructor, which is capitalized, must return a sort. Compositional interfaces are implemented by *traits* [Bi and Oliveira 2018; Ducasse et al. 2006], which serve as the unit of code reuse in CP. Traits play a similar role to classes in traditional object-oriented languages and are used to provide implementations for interpretations of the region DSL. To implement the abstract interpretation of the DSL, we first define the semantic domain type `Size`. Then we define a trait that implements `HudakSig<Size>`. In the body of the trait, we use *method patterns*, such as `(Circle _).size = 1`, to define the implementation. The method patterns used in the trait body allow us to define interpretations that look very much like the corresponding definition of `size` by pattern matching in Haskell in Figure 5.1b.

5.3.2 Linguistic Reuse and Meta-Language Optimizations

The example of horizontally aligned circles in Section 5.2.3 can be translated to CP as:

```
sharing Region = trait [self: HudakSig<Region>] ⇒ open self in {
  circles = letrec go (n: Int) (offset: Double): Region =
              if n == 0 then Circle 1.0
              else let shared = go (n-1) (offset/2.0)
                   in Union (Translate { x = -offset; y = 0.0 } shared)
                            (Translate { x =  offset; y = 0.0 } shared)
            in go 20 (pow 2.0 18);
};
```

Since the region DSL can have multiple interpretations, `circles` should be oblivious of any concrete interpretation but only refer to the language interface. To achieve this, we use *self-type annotations*. The trait `sharing` is parametrized by a type parameter `Region` and annotated with a self-type `HudakSig<Region>`. Like Scala traits, CP traits can be annotated with self-types to express abstract dependencies. This serves as an elegant way to inject region constructors. All of the constructors formerly declared in the interface are available through self-references, for instance, **`open`** `self` **`in`** `... Circle 1.0`. These constructors are *virtual* [Ernst et al. 2006; Madsen and Møller-Pedersen 1989]: they are not attached to a specific implementation.

With the self-type dependency on `HudakSig<Region>`, the static type checker of CP rejects a direct instantiation like **`new`** `sharing @Size` (`@` is the prefix for a type argument). Instead, we need to compose `sharing` with a trait implementing `HudakSig<Size>`, using a *merge operator* [Dunfield 2014]:

```
test = new sharing @Size , sz;            -- test = new ((sharing @Size) , sz);
test.circles.size           -- Above is equivalent with redundant parentheses.
```

The merge of the two traits (`sharing @Size` and `sz`) is still a trait. With self-type dependencies satisfied, the merged trait can be instantiated successfully, and we can call its method. According to call-by-need evaluation, the result of `shared` is stored for subsequent use once evaluated. Thanks to the linguistic reuse of the host-language **`let`** expressions and their optimizations, evaluating `circles.size` is able to exploit sharing and terminates almost immediately. This is similar to the Haskell approach in Section 5.2.3 using shallow embeddings.

5.3.3 Adding New Language Constructs

In compositional embeddings, language constructs can be modularly added. We first create a new language interface inspired by Hofer et al. [2008] and then define a trait implementing it:

```
type HoferSig<Region> = {
  Univ: Region;
  Empty: Region;
  Scale: Vector → Region
               → Region;
};
```

```
sz' = trait implements HoferSig<Size> ⇒ {
  (Univ     ).size = 1;
  (Empty    ).size = 1;
  (Scale _ a).size = a.size + 1;
};
```

To compose multiple interfaces, we use *intersection types*. To illustrate this, we create a repository of shapes that make use of language constructs from both interfaces:

```
type RegionSig<Region> = HudakSig<Region> & HoferSig<Region>;
repo Region = trait [self: RegionSig<Region>] ⇒ open self in {
  ellipse = Scale { x = 4.0; y = 8.0 } (Circle 1.0);
  universal = Union (Outside Empty) (Circle 1.0);
};
```

`RegionSig` is an intersection type combining the two interfaces into one. The definition of `repo` is similar to the definition of `sharing` in Section 5.3.2. It defines an ellipse, by using the `Scale` constructor from `HoferSig` and the `Circle` constructor from `HudakSig`. Similarly, it defines an essentially universal region by using constructors from both signatures.

5.3.4 Adding New Interpretations

Besides language constructs, it is also trivial to add new semantic interpretations in compositional embeddings. Just like creating a new interpretation function in deep embeddings, we only need to define a new trait implementing a compositional interface:

```
type Eval = { contains: Vector → Bool };
eval = trait implements RegionSig<Eval> ⇒ {
  (Circle         r).contains p = pow p.x 2 + pow p.y 2 ≤ pow r 2;
  (Outside        a).contains p = not (a.contains p);
  (Union        a b).contains p = a.contains p || b.contains p;
  (Intersect    a b).contains p = a.contains p && b.contains p;
  (Translate {..} a).contains p = a.contains { x = p.x - x; y = p.y - y };
  (Univ            ).contains _ = true;
  (Empty           ).contains _ = false;
  (Scale     {..} a).contains p = a.contains { x = p.x / x; y = p.y / y };
};
```

Similarly to Haskell, CP supports record wildcards `{..}`, which bring record fields into scope. We instantiate the sort with a function checking if a point resides in a region.

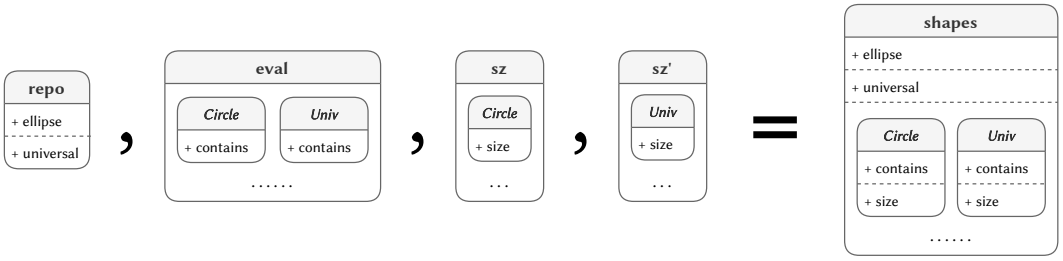


Figure 5.2: A visualization of nested trait composition.

Composing multiple interpretations. Now that we have multiple interpretations, we use *nested trait composition*, provided by the merge operator, to compose all interpretations:

```
shapes = new repo @(Eval&Size) , eval , sz , sz';  -- nested trait composition

"The elliptical region of size " ++ toString shapes.ellipse.size ++
(if shapes.ellipse.contains { x = 0.0; y = 0.0 }
 then " contains " else " does not contain ") ++ "the origin."
--> "The elliptical region of size 2 contains the origin."
```

We provide the final semantic domain (`Eval&Size`), as the type argument, for the trait `repo` and merge it with three other modularly defined traits. The trait composition is called *nested* [Bi et al. 2018] because not only different sets of constructors, but also different semantics nested in the same constructor are merged. In our case, both interpretations are available under the two sets of language constructs after nested composition, as visualized in Figure 5.2. Here, the language constructs and their semantics form a two-level hierarchy of traits. Such composition of whole hierarchies can be traced back to *family polymorphism* [Ernst 2001]. With this powerful mechanism of nested composition, we can easily develop highly modular *product lines* [Apel et al. 2013a] of DSLs, where DSL features can be composed *à la carte*.

5.3.5 Dependency Injection and Domain-Specific Optimizations

Now we illustrate CP's support for *modular dependent interpretations*. We skip the simpler example with `text` and focus on the more interesting example with `isUniv` and `isEmpty`. Similarly to the deep embedding in Haskell in Section 5.2.6, we define two compositional traits `chkUniv` and `chkEmpty`:

```
type IsUniv  = { isUniv:  Bool };
type IsEmpty = { isEmpty: Bool };
```

```
chkUniv = trait implements RegionSig<IsEmpty ⇒ IsUniv> ⇒ {
  (Univ         ).isUniv = true;
  (Outside     a).isUniv = a.isEmpty;
  (Union     a b).isUniv = a.isUniv || b.isUniv;
  (Intersect a b).isUniv = a.isUniv && b.isUniv;
  (Translate _ a).isUniv = a.isUniv;
  (Scale     _ a).isUniv = a.isUniv;
               _.isUniv = false;
};

chkEmpty = trait implements RegionSig<IsUniv ⇒ IsEmpty> ⇒ {
  (Empty        ).isEmpty = true;
  (Outside     a).isEmpty = a.isUniv;
  (Union     a b).isEmpty = a.isEmpty && b.isEmpty;
  (Intersect a b).isEmpty = a.isEmpty || b.isEmpty;
  (Translate _ a).isEmpty = a.isEmpty;
  (Scale     _ a).isEmpty = a.isEmpty;
               _.isEmpty = false;
};
```

The first thing to note is that the sort of `RegionSig` is instantiated with two types separated by a fat arrow (⇒). This is how we *refine* interface types for dependency injection. For example, `chkUniv` does not implement `RegionSig<IsEmpty>`, but we assume the latter is available. The static type checker of CP guarantees that `chkUniv` is later merged with another trait that implements `RegionSig<IsEmpty>`. That is why we can safely use `a.isEmpty` when implementing `(Outside a).isUniv`. The second trait `chkEmpty` is just the dual of `chkUniv`. It is hard to define such interpretations modularly in traditional approaches because they are dependent, but we make them compositional via dependency injection. Lastly, `_.isUniv` and `_.isEmpty` are *default patterns*, which compensate for the remaining constructors declared in the interface. For example, `_.isUniv = false` implies `Empty.isUniv = false` and `Circle.isUniv = false`. Note that in CP, for modularity, patterns are *unordered*, unlike functional languages like Haskell or ML. Therefore, default patterns are local to the traits where they are used. These patterns can be understood as a concise way to fill the missing cases in the local trait.

Another thing worth noting is that dependency injection in compositional embeddings is more modular than direct dependencies in deep embeddings because the former does not refer to concrete implementations. For example, in the compositional embedding above, `chkUniv` only requires that the dependent trait should implement the interface

`RegionSig<IsEmpty>`. However, in the deep embedding (see Section 5.2.6), `isUniv` depends on the specific `isEmpty` implementation.

Delegated method patterns. In some complicated transformations, nested pattern matching is required to inspect smaller constructs. Nested pattern matching is not directly supported in CP yet. However, there is a simple transformation that we can do whenever we would like to have nested patterns: we can delegate the inner tasks to other methods whereby only top-level patterns are needed. We call such a technique *delegated method patterns*.

Concerning the nested pattern matching in Section 5.2.6, we can implement it with two mutually dependent methods. Since delegated method patterns are not reused, the two methods are defined together for the sake of simplicity (but can always be separated into two traits):

```
type Eliminate Region = { eliminate: Region; delOutside: Region };

elim Region = trait [fself: RegionSig<Region>]
              implements RegionSig<Region⇒Eliminate Region> ⇒ open fself in {
  (Outside     a).eliminate = a.delOutside;
  (Union     a b).eliminate = Union a.eliminate b.eliminate;
  (Intersect a b).eliminate = Intersect a.eliminate b.eliminate;
  (Translate v a).eliminate = Translate v a.eliminate;
  (Scale     v a).eliminate = Scale v a.eliminate;
           [self].eliminate = self;

  -- delegated method patterns:
  (Outside a).delOutside = a.eliminate;
       [self].delOutside = Outside self.eliminate;
};
```

The translation from nested pattern matching to delegated method patterns is straightforward. Our code lifts nested patterns (`Outside (Outside a)`) to top-level delegated patterns. In this way, we do not change the underlying algorithm at all. This is in contrast with approaches like tagless-final embeddings that, as discussed in Section 5.2.6, seem unable to directly support nested patterns, requiring different algorithms to achieve the same goal. Though our approach is not as convenient as deep embeddings, it is noteworthy that we can just write the original algorithm *modularly*.

If we add new language constructs in the future, it is easy to augment the traits with more top-level patterns, but extending nested patterns is difficult because there is no

name to denote the *extension point*. With delegated method patterns, the method name `delOutside` offers an extension point for additional cases in the nested pattern matching. For instance, if we wish for an extension supporting a new kind of region and a special case for eliminating a region outside this kind of region (`eliminate (Outside (SomeRegion params))`), we can have a trait with a method pattern of the following form, modularly defined in another trait:

```
(SomeRegion params).delOutside = ...
```

Although we believe that the design of CP can be improved to better support similar logic to nested patterns, there are important differences between traditional (closed) forms of pattern matching and open pattern matching [Zhang and Oliveira 2020]. For instance, patterns in CP are unordered for compositionality, but the order of patterns matters in conventional pattern matching. Thus the design of improved language support for nested patterns in CP, which could make the use of delegated method patterns more convenient, requires further research and is left for future work.

Linguistic reuse after transformations. Before finishing the discussion of modular transformations, let us revisit CP's support for linguistic reuse: can we still have meta-language optimizations for free after complicated transformations? The answer is yes. To demonstrate this point, we can modify the definition of `circles` in Section 5.3.2 to have an inefficient `Outside (Outside (Circle 1.0))`. As usual, we compose all the necessary traits together to make sure no dependencies are missing:

```
test' = new sharing' @(Size & Eliminate Size) , sz , sz' , elim @Size;
test'.circles.eliminate.size
```

The evaluation terminates as quickly as before, meaning that meta-language optimizations are still performed even after a transformation.

We should remark that there are limits to the form of implicit sharing (and linguistic reuse) that shallow embeddings or compositional embeddings provide. For both shallow and compositional embeddings, implicit sharing will not work if the interpretation is a function, such as `contains`. In such cases, the sharing is still lost. Nevertheless, implicit sharing is an important feature that is often exploited in shallow DSLs. With compositional embeddings, we can take this idea further and make it work even after some transformations and optimizations have been applied. Moreover, it is also possible to adopt solutions with explicit sharing by modeling a `Let` construct in the DSL.

Table 5.1: A detailed comparison between different embedding approaches.

	Shallow	Deep	Hybrid	Poly.	Comp.
Transcoding free	●	●	○	●	●
Linguistic reuse	●	○	●	●	●
Language construct extensibility	●	○	◐[1]	●	●
Interpretation extensibility	○	●	●	●	●
Transformations and optimizations	○	●	●	◐[2]	◐[2]
Linguistic reuse after transformations	*n/a*	○	○	●	●
Modular dependencies	○	◐[3]	◐[3]	○	●
Nested pattern matching	○	●	●	○	◐[4]

[1] The extensibility of language constructs is limited or precludes exhaustive pattern matching.
[2] Transformations require some ingenuity and are sometimes awkward to write.
[3] Dependencies do not require code duplication but still refer to concrete implementations.
[4] Nested pattern matching is implemented as delegated method patterns.

5.3.6 A Detailed Comparison between Different Embedding Approaches

In Table 5.1, we compare existing embedding approaches in the literature with compositional embeddings. Shallow and deep embeddings have been discussed in detail in Section 5.2, and the table summarizes the points that we have made. Hybrid approaches [Jovanović et al. 2014; Rompf et al. 2012; Svenningsson and Axelsson 2015] employ both embeddings together. However, transcoding from shallow to deep is generally needed, as both shallow and deep implementations must coexist side by side. There is a clear boundary between the two parts: linguistic reuse is available only in the shallow part, while complex transformations are available only in the deep part. Therefore, it is still hard to exploit host-language features after transformations. In work by Svenningsson and Axelsson, we cannot add more constructs to the deeply embedded AST but can only extend shallowly embedded syntactic sugar. In Scala, if *open* case classes [Emir et al. 2007] are used for deep embeddings, ASTs are also extensible but do not ensure the exhaustiveness of pattern matching. Compositional embeddings offer a unified approach directly capable of all these functionalities.

Non-modular dependencies in modular embeddings. Polymorphic embeddings [Hofer et al. 2008], tagless-final embeddings [Carette et al. 2009; Kiselyov 2010], and object algebras [Oliveira and Cook 2012] (denoted as Poly. in the table) also provide a unified encoding where most advantages of shallow and deep embeddings are available. However, they cannot deal with dependencies modularly. Let us take tagless-final embeddings for exam-

ple, which can also be implemented in Haskell. We refer curious readers to Appendix A for the complete code.

In the tagless-final embedding, region constructors are defined in a type class instead of a closed algebraic data type, and `Size` can be modularly defined as an instance of the type class since it has no dependency:

```
class RegionHudak repr where
  circle    :: Double → repr
  outside   :: repr → repr
  union     :: repr → repr → repr
  intersect :: repr → repr → repr
  translate :: Vector → repr → repr
```

```
newtype Size = S { size :: Int }
instance RegionHudak Size where
  circle      _ = S 1
  outside     a = S $ a.size + 1
  union     a b = S $ a.size + b.size + 1
  intersect a b = S $ a.size + b.size + 1
  translate _ a = S $ a.size + 1
```

But how about the textual representation? As a first try, we might write:

```
newtype Text = T { text :: String }
instance RegionHudak Text where
  circle  r = T { text = "circular region of radius " ++ show r }
  outside a = T { text = "outside a region of size " ++ show a.size }
  ......
```

Unfortunately, we will get a type error concerning `a.size` because `a` has type `Text` and thus does not contain a field named `size`. Once we need operations that have some dependencies on other operations, we get into trouble!

A simple workaround is to pack the two operations together and duplicate the code of size calculation. We have already described this approach for shallow embeddings in Section 5.2.6, and the same workaround works for tagless-final embeddings:

```
data SizeAndText = ST { size :: Int, text :: String }
instance RegionHudak SizeAndText where
  circle  r = ST { size = 1, text = "circular region of radius " ++ show r }
  outside a = ST { size = a.size + 1
                 , text = "outside a region of size " ++ show a.size }
  ......
```

However, this is not modular since we have to duplicate code for size calculation, even though the size calculation is completely independent of the textual representation. If we have another operation that depends on `size`, we have to repeat the same code even again. This workaround is an *anti-pattern*, which violates basic principles of software engineering. Duplicating code every time we need a dependent interpretation is a serious problem since dependencies are extremely common. Nearly all non-trivial code will have dependencies. In functional programming, it is common to have functions defined by

Table 5.2: A briefer comparison regarding the three dimensions of the expression problem.

	FP	OOP	Poly.	Comp.
New constructs	○	●	●	●
New functions	●	○	●	●
Dependencies	●	●	○	●

● no code duplication ○ code duplication needed

pattern matching that call other functions defined by pattern matching, which is where an important form of dependencies appears. As shown in Appendix B, some workarounds that avoid code duplication in Haskell are possible, but this typically comes at the cost of more code and the use of advanced language features.

The third dimension of the expression problem. In essence, dependencies are a third dimension that is not discussed in the formulation of the expression problem by Wadler [1998]: whether dependencies require code duplication or not. If we take this into consideration, we can get a revised formulation of the expression problem, where instead of only two dimensions, we have a third dimension that evaluates the ability of an approach to deal with dependencies. With this extra dimension taken into account, what we get is Table 5.2. While tagless-final or other embeddings address the two original dimensions of extensibility modularly, they become non-modular with respect to dependencies, and programming with dependencies becomes significantly more awkward compared to conventional FP or OOP.

Note that we are not the first to observe the problem with dependencies. The problem above is known and discussed widely in the literature. In the context of polymorphic embeddings, Hofer et al. [2008] face the problem of dependencies and use the non-modular approach with tuples. Later Hofer and Ostermann [2010] adopt an extensible visitor pattern to improve modularity, but this results in much more boilerplate code. In his lecture notes on tagless-final interpreters, Kiselyov [2010] has a strong focus on discussing how to write non-compositional interpretations, which basically include operations with dependencies and nested pattern matching. He shows that non-compositional programs can often be rewritten as programs that use contexts instead. The problem of modular dependencies has been widely discussed within the context of object algebras. Most recently, Zhang and Oliveira [2017, 2020] propose solutions to the problem using meta-programming techniques. Zhang and Oliveira [2019] further explore modular solutions in both Scala and Haskell, but their approaches still require a lot of boilerplate code. In Scala, it is due to

a lack of language support for nested composition, whereas Haskell needs to encode dependencies via type classes to modularize dependent interpretations (see Appendix B for a similar solution). The drawbacks of the existing solutions above motivate the design of CP and compositional embeddings.

Transformations. It is widely believed that transformations and optimizations are much easier to write in deep embeddings [Jovanović et al. 2014; Scherr and Chiba 2014]. We agree that it requires some ingenuity and is sometimes awkward to write complex transformations in compositional embeddings, but Kiselyov [2010] has demonstrated that several challenging transformations can be written with tagless-final embeddings. Compositional programming is a generalization of such forms of embeddings, and therefore all the transformations are possible with tagless-final embeddings are also possible in CP. Even for structurally non-local transformations, we can instantiate a sort similarly to the `Reader` or `State` monad in Haskell, passing down and possibly updating a context when traversing the AST. In Figure 5.3, we show an example of common subexpression elimination based on hash consing, which is inspired by Kiselyov [2011]. Here the sort is instantiated with type `HashCons`, which imitates the `State` monad by taking a hash table of `Nodes` as a parameter and returning a pair of `NodeID` and the updated hash table. The method pattern for `Add` demonstrates how to chain the "monadic" computations. The implementation of `hashcons` is omitted for brevity, but the complete code can be found online.[2] Both our code and Kiselyov's are not fully modular with respect to type `Node`: his definition of `Node` is a *closed* algebraic data type; our definition of `Node` cannot be extended either, because nested composition for recursive types is currently not supported, which relies on future theoretical progress of reconciling distributive subtyping and recursive types. Nevertheless, our approach is less entangled than tagless final embeddings since we can handle dependent transformations modularly.

A final note is that if the context (e.g. `[Node]`) needs extending during the extension of the AST, CP can still handle it modularly with the help of *polymorphic contexts* [Zhang et al. 2021], though this technique is not employed in Figure 5.3 to keep the code simple. The basic idea is to make the definition of `HashCons` polymorphic:

```
type HashCons Ctx = { hc : Ctx → Pair NodeId Ctx };
hc (Ctx * [Node]) = trait implements ExpSig<HashCons ([Node] & Ctx)> ⇒ ...
```

In this way, `hc` can make use of the `[Node]` part while leaving the unknown `Ctx` part open for extension.

[2] https://plground.org/nobody/CSE

```
type ExpSig<Exp> = {
  Var : String → Exp;
  Lit : Int → Exp;
  Add : Exp → Exp → Exp;
};

type HashCons = { hc : [Node] → Pair NodeId [Node] };

hc = trait [self : NodeSig<Node>] implements ExpSig<HashCons> ⇒ open self in {
  (Var s).hc m = hashcons (NVar s) m;
  (Lit n).hc m = hashcons (NLit n) m;
  (Add e1 e2).hc m = let p1 = e1.hc m in
                     let p2 = e2.hc p1.snd in
                     hashcons (NAdd p1.fst p2.fst) p2.snd;
};
```

Figure 5.3: Common subexpression elimination implemented in CP.

5.4 ExT: A DSL for Document Authoring

We have presented the advantages of compositional embeddings using a relatively small DSL. One may wonder if our approach scales to larger DSLs. As our answer to this question, we developed a larger DSL for document authoring called ExT.

ExT applies compositional embeddings to a document DSL, inspired by LaTeX and Scribble [Flatt et al. 2009]. We have also done minor syntactic extensions to the CP language to support writing documents more naturally. Since documents usually involve large portions of text, it would be awkward to write such portions of text using string literals adopted by most programming languages. CP does not yet have facilities for syntactic extension that other languages (e.g. Racket, Haskell, or Coq) offer, so we have extended the parser directly. Thus, CP parses some document-specific syntax and desugars it to compositionally embedded fragments during parsing. Such a generative approach is similar to Racket macros [Ballantyne et al. 2020] and Template Haskell [Sheard and Peyton Jones 2002], which provide more flexible facilities for performing such desugaring. Nevertheless, the surface syntax of ExT is essentially a set of lightweight syntactic sugar for its underlying compositional embeddings (similar, for instance, to Scribble in Racket).

5.4.1 Design Goals and Non-Goals

There are already a large number of document DSLs, so why shall we create yet another one? We explain why by identifying important design goals and non-goals of ExT.

A document language for the web. A first goal is to have a lightweight but powerful document language tailored mostly for the web. We view EXT as an alternative to Wikitext [MediaWiki 2003], the markup language used by Wikipedia, which shares similar goals. Similar to Wikipedia pages, users can directly edit EXT documents on a web page. But different from Parsoid [MediaWiki 2011], the official Wikitext parser that runs on the server side, our implementation can directly render documents on the browser side.

General-purpose computation. The majority of existing document DSLs (e.g. CommonMark [MacFarlane 2014], Textile [Allen 2002], and reStructuredText [Goodger 2002]) have a fixed set of language features and do not provide general-purpose computation. For instance, we may want to compute a table from some data, like a table listing the largest cities in descending order of population. For such kinds of tasks, it is useful to have mechanisms for general-purpose computation that allow us to *compute* documents rather than manually write them.

Although some other document DSLs provide language constructs that enable some computation, they still have significant limitations. For instance, Wikitext provides templates, which play a role similar to functions in conventional programming languages. However, templates cannot perform arbitrary computation as CP does. The documentation says,[3] "A template can call itself but will stop after one iteration to prevent an infinite loop." In other words, templates by themselves are incapable of recursion or loops, but in practice, loops are often needed in Wikipedia. For instance, `{{loop}}` has been used on approximately 99,000 pages.[4] A Lua extension was introduced to bypass such restrictions, and the widely-used template `{{loop}}` invokes `string.rep` in Lua. Instead of handing it over to an external language, CP directly supports functions, and thus general-purpose computation is easily done in EXT, including recursion.

Static type checking. In contrast to many document languages, CP and EXT are statically type checked. Static type checking enables us to detect some errors that may arise when processing documents. In applications like wikis, it is very frequent that we need some form of structured data. States or cities, for example, may have infoboxes associated with them that list several pieces of data, such as areas, populations, official languages, etc. Using simple text to store such data can easily lead to simple errors. In addition, if we want to enable computed information as previously described, then it is useful to have a type system that allows us to write functions that take a well-defined form of structured

[3] The sentence is extracted from https://en.wikipedia.org/wiki/Wikipedia:TEMPLATE_LOOP.
[4] The number of transclusions (99,000 as of July 2022) is shown at https://en.wikipedia.org/wiki/Template:Loop.

data. Most document languages that we know of, including Wikitext, LaTeX, and Scribble, are not statically typed. We will discuss some examples where static typing is helpful in ExT in Section 5.4.2.

Extensible meta-programming and customizability. A benefit of modeling a DSL as an algebraic data type or a compositional interface is that we can easily write meta-functions over the abstract syntax of the DSL. For instance, in the context of documents, such meta-functions could include rendering to various backends (such as LaTeX or HTML) or computing a table of contents based on the document tree. In many external DSLs, such meta-functions are typically much harder to write, requiring us to directly modify the implementation. However, by embedding a DSL, the host language can act as a meta-language for the DSL and allow us to easily write meta-programs, for instance, as functions to process the AST by pattern matching.

In addition, the extensibility of compositional embeddings enables a form of extensible meta-programming: not only can we write meta-functions, but those meta-functions can be extended later to cater for new language constructs modularly defined by users. This enables DSL users to *modularly* extend the basic document language to add their own extensions, including infoboxes, vector graphics, etc. While Scribble does provide good facilities for syntactic extension and meta-programs, the document structure is fixed [Flatt 2009], so the DSL is not fully extensible. In short, ExT is designed to be extensible and highly customizable by users.

Non-goals. Currently, our implementation of CP is a fairly naive call-by-need interpreter, so the performance of ExT is not competitive with well-established tools like LaTeX. Furthermore, unlike LaTeX, which has extensive libraries developed over many years, there are no third-party libraries for ExT. Thus, ExT is *not* yet production-ready. Another non-goal is security. Since we never sanitize user-generated HTML, it is easy to perform JavaScript code injection on our website. It is off-topic to consider how to ensure web security here.

5.4.2 Syntax Overview

The syntax of ExT is designed for easy document authoring, as shown in Figure 5.4. It is reminiscent of LaTeX, but still has significant differences. The similarity is that plain text can be directly written while commands start with a backslash. But we choose different brackets for command arguments:

```
\Section[
  Welcome to
  \Href("https://plground.org")[PLGround]!
]
\Bold[CP] is a \Emph[compositional]
programming language. \\\\
There are \(1+1+1+1) key concepts in CP:
\Itemize[
  \Item[Compositional interfaces;]
  \Item[Compositional traits;]
  \Item[Method patterns;]
  \Item[Nested trait composition.]
]
```

(a) ExT code.

Welcome to PLGround!

CP is a *compositional* programming language.

There are 4 key concepts in CP:

- Compositional interfaces;
- Compositional traits;
- Method patterns;
- Nested trait composition.

(b) Rendered document.

Figure 5.4: A sample document illustrating ExT syntax.

- `\Cmd[...]` encloses document arguments. Square brackets indicate that a sequence of nested elements is recursively parsed. (Most commands in Figure 5.4 are in such a form, for example, `\Section` and `\Bold`.)

- `\Cmd(...)` encloses positional arguments. Parentheses indicate that the whole construct is desugared to function application. (`\Href` in Figure 5.4 is an example.)

- `\Cmd{...}` encloses labeled arguments. Braces indicate that these arguments are wrapped in a record. (As mentioned in Section 5.4.2, `\Line{ x1 = 0.0; y1 = 0.0; x2 = 4.6; y2 = 4.8 }` uses this form to distinguish different attributes.)

A command can be followed by an arbitrary number of different arguments in any order. This syntax is more flexible than the fixed form `@op[...]{...}` in Scribble. Although both arguments are optional in Scribble, they can neither occur more than once nor be shuffled. In addition, there are two more useful ExT constructs. One is string interpolation with the form `\(...)`, which has exactly the same syntax as the Swift programming language. We can see an example of string interpolation in Figure 5.4: `\(1+1+1+1)`. It will be desugared to `Str (toString (1+1+1+1))`. This example will execute the code, which computes "4", and its string form will be interpolated in the document. The other construct is double backslashes (`\\`), a shorthand for newline, which is borrowed from LaTeX. In ExT, it is equivalent to the `\Endl` command. There is also an example in Figure 5.4, which inserts a newline character between the first and second sentences in the body.

Desugaring. As mentioned earlier, all ExT constructs are desugared to normal CP code. For example, the following two expressions are equivalent (the document DSL is enclosed within backticks):

```
`\Entry("id"){ hidden = true; level = 1 }[This entry is \Emph[hidden]]`
-- is desugared to:
Entry "id" { hidden = true; level = 1 } (Comp (Str "This entry is ")
                                             (Emph (Str "hidden")))
```

Here, `Comp` and `Str` are two special commands that are assumed to be implemented by library authors. `Comp` is used for composing an AST of document elements, while `Str` is for plain text.

Specification. The grammar of ExT can be summarized in whole as follows (`<arg>*` means that `<arg>` may occur zero or more times):

```
<doc> ::= <str> | "\" <esc>
<esc> ::= <cmd> <arg>* | "(" <exp> ")" | "\"
<arg> ::= "[" <doc> "]" | "(" <exp> ")" | "{" <rcd> "}"

<str> ::= text without "\" and "]"
<cmd> ::= identifier
<exp> ::= expression
<rcd> ::= record fields
```

5.4.1 Extensible Commands with Multi-Backend Semantics

ExT is so flexible that only a few commands are really compulsory. Additional commands and their semantics can be extended as needed. At the very beginning, we only need to define the aforementioned three special commands:

```
type DocSig<Element> = {
  Comp: Element → Element → Element;
  Str:  String → Element;
  Endl: Element;
};
```

However, for rich text in real-world documentation, these commands are not sufficient for various document elements. Adding new commands and their semantics is not hard at all in compositional embeddings, like what we have done in Section 5.3.3 and Section 5.3.4. Language interfaces can be separately declared and then intersected with each other. Be-

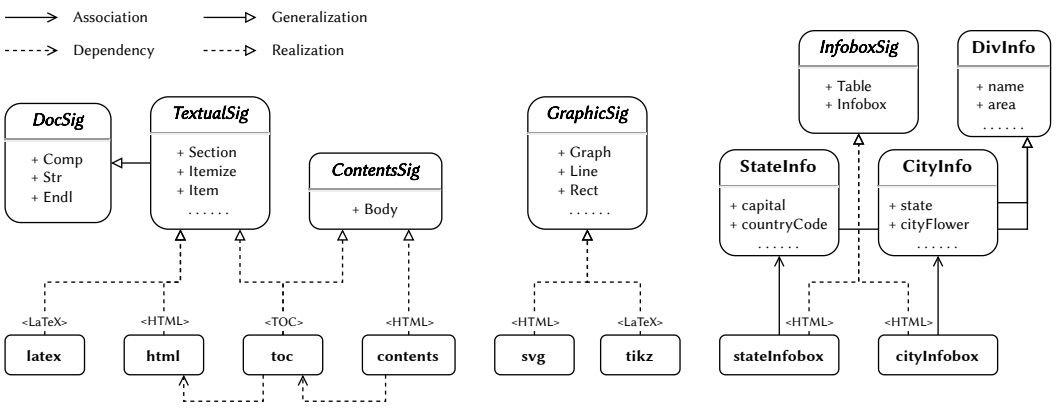


Figure 5.5: A simplified diagram of ExT components.

sides extensible commands, their semantics are also retargetable. We can modularly create different traits for different backends, such as HTML and LaTeX:

```
type HTML = { html: String };
html = trait implements DocSig<HTML> ⇒ {
  (Comp l r).html = l.html ++ r.html;
  (Str    s).html = s;
  (Endl    ).html = "<br>";
};

type LaTeX = { latex: String };
latex = trait implements DocSig<LaTeX> ⇒ {
  (Comp l r).latex = l.latex ++ r.latex;
  (Str    s).latex = s;
  (Endl    ).latex = "\\\\";
};
```

The components of ExT. We have implemented several extensions of ExT, as shown in Figure 5.5. In the upper part of the diagram, the capitalized names in a bold italic font (e.g. ***DocSig***) are compositional interfaces, while those in a bold upright font (e.g. **DivInfo**) are normal types. The smaller boxes at the bottom of the diagram are compositional traits. They implement different interfaces with the sorts specified on the dashed arrows, (e.g. <HTML>). Specifically, `TextualSig` adds a set of common commands for rich text, such as `Section`, `Itemize`, `Item`, etc. It extends `DocSig` and targets both HTML and LaTeX. `GraphicSig` is used to draw vector graphics (including lines, rectangles, circles, etc.), which are supported in HTML and LaTeX via SVG and Ti*k*Z respectively. `InfoboxSig` targets only HTML

and adds Wikipedia-like infoboxes (containing, for instance, information about areas and populations of states or cities). All these compositional interfaces and traits can be combined to form a heterogeneous composition. Notably, the implementation of `ContentsSig`, which is used to generate a table of contents, introduces some non-trivial dependencies (see Section 5.5.1).

5.4.2 Static Typing

A major difference of ExT from other document DSLs is that it is statically type checked. With static typing, potential type errors can be detected ahead of time, saving users' time for debugging. For example, when drawing a line using SVG, we need to use the `<line>` element with four numeric coordinates (`x1`, `y1`, `x2`, and `y2`). However, since additional information of an SVG element is represented as XML attributes, all of the coordinates have to be quoted as strings instead of numbers. Only after rendering it with a web browser and opening developer tools can one check if attributes are valid. In ExT, we model the line construct like this:

```
type GraphicSig<Graphic> = {
  Line: { x1: Int; y1: Int; x2: Int; y2: Int } → Graphic;
  -- and more constructors
};
```

Before rendering lines via SVG, the types of arguments are checked against the language interface, and invalid values are rejected in advance. Note that the error messages are reported in terms of the ExT surface syntax, which is more friendly to users than some meta-programming techniques, where errors are reported in terms of the generated code.

When modeling infoboxes and bibliography in Section 5.5.1, we also make use of data types to represent structured information. For example, we use `DivInfo` to model common information required by all political divisions, as well as two specific types for states and cities respectively, which extend `DivInfo` using intersection types:

```
type DivInfo = {
  name:       String;
  area:       Int;
  population: Int;
  languages: [String];
};
```

```
type StateInfo =
       DivInfo & {
  countryCode: String;
  religions:  [String];
  -- and more fields
};
```

```
type CityInfo =
      DivInfo & {
  cityFlower: String;
  timeZone:   String;
  -- and more fields
};
```

In this case, there is quite a bit of structure in the data. It statically describes what fields are allowed and what types these fields have.

5.5 Evaluation of Compositionally Embedded ExT

In this section, we describe three applications of CP, compare them with alternatives in other document languages, and evaluate the use of compositionally embedded ExT for the three applications.

5.5.1 Minipedia

Minipedia is a mini document repository of states and cities, which reconstructs a small portion of Wikipedia. Minipedia currently contains pages of the world's ten smallest countries and their capitals, as well as a structured data file storing all facts about them used for infoboxes. There are also pages consisting of sorted tables of microstates by area or population, which are computed using the information stored in the data file. Minipedia comprises over 4,000 lines of code in total, most of which is accounted for by document pages. Among the rest, the data file and ExT libraries account for around 300 and 200 lines of code, respectively.

Drawbacks of Wikipedia. Compared to our approach, Wikipedia and its markup language Wikitext have several drawbacks, as partly mentioned in Section 5.4.1:

- All data in Wikitext is in string format. Wikitext does not differentiate data types like `Int`, `Bool`, etc. This makes type errors remain uncaught in Wikipedia infoboxes. For example, in the infobox of a country, the population field can be changed to a non-numeric meaningless value, and Wikipedia will not warn the editor at all.
- Statistical data is manually written on every Wikipedia page. This can easily result in data inconsistency for the same field across different pages, and even for different places on the same page. For example, the population size on a country page is often different from that in a statistical page listing all countries by population, owing to different data sources.
- Wikitext provides user-defined templates as a reusable unit of document fragments. They can be parametrized and play a similar role to functions. However, they do not natively support general-purpose computation like recursion. Along with the Lua extension and parser functions, Wikitext offers some computational power to Wikipedia documents, but it is not as easy to use as ExT is.

Type safety and data consistency. Minipedia has a structured data file containing the information of states and cities, each piece of which is modeled as a record. The record

types have been shown in Section 5.4.2. Every record for states or cities is type checked against their corresponding types and collected in two arrays in the data file:

```
tuvalu = {
  name       = "Tuvalu";
  area       = 26;
  population = 10645;
  -- and more fields
} : StateInfo;
states = [tuvalu; {- and more -}];
```

```
funafuti = {
  name       = "Funafuti";
  area       = 2;
  population = 6320;
  -- and more fields
} : CityInfo;
cities = [funafuti; {- and more -}];
```

Such a centralized data file forces different documents to read from the same data source and prevents data inconsistency. Not only infoboxes but also computed documents like sorted lists of countries can be created using the information from the data file. Moreover, if the area of Tuvalu, for example, is assigned a string value, there will be a type error before rendering.

Writing Minipedia pages in ExT. Writing Minipedia documents is enabled by the ExT libraries shown in Figure 5.5. Different features of ExT are defined in different libraries, and we can import whatever libraries we want when writing a document. For Minipedia, we only need HTML-related libraries. There are two modes for document authoring in Minipedia: "doc-only" and "program". In the "doc-only" mode, with required libraries specified, documents are written directly in ExT without any wrapping code in CP, as shown in Figure 5.4. In the "program" mode, a document is created programmatically in a mixture of CP and ExT code. This mode makes general-purpose computation easier to write in a document.

For example, a sorted table of the smallest countries by area demonstrates general-purpose computation in the "program" mode. The program begins with an **open** directive, which imports definitions from the specified libraries. We sort the array of countries imported from `Database` using a predefined function `sort` and then iterate it to generate ExT code. As explained in Section 5.3.2, we use self-type annotations to inject dependencies on the library commands. The intersection type `DocSig<T>&TableSig<T>` allows ExT commands from both compositional interfaces to be used in the trait `doc`:

```
open LibDoc;
open LibTable;
open Database;

sortedStates = sort states;
```

No.	Country	Area (km²)
1	Vatican City State	0
2	Principality of Monaco	2
3	Republic of Nauru	21
4	Tuvalu	26
5	Republic of San Marino	61
6	Principality of Liechtenstein	160
7	Marshall Islands	181
8	Federation of Saint Christopher and Nevis	261
9	Republic of Maldives	300
10	Republic of Malta	316

(a) The smallest countries by area.

(b) Table of contents of the Tuvalu page.

Figure 5.6: Screenshots of Minipedia pages.

```
doc T = trait [self: DocSig<T>&TableSig<T>] ⇒ open self in {
  table = letrec rows (i: Int): T =
      if i == 0 then `\Trow[
        \Theader[No.]   \Theader[Country]      \Theader[Area (km^2)]
      ]` else let state = sortedStates!!(i-1) in `\rows(i-1) \Trow[
        \Tdata[\(i+1)]  \Tdata[\(state.name)]  \Tdata[\(state.area)]
      ]` in
  `\Tbody[ \rows(#sortedStates) ]`;
};
document = new doc @HTML , html , table;
document.table.html
```

A table of the smallest countries by area rendered by the code above is shown in Figure 5.6a.

Adding a table of contents (TOC). We have introduced how to write Minipedia pages using EXT libraries, but now let us go back and see how we implement a EXT library. The library of TOC adds a new command `\Body[...]`, which prepends a TOC to the document body. The difficulty here is that HTML rendering depends on the TOC, whereas the TOC in turn depends on the HTML rendering of section (and sub^{n}section) titles. As we have demonstrated in Section 5.3.5, we can modularly tackle such complex dependencies by refining the interface types:

```
type ContentsSig<Element> = { Body: Element → Element };
```

```
type TOC = { toc: String };

contents = trait implements ContentsSig<TOC ⇒ HTML> ⇒ {
  [self]@(Body e).html = self.toc ++ e.html;
};
toc = trait implements TextualSig<HTML ⇒ TOC> & ContentsSig<TOC> ⇒ {
  (Comp        l r).toc = l.toc ++ r.toc;
  (Section       e).toc = listItem 0 e.html;
  (SubSection    e).toc = listItem 1 e.html;
  (SubSubSection e).toc = listItem 2 e.html;
  (Body          e).toc = "<ul>" ++ e.toc ++ "</ul>";
                  _.toc = "";
};
```

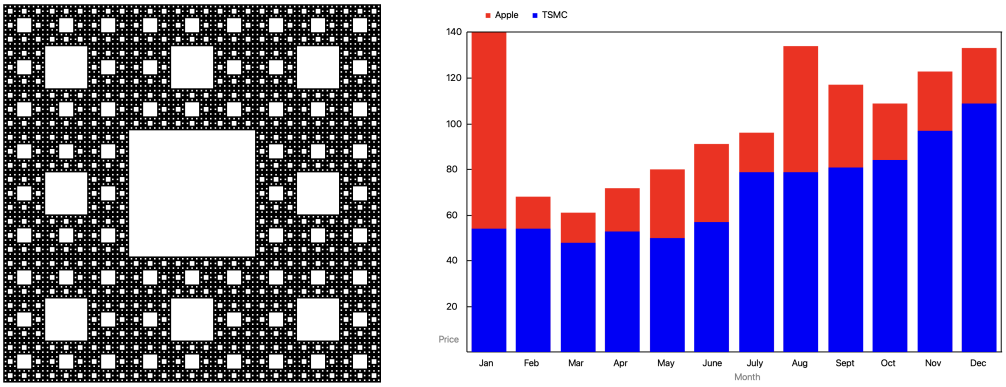
With `TOC` specified in `ContentsSig<TOC ⇒ HTML>` and the self-reference specified in `[self]`, we can call `self.toc` to generate a TOC when rendering HTML. Similarly, we can call `e.html` when generating the TOC since the interface type is refined. A minor remark is that `listItem` is an auxiliary function to generate an appropriate `<li>` wrapping for a given nesting level.

An example of a TOC is shown in Figure 5.6b. The TOC is generated automatically from the section titles in the Tuvalu page. The TOC is a compositional part of the document and is not hard-coded in the document body. This is a good example of how to modularly handle dependencies in compositional embeddings.

5.5.2 Fractals and Sharing

In the second application, we briefly introduce how to implement computational graphics like fractals in ExT with the help of linguistic reuse. Fractal drawing is all about repeating the same pattern. This drives us to exploit meta-language sharing to avoid redundant computation. We have implemented several well-known fractals, such as the Koch snowflake, the T-square, and the Sierpiński carpet. For the sake of simplicity, we take the Sierpiński carpet as an example:

```
fractal T C =
  trait [self: DocSig<T> & GraphicSig<T><C> & ColorSig<C> & Draw T C]
  implements Draw T C ⇒ open self in {
    draw {..} =
      let center = Rect { x = x + width/3; y = y + height/3;
                          width = width/3; height = height/3; color = White} in
      if level == 0 then center else
```



(a) Sierpiński carpet.

(b) A bar chart of stock prices.

Figure 5.7: Screenshots of fractals and charts.

```
    let w = width/3 in let h = height/3 in let l = level-1 in
    let shared = draw { x = x; y = y; width = w; height = h; level = l } in
    `\Group(id)[
      \shared
        \Translate{ x = w; y = 0 }(shared)
          \Translate{ x = 2*w; y = 0 }(shared)
      \Translate{ x = 0; y = h }(shared)
        \center
          \Translate{ x = 2*w; y = h }(shared)
      \Translate{ x = 0; y = 2*h }(shared)
        \Translate{ x = w; y = 2*h }(shared)
          \Translate{ x = 2*w; y = 2*h }(shared)
    ]`
};
```

We make variables shared via the **let** expressions in CP. There are many uses in the code above but, among them, `shared` accelerates fractal generation the most. The variable `shared` is later used eight times to constitute the repeating patterns in the Sierpiński carpet. `\Translate` is a command declared in `GraphicSig` for geometric translations. With automatic linguistic reuse in EXT, we only need to generate a pattern once instead of repeating it eight times. Besides the efficiency in generating fractals, our implementation of `\Translate` makes use of the <use> element in SVG to avoid the exponential growth of output. Therefore, the duplication of SVG elements is also eliminated. These optimizations are available for free thanks to the linguistic reuse in compositional embeddings.

A screenshot of the Sierpiński carpet with `level = 4` is shown in Figure 5.7a.

5.5.3 Customizing Charts

In the last application, we illustrate how line charts and bar charts are modularly rendered using external data. Charts can be rendered into a document using the ExT language and its support for vector graphics. What is interesting about charts is that there are many alternative ways to present charts and customize them. The flexibility of the CP language, in terms of modularity and compositionality, is then very useful here. We will show how to adapt traditional object-oriented design patterns for compositional embeddings when modeling graphic components.

Charting stocks. Taking stock prices as an example, we have a separate file containing data from big companies, as well as a few configuration items for chart rendering. The configuration includes the choice between lines and bars, whether to show borders or legends, what labels to display, and so on. Serving as an infrastructure, a base chart is created with the drawing functions for the caption and axes:

```
type Base = { caption: HTML; xAxis: HTML; yAxis: HTML };
baseChart (data: [Data]) = trait implements Base ⇒ {
  caption = `...`;  xAxis = `...`;  yAxis = `...`;
};
```

The simplified code above shows a rough sketch, where a base trait takes data as its parameter and implements the base interface. However, it does not implement the primary rendering function. As we have two options to visualize stock prices, we leave the choice to the Strategy pattern [Gamma et al. 1995].

Strategy pattern. Since the configuration is unknown until run time, the rendering strategy cannot be hard-coded in the base trait. Instead, we define two strategy traits implementing the rendering, respectively for lines and bars. They constitute two variants of the base chart. It is not the programmer but the configurator that decides which variant of charts should be rendered. In the original Strategy pattern, a chart object delegates to a desired strategy object, to which its mutable field points. That mutable field can be changed from clients who use the chart. In CP, we just need to merge the base trait with the strategy we want, without the need for mutable references:

```
type Render = { render: HTML };
lineStrategy = trait [self: Base] implements Render ⇒ open self in {
  render = let lines = ... in `\xAxis \yAxis \lines \caption`
};
barStrategy = trait [self: Base] implements Render ⇒ open self in {
```

```
  render = let bars = ... in `\xAxis \yAxis \bars \caption`
};
chart = baseChart data , if config.line then lineStrategy else barStrategy;
```

After merging, `chart` is still a *reusable trait*. This is impossible in traditional object-oriented languages, like Java, because they lack a dynamic way to compose classes at run time. In contrast, such a dynamic trait composition is ubiquitous in CP. It is also easy to add a new strategy, like switching to pie charts, and merge the base trait with the new one instead. If a client forgets to pick any strategy, the type checker will notify them because the trait types before and after merging are different. Moreover, another advantage of our approach over the original strategy pattern is that the self-type annotations make methods in the base trait accessible to strategy traits. So `caption`, `xAxis` and `yAxis` are directly shared with strategies, requiring no extra effort to pass them as arguments when delegating. This remedies the "communication overhead between Strategy and Context" caused by the original Strategy pattern [Gamma et al. 1995].

Decorator pattern. Besides the Strategy pattern, the Decorator pattern [Gamma et al. 1995] is also adapted for CP. Since decorations are also determined by external configuration at run time, we need a modular way to add additional drawing processes to the chart trait dynamically. When multiple decorators are enabled, their functionalities should be added. In the original Decorator pattern, the decorator class should inherit the chart class and be instantiated with a reference to a chart object. The decorator will store the chart object as a field and forward all methods to it. In concrete decorators, the rendering function will be overridden to add decorations. The decorator pattern sounds complicated in traditional object-oriented programming, but the same goal can be achieved easily in CP using the *dynamic inheritance* provided by CP:

```
type Chart = Base & Render;
borderDecorator (chart: Trait<Chart>) = trait [self: Chart]
                                        implements Chart inherits chart ⇒ {
  override render = let sr = super.render in let border = ... in `\sr \border`
};
legendDecorator (chart: Trait<Chart>) = trait [self: Chart]
                                        implements Chart inherits chart ⇒ {
  override caption = let legends = ... in `\legends`
};
chart' = if config.border then borderDecorator chart else chart;
chart'' = if config.legend then legendDecorator chart' else chart';
```

A decorator takes a chart trait as the argument and creates another trait inheriting it dynamically. In the implementation, we can override any methods as usual, and static type safety is still guaranteed. It is worth noting that, in `legendDecorator`, only `caption` is overridden while `render` is just inherited. In the original DECORATOR pattern, `render` will never be conscious of the overridden `caption` since the decorator hands over control to `chart` after forwarding `render`. This is partly why Gamma et al. [1995] warn readers that "a decorator and its component aren't identical". However, in CP, `chart.render` can access the overridden `caption` through the late-bound self-reference. This solution makes our code clean and easy to maintain.

A bar chart of stock prices with borders and legends is shown in Figure 5.7b.

True delegation via trait composition. The chart application shows how other aspects of the modularity of CP are helpful to create highly customizable DSLs. We have shown how CP adapts and simplifies object-oriented design patterns. In general, we avoid complicated class hierarchies in the original versions and replace them with relatively simple trait composition. It showcases that we can get rid of verbose design patterns if a proper language feature fills in the gap. In both patterns, component adaptation is needed at run time, so delegation is at the heart of the challenges.

5.6 Conclusion

We have presented compositional embeddings and shown their advantages compared to existing embedding techniques in terms of modularity and compositionality. CP's support for dependency injection, pattern matching, and nested composition makes *seemingly* non-compositional interpretations feasible in compositional embeddings. Thus, dependent interpretations and complex transformations can be modularly defined. These virtues of compositional embeddings lay down the foundation for the ExT DSL. The viability of compositionally embedded ExT has been evaluated with three applications: *Minipedia* shows its extensibility, type safety, data consistency, and ability to modularly handle dependencies; *fractals* show that general-purpose computation and linguistic reuse are both available; finally, *charts* show a simpler approach to delegation compared to object-oriented design patterns.

Part IV

Compilation of Compositional Programming

6 Key Ideas of the CP Compiler

In this part, we propose an efficient compilation scheme that translates merges in CP to extensible records, where types are used as record labels to perform lookup on merges. We first introduce the key ideas under the hood of the CP compiler and describe why and how to compile CP in this way. We also discuss the major challenges that we had to overcome. Although our implementation targets JavaScript, the design can be adapted to any other language that supports some kind of extensible records. We refer the reader to Chapter 7 for a formal description of our compilation scheme and Chapter 8 for the details of our implementation targeting JavaScript. Chapter 9 conducts an empirical evaluation of the CP compiler.

6.1 Dunfield's Elaboration Semantics

In previous work by Dunfield [2014] and its follow-up work by Oliveira et al. [2016], the semantics of the merge operator is well studied. According to the *non-deterministic* operational semantics given by Dunfield, a merge $48 \,,\, \mathbf{true}$ may reduce to 48 or **true**; both are valid reductions. However, such reductions may not preserve types. For instance, in a context like $(48 \,,\, \mathbf{true}) - 2$, the merge should reduce to an integer. Alternatively, Dunfield proposes an elaboration semantics into a target calculus with pairs, which is also used by Oliveira et al. Within this framework, an intersection type $A \,\&\, B$ is elaborated into a product type $A \times B$, and a merge $e_1 \,,\, e_2$ is elaborated into a pair $\langle e_1, e_2 \rangle$. While an elaboration to pairs offers a simple model for merges, it also imposes significant runtime overhead. We identify three limitations in previous work.

Indirect coercions. Following the elaboration model to pairs, $(48 \,,\, \mathbf{true}) - 2$ should be elaborated into $\langle 48, \mathbf{true} \rangle.\mathrm{fst} - 2$. That is, we need to select the first element from the elaborated pair to obtain a well-typed expression. Merges, due to their flexible nature, do not have an explicit elimination form. Then how can we determine where to insert ".fst"?

In a type-directed elaboration, we can generate coercion functions according to subtyping judgments in the typing derivation. A rule DTyp-Sub can be found in previous work.

$$\textsc{DTyp-Sub}\quad \frac{\Gamma \vdash e \Rightarrow A \rightsquigarrow \epsilon \qquad A <: B \rightsquigarrow c}{\Gamma \vdash e \Leftarrow B \rightsquigarrow c\,\epsilon} \qquad \textsc{Ela-Sub}\quad \frac{\Gamma \vdash e \Rightarrow A \rightsquigarrow \epsilon_1 \qquad \epsilon_1 : A <: B \rightsquigarrow \epsilon_2}{\Gamma \vdash e \Leftarrow B \rightsquigarrow \epsilon_2}$$

The rule DTyp-Sub means that if a source term e is inferred to have type A and elaborated into a target term ϵ, and the subtyping judgment $A <: B$ implies the coercion function c, then e can be checked against type B and elaborated into $c\,\epsilon$. In the aforementioned example, the merge of type $\mathbf{Int} \,\&\, \mathbf{Bool}$ is cast to type $\mathbf{Int}$. Therefore, a coercion function should be implicitly inserted for the subtyping relation $\mathbf{Int} \,\&\, \mathbf{Bool} <: \mathbf{Int}$. Since the latter is the first half of the former, the coercion $\lambda x.\, x.\mathrm{fst}$ is inserted.

A careful reader may notice that rule DTyp-Sub does not produce $\langle 48, \mathbf{true}\rangle.\mathrm{fst} - 2$ as we expect. Instead, it produces $((\lambda x.\, x.\mathrm{fst})\,\langle 48, \mathbf{true}\rangle) - 2$, which is less efficient as it introduces a spurious application. To fill the gap, we propose an alternative rule Ela-Sub in our work and a novel coercive subtyping judgment, which directly coerces ϵ_1 into ϵ_2. In the aforementioned example, the subtyping relation $\mathbf{Int} \,\&\, \mathbf{Bool} <: \mathbf{Int}$ will coerce $\langle 48, \mathbf{true}\rangle$ to produce the more efficient $\langle 48, \mathbf{true}\rangle.\mathrm{fst}$. Although we only avoid one step of beta reduction in this case, a more complicated subtyping judgment will lead to many coercion functions composed together and introduce many spurious applications.

Linear merge lookup. A second important drawback of Dunfield's approach is the representation of merges as nested pairs. The merge operator composes expressions in a binary manner, so extracting one component from nested merges of n components requires $n - 1$ projections in the worst case. For example, when adding one more element to the previous merge, $48 \,,\, \mathbf{true} \,,\, \text{`a'}$ for example, one more projection must be added to the elaborated result as well: $\langle\langle 48, \mathbf{true}\rangle, \text{`a'}\rangle.\mathrm{fst}.\mathrm{fst} - 2$. Note that we have simplified the coercion application from $((\lambda x.\, x.\mathrm{fst}) \circ (\lambda x.\, x.\mathrm{fst}))\,\langle\langle 48, \mathbf{true}\rangle, \text{`a'}\rangle$ to $\langle\langle 48, \mathbf{true}\rangle, \text{`a'}\rangle.\mathrm{fst}.\mathrm{fst}$. Compared with array access or dictionary lookup, such projections are more expensive in terms of both code length and runtime performance.

Pairs are order-sensitive. What is worse, a representation based on pairs has another disadvantage: unnecessary coercions are never optimized. Consider two merges $48 \,,\, \mathbf{true}$ and $\mathbf{true} \,,\, 48$. These two merges are equivalent in any context. Although they lead to a different order in the elaborated pairs, permutation of components does not matter as

long as it is consistent with the projection. For example, $\langle 48, \textbf{true}\rangle.\mathsf{fst} - 2$ is the same as $\langle \textbf{true}, 48\rangle.\mathsf{snd} - 2$. However, permutation can lead to expensive coercions. To cast $48 \,,\, \textbf{true} \,,\, \text{`a'}$ to type $\textbf{Char} \,\&\, \textbf{Int} \,\&\, \textbf{Bool}$, every single component needs to be extracted and rearranged:

$$\textbf{let } e = \langle\langle 48, \textbf{true}\rangle, \text{`a'}\rangle \textbf{ in } \langle\langle e.\mathsf{snd}, e.\mathsf{fst}.\mathsf{fst}\rangle, e.\mathsf{fst}.\mathsf{snd}\rangle$$

Thus, it is desirable to replace nested pairs with other representations that support more efficient merge lookup and avoid conversions between equivalent types.

6.2 Our Representation of Merges

Prologue: compiling overloaded functions. In programming languages that support function overloading, C++ for example, the compiler generates different names for overloaded functions. This process is usually called *name mangling*. If we have a function `f` with two overloaded versions:

```
void f(int x)  { ... }  // f → __Z1fi
void f(bool x) { ... }  // f → __Z1fb
```

Two different names are generated based on the parameter types: the postfix `i` in `__Z1fi` is short for **`int`** and `b` in `__Z1fb` for **`bool`**. After name mangling, the overloaded versions are disambiguated, and the linker can easily associate each call site with a specific version.

Key idea: compiling merges to type-indexed records. When it comes to merging, the situation is similar: a merge contains "overloaded" terms of different types. For example, the merge $48 \,,\, \textbf{true}$ contains both an integer and a boolean value. When compiling the merge, we adopt a similar technique to name mangling. We generate a unique name for every type, which is used to look up the corresponding component. More specifically, a merge is compiled to a record, and the components of the merge become its fields. For example, $48 \,,\, \textbf{true}$ will compile to $\{\mathtt{int} \Mapsto 48; \mathtt{bool} \Mapsto \textbf{true}\}$. The labels in the record, which we call *type indices*, are generated from the type of each term. As for nested merges, we also flatten them in one record. Instead of the nested pairs $\langle\langle 48, \textbf{true}\rangle, \text{`a'}\rangle$, $48 \,,\, \textbf{true} \,,\, \text{`a'}$ is translated into a record of three fields: $\{\mathtt{int} \Mapsto 48; \mathtt{bool} \Mapsto \textbf{true}; \mathtt{char} \Mapsto \text{`a'}\}$. The disjointness constraint on merging ensures that the components of a merge have non-overlapping types, hence the fields of the elaborated record are conflict-free (e.g. a merge cannot contain both 48 and 46). The idea of using labels based on types is similar to *type-*

indexed rows [Shields and Meijer 2001], though their type system does not involve subtyping at all.

The record design significantly reduces the cost of projections. For 48 , **true** , 'a', we would not need to project twice to find the exact position when selecting the integer. With a single projection, a component in an n-level merge can be extracted. Besides, record fields are order-irrelevant, which allows us to treat permuted intersection types equivalently. Using our approach, coercing a term from type **Int** & **Bool** & **Char** to type **Char** & **Int** & **Bool** has *no cost*, because the elaborated record does not change. In other words, {`int` ⟾ 48; `bool` ⟾ **true**; `char` ⟾ 'a'} and {`char` ⟾ 'a'; `int` ⟾ 48; `bool` ⟾ **true**} are equivalent. In CP, multi-field record types are also represented as intersection types. For example, $\{\ell_1 : \textbf{Int}; \ell_2 : \textbf{Int}\}$ is syntactic sugar for $\{\ell_1 : \textbf{Int}\} \,\&\, \{\ell_2 : \textbf{Int}\}$. Therefore, the order of fields in a record type does not matter either. We will develop a comprehensive theory that accounts for type equivalence and handles all possible cases next.

6.3 Reducing Coercions for Equivalent Types

Coercive subtyping is inevitable in CP, so the performance penalties caused by coercions cannot be neglected. Following the line of discussion above, an important optimization that we identify is to avoid coercions for subtyping between equivalent types, whose impact will be benchmarked in Section 9.1. In our translation scheme, some syntactically different types are translated to the same type index. These types that are treated equivalently after compilation are called *equivalent types* (denoted by $A \fallingdotseq B$). The design of equivalent types is inherently determined by the fact that we represent merges as records. We do not need to distinguish two types after compilation if their terms are compiled to records of exactly the same shape. The most interesting types in our compilation scheme are:

- *Top-like types* [Oliveira et al. 2016], which correspond to empty records because they do not convey any information.
- *Intersection types*, which correspond to multi-field records. Generally speaking, records are order-irrelevant and contain no duplicate labels (or duplicate labels are allowed but fields with the same label have equivalent values).

Considering the characteristics of our record-based representation, we can first derive that all top-like types are equivalent. In addition, two intersection types are considered equivalent if and only if they are formed using any combination of the following three criteria:

- They are permutations of the same set of types, or
- They are equivalent after deduplicating type components, or
- They are equivalent after removing top-like components.

The rules for other types are structural, ensuring that the type equivalence is a congruence.

Although we work hard to reduce the number of coercions, coercions cannot be fully eliminated. Next, we will explain the reason why they are still necessary to CP.

6.4 Necessity of Coercions

In CP, our interpretation of subtyping is *coercive* [Luo et al. 2013], in contrast to the inclusive (also called subsumptive) view of subtyping. That is, a value of a subtype is not a value of a supertype directly, but it contains sufficient information so that it can be converted into a value of a supertype. Such conversions are generated by subtyping derivations and are inserted by the subsumption rule during type checking.

The need for coercive subtyping in CP mainly comes from the unambiguity constraint on merging, for which the redundant information in expressions could be harmful. For example,

$$\textbf{let } x = 48 \,,\, \textbf{true in not } (x : \textbf{Int} \,,\, \textbf{false})$$

can evaluate to both **true** and **false** if the boolean component in x is kept. During typing, we use disjointness checks to ensure the static types of the components to be merged (**Int** and **Bool** in this example) do not overlap. But the soundness of such checks is based on the assumption that any expression's dynamic type corresponds to its static type. That is, $x : \textbf{Int}$ should contain nothing other than an integer at run time. So we have to coerce x from $\{\texttt{int} \mapsto 48; \texttt{bool} \mapsto \textbf{true}\}$ to a record that only contains the integer field. With some simplification, the whole expression should compile to:

$$\textbf{let } x = \{\texttt{int} \mapsto 48; \texttt{bool} \mapsto \textbf{true}\} \textbf{ in not } (\{\texttt{int} \mapsto x.\texttt{int}\} +\!\!+ \{\texttt{bool} \mapsto \textbf{false}\}).\texttt{bool}$$

where $+\!\!+$ denotes runtime record concatenation, which is a key feature of extensible records. In summary, there is a strong correspondence between the value and its static type in CP. So we can directly tell from the declared type how many fields the compiled record has and what the labels are. This design resolves the issue of interaction between merging and subtyping in **??** and is key to the type safety of dynamic trait inheritance.

Distributive subtyping. Normally, coercions are just removing redundant fields from a compiled record. For example, we coerce x from $\{\texttt{int} \Longmapsto 48; \texttt{bool} \Longmapsto \textbf{true}\}$ to $\{\texttt{int} \Longmapsto 48\}$ in the previous example. This is because a supertype of an intersection type consists of part of the component types, so the compiled record of the supertype contains a subset of the original fields. However, the situation becomes complicated in the presence of *distributive* subtyping. For example, a function of type $(\top \to \textbf{Int}) \,\&\, (\top \to \textbf{Bool})$ can be coerced to type $\top \to \textbf{Int} \,\&\, \textbf{Bool}$ because the former is a subtype of the latter via distributivity. The coercion is not removing fields but merging two functions into a single one.

Let us consider a more practical example based on the expression problem in Section 4.3.3. Here is a *simplified* version of what happens to the constructors for numeric literals when we compose the evaluation and pretty-printing operations:

```
ep = { Lit = \n → { eval = n } } , { Lit = \n → { print = toString n } };
-- : { Lit: Int → Eval }         & { Lit: Int → Print }
```

As the intersection type indicates, `ep` should compile to a two-field record: one field stores the constructor for `Eval` and the other for `Print`. According to the subtyping relation, via distributivity, it can be used as if it has type `{ Lit: Int → Eval&Print }`:

```
ep.Lit 48  --> { eval = 48; print = "48" }
```

However, such usage expects that the compiled record from `ep` only has one field, whose label corresponds to `{ Lit: Int → Eval&Print }`. Unfortunately, as we showed before, the compiled record actually contains two different labels from the expected one, so the subtyping does not automatically work. That is why we need to insert a coercion here to convert the two-field record to a new one with one single field, which is similar to the previous example of merging two functions.

Our compilation scheme is designed to avoid coercions as much as possible. The aforementioned coercion is not inserted for *direct* usage of record projections or function applications. Instead, the compiled code will select the two functions from the two fields for `ep.Lit` and apply both to 48. The results are then combined into a record so that both `eval` and `print` fields are present.

6.5 Implementation in JavaScript

The extensible records that we have been mentioning are an abstract data type that supports construction, concatenation, and projection. They do not imply any concrete data structure in any particular programming language. They can be implemented as hash tables, binary search trees, or even association lists, and most mainstream languages have

built-in and highly optimized support for these data structures. In our implementation, extensible records are implemented as *JavaScript objects*, whose underlying data structure still varies among JavaScript engines. Nevertheless, one thing we are certain of is that accessing properties of an object, which corresponds to record projection in our terminology, is highly optimized in the various engines.

The CP compiler supports modular type checking and separate compilation. In other words, compiling a CP file does not require access to the source code of the libraries that it depends on. What is needed is only the header files of the libraries, which mainly contain type information. Separate compilation largely decreases the rebuilding time since it avoids recompiling its dependencies, and it allows closed-source distribution of libraries. More details about the implementation of separate compilation can be found in Section 8.6.

Type indices. In our implementation, type indices are represented by JavaScript strings (hereinafter, `"string"` is in violet and monospaced). Below is how we represent different types:

- Primitive types are simply represented by their names, e.g. `"int"` for **Int**.
- Functions are represented by their return types, e.g. `"func_int"` for **String** → **Int**.
- Records are represented by both labels and field types, e.g. `"rcd_l:int"` for $\{\ell : \mathbf{Int}\}$.
- Intersections are represented by joining the representations of their components after alphabetical sorting, deduplication, and removal of top-like types, e.g. `"(bool&int)"` for **Int** & **Bool** & ⊤ & **Bool**. Note that such type indices only occur when intersection types are nested within functions or records. A top-level intersection corresponds to a multi-field record, which has separate type indices for each component.

The representation for function types may be a bit surprising. It originates from the disjointness rule for function types: two function types are disjoint if and only if their return types are disjoint (rule D-ArrowArrow). This rule is derived from the specification of disjointness (Theorem 7.7), which basically means that two disjoint types do not overlap on any meaningful types. For example, `Int → Int` and `Bool → Int` shares a common supertype `Int&Bool → Int`, so these two types are *not* disjoint. If those types are considered to be disjoint, we could have the following application:

```
((\(x: Int) → x + 1),(\(x: Bool) → if x then 1 else 0)) (1,false)
```

Note that both functions can be selected, and we get either 2 or 0 depending on which function we pick. The semantics would be ambiguous in this way. Thus, allowing such

merges is unsafe. That is why `Int → Int` and `Bool → Int` are not disjoint, and `"func_int"` cannot occur twice. The disjointness checks in CP rule out the possibility of type index conflicts between two functions in a merge. Our design that includes only return types also avoids very long property names in JavaScript, which may lead to performance issues.

Compiling parametric polymorphism. As we have discussed previously, dynamic inheritance and family polymorphism are already difficult to handle. In those examples, parametric polymorphism also plays an important role, yet we have not mentioned the difficulty of compiling it. The reason why this feature is challenging to compile is a bit more technical: it relates to when to build type indices, namely the labels of the compiled records.

For non-polymorphic types, the labels remain fixed throughout the program execution. However, for polymorphic types, we have to deal with *type instantiation*. For example, we may have a source type `{ f: A → A }`, where the type `A` is a type variable. After the instantiation of `A`, we may have the type `{ f: Int → Int }` or perhaps the type `{ f: Bool → Bool }`. The problem is that different instantiations of polymorphic type variables will produce different labels. So for polymorphic types, the labels cannot be statically computed. To solve this problem, first-class labels [Leijen 2004] are needed so that polymorphic instantiation can build a label at run time and propagate the label that corresponds to the instantiated type. A more detailed explanation with examples can be found in Section 8.2.

Important optimizations. In our implementation, we have applied several optimizations to improve the performance of the generated JavaScript code. Besides the elimination of redundant coercions based on equivalent types in Section 6.3, some important optimizations are:

1. Reducing intermediate objects using destination-passing style [Shaikhha et al. 2017];
2. Reducing object copying by detecting whether the compiled term is part of a merge;
3. Limiting lazy evaluation to certain trait fields to improve performance;
4. Preventing primitive values from boxing/unboxing;
5. Avoiding the insertion of coercions for record projections.

These optimizations will be elaborated with examples in Chapter 8, and their impact on performance will be evaluated in Section 9.1. Among the five optimizations, the last one (5^{th}) is formalized.

7 Formalization of the Compilation Scheme

To demonstrate and validate the key ideas of the compilation scheme, this chapter introduces two calculi for the source and the target languages, respectively, and the elaboration between them.

The *source calculus* is a variant of λ_{i}^{+} [Bi et al. 2018; Huang et al. 2021], which mainly omits parametric polymorphism from F_{i}^{+} [Bi et al. 2019; Fan et al. 2022], the core calculus for CP. Polymorphism is supported in our compiler, and its compilation is informally explained in Section 8.2. We omit polymorphism here because it adds considerable complications that would distract us from the key ideas of the compilation scheme. Furthermore, our formalization does not include most optimizations.

The *target calculus* λ_r is a standard λ-calculus that supports extensible records, which can be regarded as a functional subset of JavaScript.

In summary, the formalization includes the key idea of compiling merges to type-indexed records, and the following improvements:

- The use of a new coercive style that avoids modeling coercions as function terms.
- Avoiding coercions for record projections, which were needed by Fan et al. [2022].

Technical results include proofs of type safety, as well as several interesting properties about our translation of types into record labels. All proofs are mechanically checked using the Coq proof assistant.

7.1 Target Calculus with Extensible Records

As we have emphasized, our source language CP only allows disjoint traits in trait composition. Correspondingly, our source calculus λ_{i}^{+} enforces the disjointness constraint on merges and does not accept records with overlapping fields. In contrast, the main characteristic of our *target* calculus λ_r is that it allows duplicate labels in records. When labels conflict, overriding happens, like the design of scoped labels by Leijen [2005]. But this

overriding does not affect type safety (with the existence of subtyping) or the coherence of the elaboration semantics. This is because we only need to consider the terms that are generated by the elaboration from our source calculus λ_i^+. Since labels are computed from the corresponding source types of the fields, the type system of λ_r can require that duplicate labels in one record must be associated with fields of *equivalent* types. Besides, these fields are semantically equivalent because they *originate* from the same terms.

For instance, $1 \,,\, 2$ and even $1 \,,\, 1$ are forbidden in λ_i^+ (and our source language CP). Consequently, the elaborated terms in λ_r cannot have conflicting fields like $\{\texttt{int} \Mapsto 1;\ \texttt{int} \Mapsto 2\}$. However, it is possible, as part of evaluation, that harmless forms of duplicate fields arise, leading to duplicate fields where the values are the same, such as $\{\texttt{int} \Mapsto 1;\ \texttt{int} \Mapsto 1\}$. We will discuss this harmless duplication and the coherence of the elaboration semantics in Section 7.3 and Section 7.4 after presenting both calculi and the elaboration rules.

Syntax. We use the integer type as a representative of base types. $\mathbb{Z}$ denotes the integer type, and n represents any integer literal. The meta-variable ρ stands for record types, including the empty record type $\{\,\}$. The type ρ extended by a field of type $\mathcal{A}$ with label ℓ is written as $\{\ell \Mapsto \mathcal{A} \mid \rho\}$. For example, $\{\ell_1 \Mapsto \mathcal{A} \mid \{\ell_2 \Mapsto \mathcal{B} \mid \{\,\}\}\}$ is a record type with two fields and is abbreviated as $\{\ell_1 \Mapsto \mathcal{A};\ \ell_2 \Mapsto \mathcal{B}\}$. In general, abbreviations $\{\ell_1 \Mapsto \mathcal{A}_1;\ \ldots;\ \ell_n \Mapsto \mathcal{A}_n\}$ represents a multi-field record type, and $\{\ell_1 \Mapsto \mathcal{A}_1;\ \ldots;\ \ell_n \Mapsto \mathcal{A}_n \mid \rho\}$ is the record type ρ being extended by n fields. At the term level, records can be concatenated using $+\!\!+$, and $\epsilon.\ell$ extracts the first ℓ field from ϵ. The full syntax of λ_r is as follows:[1]

Types	$\mathcal{A}, \mathcal{B}, C ::= \mathbb{Z} \mid \mathcal{A} \to \mathcal{B} \mid \rho$
Record types	$\rho ::= \{\,\} \mid \{\ell \Mapsto \mathcal{A} \mid \rho\}$
Expressions	$\epsilon ::= n \mid x \mid \lambda x.\, \epsilon \mid \epsilon_1\, \epsilon_2 \mid \{\ell_1 \Mapsto \epsilon_1;\ \ldots;\ \ell_n \Mapsto \epsilon_n\} \mid \epsilon.\ell \mid \epsilon_1 +\!\!+ \epsilon_2$
Values	$v ::= n \mid \lambda x.\, \epsilon \mid \{\ell_1 \Mapsto v_1;\ \ldots;\ \ell_n \Mapsto v_n\}$
Typing contexts	$\Delta ::= \cdot \mid \Delta, x : \mathcal{A}$

Small-step semantics. The dynamic semantics of target expressions is defined at the top of Figure 7.1. For conciseness, we also use a list comprehension representation $\{\,\overline{\ell_i \Mapsto \epsilon_i}^{\,i}\,\}$ for multi-field records. The evaluation is call-by-value, and record fields are eagerly evaluated. To concatenate two records, they have to be fully reduced to values and then merged in rule TStep-Concat. For example, $\{\ell \Mapsto 1 + 1\} +\!\!+ \epsilon$ evaluates to $\{\ell \Mapsto 2;\ \ell_1 \Mapsto$

[1] In our Coq formalization, the bottom type and fixpoint expressions are also formalized in both source and target calculi. We omit them here to better align with λ_i^+ [Bi et al. 2018], which does not support these features.

$\boxed{\epsilon \to \epsilon'}$ *(Small-step semantics)*

$$\textsc{TStep-Proj}\ \frac{\epsilon \to \epsilon'}{\epsilon.\ell \to \epsilon'.\ell} \qquad \textsc{TStep-AppL}\ \frac{\epsilon_1 \to \epsilon_1'}{\epsilon_1\,\epsilon_2 \to \epsilon_1'\,\epsilon_2} \qquad \textsc{TStep-AppR}\ \frac{\epsilon \to \epsilon'}{v\,\epsilon \to v\,\epsilon'}$$

$$\textsc{TStep-ConcatL}\ \frac{\epsilon_1 \to \epsilon_1'}{\epsilon_1 +\!\!+ \epsilon_2 \to \epsilon_1' +\!\!+ \epsilon_2} \qquad \textsc{TStep-ConcatR}\ \frac{\epsilon \to \epsilon'}{v +\!\!+ \epsilon \to v +\!\!+ \epsilon'}$$

$$\textsc{TStep-ProjRcd}\ \frac{\mathbf{lookup}\ \ell\ v_1 \Rightarrow v_2}{v_1.\ell \to v_2} \qquad \textsc{TStep-Rcd}\ \frac{\epsilon \to \epsilon'}{\{\overline{\ell_i \mapsto v_i}^{\,i};\ \ell \mapsto \epsilon;\ \overline{\ell_j \mapsto \epsilon_j}^{\,j}\} \to \{\overline{\ell_i \mapsto v_i}^{\,i};\ \ell \mapsto \epsilon';\ \overline{\ell_j \mapsto \epsilon_j}^{\,j}\}}$$

$$\textsc{TStep-Concat}\ \frac{}{\{\overline{\ell_i \mapsto v_i}^{\,i}\} +\!\!+ \{\overline{\ell_j \mapsto v_j}^{\,j}\} \to \{\overline{\ell_i \mapsto v_i}^{\,i};\ \overline{\ell \mapsto v_j}^{\,j}\}} \qquad \textsc{TStep-AppAbs}\ \frac{}{(\lambda x.\,\epsilon)\ v \to \epsilon[x \mapsto v]}$$

$\boxed{\mathbf{lookup}\ \ell\ v_1 \Rightarrow v_2}$ *(Label lookup on records)*

$$\mathbf{lookup}\ \ell\ \{\ell_1 \mapsto v_1;\ \ldots;\ \ell_n \mapsto v_n\} \Rightarrow v_k \qquad \text{if } \ell_k = \ell \text{ and } \forall j \in 1..k\text{–}1, \ell_j \neq \ell$$

$\boxed{\mathbf{lookup}\ \ell\ \rho \Rightarrow \mathcal{B}}$ *(Label lookup on record types)*

$$\begin{array}{ll}
\mathbf{lookup}\ \ell\ \{\ell \mapsto \mathcal{A} \mid \rho\} \Rightarrow \mathcal{A} & \\
\mathbf{lookup}\ \ell_1\ \{\ell_2 \mapsto \mathcal{A} \mid \rho\} \Rightarrow \mathcal{B} & \text{if } \ell_1 \neq \ell_2 \text{ and } \mathbf{lookup}\ \ell_1\ \rho \Rightarrow \mathcal{B} \\
\mathbf{lookup}\ \ell_1\ \{\ell_2 \mapsto \mathcal{A} \mid \rho\} \Rightarrow\!\!\!\!\Rightarrow & \text{if } \ell_1 \neq \ell_2 \text{ and } \mathbf{lookup}\ \ell_1\ \rho \Rightarrow\!\!\!\!\Rightarrow \\
\mathbf{lookup}\ \ell\ \{\,\} \Rightarrow\!\!\!\!\Rightarrow &
\end{array}$$

Figure 7.1: Dynamic semantics and meta-functions for λ_r.

$v_1;\ \ldots;\ \ell_n \mapsto v_n\}$, assuming that ϵ evaluates to $\{\ell_1 \mapsto v_1;\ \ldots;\ \ell_n \mapsto v_n\}$. Rule TStep-ProjRcd uses the lookup function (**lookup** $\ell\ v_1 \Rightarrow v_2$) defined in the middle of Figure 7.1 to extract the first field with a matched label.

Type-level lookup. Besides the value-level lookup function, we define a meta-function on record types at the bottom of Figure 7.1 to reflect the behavior of field selection. It finds the first field type that matches the given label, just like the value-level one. We use **lookup** $\ell\ \rho \Rightarrow\!\!\!\!\Rightarrow$ to represent the case where no field in ρ matches ℓ.

Width subtyping. We define a form of width subtyping for record types at the top of Figure 7.2, while depth subtyping is not supported in λ_r. Intuitively, $\rho_1 \subseteq \rho_2$ holds if, for any projection that can be performed on a term of ρ_2, it can also be performed on any term of ρ_1, and their results have equivalent types. The subtyping relation will be used after

Type equivalence $\mathcal{A} \approx \mathcal{B} \triangleq \mathcal{A} \subseteq \mathcal{B} \wedge \mathcal{B} \subseteq \mathcal{A}$

$\boxed{\mathcal{A} \subseteq \mathcal{B}}$ *(Width subtyping)*

$$\text{RTS-Refl}\ \frac{}{\mathcal{A} \subseteq \mathcal{A}} \qquad \text{RTS-Arrow}\ \frac{\mathcal{A} \approx \mathcal{A}' \qquad \mathcal{B} \approx \mathcal{B}'}{\mathcal{A} \to \mathcal{B} \subseteq \mathcal{A}' \to \mathcal{B}'} \qquad \text{RTS-Rcd}\ \frac{\forall \ell\ C,\ \text{if } \mathbf{lookup}\,\ell\,\rho_2 \Rightarrow C \text{ then } \mathbf{lookup}\,\ell\,\rho_1 \approx C}{\rho_1 \subseteq \rho_2}$$

$\boxed{\mathbf{lookup}\,\ell\,\rho \approx \mathcal{A}}$, $\boxed{\mathbf{lookup}\,\ell\,\rho \sim \mathcal{A}}$ *(Abbreviations for lookup)*

$$\mathbf{lookup}\,\ell\,\rho \approx \mathcal{A} \triangleq \exists \mathcal{A}',\ \mathbf{lookup}\,\ell\,\rho \Rightarrow \mathcal{A}' \wedge \mathcal{A}' \approx \mathcal{A}$$
$$\mathbf{lookup}\,\ell\,\rho \sim \mathcal{A} \triangleq \mathbf{lookup}\,\ell\,\rho \approx \mathcal{A} \vee \mathbf{lookup}\,\ell\,\rho \twoheadrightarrow$$

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

$$\text{WFC-Nil}\ \frac{}{\vdash \cdot} \qquad \text{WFC-Cons}\ \frac{\vdash \mathcal{A} \qquad \vdash \Delta}{\vdash \Delta, x : \mathcal{A}}$$

$\boxed{\vdash \mathcal{A}}$ *(Type well-formedness)*

$$\text{WF-Nil}\ \frac{}{\vdash \{\,\}} \qquad \text{WF-Int}\ \frac{}{\vdash \mathbb{Z}} \qquad \text{WF-Arrow}\ \frac{\vdash \mathcal{A} \qquad \vdash \mathcal{B}}{\vdash \mathcal{A} \to \mathcal{B}} \qquad \text{WF-Rcd}\ \frac{\vdash \mathcal{A} \qquad \vdash \rho \qquad \mathbf{lookup}\,\ell\,\rho \sim \mathcal{A}}{\vdash \{\ell \Mapsto \mathcal{A} \mid \rho\}}$$

Figure 7.2: Width subtyping, type equivalence, and well-formedness in λ_r.

we introduce our source calculus and its elaboration semantics in the next section. In the metatheory proofs, we will need to relate record expressions to parts of their types, like $\{\ell = 1;\ \ell' = \mathbf{true}\}$ to $\{\ell \Mapsto \mathbb{Z}\}$. The relation between types and their parts is characterized by width subtyping.

Equivalence of target types. An equivalence relation $\approx$ is derived from width subtyping to allow permutation of record fields. $\mathbf{lookup}\,\ell\,\rho \approx \mathcal{A}$ is an abbreviation for the case where looking up ℓ in ρ produces a type equivalent to $\mathcal{A}$. A similar abbreviation

lookup $\ell\ \rho \sim \mathcal{A}$ additionally includes the case where ℓ is absent in ρ. An important property of equivalent types is that they preserve the results of lookup:

Lemma 7.1 (Lookup on equivalent types). *Given* $\rho_1 \approx \rho_2$*:*

- *If* **lookup** $\ell\ \rho_1 \Rightarrow C$ *then* **lookup** $\ell\ \rho_2 \approx C$.
- *If* **lookup** $\ell\ \rho_1 \twoheadRightarrow$ *then* **lookup** $\ell\ \rho_2 \twoheadRightarrow$.

Type well-formedness. Well-formed types are defined at the bottom of Figure 7.2. Record type extension must be consistent: duplicate labels must be associated with equivalent field types. Specifically, as shown by rule WF-Rcd, to safely extend type ρ by a new field of label ℓ, either the old field type in ρ is equivalent to the new field type, or ρ lacks label ℓ. This is also enforced in the typing rule Typ-RcdCons.

As we will explain later, every type in the source language, including an intersection type, is translated into a record type in the target language. All the record labels are generated from source types in the translation process, where disjoint source types are converted to distinct labels. Although overlapping is forbidden in merges, overlapping is *not* forbidden in intersection types. For example, $1 \,,\, 2$ is forbidden but $\mathbb{Z} \,\&\, \mathbb{Z}$ is a valid type in λ_{i}^{+}. Therefore, it is natural for corresponding record types to contain duplicate labels. The properties of the source calculus also ensure that the translated types are well-formed. With the well-formedness restriction, permuting any fields in a record type does not affect type safety.

Typing. The typing rules of target expressions are presented in Figure 7.3. A set of auxiliary rules is defined to concatenate two record types ($\rho_1 \uplus \rho_2 \Rightarrow \rho_3$). The premise of rule CT-Rcd guarantees the well-formedness of the result type. Given the types of two record expressions, the concatenation of the two record types directly reveals the shape of the result of concatenating the two expressions.

In our type system, there is no subsumption rule or a rule that allows conversion between $\approx$-equivalent types. Every expression under the given typing context has a unique type. That is, from a record type, it is straightforward to tell the shape of its value: how many fields it has, what the labels are, and how the fields are arranged. On the other hand, in rule Typ-App, the requirement on argument type is relaxed to $\approx$-equivalence.

Type soundness. The λ_r calculus is proven to be type-sound via progress and type preservation. However, we should emphasize that type preservation (and the substitution lemma) is proven with respect to $\approx$-equivalence.

$\boxed{\rho_1 \uplus \rho_2 \Rrightarrow \rho_3}$ *(Record type concatenation)*

$$\frac{}{\{\,\} \uplus \rho \Rrightarrow \rho}\ \textsc{CT-Nil} \qquad \frac{\mathbf{lookup}\ \ell\ \rho_2 \sim \mathcal{A} \qquad \rho_1 \uplus \rho_2 \Rrightarrow \rho_3}{\{\ell \Mapsto \mathcal{A} \mid \rho_1\} \uplus \rho_2 \Rrightarrow \{\ell \Mapsto \mathcal{A} \mid \rho_3\}}\ \textsc{CT-Rcd}$$

$\boxed{\Delta \vdash \epsilon : \mathcal{A}}$ *(Typing)*

$$\frac{\vdash \Delta}{\Delta \vdash n : \mathbb{Z}}\ \textsc{Typ-Int} \qquad \frac{\Delta, x : \mathcal{A} \vdash \epsilon : \mathcal{B}}{\Delta \vdash \lambda x.\, \epsilon : \mathcal{A} \to \mathcal{B}}\ \textsc{Typ-Abs} \qquad \frac{\Delta \vdash \epsilon_1 : \mathcal{A} \to \mathcal{B} \qquad \Delta \vdash \epsilon_2 : \mathcal{A}' \qquad \mathcal{A} \approx \mathcal{A}'}{\Delta \vdash \epsilon_1\ \epsilon_2 : \mathcal{B}}\ \textsc{Typ-App}$$

$$\frac{\vdash \Delta}{\Delta \vdash \{\,\} : \{\,\}}\ \textsc{Typ-RcdNil} \qquad \frac{\Delta \vdash \epsilon : \mathcal{A} \qquad \Delta \vdash \{\ell_1 \Mapsto \epsilon_1;\ \ldots;\ \ell_n \Mapsto \epsilon_n\} : \rho \qquad \mathbf{lookup}\ \ell\ \rho \sim \mathcal{A}}{\Delta \vdash \{\ell \Mapsto \epsilon;\ \ell_1 \Mapsto \epsilon_1;\ \ldots;\ \ell_n \Mapsto \epsilon_n\} : \{\ell \Mapsto \mathcal{A} \mid \rho\}}\ \textsc{Typ-RcdCons}$$

$$\frac{\vdash \Delta \qquad x : \mathcal{A} \in \Delta}{\Delta \vdash x : \mathcal{A}}\ \textsc{Typ-Var} \qquad \frac{\Delta \vdash \epsilon : \rho \qquad \mathbf{lookup}\ \ell\ \rho \Rrightarrow \mathcal{B}}{\Delta \vdash \epsilon.\ell : \mathcal{B}}\ \textsc{Typ-RcdProj} \qquad \frac{\rho_1 \uplus \rho_2 \Rrightarrow \rho_3 \qquad \Delta \vdash \epsilon_1 : \rho_1 \qquad \Delta \vdash \epsilon_2 : \rho_2}{\Delta \vdash \epsilon_1 +\!\!+ \epsilon_2 : \rho_3}\ \textsc{Typ-RcdMerge}$$

Figure 7.3: Typing of λ_r.

Theorem 7.2 (Progress). *If $\cdot \vdash \epsilon : \mathcal{A}$, then either ϵ is a value or $\exists \epsilon', \epsilon \longrightarrow \epsilon'$.*

Lemma 7.3 (Substitution preserves typing). *If $\Delta, x : \mathcal{A}, \Delta' \vdash \epsilon : \mathcal{B}$ and $\Delta \vdash \epsilon' : \mathcal{A}'$ and $\mathcal{A}' \approx \mathcal{A}$, then $\exists\, \mathcal{B}'$ such that $\Delta, \Delta' \vdash \epsilon[x \mapsto \epsilon'] : \mathcal{B}'$ and $\mathcal{B}' \approx \mathcal{B}$.*

Theorem 7.4 (Type preservation). *If $\cdot \vdash \epsilon : \mathcal{A}$ and $\epsilon \longrightarrow \epsilon'$, then $\exists \mathcal{A}', \cdot \vdash \epsilon' : \mathcal{A}'$ and $\mathcal{A}' \approx \mathcal{A}$.*

7.2 Source Calculus and Elaboration

The source calculus is a variant of λ_i^+ [Bi et al. 2018; Huang et al. 2021]. It includes type $\top$, the maximal element in subtyping, as well as its canonical value $\top$. Functions ($\lambda x.\, e : A \to B$) always have type annotations. $\{\ell = e\}$ stands for single-field records, which has type $\{\ell : A\}$ if e has type A. The full syntax of λ_i^+ is as follows:

Types	$A, B, C ::= \top \mid \mathbb{Z} \mid A \to B \mid \{\ell : A\} \mid A \,\&\, B$
Expressions	$e ::= \top \mid n \mid x \mid \lambda x.\, e : A \to B \mid e_1\ e_2 \mid \{\ell = e\} \mid e.\ell \mid e_1\ ,\ e_2 \mid e : A$

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

$$\text{TL-Top}\ \frac{}{\lceil \top \rceil} \qquad \text{TL-And}\ \frac{\lceil A \rceil \quad \lceil B \rceil}{\lceil A \,\&\, B \rceil} \qquad \text{TL-Arrow}\ \frac{\lceil B \rceil}{\lceil A \to B \rceil} \qquad \text{TL-Rcd}\ \frac{\lceil B \rceil}{\lceil \{\ell : B\} \rceil}$$

$\boxed{A * B}$ *(Type disjointness)*

$$\text{D-Symm}\ \frac{B * A}{A * B} \qquad \text{D-TopL}\ \frac{\lceil A \rceil}{A * B} \qquad \text{D-AndL}\ \frac{A_1 * B \quad A_2 * B}{A_1 \,\&\, A_2 * B} \qquad \text{D-IntArrow}\ \frac{}{\mathbb{Z} * A_1 \to A_2} \qquad \text{D-IntRcd}\ \frac{}{\mathbb{Z} * \{\ell : A\}}$$

$$\text{D-ArrowArrow}\ \frac{A_2 * B_2}{A_1 \to A_2 * B_1 \to B_2} \qquad \text{D-RcdEq}\ \frac{A * B}{\{\ell : A\} * \{\ell : B\}} \qquad \text{D-RcdNeq}\ \frac{\ell_1 \neq \ell_2}{\{\ell_1 : A\} * \{\ell_2 : B\}} \qquad \text{D-ArrowRcd}\ \frac{}{A_1 \to A_2 * \{\ell : A\}}$$

Figure 7.4: Top-like types and type disjointness in λ_{i}^{+}.

Merge operator and disjoint intersection types. The symmetric merge operator ($,$) is like record concatenation, with which we can construct multi-field records from single-field records. However, it is not restricted to records: as long as two expressions have *disjoint* types (i.e. they are thought to be compatible), $e_1 \,,\, e_2$ is allowed, containing the information of both expressions. Assuming that e_1 and e_2 have type A and B respectively, the whole merge has intersection type $A \,\&\, B$.

$$\begin{aligned} \{\ell_1 : A_1;\ \ldots;\ \ell_n : A_n\} &\triangleq \{\ell_1 : A_1\} \,\&\, \ldots \,\&\, \{\ell_n : A_n\} \\ \{\ell_1 = e_1;\ \ldots;\ \ell_n = e_n\} &\triangleq \{\ell_1 = e_1\} \,,\, \ldots \,,\, \{\ell_n = e_n\} \end{aligned}$$

Top-like types and disjointness. Figure 7.4 defines two relations. They follow the specifications in previous work on disjoint intersection types [Huang et al. 2021], which are defined in terms of coercive subtyping ($\epsilon_1 : A <: B \rightsquigarrow \epsilon_2$) in Figure 7.9 . Since we only need to consider the relation on types, here we use $A <: B$ to represent the subtyping relation, ignoring the terms ϵ_1 and ϵ_2.

Theorem 7.5 (Coercion-erased subtyping). $A <: B \iff \forall \epsilon_1, \exists \epsilon_2, \epsilon_1 : A <: B \rightsquigarrow \epsilon_2$.

At the top of Figure 7.4 is the algorithmic definition of top-like types ($\lceil A \rceil$). It characterizes types that are equivalent to $\top$, including any function type with a top-like return

$$
\begin{array}{lll}
\text{Type indices} & T & ::= \overline{\mathbb{Z}} \mid \overline{Ts \to Ts} \mid \overline{\{\ell : Ts\}} \\
\text{List of type indices} & Ts & ::= [T_1, \ldots, T_n]
\end{array}
$$

$\boxed{|A| = Ts}$ *(Translation to type indices)*

$$
\begin{array}{lll}
|A| = [\,] & & \text{if } \rceil A \lceil \\
|\mathbb{Z}| = [\overline{\mathbb{Z}}] & & \\
|A \to B| = [\overline{|A| \to |B|}] & & \text{if not } \rceil B \lceil \\
|\{\ell : A\}| = [\overline{\{\ell : |A|\}}] & & \text{if not } \rceil A \lceil \\
|A \,\&\, B| = \textbf{dedup}\ (\textbf{merge}\ |A|\ |B|) & &
\end{array}
$$

$\boxed{\|A\| = \rho}$ *(Translation to target types)*

$$
\begin{array}{lll}
\|A\| = \{\,\} & & \text{if } \rceil A \lceil \\
\|\mathbb{Z}\| = \{\overline{\mathbb{Z}} \mapsto \mathbb{Z}\} & & \\
\|A \to B\| = \{\overline{|A| \to |B|} \mapsto \|A\| \to \|B\|\} & & \text{if not } \rceil B \lceil \\
\|\{\ell : A\}\| = \{\overline{\{\ell : |A|\}} \mapsto \|A\|\} & & \text{if not } \rceil A \lceil \\
\|A \,\&\, B\| = \rho & & \text{if } \|A\| \uplus \|B\| \Rightarrow \rho
\end{array}
$$

Figure 7.5: Translation functions for types in λ_{i}^{+}.

type, and any record type with a top-like field type. These types are thought to be vacuous and treated in a unified way.

Theorem 7.6 (Top-like types respect the specification). $\rceil A \lceil \iff \top <: A$.

At the bottom of Figure 7.4 is the algorithmic definition of disjointness. We say two types are disjoint ($A * B$) if they do not overlap on any meaningful types; or, any common supertypes they share are top-like. Irrelevant types are considered disjoint, such as integer and function types, or records with different labels. Function types are disjoint if and only if their return types are disjoint. Two record types with the same label are disjoint if and only if their field types are disjoint.

Theorem 7.7 (Type disjointness respects the specification). $A * B \iff \forall C$, *if* $A <: C$ *and* $B <: C$ *then* $\rceil C \lceil$.

Type indices and translation functions. Merges in λ_{i}^{+} are elaborated into records in λ_r. Each component is tagged by a label, which we call a *type index*. Type indices are computed from the component types of an intersection. Defined in Figure 7.5, the translation function $|\cdot|$ maps a type to a list of type indices Ts. For types that are neither a top-like nor an intersection type, the result is a singleton list. Values of top-like types are thought to contain no information, so these types are omitted in translation, i.e. they are converted into an empty list $[\,]$. These lists are merged in the case of intersection types: **merge** is a merge sort, taking two sorted lists and producing a merged sorted list. Then we remove

duplicates from the result list using **dedup**. For example, $\mathbb{Z} \,\&\, (\mathbb{Z} \to \mathbb{Z}) \,\&\, \mathbb{Z} \,\&\, (\top \to \top)$ is translated to $[\overline{\mathbb{Z}}, \overline{\overline{\mathbb{Z}} \to \overline{\mathbb{Z}}}]$. The list only contains the type indices for the first two elements of the intersection type because the third element is a duplicate of the first one and the last element is a top-like type. We use an injective function to map each type index to a unique string in Coq, and we use the alphabetical order of their corresponding strings to sort type indices.

Lemma 7.8 (Translation). *The mapping from type indices to strings has the following properties:*

- *If* $|A_1 \to B_1| = |A_2 \to B_2|$ *then* $|A_1| = |A_2|$ *and* $|B_1| = |B_2|$, *given that* $A_1 \to B_1$ *and* $A_2 \to B_2$ *are not top-like.*
- *If* $|\{\ell_1 : A_1\}| = |\{\ell_2 : A_2\}|$ *then* $\ell_1 = \ell_2$ *and* $|A_1| = |A_2|$, *given that* $\{\ell_1 : A_1\}$ *and* $\{\ell_2 : A_2\}$ *are not top-like.*

To the right of the type-index translation function, there is another function $\|\cdot\|$ that maps source types to target types. It uses the record type concatenation defined in Figure 7.3. The function is based on the design of elaboration, which we will introduce later (presented in Figure 7.7). It reflects the type of the elaborated target term. The result of translation is always a record type: all top-like types are converted to the empty record type; converting an intersection type is concatenating their counterparts. For the remaining types, the translation is a record type tagged by the type index associated with the type itself. Only when two field types are $\approx$-equivalent, they can have the same type index. While our typing rules use the type-index translation function $|\cdot|$, the type translation function $\|\cdot\|$ only serves the purpose of proving metatheory properties.

Equivalence of source types. Corresponding to the $\approx$-equivalence on target types, $\coloneqq$ defines an equivalence relation on source types. Likewise, it is derived from a preorder ($A \sqsubseteq B$), which is the width subtyping in the source calculus. Note that it is not the subtyping used in the type system, but rather an auxiliary relation defined to better characterize the invariant of the type index translation. As defined in Figure 7.6, this preorder relation is stricter than the coercive subtyping used in our source type system (presented in Figure 7.9). An intersection type can be intuitively understood as a set of distinct types. For example, the intersection type $\mathbf{Bool} \,\&\, \mathbf{Char} \,\&\, (\mathbf{Int} \to \mathbf{Int})$ represents a set of three distinct elements: **Bool**, **Char**, and **Int** $\to$ **Int**. Its width subtype must contain all these three elements. Generally speaking, all component types in an intersection must be present in its width subtype, excluding duplicates and top-like types. The $\coloneqq$-equivalence groups types that map to the same type index.

Type equivalence $A \fallingdotseq B \triangleq A \sqsubseteq B \wedge B \sqsubseteq A$

$\boxed{A \sqsubseteq B}$ *(Width subtyping)*

$$\text{RS-Int}\ \frac{}{\overline{\mathbb{Z}} \sqsubseteq \overline{\mathbb{Z}}} \qquad \text{RS-Top}\ \frac{\rceil B \lceil}{A \sqsubseteq B} \qquad \text{RS-Arrow}\ \frac{A_1 \fallingdotseq B_1 \qquad A_2 \fallingdotseq B_2}{A_1 \to A_2 \sqsubseteq B_1 \to B_2} \qquad \text{RS-Rcd}\ \frac{A \fallingdotseq B}{\{\ell : A\} \sqsubseteq \{\ell : B\}} \qquad \text{RS-AndL1}\ \frac{A_1 \sqsubseteq A_3}{A_1 \,\&\, A_2 \sqsubseteq A_3}$$

$$\text{RS-AndL2}\ \frac{A_2 \sqsubseteq A_3}{A_1 \,\&\, A_2 \sqsubseteq A_3} \qquad \text{RS-AndR}\ \frac{A_1 \sqsubseteq A_2 \qquad A_1 \sqsubseteq A_3}{A_1 \sqsubseteq A_2 \,\&\, A_3}$$

Figure 7.6: Width subtyping in λ_i^+.

Lemma 7.9 (Equivalent types). *Some properties of the $\fallingdotseq$-equivalence can be derived from properties of width subtyping:*

- *If* $\mathbf{lookup}\ \ell\ \|A\| \Rightarrow C_1$ *and* $\mathbf{lookup}\ \ell\ \|B\| \Rightarrow C_2$ *then* $C_1 \subseteq C_2$*. Thus, by symmetry, if* $\mathbf{lookup}\ \ell\ \|A\| \Rightarrow C_1$ *and* $\mathbf{lookup}\ \ell\ \|B\| \Rightarrow C_2$ *then* $C_1 \approx C_2$*.*
- *$A \sqsubseteq B$ if and only if all components of $|B|$ can be found in $|A|$. Thus, by symmetry, $A \fallingdotseq B$ if and only if $|A| = |B|$.*
- *If $A \sqsubseteq B$ then $\|A\| \subseteq \|B\|$. Thus, by symmetry, if $A \fallingdotseq B$ then $\|A\| \approx \|B\|$.*

The first one is a strong result about type translation: in a translated type, the type of a record field can be determined by its associated label. Hence, for any two translated types $\|A\|$ and $\|B\|$, looking up the same label ℓ will lead to equivalent types C_1 and C_1. For example, looking up an integer label $\overline{\mathbb{Z}}$ should always return an integer type $\mathbb{Z}$. With the above properties, we can prove that all translated types are well-formed.

Lemma 7.10 (Well-formedness of translated types). $\forall A, \vdash \|A\|$.

Type-directed elaboration. Defined in Figure 7.7, $\Gamma \vdash e \Leftrightarrow A \rightsquigarrow \epsilon$ relates a source expression e to a source type A under the typing context Γ and the typing mode $\Leftrightarrow$, and the typing derivation generates a target expression ϵ from e. The type system is *bidirectional* [Dunfield and Krishnaswami 2021; Pierce and Turner 2000]: under the inference mode ($\Rightarrow$), A is generated as an output; under the checking mode ($\Leftarrow$), A is given as an

Typing contexts $\quad \Gamma ::= \cdot \mid \Gamma, x : A$

Typing modes $\quad \Leftrightarrow ::= \Leftarrow \mid \Rightarrow$

$\boxed{\Gamma \vdash e \Leftrightarrow A \rightsquigarrow \epsilon}$ *(Type-directed elaboration)*

$$\textsc{Ela-Top}\ \frac{}{\Gamma \vdash \top \Rightarrow \top \rightsquigarrow \{\}} \qquad \textsc{Ela-TopAbs}\ \frac{\rceil B \lceil}{\Gamma \vdash \lambda x.\, e : A \to B \Rightarrow A \to B \rightsquigarrow \{\}}$$

$$\textsc{Ela-TopRcd}\ \frac{\Gamma \vdash e \Rightarrow A \rightsquigarrow \epsilon \qquad \rceil A \lceil}{\Gamma \vdash \{\ell = e\} \Rightarrow \{\ell : A\} \rightsquigarrow \{\}} \qquad \textsc{Ela-Int}\ \frac{}{\Gamma \vdash n \Rightarrow \mathbb{Z} \rightsquigarrow \{\overline{\mathbb{Z}} \Mapsto n\}} \qquad \textsc{Ela-Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A \rightsquigarrow x}$$

$$\textsc{Ela-Abs}\ \frac{\neg\rceil B \lceil \qquad \Gamma, x : A \vdash e \Leftarrow B \rightsquigarrow \epsilon}{\Gamma \vdash \lambda x.\, e : A \to B \Rightarrow A \to B \rightsquigarrow \{|A \to B| \Mapsto \lambda x.\, \epsilon\}} \qquad \textsc{Ela-Sub}\ \frac{\Gamma \vdash e \Rightarrow A \rightsquigarrow \epsilon_1 \qquad \epsilon_1 : A <: B \rightsquigarrow \epsilon_2}{\Gamma \vdash e \Leftarrow B \rightsquigarrow \epsilon_2}$$

$$\textsc{Ela-App}\ \frac{\Gamma \vdash e_1 \Rightarrow A \rightsquigarrow \epsilon_1 \qquad \Gamma \vdash e_2 \Rightarrow B \rightsquigarrow \epsilon_2 \qquad \epsilon_1 : A \bullet \epsilon_2 : B \rightsquigarrow \epsilon_3 : C}{\Gamma \vdash e_1\, e_2 \Rightarrow C \rightsquigarrow \epsilon_3} \qquad \textsc{Ela-Rcd}\ \frac{\neg\rceil A \lceil \qquad \Gamma \vdash e \Rightarrow A \rightsquigarrow \epsilon}{\Gamma \vdash \{\ell = e\} \Rightarrow \{\ell : A\} \rightsquigarrow \{|\{\ell : A\}| \Mapsto \epsilon\}}$$

$$\textsc{Ela-Proj}\ \frac{\Gamma \vdash e \Rightarrow A \rightsquigarrow \epsilon_1 \qquad \epsilon_1 : A \bullet \{\ell\} \rightsquigarrow \epsilon_2 : B}{\Gamma \vdash e.\ell \Rightarrow B \rightsquigarrow \epsilon_2} \qquad \textsc{Ela-Merge}\ \frac{\Gamma \vdash e_1 \Rightarrow A \rightsquigarrow \epsilon_1 \qquad \Gamma \vdash e_2 \Rightarrow B \rightsquigarrow \epsilon_2 \qquad A * B}{\Gamma \vdash e_1 \,,\, e_2 \Rightarrow A \,\&\, B \rightsquigarrow \epsilon_1 +\!\!+ \epsilon_2} \qquad \textsc{Ela-Anno}\ \frac{\Gamma \vdash e \Leftarrow A \rightsquigarrow \epsilon}{\Gamma \vdash e : A \Rightarrow A \rightsquigarrow \epsilon}$$

Figure 7.7: Type-directed elaboration of λ_i^+.

input. Given the typing context, every well-typed e has a unique inferred type; all the types that e can be checked against are supertypes of this inferred type.

Rule Ela-Merge is the signature rule of calculi with *disjoint intersection types*. The disjointness restriction on types ($A * B$, defined in Figure 7.4) prevents the overlapping of components in a merge. Thus, in a well-typed term like $e_1 \,,\, \ldots \,,\, e_n$, every subterm in the merge has disjoint types. Rule Ela-Anno allows upcasting expressions to any supertypes. The subtyping relation is checked in rule Ela-Sub via the subtyping judgment $\epsilon_1 : A <: B \rightsquigarrow \epsilon_2$, which also coerces the target term ϵ_1 to ϵ_2. Rule Ela-App relies on *distributive application*, which is defined in Figure 7.8. It takes the function type A and the

$\boxed{\epsilon_1 : A \bullet \epsilon_2 : B \rightsquigarrow \epsilon_3 : C}$ *(Distributive application)*

$$\textsc{A-Top}\ \frac{\rceil A \lceil}{\epsilon_1 : A \bullet \epsilon_2 : B \rightsquigarrow \{\} : \top} \qquad \textsc{A-Arrow}\ \frac{\neg\rceil B \lceil \quad \epsilon_2 : C <: A \rightsquigarrow \epsilon_3}{\epsilon_1 : A \to B \bullet \epsilon_2 : C \rightsquigarrow (\epsilon_1.|A \to B|)\ \epsilon_3 : B}$$

$$\textsc{A-And}\ \frac{\neg\rceil A \,\&\, B \lceil \quad \epsilon_1 : A \bullet \epsilon_2 : C \rightsquigarrow \epsilon_3 : A' \qquad \epsilon_1 : B \bullet \epsilon_2 : C \rightsquigarrow \epsilon_4 : B'}{\epsilon_1 : A \,\&\, B \bullet \epsilon_2 : C \rightsquigarrow \epsilon_3 ,, \epsilon_4 : A' \,\&\, B'}$$

$\boxed{\epsilon_1 : A \bullet \{\ell\} \rightsquigarrow \epsilon_3 : C}$ *(Distributive projection)*

$$\textsc{P-Top}\ \frac{\rceil A \lceil}{\epsilon_1 : A \bullet \{\ell\} \rightsquigarrow \{\} : \top} \qquad \textsc{P-RcdEq}\ \frac{\neg\rceil A \lceil}{\epsilon : \{\ell : A\} \bullet \{\ell\} \rightsquigarrow \epsilon.|\{\ell : A\}| : A}$$

$$\textsc{P-RcdNeq}\ \frac{\neg\rceil A \lceil \quad \ell_1 \neq \ell_2}{\epsilon : \{\ell_1 : A\} \bullet \{\ell_2\} \rightsquigarrow \{\} : \top} \qquad \textsc{P-And}\ \frac{\neg\rceil A \,\&\, B \lceil \quad \epsilon_1 : A \bullet \{\ell\} \rightsquigarrow \epsilon_2 : A' \quad \epsilon_1 : B \bullet \{\ell\} \rightsquigarrow \epsilon_3 : B'}{\epsilon_1 : A \,\&\, B \bullet \{\ell\} \rightsquigarrow \epsilon_2 ,, \epsilon_3 : A' \,\&\, B'}$$

Figure 7.8: Distributive application and projection in λ_{i}^{+}.

argument type B and, if A can be applied to B, produces the return type C. Distributive application additionally allows intersection types and top-like types (can be regarded as 0-ary intersections) to be applicable due to the distributivity of functions over intersections. For example, $(A_1 \to B_1) \,\&\, (A_2 \to B_2)$ can be applied to $A_1 \,\&\, A_2$ and produces $B_1 \,\&\, B_2$. Besides, $\epsilon_1 : A \bullet \epsilon_2 : B \rightsquigarrow \epsilon_3 : C$ uses ϵ_1 and ϵ_2 to generate the target term ϵ_3, reflecting the application in the target language. Similarly, rule Ela-Proj relies on *distributive projection* to obtain the result type. Given a label ℓ, the relation $\epsilon_1 : A \bullet \{\ell\} \rightsquigarrow \epsilon_2 : B$ finds all field types in A that match ℓ and returns them as an intersection type B, if there is more than one matched field. Similarly, ϵ_2 is the target expression that extracts the corresponding fields in ϵ_1.

Rules Ela-Top, Ela-TopAbs, and Ela-TopRcd generate an empty record for top-like types, which is a counterpart of the canonical top value. For non-top-like types, rules Ela-Int, Ela-Abs, and Ela-Rcd produces records with a single label translated from the type directly. Consequently, all elaborated terms are either reducible or are in a record form.

Ordinary types $\quad A^\circ, B^\circ, C^\circ ::= \top \mid \mathbb{Z} \mid A \to B^\circ \mid \{\ell : A^\circ\}$

$\boxed{\epsilon_1 : A <: B \rightsquigarrow \epsilon_2}$ *(Coercive subtyping)*

$$\text{S-Top}\ \frac{\rceil B^\circ \lceil}{\epsilon : A <: B^\circ \rightsquigarrow \{\}} \qquad \text{S-Int}\ \frac{}{\epsilon : \mathbb{Z} <: \mathbb{Z} \rightsquigarrow \{\overline{\mathbb{Z}} \Mapsto \epsilon.\overline{\mathbb{Z}}\}}$$

$$\text{S-Arrow}\ \frac{\begin{array}{c}\neg\rceil B_2^\circ \lceil \\ x : B_1 <: A_1 \rightsquigarrow \epsilon_1 \\ (\epsilon.|A_1 \to A_2|)\ \epsilon_1 : A_2 <: B_2^\circ \rightsquigarrow \epsilon_2\end{array}}{\epsilon : A_1 \to A_2 <: B_1 \to B_2^\circ \rightsquigarrow \{|B_1 \to B_2^\circ| \Mapsto \lambda x.\, \epsilon_2\}} \qquad \text{S-AndL}\ \frac{\epsilon : A <: C^\circ \rightsquigarrow \epsilon'}{\epsilon : A \,\&\, B <: C^\circ \rightsquigarrow \epsilon'}$$

$$\text{S-Rcd}\ \frac{\neg\rceil B^\circ \lceil \qquad \epsilon.|\{\ell : A\}| : A <: B^\circ \rightsquigarrow \epsilon_2}{\epsilon : \{\ell : A\} <: \{\ell : B^\circ\} \rightsquigarrow \{|\{\ell : B^\circ\}| \Mapsto \epsilon_2\}} \qquad \text{S-AndR}\ \frac{\epsilon : B <: C^\circ \rightsquigarrow \epsilon'}{\epsilon : A \,\&\, B <: C^\circ \rightsquigarrow \epsilon'}$$

$$\text{S-Split}\ \frac{B_1 \lhd B \rhd B_2 \qquad \epsilon : A <: B_1 \rightsquigarrow \epsilon_1 \qquad \epsilon : A <: B_2 \rightsquigarrow \epsilon_2 \qquad \epsilon_1 : B_1 \rhd B \lhd \epsilon_2 : B_2 \rightsquigarrow \epsilon_3}{\epsilon : A <: B \rightsquigarrow \epsilon_3}$$

Figure 7.9: Coercive subtyping in λ_{i}^{+}.

Coercive subtyping. Defined in Figure 7.9, $\epsilon_1 : A <: B \rightsquigarrow \epsilon_2$ takes a target expression ϵ_1 and two source types A and B and produces a target term ϵ_2. Intuitively, when ϵ_1 has type $\|A\|$, the generated ϵ_2 will have a type that is equivalent to $\|B\|$. The formal theorem will be given later (Theorem 7.11) when establishing type soundness. Besides producing the coerced target term, this relation also checks whether A is a subtype of B. For coercive subtyping, a more common form of the judgment is $A <: B \rightsquigarrow c$, where c is a coercion function in the target language with type $\|A\| \to \|B\|$. Instead of generating a coercion function, we directly transform the term. The main motivation behind our design is to generate more efficient terms: ϵ_2 can be understood as a simplified result of the application $c\ \epsilon_1$. By adopting this technique, we aim to skip some reduction steps, ultimately improving the performance of code that relies on coercions. This idea has been discussed in Section 6.1, and it is further optimized in Section 8.4.

To understand the subtyping check, we can ignore ϵ_1 and ϵ_2. In our formulation of subtyping, type constructors like arrows and records distribute over intersections, e.g. $A \to B \,\&\, C$ is equivalent to $(A \to B) \,\&\, (A \to C)$. Such distributivity first appeared in the

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

$$\textsc{Sp-And}\ \frac{}{A \lhd A \,\&\, B \rhd B} \qquad \textsc{Sp-Arrow}\ \frac{B_1 \lhd B \rhd B_2}{A \to B_1 \lhd A \to B \rhd A \to B_2} \qquad \textsc{Sp-Rcd}\ \frac{B_1 \lhd B \rhd B_2}{\{\ell : B_1\} \lhd \{\ell : B\} \rhd \{\ell : B_2\}}$$

$\boxed{\epsilon_1 : A \rhd C \lhd \epsilon_2 : B \rightsquigarrow \epsilon}$ *(Coercive merging)*

$$\textsc{M-Top}\ \frac{\rceil C \lceil \qquad A \lhd C \rhd B}{\epsilon_1 : A \rhd C \lhd \epsilon_2 : B \rightsquigarrow \{\,\}} \qquad \textsc{M-And}\ \frac{\neg\rceil A \,\&\, B \lceil}{\epsilon_1 : A \rhd A \,\&\, B \lhd \epsilon_2 : B \rightsquigarrow \epsilon_1 +\!\!+ \epsilon_2}$$

$$\textsc{M-Arrow}\ \frac{\neg\rceil B_1 \lceil \qquad \neg\rceil B_2 \lceil \qquad (\epsilon_1.|A \to B_1|)\ x : B_1 \rhd B \lhd (\epsilon_2.|A \to B_2|)\ x : B_2 \rightsquigarrow \epsilon}{\epsilon_1 : A \to B_1 \rhd A \to B \lhd \epsilon_2 : A \to B_2 \rightsquigarrow \{|A \to B| \Rrightarrow \lambda x.\, \epsilon\}}$$

$$\textsc{M-ArrowL}\ \frac{\neg\rceil B_1 \lceil \qquad \rceil B_2 \lceil \qquad (\epsilon_1.|A \to B_1|)\ x : B_1 \rhd B \lhd \{\,\} : B_2 \rightsquigarrow \epsilon}{\epsilon_1 : A \to B_1 \rhd A \to B \lhd \epsilon_2 : A \to B_2 \rightsquigarrow \{|A \to B| \Rrightarrow \lambda x.\, \epsilon\}}$$

$$\textsc{M-ArrowR}\ \frac{\rceil B_1 \lceil \qquad \neg\rceil B_2 \lceil \qquad \{\,\} : B_1 \rhd B \lhd (\epsilon_2.|A \to B_2|)\ x : B_2 \rightsquigarrow \epsilon}{\epsilon_1 : A \to B_1 \rhd A \to B \lhd \epsilon_2 : A \to B_2 \rightsquigarrow \{|A \to B| \Rrightarrow \lambda x.\, \epsilon\}}$$

$$\textsc{M-Rcd}\ \frac{\neg\rceil A_1 \lceil \qquad \neg\rceil A_2 \lceil \qquad \epsilon_1.|\{\ell : A_1\}| : A_1 \rhd A \lhd \epsilon_2.|\{\ell : A_2\}| : A_2 \rightsquigarrow \epsilon}{\epsilon_1 : \{\ell : A_1\} \rhd \{\ell : A\} \lhd \epsilon_2 : \{\ell : A_2\} \rightsquigarrow \{|\{\ell : A\}| \Rrightarrow \epsilon\}}$$

$$\textsc{M-RcdL}\ \frac{\neg\rceil A_1 \lceil \qquad \rceil A_2 \lceil \qquad \epsilon_1.|\{\ell : A_1\}| : A_1 \rhd A \lhd \{\,\} : A_2 \rightsquigarrow \epsilon}{\epsilon_1 : \{\ell : A_1\} \rhd \{\ell : A\} \lhd \epsilon_2 : \{\ell : A_2\} \rightsquigarrow \{|\{\ell : A\}| \Rrightarrow \epsilon\}}$$

$$\textsc{M-RcdR}\ \frac{\rceil A_1 \lceil \qquad \neg\rceil A_2 \lceil \qquad \{\,\} : A_1 \rhd A \lhd \epsilon_2.|\{\ell : A_2\}| : A_2 \rightsquigarrow \epsilon}{\epsilon_1 : \{\ell : A_1\} \rhd \{\ell : A\} \lhd \epsilon_2 : \{\ell : A_2\} \rightsquigarrow \{|\{\ell : A\}| \Rrightarrow \epsilon\}}$$

Figure 7.10: Type splitting and coercive merging in λ_{i}^{+}.

system proposed by Barendregt et al. [1983] and is supported by λ_i^+ and F_i^+. We follow the subtyping algorithm design in λ_i^+ [Huang et al. 2021]. It distinguishes types that have a form equivalent to intersection types from others. Such types are called *splittable types* and can be separated into two via *type splitting*, which is defined in Figure 7.10. For example, $A \to B \lhd A \to B \,\&\, C \rhd A \to C$ represents that $A \to B \,\&\, C$ is equivalent to the intersection of $A \to B$ and $A \to C$. The notation A° stands for types that are not splittable, which are called *ordinary types*.

In type splitting, the two split types are outputs. However, they are then used as inputs in the *coercive merging* judgment $\epsilon_1 : A \rhd C \lhd \epsilon_2 : B \rightsquigarrow \epsilon$, as defined in Figure 7.10. If we omit ϵ_1 and ϵ_2 in this judgment, coercive merging characterizes the same relation as type splitting. In other words, removing $B_1 \lhd B \rhd B_2$ from rule S-Split does not change the idea of subtyping. We retain it to better represent the information flow in the subtyping algorithm: in rule S-Split, after being generated from type splitting, B_1 and B_2 are used to coerce the same term e individually, and the coerced results e_1 and e_2 are merged back, guided by the types. Note that, in this process, it is possible to duplicate terms and lead to duplicate labels in the corresponding target record terms.

Soundness of elaboration. The semantics of our source calculus is given via an elaboration, which reflects the compilation of CP. We establish our type-safety proofs on (1) the type safety of our target calculus, and (2) the soundness of elaboration, which connects the source calculus to the target calculus. Specifically, for every well-typed source expression, the typing derivation produces a target term, and we prove that the target term is well-typed in the record calculus. In addition, its type is equivalent to the translated type of the original source expression. The soundness of elaboration is based on the soundness of coercive subtyping, distributive application, and distributive projection, which guarantee that the terms generated from these judgments have the desired types. Note that the premises of these soundness lemmas are coarser than their conclusion: the actual type of the input term does not have to be equivalent to the annotated type. For example, in $\epsilon_1 : A <: B \rightsquigarrow \epsilon_2$, ϵ_1 only needs to have a subtype of $\|A\|$ for ϵ_2 to be correctly typed. This is because, in these coercive relations, the input terms are always used for projection (e.g. in rules S-Arrow and S-Rcd) but never for concatenation. As long as the input terms have sufficient fields, other fields that they have are unimportant.

Theorem 7.11 (Elaboration soundness). *We have that:*

- *If* $\epsilon_1 : A <: B \rightsquigarrow \epsilon_2$ *and* $\Delta \vdash \epsilon_1 : \mathcal{A}$ *and* $\mathcal{A} \subseteq \|A\|$, *then* $\exists \mathcal{B}, \Delta \vdash \epsilon_2 : \mathcal{B}$ *and* $\mathcal{B} \approx \|B\|$.

- *If* $\epsilon_1 : A \bullet \epsilon_2 : B \rightsquigarrow \epsilon_3 : C$ *and* $\Delta \vdash \epsilon_1 : \mathcal{A}$ *and* $\mathcal{A} \subseteq \|A\|$ *and* $\Delta \vdash \epsilon_2 : \mathcal{B}$ *and* $\mathcal{B} \subseteq \|B\|$, *then* $\exists C, \Delta \vdash \epsilon_3 : C$ *and* $C \approx \|C\|$.

- *If* $\epsilon_1 : A \bullet \{\ell\} \rightsquigarrow \epsilon_2 : B$ *and* $\Delta \vdash \epsilon_1 : \mathcal{A}$ *and* $\mathcal{A} \subseteq \|A\|$, *then* $\exists \mathcal{B}, \Delta \vdash \epsilon_2 : \mathcal{B}$ *and* $\mathcal{B} \approx \|B\|$.

- *If* $\Gamma \vdash e \Leftrightarrow A \rightsquigarrow \epsilon$ *then* $\exists \mathcal{A}, |\Gamma| \vdash \epsilon : \mathcal{A}$ *and* $\mathcal{A} \approx \|A\|$.

7.3 Duplicates in Translation and Coherence of Subtyping

With the disjointness constraint enforced in rule Ela-Merge, all elaborated records originating from that rule have distinct labels. However, we still have to take duplicates into account because they can be generated by coercive subtyping. For example, $\epsilon : \mathbb{Z} <: \mathbb{Z} \,\&\, \mathbb{Z} \rightsquigarrow \epsilon +\!\!+ \epsilon$ duplicates ϵ. Note that given ϵ_1, A, and B, there could be more than a single possible ϵ_2, generated by different derivations of subtyping $\epsilon_1 : A <: B \rightsquigarrow \epsilon_2$. Therefore, it is possible to have $\epsilon_1 : A <: B \,\&\, B \rightsquigarrow \epsilon_2 +\!\!+ \epsilon_3$ where ϵ_2 and ϵ_3 are different. In the proof of Theorem 7.11, we show such results do not violate type well-formedness: ϵ_2 and ϵ_3 have equivalent types. Moreover, we conjecture that ϵ_2 and ϵ_3 have the same behavior, since similar results have been proven in the past (semantic coherence or determinism) for variants of λ_i^+ [Bi et al. 2018; Huang et al. 2021]. Because the disjointness restriction in rule Ela-Merge ensures that semantically different terms with equivalent types cannot be introduced into one merge, it is sufficient to distinguish terms by the type indices.

Past coherence results. The technical reason for our coercive subtyping not producing a unique result is that we allow types like $A_1 \,\&\, A_2$ even if a non-top-like type B exists such that $A_1 <: B$ and $A_2 <: B$ both hold, leading to two different subtyping derivation paths for $A_1 \,\&\, A_2 <: B$. One way to ensure the uniqueness of coercions, as utilized by previous work, is to reject such intersection types via a disjointness constraint in type well-formedness [Alpuim et al. 2017]. If intersection types are restricted in this way, we do not even need to worry about duplicates at all. However, unrestricted intersection types are more expressive and are required when encoding bounded polymorphism [Xie et al. 2020]. Moreover, imposing a disjoint constraint on all intersections significantly complicates the proof of type soundness [Alpuim et al. 2017]. That is why all subsequent work [Bi et al. 2018, 2019; Fan et al. 2022; Huang et al. 2021], besides ours, relaxed the restriction on intersections.

For our source calculus to be coherent, it is necessary to ensure that all the coercions generated from the same subtyping judgment are equivalent. Proving this property is challenging, especially considering the main focus we had when designing the formal calculi is to validate the compilation, for which the efficiency is more important. In other words, many design choices in the formalization are driven by efficiency considerations, rather than by considerations for making a proof of coherence easier. For instance, we use a novel and non-standard form of coercive subtyping. Nevertheless, the coherence of the subtyping relation (and the determinism of the casting semantics implied by subtyping) has already been proven in previous work on λ_i^+ [Bi et al. 2018; Huang et al. 2021]. Although our setting differs slightly due to our use of a different target language, the previous results about coherence provide us confidence that the elaboration here is also coherent.

Determinism in a direct operational semantics. Huang et al.'s variant of λ_i^+ uses a direct operational semantics where annotations trigger subtyping checks and act as upcasts at run time, directly manipulating source values. The upcasting process mirrors the approach of algorithmic subtyping, which is similar to how coercions are generated in our subtyping judgments. For instance, the expression $1 : \mathbf{Int} \,\&\, \mathbf{Int}$ evaluates to $1 \,,\, 1$. When an integer is expected, either component can be selected. This type system permits duplicate components in merges. The operational semantics of this variant has been proven to be deterministic and type-safe, providing evidence that no ambiguity arises from subtyping when using disjoint merges.

Coherence for an elaboration semantics. Closer to our work, Bi et al.'s variant of λ_i^+, also known as NeColus, employs an elaboration semantics that is proven to be coherent. The NeColus calculus covers the same set of expressions as our source calculus and also features a syntax-directed bidirectional type system. It uses a different algorithm to decide subtyping, and provides a formulation in declarative style, which is equivalent to ours (see Huang et al.'s work for a formalization of this result). Most of the typing rules are also the same as ours, including the rule for merges and the subsumption rule. The rules for lambda abstraction, application, and record projection are slightly different in terms of requiring more or fewer type annotations, and NeColus does not have the separate relations for distributive application and projection.

The main difference between Bi et al.'s elaboration and ours, is that the target language for NeColus is a different calculus called λ_c. λ_c is a variant of the simply typed λ-calculus extended with records, products, and explicit coercions. In NeColus, merges are translated into pairs. For example, $1 \,,\, \mathbf{true}$ is translated into $\langle 1, \mathbf{true} \rangle$. In the coherence theorem, Bi

et al. prove that such elaboration always leads to equivalent terms in the target language. Their proofs show that the duplication that arises in coercive subtyping does not cause ambiguity in the target language.

Before discussing the coherence proof, let us compare the two elaboration frameworks with some examples. Both λ_c and our λ_r only include the integer type $\mathbb{Z}$ as a representative of primitive types, but we will use **Bool** and **Int** in our examples for demonstration. Besides, we replace the coercions in λ_c by lambda terms and simplify all the elaboration results for easier comparison. Furthermore, we include JavaScript code to show the same situation in the code generated by our compiler.

Example 1. Consider the source expression:

$$1 : \mathbf{Int} \,\&\, \mathbf{Int} : \mathbf{Int}$$

The translation to λ_r is unique:

$$\{|\mathbf{Int}| \Longmapsto \{|\mathbf{Int}| \Longmapsto 1;\ |\mathbf{Int}| \Longmapsto 1\}.|\mathbf{Int}|\}$$

The translation to λ_c has multiple possibilities, including:

$$(\lambda x.\, x.\mathsf{fst})\ \langle 1, 1\rangle \qquad\qquad (\lambda x.\, x.\mathsf{snd})\ \langle 1, 1\rangle$$

This example illustrates a challenging situation that involves two steps: first creating a term corresponding to type **Int** & **Int**, and then selecting a component of type **Int**. In λ_r, the duplication in the first step causes a label conflict, and the second step is deterministic because of the overriding semantics; while in λ_c, it is the second step that brings potential ambiguity. With pairs serving as the target for merges in λ_c, every component in merges can be identified and extracted by position. The position information is analyzed from types during the elaboration. If the merge consists of two terms of the same type, the two positions can be used interchangeably. Therefore, there are two possible coercions that can upcast **Int** & **Int** to **Int**, namely $\lambda x.\, x.\mathsf{fst}$ and $\lambda x.\, x.\mathsf{snd}$. Note that, in a calculus with pairs, these two functions are clearly semantically different since they can be applied to a pair argument like $\langle 1, 2\rangle$, which would produce two different results. But the point is that such pairs with different elements of the same type are never produced by the elaboration, so these two coercions behave identically for the *pairs that can be generated from the elaboration.*

This idea applies, more generally, to types with the same type index in our elaboration, and not just syntactically equal types. Types with the same type index are mutual subtypes and can interchange with each other in subtyping derivation. Another observation from this example is that, whenever there is an overriding in λ_r, there will be multiple possibilities in the translation to λ_c.

Finally, the compiled JavaScript code (without optimization) is as follows:

```
const $1 = {};  $1.int = 1;  $1.int = 1;
const $2 = {};  $2.int = $1.int;
```

The JavaScript code generated by our compiler shares the same overriding semantics as λ_r: `$1.int` is assigned twice, and the second assignment overrides the first one. Note that we have implemented several optimizations in our compiler, including eliminating redundant coercions, so the actual code generated by our compiler is more concise than the one shown here. Since $\mathbf{Int} \,\&\, \mathbf{Int}$ and $\mathbf{Int}$ are equivalent types, no coercions will be inserted in the optimized code. We keep the code unoptimized here to better illustrate the correspondence between the JavaScript code and the λ_r term.

Example 2. Consider another source expression:

$$(\lambda f.\, f) : (\mathbf{Int} \to \mathbf{Int}) \,\&\, (\mathbf{Int} \to \mathbf{Int} \,\&\, \mathbf{Bool}) \to \mathbf{Int} \to \mathbf{Int}$$

This time the translation to λ_r has two possibilities. Here we use A to denote the type $(\mathbf{Int} \to \mathbf{Int}) \,\&\, (\mathbf{Int} \to \mathbf{Int} \,\&\, \mathbf{Bool}) \to \mathbf{Int} \to \mathbf{Int}$:

$$\{|A| \Longrightarrow \lambda f.\, \{|\mathbf{Int} \to \mathbf{Int}| \Longrightarrow \lambda x.\, \{|\mathbf{Int}| \Longrightarrow ((f.|\mathbf{Int} \to \mathbf{Int}|)\, \{|\mathbf{Int}| \Longrightarrow x.|\mathbf{Int}|\}).|\mathbf{Int}|\}\}\}$$
$$\{|A| \Longrightarrow \lambda f.\, \{|\mathbf{Int} \to \mathbf{Int}| \Longrightarrow \lambda x.\, \{|\mathbf{Int}| \Longrightarrow ((f.|\mathbf{Int} \to \mathbf{Int} \,\&\, \mathbf{Bool}|)\, \{|\mathbf{Int}| \Longrightarrow x.|\mathbf{Int}|\}).|\mathbf{Int}|\}\}\}$$

The two possibilities correspond to the translation to λ_c as follows:[2]

$$\lambda f.\, \lambda x.\, f.\mathsf{fst}\; x \qquad\qquad \lambda f.\, \lambda x.\, (f.\mathsf{snd}\; x).\mathsf{fst}$$

This is a case where λ_r has multiple syntactically different translation results. The higher-order function expects a parameter f, which is a record in λ_r that has two fields with label

[2]The λ_c terms look much simpler because they are derived from declarative subtyping for brevity. If we derive the coercions from algorithmic subtyping, the terms will be more complicated:

$$\lambda x_1.\, (\lambda x_2.\, (\lambda x_3.\, \lambda x_4.\, (x_3\; x_4))\; ((\lambda x_5.\, x_5.\mathsf{fst})\; x_2))\; x_1$$

$$\lambda x_1.\, (\lambda x_2.\, (\lambda x_4.\, \lambda x_5.\, ((\lambda x_6.\, x_6.\mathsf{fst})\; (x_4\; x_5)))\; ((\lambda x_7.\, x_7.\mathsf{snd})\; x_2))\; x_1$$

$|\mathbf{Int} \to \mathbf{Int}|$ and label $|\mathbf{Int} \to \mathbf{Int} \,\&\, \mathbf{Bool}|$. These two type indices are different, but the two types are not disjoint. Therefore, if their common supertype is desired in a source context, both fields can be selected. For the purpose of coherence, the two results originating from both sides should behave the same; that is to say, they should have equivalent semantics for the overlapping part of their types (i.e. $\mathbf{Int} \to \mathbf{Int}$). This is also needed for λ_c: the two translated terms should only apply to a pair of functions that have the same behavior, if we only consider the integer part in their return results. Because of disjointness, the only way to create a term with type $(\mathbf{Int} \to \mathbf{Int}) \,\&\, (\mathbf{Int} \to \mathbf{Int} \,\&\, \mathbf{Bool})$ is using one lambda abstraction, such as $(\lambda x.\, x\ ,\ \mathbf{true}) : \mathbf{Int} \to \mathbf{Int} \,\&\, \mathbf{Bool} : (\mathbf{Int} \to \mathbf{Int}) \,\&\, (\mathbf{Int} \to \mathbf{Int} \,\&\, \mathbf{Bool})$. It does not type-check to use a merge of two functions with types $\mathbf{Int} \to \mathbf{Int}$ and $\mathbf{Int} \to \mathbf{Int} \,\&\, \mathbf{Bool}$ as these two types are not disjoint. Therefore, the function that can be selected by the multiple coercions is the same function, which was just duplicated twice.

The compilation to JavaScript may have two possible versions as well:

```
const $1 = {};
$1.fun_fun_int = function ($f) {
  const $2 = {};
  $2.fun_int = function ($3) {
    const $4 = {};
    $4.int = $3.int;
    const $5 = $f.fun_int($4);
    const $6 = {};
    $6.int = $5.int;
    return $6;
  };
  return $2;
};
```

```
const $1 = {};
$1.fun_fun_int = function ($f) {
  const $2 = {};
  $2.fun_int = function ($3) {
    const $4 = {};
    $4.int = $3.int;
    const $5 =
    $f['fun_(bool&int)']($4);
    const $6 = {};
    $6.int = $5.int;
    return $6;
  };
  return $2;
};
```

Note that the only difference is the value of `$5`: one is created by applying `$f.fun_int` and the other by applying `$f['fun_(bool&int)']` (n.b. `$f.fun_(bool&int)` does not work because the label contains special characters). The two versions will behave identically, so it does not matter which one is actually generated. Similarly to the situation in λ_r, as long as the CP code is well-typed and is compiled to JavaScript by our compiler, `$f.fun_int` and `$f['fun_(bool&int)']` originate from the same function and behave the same as only the integer part of the return value (i.e. `$5.int`) is used. Again, the JavaScript code is deliberately kept unoptimized to show the correspondence with the λ_r terms. In the optimized code for the first version, for instance, the creation of `$4` and `$6` can be eliminated.

7.4 A Sketch of the Coherence Proof

In this section, we will provide a summary of the key steps in the coherence proof for NeColus [Bi et al. 2018]. After that, we will discuss how to adapt the proof to our setting.

The coherence result in NeColus is established using a semantic approach based on *logical relations* [Biernacki and Polesiuk 2015; Tait 1967]. We will use E, V, τ, and Δ to represent the expressions, values, types, and typing contexts in λ_c, the target language for NeColus.

Coherence via contextual equivalence. The coherence theorem proven for NeColus is as follows:

Theorem 7.12 (Coherence of NeColus). *If* $\Gamma \vdash e \Leftrightarrow A$ *then* $\Gamma \vdash e \simeq_{ctx} e : A$.

Here e is a source expression, so it is to say that a well-typed source expression is contextually equivalent to itself. The notation $\Leftrightarrow$ stands for both bidirectional typing modes, namely inference ($\Rightarrow$) and checking ($\Leftarrow$). Contextual equivalence is defined as:

Definition 7.13 (Contextual equivalence in NeColus). $\Gamma \vdash e_1 \simeq_{ctx} e_2 : A \triangleq$

$\forall E_1\, E_2\, C\, D$, *if* $\Gamma \vdash e_1 \Leftrightarrow A \rightsquigarrow E_1$ *and* $\Gamma \vdash e_2 \Leftrightarrow A \rightsquigarrow E_2$ *and* $C : (\Gamma \Leftrightarrow A) \mapsto (\cdot \Leftrightarrow \mathbb{Z}) \rightsquigarrow D$
then $D[E_1] \simeq D[E_2]$.

The intuition behind the definition is that two elaborated terms (E_1 and E_2) should be considered equivalent if, for any well-typed contexts (that could be generated during the elaboration), plugging either of them in makes no difference to the final evaluation result. Here C and D stand for source and target *expression contexts*, respectively. An expression context is an expression that contains a hole $[\cdot]$. The typing judgment for contexts $C : (\Gamma \Leftrightarrow A) \mapsto (\cdot \Leftrightarrow \mathbb{Z}) \rightsquigarrow D$ means that, given any well-typed NeColus expression $\Gamma \vdash e \Leftrightarrow A$, we have $\cdot \vdash C[e] \Leftrightarrow \mathbb{Z}$, and the source context C corresponds to a target context D in elaboration, which, similarly, make $D[E_1]$ and $D[E_2]$ have type $\mathbb{Z}$. In this definition, $\simeq$ stands for Kleene equality. That is to say, $D[E_1]$ and $D[E_2]$ evaluate to the same integer. Here the definition intentionally excludes the contexts that cannot be obtained from a well-typed NeColus expression. For example, the source expression $(\lambda x.\, x) : \mathbb{Z} \,\&\, \mathbb{Z} \to \mathbb{Z}$ can be elaborated into $\lambda x.\, x.\mathsf{fst}$ or $\lambda x.\, x.\mathsf{snd}$. To judge whether they have the same contribution to computation, we should not consider the target expression context $[\cdot]\langle 1, 2\rangle$, that is, applying the elaborated function to $\langle 1, 2\rangle$, because its corresponding source term violates the disjointness restriction and thus is not well-typed.

$$
\begin{aligned}
(V_1, V_2) \in \mathcal{V}[\![\mathbb{Z}; \mathbb{Z}]\!] &\triangleq \exists n,\ V_1 = V_2 = n \\
(V_1, V_2) \in \mathcal{V}[\![\tau_1 \to \tau_2; \tau_1' \to \tau_2']\!] &\triangleq \forall (V, V') \in \mathcal{V}[\![\tau_1; \tau_1']\!],\ (V_1\, V, V_2\, V') \in \mathcal{E}[\![\tau_2; \tau_2']\!] \\
(\{\ell = V_1\}, \{\ell = V_2\}) \in \mathcal{V}[\![\{\ell : \tau_1\}; \{\ell : \tau_2\}]\!] &\triangleq (V_1, V_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!] \\
(\langle V_1, V_2\rangle, V_3) \in \mathcal{V}[\![\tau_1 \times \tau_2; \tau_3]\!] &\triangleq (V_1, V_3) \in \mathcal{V}[\![\tau_1; \tau_3]\!] \wedge (V_2, V_3) \in \mathcal{V}[\![\tau_2; \tau_3]\!] \\
(V_3, \langle V_1, V_2\rangle) \in \mathcal{V}[\![\tau_3; \tau_1 \times \tau_2]\!] &\triangleq (V_3, V_1) \in \mathcal{V}[\![\tau_3; \tau_1]\!] \wedge (V_3, V_2) \in \mathcal{V}[\![\tau_3; \tau_2]\!] \\
(V_1, V_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!] &\triangleq \textbf{true} \quad \text{otherwise} \\
(E_1, E_2) \in \mathcal{E}[\![\tau_1; \tau_2]\!] &\triangleq \exists V_1\, V_2,\ E_1 \to^* V_1 \wedge E_2 \to^* V_2 \wedge (V_1, V_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!]
\end{aligned}
$$

Figure 7.11: Logical relations for λ_c (the target calculus of NeColus).

Heterogeneous logical relations. NeColus introduces two heterogeneous logical relations that relates values and terms, respectively, in the target language λ_c, as shown in Figure 7.11. The logical relation is designed to capture the intuition of coherent expressions – those that are safe to coexist in pairs, as they are unambiguous in every valid context.

First, $\mathcal{V}[\![\tau_1; \tau_2]\!]$ relates all duplicate values. For example, $(1, 1) \in \mathcal{V}[\![\mathbb{Z}; \mathbb{Z}]\!]$ because it never leads to ambiguity. Second, it is proven that all disjoint values are also related. For example, as long as we have $\mathbb{Z} * \{\ell : \mathbb{Z}\}$, we also have $(1, \{\ell = 1\}) \in \mathcal{V}[\![\mathbb{Z}; \{\ell : \mathbb{Z}\}]\!]$. For product types, the relation distributes over the product constructor $\times$. This reflects the disjointness of intersection types, that is, $A \,\&\, B * C$ if and only if $A * C$ and $B * C$. Finally, $\mathcal{E}[\![\tau_1; \tau_2]\!]$ states that E_1 and E_2 are related if they evaluate to related values. The relation is then lifted to open terms via a semantic interpretation of typing contexts, and the logical equivalence is defined in a standard way: two open terms are related if every pair of related closing substitutions make them related. $\Delta \vdash E_1 \simeq_{log} E_2 : \tau$ denotes that two well-typed expressions are logically equivalent with respect to the same typing context and the same type.

Fundamental property. In NeColus, a fundamental property is proven, stating that any two λ_c terms elaborated from the same NeColus expression are related by the logical relation. Here $\|\Gamma\|$ and $\|A\|$ stand for the translation of typing contexts and types from NeColus to λ_c.

Theorem 7.14 (Fundamental property of NeColus). *If* $\Gamma \vdash e \Leftrightarrow A \rightsquigarrow E_1$ *and* $\Gamma \vdash e \Leftrightarrow A \rightsquigarrow E_2$, *then* $\|\Gamma\| \vdash E_1 \simeq_{log} E_2 : \|A\|$.

Note that $\Delta \vdash E \simeq_{log} E : \tau$ does not hold for every well-typed target E. For example, the logical relation does not consider $\langle 1, 2\rangle$. Since there is no possible elaboration (the source expression $1 \,,\, 2$ violates the disjoint constraint), this does not prevent the fundamental

property. Actually the limitation on coherent products helps the logical relation to accept some semantically equivalent terms like $\lambda x.\, x.\mathsf{fst}$ and $\lambda x.\, x.\mathsf{snd}$: they are two translations of $(\lambda x.\, x) : \mathbf{Int} \,\&\, \mathbf{Int} \to \mathbf{Int}$. With the assumption that the two integers in the pair argument are related by the logical relation, choosing either one leads to the same result.

Soundness. Since the fundamental property has shown that different elaborations of the same NeColus expression are logically equivalent, the remaining step is to show that logical equivalence is sound with respect to contextual equivalence.

A compatibility lemma for coercions is proven during the establishment of the fundamental property: if two terms are related by the logical relation, after applying a coercion to one term, they are still related. Based on this lemma, we can prove that the logical relation is preserved by all well-typed NeColus contexts, including the contexts that coverts a term to an integer result, for which the logical relation implies Kleene equality. Then it is straightforward to show that the logical relation is sound.

Theorem 7.15 (Soundness w.r.t. contextual equivalence in NeColus). *If* $\Gamma \vdash e_1 \Leftrightarrow A \rightsquigarrow E_1$ *and* $\Gamma \vdash e_2 \Leftrightarrow A \rightsquigarrow E_2$ *and* $\|\Gamma\| \vdash E_1 \simeq_{log} E_2 : \|A\|$, *then* $\Gamma \vdash e_1 \simeq_{ctx} e_2 : A$.

Finally, the coherence theorem follows directly from the fundamental property and the soundness result. In other words, we can conclude that any two λ_c terms elaborated from the same NeColus expression are contextually equivalent.

Adapting the proof to our source language: a sketch. While our compilation scheme makes some different design choices for efficiency, it essentially shares the same principle of coherence with NeColus. Here we provide a sketch of how to adapt the ideas from the NeColus proof to our work, although we do not provide a full proof.

We can define two logical relations for values ($\mathcal{V}_\nabla[\![\mathcal{A}; \mathcal{B}]\!]$) and terms ($\mathcal{E}_\nabla[\![\mathcal{A}; \mathcal{B}]\!]$) in λ_r, as presented in Figure 7.12. In this definition, we relate two records if they do not cause ambiguity under any contexts that can be elaborated from our source calculus. That is to say, for two records to be related, any pair of fields from them, as long as their labels correspond to overlapping source types, must have related fields. For example, relating $\{|\mathbf{Int} \to \mathbf{Int} \,\&\, \mathbf{Bool}| \Longmapsto v_1\}$ and $\{|\mathbf{Int} \to \mathbf{Int}| \Longmapsto v_2\}$ in the logical relation requires v_1 and v_2 to be related. Considering a source expression context $[\cdot] : \mathbf{Int} \to \mathbf{Int}$, it applies to both records and leads to two projections that extract v_1 and v_2, respectively. So v_1 and v_2 should be equivalent. Besides, disjoint terms are also related. For example, $\{|\{\ell_1 : A\}| \Longmapsto v_1\}$ and $\{|\{\ell_2 : B\}| \Longmapsto v_2\}$ correspond to two source records of type $\{\ell_1 : A\}$ and type $\{\ell_2 : B\}$. If $\ell_1 = \ell_2$, their relation should be decided by the fields; if $\ell_1 \neq \ell_2$, they are always related

$$
\begin{aligned}
(v_1, v_2) \in \mathcal{V}_\nabla[\![\mathbb{Z}; \mathbb{Z}]\!] &\triangleq \exists n,\ v_1 = v_2 = n \\
(v_1, v_2) \in \mathcal{V}_\nabla[\![\mathcal{A}_1 \to \mathcal{B}_1; \mathcal{A}_2 \to \mathcal{B}_2]\!] &\triangleq \forall (v, v') \in \mathcal{V}_\nabla[\![\mathcal{A}_1; \mathcal{A}_2]\!], \\
&\quad (v_1\ v, v_2\ v') \in \mathcal{E}_\nabla[\![\mathcal{B}_1; \mathcal{B}_2]\!] \\
(\{\ell_1 \Longmapsto v_1\}, \{\ell_2 \Longmapsto v_2\}) \in \mathcal{V}_\nabla[\![\{\ell_1 \Longmapsto \mathcal{A}_1\}; \{\ell_2 \Longmapsto \mathcal{A}_2\}]\!] &\triangleq (v_1, v_2) \in \mathcal{V}_\nabla[\![\mathcal{A}_1; \mathcal{A}_2]\!] \quad \text{if } \ell_1 = \ell_2 = \overline{\mathbb{Z}} \\
&\quad \text{or } (\exists Ts_i,\ \ell_1 = \overline{Ts_1 \to Ts_2} \wedge \ell_2 = \overline{Ts_3 \to Ts_4}) \\
&\quad \text{or } (\exists \ell\ Ts_i,\ \ell_1 = \overline{\{\ell : Ts_1\}} \wedge \ell_2 = \overline{\{\ell : Ts_2\}}) \\
(\{\ell_1 \Longmapsto v_1;\ \ldots;\ \ell_n \Longmapsto v_n\}, v') \in \mathcal{V}_\nabla[\![\{\ell_1 \Longmapsto \mathcal{A} \mid \rho\}; \mathcal{B}]\!] &\triangleq (\{\ell_1 \Longmapsto v_1\}, v') \in \mathcal{V}_\nabla[\![\{\ell_1 \Longmapsto \mathcal{A}\}; \mathcal{B}]\!] \wedge \\
&\quad (\{\ell_2 \Longmapsto v_2;\ \ldots;\ \ell_n \Longmapsto v_n\}, v') \in \mathcal{V}_\nabla[\![\rho; \mathcal{B}]\!] \\
(v', \{\ell_1 \Longmapsto v_1;\ \ldots;\ \ell_n \Longmapsto v_n\}) \in \mathcal{V}_\nabla[\![\mathcal{B}; \{\ell_1 \Longmapsto \mathcal{A} \mid \rho\}]\!] &\triangleq (v', \{\ell_1 \Longmapsto v_1\}) \in \mathcal{V}_\nabla[\![\mathcal{B}; \{\ell_1 \Longmapsto \mathcal{A}\}]\!] \wedge \\
&\quad (v', \{\ell_2 \Longmapsto v_2;\ \ldots;\ \ell_n \Longmapsto v_n\}) \in \mathcal{V}_\nabla[\![\mathcal{B}; \rho]\!] \\
(v_1, v_2) \in \mathcal{V}_\nabla[\![\mathcal{A}_1; \mathcal{A}_2]\!] &\triangleq \textbf{true} \quad \text{otherwise} \\
(\epsilon_1, \epsilon_2) \in \mathcal{E}_\nabla[\![\mathcal{A}_1; \mathcal{A}_2]\!] &\triangleq \exists v_1\ v_2,\ \epsilon_1 \to^* v_1\ \wedge\ \epsilon_2 \to^* v_2\ \wedge \\
&\qquad (v_1, v_2) \in \mathcal{V}_\nabla[\![\mathcal{A}_1; \mathcal{A}_2]\!]
\end{aligned}
$$

Figure 7.12: Logical relations for λ_r (our target calculus).

because of disjointness. The relation of terms can be lifted to open terms like in NeColus. We expect the logical equivalence derived from the logical relation to be compatible with our coercive subtyping, and a fundamental property should follow. The main rationale is that the overlapping part must have the same origin, restricted by type disjointness. What coercions do is to decompose and recompose the behaviors according to the types (labels). Disjoint terms will be distinguished clearly in the process. A field of $|\{\ell_1 : A\}|$, for example, will not be used to build a field of $|\{\ell_2 : A\}|$ if $\ell_1 \neq \ell_2$. No λ_{i}^+ context should violate the derived logical equivalence. We anticipate a similar theorem asserting that logical equivalence implies contextual equivalence, provided that contextual equivalence is defined in a manner analogous to NeColus. Consequently, the coherence theorem would follow, in the style of Theorem 7.12.

In summary, this chapter presents a mechanized formalization of a compilation scheme that ensures type safety in the translation from λ_{i}^+ to λ_r. By focusing on the key ideas of the elaboration semantics, it establishes a foundation for understanding the compilation based on type-indexed records. The issue of coherence is also discussed, and a sketch of the coherence proof is provided. However, the exclusion of parametric polymorphism and optimizations limits the scope of this chapter, leaving room for future work.

8 Implementation Details

This chapter introduces our concrete implementation of the CP compiler that targets JavaScript. The full set of compilation rules can be found in Appendix C.

8.1 From Elaboration Semantics to JavaScript Code

In the elaboration semantics presented in Chapter 7, we use a λ-calculus with extensible records as the target. In the actual compilation, records are modeled as JavaScript objects, and type indices are realized as objects' property names. For example, $48 \,,\, \textbf{true}$ compiles to `{ int: 48, bool: true }` in JavaScript. Besides, record concatenation can be directly represented by object merging using the spread syntax like `{ ...obj1, ...obj2 }`.

As we have mentioned, the formalized target language is a functional calculus, but JavaScript is an imperative language. The mismatch in programming paradigms is an important consideration when implementing the compilation to JavaScript. We consider two designs of target forms: one is based on *static single assignment* (SSA) [Cytron et al. 1991], and the other is based on *destination-passing style* (DPS) [Shaikhha et al. 2017]. We eventually choose the latter due to performance reasons, which we will explain next.

Reducing intermediate objects. In our initial design based on the SSA form, every subterm in a merge creates a new object in the compiled JavaScript code. Consequently, there will be too many intermediate objects that are useless. For example, consider the merge $48 \,,\, \textbf{true} \,,\, \texttt{chr}\ 32$, where `chr` is a function that converts an integer to a character, and the compiled function has been stored in `$chr`. We need to create six JavaScript objects in the SSA-based form, as shown in Figure 8.1a. As mitigation, we adopt a new design based on DPS. We just create one object for the merge (e.g. `$1` in Figure 8.1b) and pass the variable down to subterms to update their corresponding properties. To further prevent the function application (e.g. `chr 32`) from creating any intermediate object, we add an extra parameter when compiling all functions, including `chr`. The destination object (e.g. `$1`) is passed to the compiled function as the last argument, and the function body will directly write to that argument instead of creating a new object. In other words, `$1` is an

```
const $1 = { int: 48 };
const $2 = { bool: true };
const $3 = { ...$1, ...$2 };
const $4 = { int: 32 };
const $5 = $chr.fun_char($4);
const $6 = { ...$3, ...$5 };
```

(a) Before optimization: 6 objects.

```
const $1 = {};
$1.int = 48;
$1.bool = true;
const $2 = {};
$2.int = 32;
$chr.fun_char($2, $1);
```

(b) After optimization: 2 objects.

Figure 8.1: Simplified JavaScript code for $48 \, , \, \textbf{true} \, , \, \mathsf{chr}\ 32$.

output parameter while `$2` is an input. As a result, we reduce the creation of six objects to only two objects and avoid two object concatenations, as shown in Figure 8.1b.

8.2 Parametric Polymorphism

The compilation of polymorphic terms is not formalized in Chapter 7, so here we introduce our solution in more detail. The challenge posed by parametric polymorphism is mainly because type arguments are unknown until type application. For those terms whose types contain free variables, we can only generate type indices at run time. For example, Figure 8.2a shows the simplified JavaScript code for the following definition in CP:

```
poly = /\(A * Int). \(x: A) → x , 48;
```

Here the notation `/\(A * Int)` represents a type parameter `A` bound by a Λ-function where `A` is disjoint with **Int**. The `poly` function can be typed as $\forall A * \textbf{Int}.\ A \to A \,\&\, \textbf{Int}$. We employ a *de Bruijn index* [de Bruijn 1972] to represent the bound variable `A`, so the Λ-function's type index is `"forall_fun_(1&int)"`. In contrast, the inner λ-function's type index is more intriguing. Before we introduce our approach, let us first see an example of applying `poly`:

```
poly @(String & Bool) ("foo" , true)
```

The type parameter is instantiated with an intersection type (**String** & **Bool**), and Figure 8.2b shows the simplified JavaScript code for the application. After `poly` is instantiated, the inner λ-function will be used with concrete type indices (e.g. `"fun_(bool&int&string)"`) instead of something like `"fun_(A&int)"`. To achieve this, we pass the instantiated type as an argument to the outer Λ-function. The argument is in the form of a JavaScript array, which consists of the component types in the case of an intersection type. The array is empty for top-like types, or it is a singleton array for non-intersection, non-top-like cases. We also predefine a `toIndex` function to help generate the type index based on the runtime instantiation, whose definition is shown in Figure 8.2c. The `toIndex` function accepts an

```
const $poly = {};
$poly['forall_fun_(1&int)'] = function ($A, $1) {
  $1['fun_' + toIndex([ ...$A, 'int' ])] = function ($x, $2) {
    for (const $A$elem of $A) $2[$A$elem] = $x[$A$elem];
    $2.int = 48;
  };
};
```

(a) `poly = /\(A * Int). \(x: A) → x , 48.`

```
const $3 = {};  const $4 = {};
$poly['forall_fun_(1&int)']([ 'string', 'bool' ], $4);
const $5 = {};  $5.string = 'foo';  $5.bool = true;
$4['fun_(bool&int&string)']($5, $3);
```

(b) `poly @(String & Bool) ("foo" , true).`

```
function toIndex(tt) {
  const ts = tt.sort().filter((t, i) ⇒ t === 0 || t !== tt[i-1]);
  if (ts.length === 1) return ts[0];
  else return '(' + ts.join('&') + ')';
};
```

(c) `toIndex`: an auxiliary function for generating type indices at run time.

Figure 8.2: Simplified JavaScript code for polymorphic terms.

array of types and generates their intersection's type index. To fit in with the notion of equivalent types in Section 6.3, it will sort and deduplicate the component types. Now that some type indices are dynamically generated, we have to use `obj[toIndex($A)]` instead of `obj.name` directly. Such a feature is called first-class labels [Leijen 2004] and is supported in JavaScript via computed property names with brackets. The situation in Figure 8.2a is a bit more complicated because the type of the inner λ-function is $A \to A \,\&\, \mathbf{Int}$, so the dynamically computed type index is `"fun_" + toIndex([ ...$A, "int" ])`.

As illustrated above, the compiled code for polymorphic definitions incurs overhead due to the dynamic computation of type indices. In mainstream languages, parametric polymorphism is implemented via either *erasure* [Igarashi et al. 2001] or *monomorphization* [Griesemer et al. 2020]. Our compilation scheme cannot erase type information at run time, so monomorphization is a potential direction for improving the performance of polymorphic code. We leave this for future work.

```
const $1 = {};
$1['rcd_x:int'] =
    $1['rcd_y:int'];
```

(a) By value.

```
const $1 = {};
$1.__defineGetter__(
  'rcd_x:int',
  function () {
    return $1['rcd_y:int'];
  });
```

(b) By thunk.

```
const $1 = {};
$1.__defineGetter__(
  'rcd_x:int',
  function () {
    delete this['rcd_x:int'];
    return this['rcd_x:int'] =
    $1['rcd_y:int'];
  });
```

(c) By memoized thunk.

Figure 8.3: Simplified JavaScript code for `{ x = this.y }`.

8.3 Lazy Evaluation

In the most recent work by Fan et al. [2022], F_i^+ is formalized as a call-by-name calculus to correctly model trait instantiation. A simple example is as follows:

```
type Rcd = { x: Int; y: Int };
new (trait [this: Rcd] ⇒ { x = this.y; y = 48 })
```

The program will not terminate if evaluated using the call-by-value strategy. That is why Fan et al. go for a call-by-name semantics and evaluate record fields lazily. However, a naive call-by-name implementation may evaluate the same record field more than once and cause a significant slowdown. Even with proper memoization (call-by-need), the performance of generated code is still not ideal as JavaScript does not support lazy evaluation natively. In our implementation, we employ a hybrid strategy: only self-annotated trait fields are lazily evaluated, and other language constructs including function applications are strictly evaluated. This approximates the semantics in conventional OOP languages, which are call-by-value in terms of initializing fields and calling methods, except for lazy fields.

To better illustrate different evaluation strategies for record fields, we show three code snippets generated for the field `{ x = this.y }`. Figure 8.3a employs strict evaluation, but instead of non-termination, the issue here is that the field `y` is not yet available, so field `x` is unexpectedly assigned **`undefined`**. This issue severely limits self-references, and the code in Figure 8.3a would be broken. Thus we need a better approach. Figure 8.3b resolves the issue by adding a *thunk*. The field is wrapped in a getter, and thus the computation of `this.y` is delayed until the whole record is constructed with the field `y` available. But note that the getter is called every time the field is accessed. This is undesirable as the computation in the thunk would be triggered on every access to the field. We optimize this by

memoizing fields using *smart getters*[1]. As shown in Figure 8.3c, once the field is evaluated, the getter is deleted, and the value is stored instead. Our implementation automatically detects self-annotated trait fields and applies the last approach to them; for other cases, the first approach is applied. By this means, self-references and trait instantiation in CP are correctly supported while maintaining good performance for other language constructs.

8.4 Important Optimizations

Eliminating redundant coercions. In the rules of coercive subtyping shown in Figure 7.9, even $A <: A$ may go through a lot of rules when A is a complex intersection type. However, as long as we encounter subtyping between *equivalent* types, we do not need any coercion code. To implement this optimization, an immediate idea would be:

- Adding a special case to rule Ela-Sub such that if $A \coloneqq B$ then $\epsilon_1 = \epsilon_2$.

Unfortunately, this rule does not deal with many important cases. For instance, when checking $A \to B <: A \to C$, we would like to avoid applying coercions to the inputs of the function (since they have the same type). Therefore, besides the rule above, we may consider:

- Adding an extra rule (say S-Equiv in Figure 8.4) such that if $A \coloneqq B$ then $\epsilon : A <: B \rightsquigarrow \epsilon$.

However, this idea is incorrect when an intersection type occurs on the left-hand side. For example, when upcasting 48 , **true** to type **Int**, the coercion is missing in the subtyping derivation:

$$\dfrac{\dfrac{}{\textbf{Int} \coloneqq \textbf{Int}}}{\{\texttt{int} \mapsto 48; \texttt{bool} \mapsto \textbf{true}\} : \textbf{Int} <: \textbf{Int} \rightsquigarrow \{\texttt{int} \mapsto 48; \texttt{bool} \mapsto \textbf{true}\}}\ \text{S-Equiv}}{\{\texttt{int} \mapsto 48; \texttt{bool} \mapsto \textbf{true}\} : \textbf{Int} \,\&\, \textbf{Bool} <: \textbf{Int} \rightsquigarrow \{\texttt{int} \mapsto 48; \texttt{bool} \mapsto \textbf{true}\}}\ \text{S-AndL}$$

Observing that rules S-AndL and S-AndR are the root cause of missing coercions, we add an extra flag that indicates the applicability of rule S-Equiv to fix the latter idea. As shown in Figure 8.4, by default ($<:^{+}$) the optimization S-Equiv can apply, but it will be disabled ($<:^{-}$) in the derivations of rule S-AndL (as well as rule S-AndR), and re-enabled in derivations of rule S-Arrow (as well as rules S-All and S-Rcd). In some rules, the flag

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/get#smart_self-overwriting_lazy_getters

$\boxed{\epsilon_1 : A <:^{\pm} B \rightsquigarrow \epsilon_2}$ *(Optimized coercive subtyping)*

$$\frac{A \fallingdotseq B}{\epsilon : A <:^{+} B \rightsquigarrow \epsilon}\ \text{S-Equiv} \qquad \frac{\epsilon : A <:^{-} C \rightsquigarrow \epsilon'}{\epsilon : A \,\&\, B <:^{\pm} C \rightsquigarrow \epsilon'}\ \text{S-AndL}$$

$$\frac{x : B_1 <:^{+} A_1 \rightsquigarrow \epsilon_1 \qquad (\epsilon.|A_1 \to A_2|)\ \epsilon_1 : A_2 <:^{+} B_2 \rightsquigarrow \epsilon_2}{\epsilon : A_1 \to A_2 <:^{\pm} B_1 \to B_2 \rightsquigarrow \{|B_1 \to B_2| \Mapsto \lambda x.\, \epsilon_2\}}\ \text{S-Arrow}$$

Figure 8.4: Selected rules for optimized coercive subtyping.

does not matter, so we use $<:^{\pm}$ to mean that both cases apply. The complete rules targeting JavaScript can be found in Appendix C.

For a simple example of the optimized code, we consider a CP function that takes a parameter of type **Int&Bool**, and we pass a merge of type **Bool&Int** as its argument:

```
(\(x:Int&Bool) → x:Int) (true , 48)
```

The compiled JavaScript for this function application is:

```
const $1 = {};
const $2 = {};  $2.fun_int = function ($x, $3) { $3.int = $x.int };
const $4 = {};  $4.bool = true;  $4.int = 48;
$2.fun_int($4, $1);
```

Although it goes through the rule A-Arrow to check the argument of type **Bool&Int** against its supertype **Int&Bool**, there is no coercion inserted for `$4`. This is just due to the elimination of coercions for subtyping between equivalent types (**Bool** & **Int** $\fallingdotseq$ **Int** & **Bool**).

Optimizing record projection. In the formalization of F^{+}_{i} by Fan et al. [2022], a record with more than one label can only be projected after a coercion is inserted to remove irrelevant labels. For example, `{x = 1; y = 2}.x` in CP has to be elaborated into `({x = 1},{y = 2} : {x: Int}).x` in F^{+}_{i} to make the record projection work. If a similar technique is used in the CP compiler, there will be too many coercions that lead to poor performance. This flaw is because the formalization of F^{+}_{i} reuses the rules of distributive application for record projection. However, the semantics for distributive projection is slightly different: we do not require that every component of a record can be projected by the same label. Therefore, we add rule JP-RcdNeq to safely ignore irrelevant fields. The complete

```
const $id = {};
$id['fun_(double&int&string)'] = function ($x, $1) {
  $1.double = $x.double;  $1.int = $x.int;  $1.string = $x.string;
};
```

(a) Before optimization: always copying fields from $x to $1.

```
const $id = {};
$id['fun_(double&int&string)'] = function ($x, $1) {
  if ($1) { $1.double = $x.double;  $1.int = $x.int;  $1.string = $x.string; }
  return $x;
};
```

(b) After optimization: no copying when $1 is **undefined**.

Figure 8.5: Simplified JavaScript code for `id (x: Double&Int&String) = x`.

```
const $1 = {};
const $2 =
    $id['fun_(double&int&string)']($y);
const $3 = {};  $3.int = 1;
$1.int = $2.int + $3.int;
```

(a) id $y + 1$.

```
const $1 = {};
$id['fun_(double&int&string)']($y,
    $1);
$1.bool = true;
```

(b) id y , **true**.

Figure 8.6: Simplified JavaScript code for applying id.

rules have been formalized in Figure 7.8. By implementing the new rules for distributive projection, the compiled JavaScript can directly handle projection for multi-field records.

Reducing object copying. Although the DPS-based design reduces the number of intermediate objects, it sometimes introduces unnecessary object copying. For example, consider a function `id` that takes a parameter of type **`Double&Int&String`** and returns it as is:

```
id (x: Double&Int&String) = x;
```

The compiled JavaScript code based on DPS is shown in Figure 8.5a. The function always copies the fields from the parameter `$x` to the destination object `$1`. Such object copying is necessary if the result of the function call is part of a merge (e.g. id y , **true**), but it is inefficient otherwise (e.g. id $y + 1$). To optimize this, we do not pass a destination object to the function if the function call does not occur in a merge, as shown in Figure 8.6a. The function call in a merge still follows the DPS design we discussed earlier, as shown

```
const $con = {};
$con['fun_(bool&int)'] = function ($_, $1) {
  $2 = $1 || {};
  $2.bool = false;  $2.int = 0;
  return $2;
};
```

Figure 8.7: Simplified JavaScript code for `con (_: Top) = false,0`.

in Figure 8.6b. To distinguish these two cases, we add a dynamic check in the compiled function to avoid unnecessary copying when the destination (`$1`) is **`undefined`**, as shown in Figure 8.5b.

Optional destination objects. We have avoided unnecessary copying when a destination is absent, but what about the dual case: how to avoid a fresh object when a destination is present? Since the body of the previous function `id` is just a variable, no fresh object is created in any case. Let us consider another function `con`, which returns a merge of **`false`** and `0` regardless of the input:

```
con (_: Top) = false,0;
```

The compiled JavaScript code is shown in Figure 8.7. `$1 || {}` on the first line of the function body checks if the destination (`$1`) is present. If it is, `$2` is just an alias of `$1`; otherwise, a fresh object `{}` is created and assigned to `$2`. By this means, we avoid creating a fresh object if the destination is provided.

Avoiding boxing/unboxing. Although extensible records, or more specifically, JavaScript objects, serve as an excellent target for merges in CP, they are not so efficient when only primitive values and their computation are involved. For example, when compiling $1+2$ to JavaScript, the naive approach is shown in Figure 8.8a. It would first create two objects for 1 and 2, then access their `"int"` fields to get the values, perform the addition, and finally create a new object for the result. These tedious wrapping/unwrapping of objects are similar to boxing/unboxing in Java and are unnecessary in this case. As shown in Figure 8.8b, we do not need to create any objects when compiling $1 + 2$. Instead of $\{\texttt{int} \Longmapsto 1\}$, we can directly use 1 if it is not part of a merge. This is a significant optimization for programs that heavily rely on arithmetic. In addition to integers, we also optimize the compilation for floating-point numbers, boolean values, and strings in a similar way.

The optimization becomes a bit more complicated when interacting with parametric polymorphism, as we cannot statically know the type of a polymorphic term. Therefore,

```
const $1 = {};  $1.int = 1;
const $2 = {};  $2.int = 2;
const $3 = {};  $3.int = $1.int +
    $2.int;
```

(a) Before optimization: 3 objects.

```
const $1 = 1;
const $2 = 2;
const $3 = $1 + $2;
```

(b) After optimization: 0 objects.

Figure 8.8: Simplified JavaScript code for 1 + 2.

to guarantee a consistent runtime representation of primitive values, we also add runtime primitiveness checks and perform the optimization to those polymorphic terms whose types are instantiated primitive. For an artificial example, consider the following CP code:

```
idPartly A B (x: A&B): B = x;
idPartly @String @Int        ("foo" , 48)         --> 48
idPartly @String @(Int&Bool) ("foo" , 48 , true)  --> 48 , true
```

We know that the return type of `idPartly` is `B`, but we cannot statically know if `B` is a primitive type. We should do the optimization in the first application above but not in the second one. Such a decision is made at run time by checking `B`'s primitiveness. By this means, we can make sure that the runtime representation of primitive values is consistent and efficient.

8.5 Selected Rules for Destination-Passing Style

To help to understand the implementation of destination-passing style, we present the compilation process in the form of type-directed rules. The full set of rules can be found in Appendix C. We select a few representative ones here. Besides destination-passing style, the design of the compilation rules is greatly influenced by the optimization that reduces object copying (discussed in Section 8.4). We will revisit the examples in Figure 8.5, Figure 8.6, and Figure 8.7 to show how the optimized code is generated systematically.

In the compilation rules, we have three kinds of destinations:

- z stands for a non-empty destination, where the context is a merge. We will store the current result as a field in the destination object z.
- $y?$ stands for an optional destination, where the context is a function body. Since the last parameter of a compiled function can be **`undefined`**, we do not statically know if the destination is present.

Destinations $dst ::= \mathbf{nil} \mid y? \mid z$

JavaScript code $\mathtt{J} ::= \varnothing \mid \mathtt{J}_1; \mathtt{J}_2 \mid \texttt{code}$

$\boxed{\Gamma; dst \vdash e \Leftrightarrow A \rightsquigarrow \mathtt{J} \mid z}$ *(Type-directed compilation)*

J-Var
$$\frac{x : A \in \Gamma}{\Gamma; z \vdash x \Rightarrow A \rightsquigarrow \texttt{copy(z, x);} \mid z}$$

J-VarOpt
$$\frac{x : A \in \Gamma}{\Gamma; y? \vdash x \Rightarrow A \rightsquigarrow \texttt{if (y) copy(y, x);} \mid x}$$

J-VarNil
$$\frac{x : A \in \Gamma}{\Gamma; \mathbf{nil} \vdash x \Rightarrow A \rightsquigarrow \varnothing \mid x}$$

J-Abs
$$\frac{\Gamma, x : A; y? \vdash e \Leftarrow B \rightsquigarrow \mathtt{J} \mid y_0}{\Gamma; z \vdash \lambda x{:}A.\ e{:}B \Rightarrow A \to B \rightsquigarrow \texttt{z["func_|B|"] = (x, y) => \{ J; return y0; \};} \mid z}$$

J-Opt
$$\frac{\Gamma; z \vdash e \Leftrightarrow A \rightsquigarrow \mathtt{J} \mid z}{\Gamma; y? \vdash e \Leftrightarrow A \rightsquigarrow \texttt{var z = y || \{\}; J;} \mid z}$$

J-Merge
$$\frac{\Gamma; z \vdash e_1 \Rightarrow A \rightsquigarrow \mathtt{J}_1 \mid z \quad \Gamma; z \vdash e_2 \Rightarrow B \rightsquigarrow \mathtt{J}_2 \mid z \quad \Gamma \vdash A * B}{\Gamma; z \vdash e_1 \,,\, e_2 \Rightarrow A \,\&\, B \rightsquigarrow \mathtt{J}_1; \mathtt{J}_2 \mid z}$$

J-Int
$$\Gamma; z \vdash n \Rightarrow \mathbb{Z} \rightsquigarrow \texttt{z.int = n;} \mid z$$

(a) Type-directed compilation.

$\boxed{\Gamma; dst \vdash x : A \bullet y : B \rightsquigarrow \mathtt{J} \mid z : C}$ *(Function application)*

JA-ArrowEquiv
$$\frac{A \fallingdotseq C}{\Gamma; z \vdash x : A \to B \bullet y : C \rightsquigarrow \texttt{x["func_|B|"](y, z);} \mid z : B}$$

JA-ArrowOpt
$$\frac{A \fallingdotseq C}{\Gamma; z_0? \vdash x : A \to B \bullet y : C \rightsquigarrow \texttt{var z = x["func_|B|"](y, z0);} \mid z : B}$$

JA-ArrowNil
$$\frac{A \fallingdotseq C}{\Gamma; \mathbf{nil} \vdash x : A \to B \bullet y : C \rightsquigarrow \texttt{var z = x["func_|B|"](y);} \mid z : B}$$

(b) Function application.

Figure 8.9: Selected rules for destination-passing style.

- **nil** stands for no destination, which means that the context is neither a function body nor a merge.

Seven rules for type-directed compilation are selected in Figure 8.9a. A rule ($\Gamma; dst \vdash e \Leftrightarrow A \rightsquigarrow \mathtt{J} \mid z$) basically reads as: given a typing context Γ and a destination dst, the F_{i}^{+} term e is checked/inferred to have type A and is compiled to variable z in JavaScript code $\mathtt{J}$. That is, after running $\mathtt{J}$, the result is stored in the JavaScript variable z. Let us take the variable access in F_{i}^{+} as an example. We perform case analysis on the destination: if the destination is present, we copy the contents of x to z (J-Var); if the destination is absent, we directly return the variable x (J-VarNil); if the destination is optional, we dynamically check the presence of y and copy the contents of x only if y is present (J-VarOpt). Since the destination is set optional for the function body (J-Abs), J-VarOpt is used instead of the other two if the function body is a variable access. This is how we get the optimized code for an identity function in Figure 8.5b. As for the function presented in Figure 8.7, the body is a merge of two literals. There is only one version of J-Merge, which assumes that a destination is provided, so a bridge rule J-Opt is used to properly set the destination. Subsequently, J-Merge delegates the compilation to the two subterms (e.g. J-Int) and concatenates the JavaScript code.

Concerning function application, we select three rules in Figure 8.9b. A rule ($\Gamma; dst \vdash x : A \bullet y : B \rightsquigarrow \mathtt{J} \mid z : C$) basically reads as: given a typing context Γ and a destination dst, applying the compiled function in x of type A to the compiled argument y of type B yields variable z of type C in JavaScript code $\mathtt{J}$. All three rules deal with the simple cases where the parameter type is equivalent to the argument type, so we do not need to insert any coercion for the argument. Again, we perform case analysis on the destination (JA-ArrowEquiv, JA-ArrowNil, and JA-ArrowOpt). This explains why we get different JavaScript code for the two function calls in Figure 8.6.

8.6 Separate Compilation

Lastly, our implementation supports separate compilation. This is usually difficult to achieve in a programming language with a high level of extensibility and modularity that can solve the expression problem (CP's solution is covered in Section 4.3.3). The difficulty of separate compilation in the presence of modularity has been previously studied in the context of *feature-oriented programming* [Apel and Kästner 2009; Prehofer 1997]. As identified by Kästner et al. [2011], modularity can be divided into two categories: *cohesion* and *information hiding*. The cohesive approach often employs source-to-source transformations, which require the whole source code to be available. As a result, they achieve

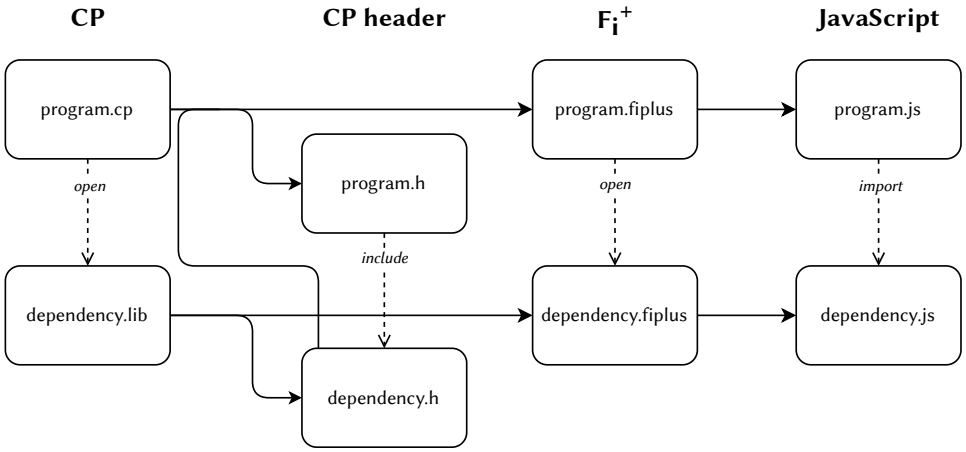


Figure 8.10: A flowchart of separate compilation in CP.

modularity at the cost of modular type checking and separate compilation. Many existing feature-oriented tools, such as AHEAD [Batory et al. 2004], FeatureC++ [Apel et al. 2005], and FeatureHouse [Apel et al. 2013b], fall into this category. The second approach is based on strong interfaces and information hiding. This notion of modularity is underrepresented in feature-oriented software development, but it is emphasized in the community of programming languages and is employed by `gbeta` [Ernst 2000] and CP.

Both `gbeta` and CP can solve the expression problem via family polymorphism [Ernst 2004] without sacrificing separate compilation. However, separate compilation affects the performance of attribute lookup in `gbeta`. Since an object in `gbeta` may have more mixin instances at run time than what is statically known, and the mixin instances may occur in a different order, the offset of an attribute cannot be determined statically. Especially when separate compilation is desired, we cannot do whole program analysis to optimize attribute lookup for some specific inherited classes. As a result, `gbeta` has to perform a linear search through super-mixins to look up inherited attributes. In contrast, attribute lookup in CP, even with dynamic inheritance, is much more efficient, and no linear search is needed.

The flowchart in Figure 8.10 gives an overview of the compilation process from CP all the way down to JavaScript, which is almost a textbook example of separate compilation. Like most programming languages, the compilation unit of CP is a file. One can refer to another file using **`open`** directives in CP, which compile to JavaScript module **`import`** statements. To provide sufficient type information for modular type checking, a CP file compiles to a JavaScript file as well as a CP header file. CP header files are similar to `.mli` files in OCaml and consist of type definitions, type signatures for terms, and references

```
-- example.cp
open strings;
type Rcd = { m: String; n: String };
mkA = trait [this: Rcd] ⇒ {
  m = "foobar";
  n = toUpperCase this.m;
};
```

```
-- example.cp.h
include "strings.lib.h";
type Rcd = { m: String; n: String };
term mkA :
  Trait<{ m: String }&{ n: String }
     ⇒ { m: String }&{ n: String }>;
```

Figure 8.11: A CP file and its corresponding header file.

to other header files. See Figure 8.11 for a slightly simplified example. Normally, header files are automatically generated by the CP compiler, but users can also edit them to hide some definitions that are supposed to be private. The compilation only depends on the file to be compiled and the headers files of its dependencies. As a result, compiling a file does not require recursively compiling its dependencies, and its dependents do not need recompilation as long as its header file is not changed (though its implementations may have changed).

Between CP and JavaScript, there is a core calculus F_{i}^{+} [Bi et al. 2019; Fan et al. 2022]. Since our implementation of CP is based on the elaboration semantics formalized by Zhang et al. [2021], CP language constructs are first desugared into F_{i}^{+} terms, and then these F_{i}^{+} terms are compiled into JavaScript code. Both sets of elaboration rules are syntax-directed and compositional, and the elaboration contexts only include type information from header files in our implementation. That is why CP code can be separately compiled with the help of header files.

In summary, the chapter provides a detailed explanation of how the CP compiler translates a functional language into efficient JavaScript code while tackling key challenges like language paradigm mismatch, performance optimizations, and maintaining modularity with separate compilation. While the implementation successfully meets performance goals, certain areas, such as parametric polymorphism and further optimizations, remain as potential future directions for improvement.

9 Empirical Evaluation

In this chapter, we conduct an empirical evaluation of the CP compiler. We analyze the impact of various optimizations in the CP compiler. Furthermore, we compare the efficiency of dynamic inheritance in CP with that in handwritten JavaScript code. The key takeaway from our empirical evaluation is that using a naive compilation scheme for merges can be orders of magnitude slower than optimized code. Our optimizations lead to code that can be competitive with similar handwritten JavaScript code.

Experimental setup and benchmark programs. We performed experiments on a system featuring an Apple M1 Pro chip and 16GB RAM. JavaScript code was executed using Node.js 20.12.2 LTS. The outline of benchmark programs is presented in Table 9.1. The initial four benchmarks focus on general-purpose computations, while the latter four are adapted from examples in Chapter 5, showcasing CP's novel features. Among them, chart is the biggest program with around 300 lines of code. Challenges discussed in Section 4.3, including dynamic inheritance and family polymorphism, are prominent in the latter four benchmarks.

9.1 Ablation Study on Optimizations

The coercive subtyping semantics of CP raises important questions about efficiency since coercions have runtime costs and they are pervasively employed in generated code. There are essentially three main concerns that need to be addressed in obtaining an efficient compilation scheme for CP:

- **Efficient lookup.** Since merge lookup is pervasive, it is important to use a runtime representation for merges that enables efficient lookup.
- **Efficient merging and copying.** Since merging is frequent, it is important that the merging process is efficient and minimizes the amount of copying involved in merging.

- **Minimizing the cost of coercions.** Since our subtyping is coercive, it is fundamental that the cost of coercions is minimized. Furthermore, optimizations should avoid coercions when possible.

In our work, we have addressed the above three points. Our representation of merges as type-indexed records makes the cost of a merge lookup essentially the same as the cost of a JavaScript field lookup, which is very efficient. This provides a major source of improvement over a representation with pairs, where lookup time can be linear. We believe that it is hard to do better in this dimension, at least if the goal is to target JavaScript. For merging, we rely on JavaScript's ability to copy object fields. An important concern for merging is to avoid the creation of intermediate objects, minimizing the amount of copying. The DPS optimization is particularly important for obtaining efficient merging. Like lookup, we believe that the CP compiler also achieves efficient merging. Finally, to mitigate coercions, we employ a hybrid model that combines inclusive and coercive subtyping. We only insert coercions when necessary and try to eliminate redundant coercions as much as possible. We have mentioned several optimizations in Chapter 8, two of which are avoiding unnecessary coercions: one for equivalent types and the other for record projections.

All the implemented optimizations should improve the performance of our CP compiler in theory. Here we select four representative ones to evaluate their impact in practice:

1. Reducing intermediate objects using destination-passing style (DPS);
2. Preventing primitive values from boxing/unboxing (NoBox);
3. Eliminating coercions for subtyping between equivalent types (TyEquiv & CoElim);
4. Avoiding the insertion of coercions for record projections (ProjOptim).

We conduct an *ablation study* on the four optimizations. Figure 9.1b shows the execution time ratios (slowdowns) to the optimized JavaScript code when removing each optimization. Figure 9.1a lists the original data for Figure 9.1b in milliseconds. CP Compiler represents the most optimized version of the CP compiler, including all the aforementioned optimizations. The remaining variants are CP Compiler minus one optimization, including all other optimizations. To summarize the benchmark results, different optimizations show different degrees of speedup for different benchmarks, but we believe that the coercion-related ones are especially important when (dynamic) inheritance is concerned.

The first optimization (DPS) speeds up the execution of the latter five benchmarks by reducing the number of intermediate objects and object concatenations. In contrast, the former three benchmarks do not benefit from the optimization because they only perform

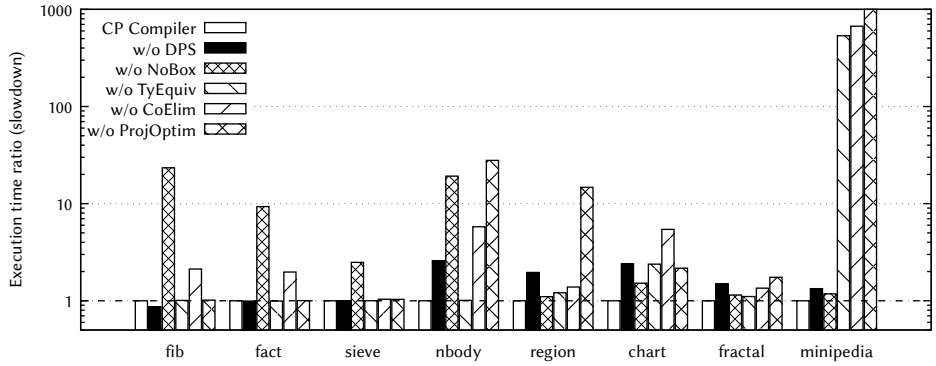
Table 9.1: Outline of the benchmark programs.[†]

fib	Calculating Fibonacci numbers without memoization.
fact	Some factorial functions multiplied together.
sieve	Sieve of Eratosthenes, an algorithm for finding prime numbers.
nbody	Numerical simulation of the *n*-body problem.
region	An embedded DSL for geometric regions.
chart	Generating SVG code for customizable charts.
fractal	Generating SVG code for a simple fractal called the Sierpiński carpet.
minipedia	Generating HTML code for a mini document with a computed table of contents.

[†] Some computations in the benchmark programs are repeated several times for longer and more stable execution time.

	fib	fact	sieve	nbody	region	chart	fractal	minipedia
CP Compiler	2837	1433	1736	1704	1944	516	4578	45
w/o DPS	2451	1422	1728	4402	3806	1243	6861	60
w/o NoBox	66348	13369	4314	32716	2144	783	5249	53
w/o TyEquiv	2860	1425	1738	1722	2349	1229	5064	24080
w/o CoElim	6020	2832	1801	9851	2693	2803	6173	30192
w/o ProjOptim	2880	1438	1795	47505	28591	1117	7990	OOM[‡]

(a) Execution time (ms) of the JavaScript code generated by variants of the CP compiler.



(b) Execution time ratios (slowdowns) of different variants to the optimized JavaScript code.[‡]

[‡] The bar that exceeds the frame represents *JavaScript heap out of memory* (OOM) for minipedia w/o ProjOptim.

Figure 9.1: Ablation study on optimizations for the CP compiler.

arithmetic operations and no objects are involved. The first benchmark (fib) even becomes a bit slower because the optimization inserts extra checks into function bodies to test if destination objects are present. Overall, the speedup ratios are 2.6× at most. This optimization clearly helps for programs involving objects and merging, although the benefits of this optimization are smaller than optimizations on coercions. In essence, intermediate objects are not the main bottleneck in the JavaScript code generated by the CP compiler, although they still have a considerable cost for many programs.

The second optimization (NoBox) is important for primitive operations such as arithmetic, which complements the first optimization. It speeds up all benchmarks since primitive operations are inevitable in practical programs. It brings around 23× speedup for fib and around 19× speedup for nbody because they involve a lot of arithmetic operations. Numbers do not need to be boxed/unboxed in the optimized JavaScript code, so the performance is improved significantly.

The analysis for the third optimization is split into two parts for a finer-grained analysis. We have a version of the CP compiler that only removes coercions for *syntactically equal* types but does not eliminate other coercions for *equivalent* types (w/o TyEquiv). The other version does not eliminate redundant coercions at all (w/o CoElim). Some benchmarks (such as chart and minipedia) make use of equivalent types a lot, hence their performance is already affected by removing TyEquiv. After further removing CoElim, most benchmarks experience significant slowdowns (up to 671× slower in the worst case for minipedia).

The last optimization (ProjOptim) targets coercions for record projections, so the benchmarks that do not use records (such as fib, fact, and sieve) are not affected at all. Among the relevant benchmarks, nbody becomes around 28× slower without this optimization. This is because the masses, velocities, and coordinates of the bodies are all stored in records. Note that the JavaScript code generated for minipedia runs out of memory, so there is no data in Figure 9.1a, and the exception is represented by a bar that exceeds the frame in Figure 9.1b.

In conclusion, all optimizations work in practice. The elimination of redundant coercions has a particularly significant impact on the performance. The representation of JavaScript objects (or extensible records in general) brings forth a class of equivalent types, whose terms share the same shape. We can then avoid the coercions between these types but still obtain an equivalent object as a result. This optimization has a significant impact but cannot be done in previous work [Dunfield 2014; Oliveira et al. 2016] because they use nested pairs as the elaboration target of merges. Since pairs are order-sensitive, they require coercions that can be avoided with order-insensitive objects (see also the discussion

in Section 6.1). Together with the faster lookup by type indices in objects, the JavaScript code generated by the CP compiler achieves reasonable performance.

9.2 Comparison with Handwritten JavaScript Code

Our focus in this thesis is on the type-safe compilation of dynamic inheritance and the efficient compilation of languages with merges. A first natural question to ask is how the new compilation scheme compares against existing compilation schemes for merges. Unfortunately, such a direct comparison is not feasible for a few different reasons. Firstly, the only other compiler for a language with merges is Stardust by Dunfield [2014]. However, Stardust targets ML, instead of JavaScript. Thus, a direct comparison of performance would not be possible. Furthermore, Stardust does not support distributive subtyping and nested composition. Thus, most of our examples and case studies cannot be encoded in Stardust. Nevertheless, in Section 6.1, we have highlighted some advantages of using our record-based representation versus using pairs (which Stardust employs) in the compilation of merges.

In spite of the above-mentioned difficulties of a direct comparison, it is still helpful to do an elementary quantitative analysis with handwritten JavaScript code to assess the impact of the coercive semantics of CP. Although we have worked hard to eliminate unnecessary coercions, the JavaScript code generated by the CP compiler still includes plenty of coercions. In contrast, handwritten JavaScript code is coercion-free, and subtyping in TypeScript has no cost. It would be unrealistic to expect a stable performance that is competitive with JavaScript, especially since our implementation is still a proof of concept for our compilation scheme. However, ideally, the performance penalty imposed by coercions should not be too high.

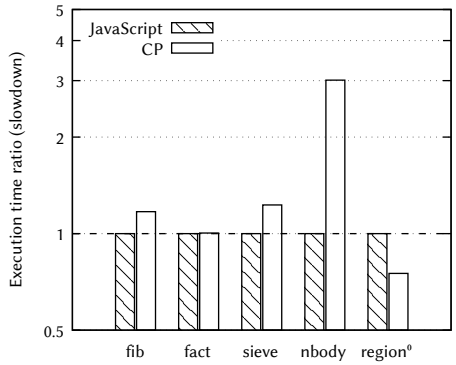
A brief comparison is made based on the former four benchmarks, namely fib, fact, sieve, and nbody (we will explain region^0 later). Figure 9.2c shows the execution time ratios (slowdowns) of the JavaScript code generated by the CP compiler compared to the handwritten JavaScript code, and Figure 9.2a lists the original data. They mainly demonstrate general-purpose computations. The handwritten JavaScript code is transliterated from the corresponding CP code in order to make an apples-to-apples comparison. It follows a functional programming style similar to CP and may not be idiomatic in JavaScript. The performance of the JavaScript code generated by the CP compiler is slightly slower than that of the handwritten code for fib, fact, and sieve. The biggest slowdown is around 3× for nbody, partly because the manipulations of records and arrays in CP are less efficient than in native JavaScript. Moreover, our treatment of **`let`** expressions is oversimplified. In

	fib	fact	sieve	nbody	region^0
JavaScript	2423	1427	1413	566	1513
CP	2837	1433	1736	1704	1137

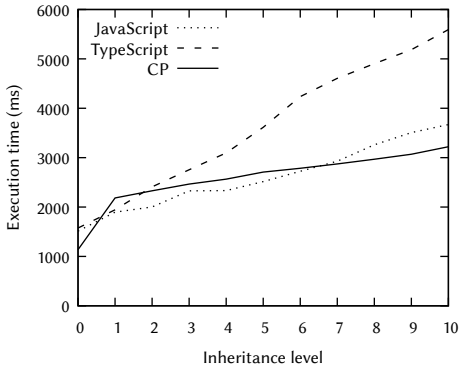
(a) Execution time (ms) for five benchmarks.

Inheritance level	0	1	2	3	4	5	6	7	8	9	10
JavaScript	1513	1896	2002	2328	2333	2515	2724	2928	3260	3507	3670
TypeScript	1575	1944	2409	2755	3096	3614	4236	4606	4903	5186	5593
CP	1137	2184	2329	2465	2565	2708	2785	2873	2968	3069	3221

(b) Execution time (ms) for region$^{0..10}$.



(c) Bar chart for five benchmarks.



(d) Line chart for region$^{0..10}$.

Figure 9.2: Comparison between JavaScript code generated by the CP compiler and hand-written code.

CP, `let x = e1 in e2` is desugared into `(\x → e2) e1`, which is much slower than `const` statements in JavaScript. In nbody, there are several nested `let`s in recursive functions, introducing significant overhead.

The latter four benchmark programs make use of CP's novel features, making transliteration to JavaScript difficult. Nevertheless, we adapt the fifth benchmark (region) to make a comparison between conceptually equivalent programs. To recap, the benchmark program is mainly an embedded DSL for geometric regions [Hudak 1998]. For modular extension, the DSL is implemented with techniques of family polymorphism, which are described in Section 4.2.4 for JavaScript and in Section 4.3.3 for CP. Both implementations heavily rely on class/trait inheritance, so the performance penalty of inheritance is well demonstrated in this benchmark. Furthermore, we change the number of inheritance levels from 0 to 10 (region^n represents that the desired method is in the n-level super-trait/-class) to see the trend of the performance penalty. In other words, region^0 is *monolithic* code with a single trait/class and no inheritance hierarchy. At level one, we introduce a slightly more

modular version of region with one level of inheritance: there is a super-trait/-class and a sub-trait/-class. Higher levels simply introduce more inheritance layers. The results are shown in Figure 9.2b and Figure 9.2d.

Besides CP and JavaScript, a TypeScript version is also included for this comparison. The source code is simply the JavaScript version plus type annotations. We use the official TypeScript compiler to compile it to JavaScript and then use Node.js to execute the JavaScript code. The TypeScript code has a different performance profile from the JavaScript code because the TypeScript compiler by default (as of the current version 5.4) desugars classes into prototypes. This is due to the default compilation target being ECMAScript 3 [ECMA 1999] for best compatibility, which does not support classes. Newer versions of Node.js (based on ECMAScript 6 [ECMA 2015] or above) natively support classes, so the handwritten JavaScript directly uses classes. To sum up, the difference between JavaScript and TypeScript in the benchmark is mainly classes versus prototypes.

Without inheritance (region^0), the JavaScript code generated by the CP compiler is *faster* than the handwritten JavaScript and TypeScript code. This is because the technique of nested anonymous classes is neither idiomatic nor efficient in JavaScript. In contrast, nested traits themselves do not introduce extra runtime overhead in CP. However, when the desired method is one level up in the inheritance hierarchy, the CP compiler generates around 2× slower code, compared to the monolithic version, because coercions are inserted for nested trait composition. For the monolithic version, there are almost no coercions in the CP code. Then the performance penalty increases more smoothly when the inheritance level is higher. The CP compiler regains its leading position when the number of inheritance levels is higher than 6. In contrast, TypeScript has the steepest curve. The desugared prototype-based code generated by the TypeScript compiler is the least efficient among the three implementations.

In conclusion, the performance penalty of coercions brought by trait inheritance is not negligible but increases more smoothly with the number of inheritance levels than in handwritten JavaScript. This is partly due to the efficient lookup by type indices in extensible records (or, more specifically, JavaScript objects). Looking up a deeply nested method in the inheritance hierarchy can be slow in JavaScript, but this is not the case in CP.

Part V

Compositional Programming with Union Types

10 Named Arguments as Intersections, Optional Arguments as Unions

Named and optional arguments are prevalent features in many mainstream programming languages, enhancing code readability and flexibility. Despite widespread use, their formalization has not been extensively studied.

This part extends compositional programming with union types, enabling a type-safe foundation for named and optional arguments. We first conduct a survey of existing languages' support for named arguments in Section 10.1. Then we identify in Section 10.2 a critical type-safety issue in popular static type checkers for Python and Ruby, particularly in handling first-class named arguments in the presence of subtyping. Our solution is informally presented in Section 10.3 and formalized in Section 10.4 through an elaboration from a functional language with named and optional arguments (Uaena) to a minimal core calculus with intersection and union types (λ_{iu}). We conclude this chapter with a detailed discussion of existing designs in Section 10.5.

10.1 Introduction

The λ-calculus, introduced by Church [1941], shows how to model computation solely with function abstraction and application. For example, natural numbers, boolean values, pairs, and lists, as well as various operations on them, can be represented by higher-order functions via Church encoding. In the λ-calculus, a function only has one parameter and can only be applied to one argument. Many programming languages in the ML family inherit this feature. If more than one argument is desired in those languages, we need to create a sequence of functions, each with a single argument, and perform an iteration of applications. This idea is called *currying*. Currying brings brevity to functional programming and naturally allows partial application, but it usually limits the flexibility of function application. For example, we cannot pass arguments in a different order nor omit some of them by providing default values. Both demands are not rare in practical programming and

```python
def exp(x, base=math.e):
  return base ** x

exp(10, 2)  #= exp(x=10, base=2) = 1024
exp(base=2, x=10)               #= 1024
exp(x=10)                       #= e^10

args = { "base": 2, "x": 10 }
exp(**args) #= exp(base=2, x=10) = 1024
```

(a) The Python way.

```ruby
def exp(x:, base: Math::E)
  base ** x
end
exp(10, 2) # ArgumentError!
exp(base: 2, x: 10) #= 1024
exp(x: 10)          #= e^10

args = { base: 2, x: 10 }
exp(**args)         #= 1024
```

(b) The Ruby way.

Figure 10.1: Named arguments in Python and Ruby.

can be met in a language that supports *named* and *optional* arguments. Named arguments also largely improve the readability of function calls. For example, it is unclear which is the source and which is the destination in `copy(x, y)`, while `copy(to: x, from: y)` is self-explanatory.

Named arguments are widely supported in mainstream programming languages, such as Python, Ruby, OCaml, C#, and Scala, just to name a few. The earliest instance, to the best of our knowledge, is Smalltalk, where every method argument *must* be associated with a *keyword* (i.e. an external name). In other words, there are no positional arguments (i.e. arguments with no keywords) in Smalltalk. The syntax of modern languages is usually less rigid, so programmers can choose whether to attach keywords to arguments or not. There are two ways to reconcile positional and named arguments. One way, employed by Python and shown in Figure 10.1a, is to make parameter names in a function definition as non-mandatory keywords. Thus, every argument can be passed with or without keywords by default. As shown in the Python code, `exp(10, 2)` is equivalent to `exp(x=10, base=2)`. To reconcile the two forms in the same call, a restriction is imposed that all named arguments must follow positional ones. The other way, shown in Figure 10.1b and used in Ruby, is to strictly distinguish named arguments from positional ones. When defining a Ruby function, a keyword parameter should always end with a colon even if it does not have a default value. By this means, they are syntactically distinct from positional parameters, and their keywords cannot be omitted in a function call. There is also a restriction in Ruby that all named arguments must follow positional ones in both function definitions and call sites. The two kinds of arguments are usually used in different scenarios: positional arguments are used when the number of arguments is small and the order is clear, while named arguments are used in more complex cases especially when settings or configurations are involved.

Table 10.1: Named arguments with different design choices in different languages.

	Smalltalk	Python	Ruby	Racket	OCaml	C#	Scala	Dart	Swift	CP
Commutativity	○	●	●	●	●	●	●	●	○	●
Optionality	○	●	●	●	●	●	●	●	●	●
Currying	○	○	○	○	●	○	○	○	○	○
Distinctness	*n/a*	○	●	●	●	○	○	●	●	●
First-class value	○	●	●	◐	○	○	○	○	○	●
Static typing	○	○	○	○	●	●	●	●	●	●
Soundness proof	○	○	○	○	●	○	○	○	○	●

n/a: Smalltalk does not support positional arguments at all.
◐: Racket's support for first-class named arguments is limited and forbids commutativity.

More interestingly, named arguments are *first-class* values in Python and Ruby: they can be assigned to a variable. As shown at the bottom of Figure 10.1, the variable `args` stores the two arguments named `base` and `x`, and we can later pass it to `exp` by unpacking it with `**` (sometimes called the splat operator). In fact, `args` is a dictionary in Python and similarly a hash in Ruby. Thus, first-class named arguments can be manipulated and passed around like standard data structures. This feature is widely used in Python and Ruby.

Including the distinctness and first-class values illustrated above, we have identified five important design choices found in existing languages that support named arguments:

1. *Commutativity*: whether the order of (actual) arguments can be different from that of (formal) parameters originally declared.

2. *Optionality*: whether some arguments can be omitted in a function call if their default values are predefined.

3. *Currying*: whether a function that takes more than one argument is always converted into a chain of functions that each take a single argument.

4. *Distinctness*: whether named arguments are distinct from positional ones in how they are defined and passed.

5. *First-class value*: whether named arguments are first-class values.

As shown in Table 10.1, the first two properties hold for most mainstream programming languages, with Smalltalk and Swift being two exceptions. Commutativity and optionality are so useful that we believe they should not be compromised. Concerning the third point, OCaml is the only language that manages to reconcile currying with commutativity,

though at the cost of introducing a very complicated core calculus. We agree that currying is very useful when we use normal positional arguments, but we argue here that currying can be temporarily dropped when we use named arguments because the most common use case for named arguments is to represent a whole chunk of parameters like settings or configurations. The fourth design, distinctness, is endorsed by Ruby, Racket, OCaml, Dart, and Swift. It improves the readability of call sites to enforce keywords whenever arguments are defined to be named. We advocate distinctness in this work also because it simplifies the language design and allows us to focus on more important topics, especially type safety with first-class named arguments.

Although named arguments are ubiquitous, they have not attracted enough attention in the research of programming languages. Among the few related papers, the work by Garrigue [1994] formalizes a label-selective λ-calculus and eventually apply it to OCaml [Garrigue 2001]. Another work by Rytz and Odersky [2010] discusses the design of named and optional arguments in Scala, but it mainly focuses on practical aspects. The core features of Scala are formalized in a family of DOT calculi, but named arguments are never included. The support for named arguments is implemented as macros in Racket [Flatt and Barzilay 2009]. So their extension is more like userland syntactic sugar and requires no changes to the core compiler. Haskell does not support named arguments natively, but the paradigm of *named arguments as records* is folklore. We will discuss OCaml, Scala, Racket, and Haskell in detail in Section 10.5. In short, named arguments are implemented in an ad-hoc manner and are not well founded from a type-theoretic perspective in most languages, especially object-oriented ones. Only OCaml and CP provide *soundness proofs* for the feature of named arguments.

An important issue that has not been explored in the literature is the interaction between subtyping and first-class named arguments. A naive design can easily lead to a type-safety issue. We will show in Section 10.2.2 that the most widely used optional type checker for Python, mypy [Lehtosalo et al. 2012], fails to detect a type-unsafe use of first-class named arguments. The same issue also exists in Ruby with Steep [Matsumoto et al. 2018] or Sorbet [Stripe 2019]. It arises from subtyping hiding some arguments from their static type and bypassing the type checking for optional arguments. As a result, an optional argument may have an unexpected type at run time, which leads to a runtime error.

In this chapter, we present a type-safe foundation for named and optional arguments. At the heart of our approach is the translation into a core calculus called λ_{iu}, which features *intersection and union types* [Barbanera et al. 1995; Dunfield 2014; Frisch et al. 2008]. Our approach supports first-class named arguments like Python and Ruby, but the type-safety issue is addressed by us. The λ_{iu} calculus has been shown to be type-sound [Rehman 2023],

and we show that our translation from our source language into λ_{iu} is type-safe. Thus, we establish the type safety of our approach.

10.2 Named and Optional Arguments: The Bad Parts

Since named and optional arguments are not well studied in most languages, the ad-hoc mechanisms employed in those languages may sometimes surprise programmers or even cause safety issues.

10.2.1 Gotcha! Mutable Default Arguments in Python

Let us consider a simple Python function that appends an element to a list. We provide a default value for the list, which is an empty list:

```
def append(x, xs=[]):
  xs.append(x)
  return xs
```

```
append(1) #= [1]
append(2) #= [1, 2]
```

After calling `append(1)` above, we get the expected result `[1]`. However, continuing to call `append(2)` gives us `[1, 2]` instead of `[2]`. This is because Python only evaluates the default value once when the function is defined, so the same list initialized for `xs` is shared across different calls to `append`. When calling `append(2)`, the default value for `xs` is no longer an empty list but the list that has been modified by the previous call `append(1)`.

This issue, while seemingly minor, highlights the importance of understanding the semantics of default arguments. Our design strives to avoid such surprises, following the principle of least astonishment, yet this is not our main focus. We will discuss the more critical issue about type safety next.

10.2.2 Caution! Type Safety with First-Class Named Arguments

As we have shown in Figure 10.1, quite a few languages, especially dynamically typed ones like Python and Ruby, treat named arguments as first-class values. This feature is particularly helpful for passing settings because they are usually stored in a separate configuration file. We can read the settings from the file and pass them as named arguments using the `**` operator. For example, we can find such code in Python to run a web server:

```
class App: # from a web server library
  def run(self, host: str, port: int, debug: bool = False):
    assert isinstance(debug, bool) # actual code omitted...
```

```
args = { "host": "0.0.0.0", "port": 80, "debug": True }
app.run(**args) #= app.run(host="0.0.0.0", port=80, debug=True)
```

Although Python is dynamically typed, there is continuous effort in the Python community to improve the detection of type errors earlier in the development process, primarily through static analysis. There is an optional static type checker for Python called mypy [Lehtosalo et al. 2012].[1] In the example above, we make use of *type hints*, introduced in Python 3.5, to specify the types of the parameters and the return value of the `run` method. The type hints have no effect at run time but can be used by external tools like mypy to statically check if the code is well-typed. Perhaps surprisingly, the code above cannot pass mypy's type checking, because the type inferred for `args` (i.e. `dict[str,object]`) is not precise enough. The type checker needs to know what keys `args` exactly has and what types the values associated with those keys have, in order to make sure that `**args` is compatible with the parameters of `app.run`.

Fortunately, `TypedDict` is added in Python 3.8 to represent a specific set of keys and their associated types. By default, every specified key is required, except when it is marked as `NotRequired`, which is a type qualifier added later in Python 3.11. With `TypedDict` and `NotRequired`, we can now define a precise dictionary type for `args` that passes mypy's type checking:

```
class Args(TypedDict):
  host: str
  port: int
  debug: NotRequired[bool]

args0: Args = { "host": "0.0.0.0", "port": 80, "debug": True }
app.run(**args0) # type-checks in mypy

args1: Args = { "host": "0.0.0.0", "port": 80 }
app.run(**args1) # type-checks in mypy, too
```

The mypy type checker will raise an error if we provide an argument with an incompatible type, such as a string for the `debug` key:

```
class In(TypedDict):
  host: str
  port: int
  debug: str
```

[1] Our code is tested against mypy 1.14.0 (released on 20 December 2024).

```
args2: In = { "host": "0.0.0.0", "port": 80, "debug": "Oops!" }
app.run(**args2) # TypeError: Argument "debug"
# has incompatible type "str"; expected "bool"  [arg-type]
```

However, mypy's type system is not completely type-safe. We can create a function `f` that takes a dictionary with three keys specified in type `In` and returns a dictionary with only two keys specified in type `Out`. The function type-checks in mypy because type `In` is compatible whenever type `Out` is expected. Roughly speaking, it means that `In` is a subtype of `Out`. Then we can use `f` to forget the `debug` key in the static type:

```
class Out(TypedDict):
  host: str
  port: int

def f(args: In) → Out: return args

args3 = f(args2) # still contains { "debug": "Oops!" }
app.run(**args3) # type-checks in mypy, but has a runtime error!
```

Here `args3` has type `Out` without the `debug` key specified. From a static viewpoint, `args3` only has two keys `host` and `port`, which are compatible with the parameters of `app.run` since `debug` is optional and has a default value. That is why `app.run(**args3)` type-checks in mypy. However, at run time, the `debug` key is still present in `args3`, so the string `"Oops!"` is passed as a named argument to `app.run`, which originally expects a boolean value. This results in a runtime error since there is an assertion in `app.run` to ensure that `debug` is boolean.

This issue is not unique to Python and mypy. We have reproduced nearly the same issue in Ruby with two popular type checkers, namely Steep [Matsumoto et al. 2018] and Sorbet [Stripe 2019], which is illustrated in Appendix D and Appendix E.[2]

In conclusion, subtyping can lead to a fundamental type-safety issue when dealing with first-class named and optional arguments. In essence, the following subsumption chain is questionable:

```
   { host: str, port: int, debug: str }
<: { host: str, port: int }
<: { host: str, port: int, debug?: bool }
```

Following this chain bypasses mypy's type compatibility checking for the `debug` key. Next we will show how to break the chain and address the type-safety issue.

[2] Our code is tested against Steep 1.9.2 (16 December 2024) and Sorbet 0.5.11708 (20 December 2024).

10.3 Our Type-Safe Approach

In this section, we informally present how we translate named and optional arguments into a core language with intersection and union types, while retaining type safety. We start by introducing the core language constructs that we need. Then we illustrate our translation scheme by example and demonstrate how it recovers type safety. After that, we showcase a practical example in the CP language, which has incorporated our approach to support named and optional arguments. Finally, we discuss how our translation scheme can be applied to other languages.

10.3.1 Core Language

The core language features intersection and union types, which establish an elegant duality in the type system. A value of the intersection type $A \wedge B$ can be assigned both A and B, whereas a value of the union type $A \vee B$ can be assigned either A or B. Intersection and union types correspond to the logical conjunction and disjunction respectively.[3] Similar calculi are widely studied [Barbanera et al. 1995; Dunfield 2014; Frisch et al. 2008] and provide a well-understood foundation for named and optional arguments.

Named arguments as intersections. Named arguments are translated to multi-field records. However, the core language does not support multi-field records directly. There are only single-field records in the core language, and multiple fields are represented as intersections of single-field record types. For example, $\{x : \mathbb{Z}\} \wedge \{y : \mathbb{Z}\}$ represents a record type with two integer fields x and y. With intersection types, width subtyping for record types comes for free, and permutations of record fields are naturally allowed [Reynolds 1997].

At the term level, a merge operator [Dunfield 2014; Rehman 2023] is used to concatenate multiple single-field records to form multi-field records, reminiscent of Forsythe [Reynolds 1997]. For example, $\{x = 1\},\{y = 2\}$ forms a two-field record from two single-field records.

Optional arguments as unions. Optional arguments are translated to nullable types. A nullable type is not implicit in the core language but is represented as a union with the null type [Nieto et al. 2020]. For example, an optional integer argument named z is translated to $\{z : \mathbb{Z} \vee \mathbf{Null}\}$.

[3] In this chapter, we use the notation $A \wedge B$ for intersections and $A \vee B$ for unions, to better align with the literature, rather than the notation $A \,\&\, B$ and $A \mid B$ in CP.

At the term level, a type-based switch expression [Frisch et al. 2008; Rehman 2023] is used to scrutinize a term of a union type, reminiscent of ALGOL 68 [van Wijngaarden et al. 1975]. For example, **switch** z **case** $\mathbb{Z} \Rightarrow e_1$ **case Null** $\Rightarrow e_2$ returns e_1 if z is an integer or e_2 if **null**.

10.3.2 Translation by Example

Let us review the previous Python function in Section 10.2.1 that appends an element to a list, which defaults to an empty list:

```
def append(x: int, xs: list[int] = []): ...
```

The function will be translated to a core function as follows:

$$
\begin{aligned}
\mathsf{append} = \lambda \mathit{args}\colon \{x : \mathbb{Z}\} \wedge \{\mathit{xs} : [\mathbb{Z}] \vee \mathbf{Null}\}.\ & \\
& \mathbf{let}\ x = \mathit{args}.x\ \mathbf{in} \\
& \mathbf{let}\ \mathit{xs} = \mathbf{switch}\ \mathit{args}.\mathit{xs}\ \mathbf{as}\ \mathit{xs}\ \mathbf{case}\ \mathbb{Z} \Rightarrow \mathit{xs}\ \mathbf{case}\ \mathbf{Null} \Rightarrow [\,]\ \mathbf{in} \\
& \ldots
\end{aligned}
$$

Here we can see that the default value (i.e. the empty list) is not shared across different calls to `append` because the default value is evaluated within the function body. Therefore, calling `append(x=1)` will consistently return `[1]` instead of surprisingly modifying the default value. This design leads to less astonishment and more predictable behavior.

Since we translate named parameter types to record types, we correspondingly translate named arguments to records. For example, the function call `append(x=1, xs=[0])` will be translated to $\mathsf{append}\,(\{x = 1\}, \{\mathit{xs} = [0]\})$.

Rewriting call sites. More importantly, we also rewrite call sites to add null values for absent optional arguments. For example, the function call `append(x=1)` will be rewritten and translated to $\mathsf{append}\,(\{x = 1\}, \{y = \mathbf{null}\})$.

Dependent default values. Another advantage of our translation scheme is that it naturally allows default values to depend on earlier arguments. Python and Ruby do not support dependent default values, but this feature can be useful in some practical scenarios. For example, when setting up I/O, we may want to output error messages to the same stream as `out` by default:

```
def setIO(in_, out, err = out): ...
```

The variable `out` can be used in the default value of `err` because it has been brought into scope by the previous let-in binding:

$$\textbf{let } \mathit{out} = \mathit{args.out} \textbf{ in}$$
$$\textbf{let } \mathit{err} = \textbf{switch } \mathit{args.err} \textbf{ as } \mathit{err} \textbf{ case IO} \Rightarrow \mathit{err} \textbf{ case Null} \Rightarrow \mathit{out} \textbf{ in}$$

10.3.3 Recovering Type Safety

The type safety of our translation scheme is essentially guaranteed by call site rewriting. Besides adding null values for absent optional arguments, we also sanitize arguments to ensure that they are expected from the parameter list. Since named arguments are first-class and can be passed as a variable, we may not have literals like `append(x=1, xs=[0])` but splats like `append(**args)`. So the matching between (formal) parameters and (actual) arguments is performed based on their static types:

- If `args` has type $\{x : \mathbb{Z}\} \wedge \{xs : [\mathbb{Z}]\}$, the call site will be rewritten to something equivalent to `append(x=args.x, xs=args.xs)`.
- If `args` only has type $\{x : \mathbb{Z}\}$, the call site will be rewritten to something equivalent to `append(x=args.x, xs=null)`.

For the `append` function, no other cases can pass the sanitization process.

Let us review the previous type-unsafe Python example in Section 10.2.2:

```
def f(args: In) → Out: return args
args = f({ "host": "0.0.0.0", "port": 80, "debug": "Oops!" })
app.run(**args) #= app.run(host="0.0.0.0", port=80, debug="Oops!")
```

Recall that `args` has type `Out`, which is similar to `{ host: str, port: int }`. The `debug` key is forgotten in the static type but is still present at run time. It passes mypy's type checking but raises a runtime error. In our translation scheme, the call site will be rewritten to the following form based on the type of `args` (i.e. `Out`):

```
app.run(host=args.host, port=args.port, debug=null)
```

Therefore, type safety is recovered in our translation scheme.

Takeaways. There are two important observations from our translation scheme:

1. `{ required: A, optional?: B }` is not equivalent to `{ required: A }` because the former contains more information that prevents `optional` from being associated with other types than `B`. In other words, the `optional` argument can be absent, but if it is present, it must have type `B`.

2. Corresponding to the above observation at the type level, we explicitly pass a null value as an optional argument if it is statically missing. The null value fills the position of a potentially forgotten argument that may have a wrong type. In other words, we implement the splat operator as per the static type of named arguments.

10.3.4 Implementation in the CP Language

Our approach to named and optional arguments has been implemented in the CP language. CP supports not only intersection and union types but also the merge operator and type-based switch expression. The implementation of named and optional arguments in CP is a direct application of our translation scheme.

More interestingly, the sanitization process during call site rewriting comes for free because CP employs a *coercive* semantics for subtyping [Luo et al. 2013]. For example, a subtyping relation between `{ host: str, port: int, debug: str }` and `{ host: str, port: int }` implies a coercion function from subtype to supertype. In CP, such coercions are implicitly inserted to remove the forgotten fields (e.g. `debug` in this case). Therefore, the only remaining work is to add null values for absent optional arguments.

To demonstrate the use of named and optional arguments in CP, we show a fractal example in Figure 10.2, which is adapted from code in Section 5.5.2. The code makes use of named and optional arguments a lot, including both the `SVG`/`Rect` constructors from the library and the `fractal` function defined by the client. For example, `fractal` has five named arguments (`level`, `x`, `y`, `width`, and `height`), among which `level` is optional with a default value of `4`.

It is worth noting that named arguments are used as first-class values in the CP code. On the first line of the `fractal` body, we store three fields `level`, `width`, and `height` in a variable `args`, which are shared arguments for later calls. When constructing the center rectangle, we merge `args` with three more fields `x`, `y`, and `color` to form a full set of named arguments we need for the `Rect` constructor. When recursively calling `fractal`, we pass `args` merged with different `x` and `y` values to draw the eight sub-copies. In the `main` function, we also use a variable `init` to avoid repeating the same set of arguments for `SVG`, `Rect`, and `fractal` calls. The `**` operator is not needed in CP when passing first-class named arguments. Note that the parameter lists of these three constructors/functions are not completely the same, but we can still use a larger set of named arguments to cover all the cases. This is possible because CP allows subtyping for named arguments while retaining type safety.

```
-- from a SVG library in CP
SVG: { width: Int; height: Int } → [Element] → Graphic;     -- <svg>
Rect: { x: Int; y: Int; width: Int; height: Int
      ; rx?: Int; ry?: Int; color?: String } → Element;     -- <rect>
......
-- client code
fractal { level = 4; x: Int; y: Int; width: Int; height: Int } =
  let args = { level = level-1; width = width/3; height = height/3 } in
  let center = Rect (args,{ x = x + width/3; y = y + height/3
                          ; color = "white" }) in
  if level == 0 then [center]
  else fractal (args,{ x = x;             y = y })
    ++ fractal (args,{ x = x + width/3;   y = y })
    ++ fractal (args,{ x = x + width*2/3; y = y })
    ++ fractal (args,{ x = x;             y = y + height/3 })
    ++ [center]
    ++ fractal (args,{ x = x + width*2/3; y = y + height/3 })
    ++ fractal (args,{ x = x;             y = y + height*2/3 })
    ++ fractal (args,{ x = x + width/3;   y = y + height*2/3 })
    ++ fractal (args,{ x = x + width*2/3; y = y + height*2/3 });
init = { x = 0; y = 0; width = 600; height = 600; color = "black" };
main = SVG init ([Rect init] ++ fractal init);
```

Figure 10.2: Sierpiński carpets implemented in the CP language.

10.3.5 Applications to Other Languages

Although we base our translation scheme on a core language with intersection and union types for type-theoretic solidness and elegance, it can work for a wider range of languages. We discuss the alternatives to intersections and unions below.

Alternative to intersections. Record types have existed long before intersection types were invented. In practice, multi-field records are rarely represented as intersections of single-field records. For example, *Software Foundations* [Pierce et al. 2008] demonstrates how to directly model multi-field records and define depth, width, and permutation subtyping without intersections, though their formalization is more complex than ours.

There is a merge operator in our translation scheme, but we only use it to construct multi-field records statically. Although the merge operator can be powerful if we want to construct first-class named arguments at run time like in CP, its absence does not disable our translation scheme. In other words, we only assume a simplified version that does not merge terms dynamically.

Alternative to unions. Nullable types are rarely represented as unions with the null type too. For example, C#, Kotlin, and Dart support nullable types as a primitive data structure. Putting a question mark behind any type makes it nullable in these languages (e.g. `int?`).

No matter how a nullable type is represented, there is usually some expression that can check whether a nullable value is null or not. For example, C# provides the **`is`** operator to examine the runtime type, which is generally known as type introspection and is similar to the type-based switch. C# also provides the null coalescing operator `??` and simplifies the common pattern **switch** e **as** x **case** $A \Rightarrow x$ **case Null** $\Rightarrow d$ as `e??d` for nullable values.

Dynamically typed languages. It may be surprising at first sight that dynamically typed languages can benefit from our work with static typing, but recall that the type-safety issue in Section 10.2.2 was found in Python. Nowadays, popular dynamically typed languages have been retrofitted with gradual typing. For example, Python has type hints and mypy [Lehtosalo et al. 2012], Ruby has RBS and Steep [Matsumoto et al. 2018], JavaScript gets typed by TypeScript [Microsoft 2012], and Lua gets typed by Luau [Roblox 2019]. All of these typed versions support record-like and union types, and all except Python also support intersection types. Our translation scheme can almost directly apply to these languages. For a concrete example, we show how the aforementioned `exp` function can be encoded in TypeScript:

```
function exp(args: { x: number } & { base: number|null }) {
  let x = args.x;
  let base = (typeof args.base === "number") ? args.base : Math.E;
  return Math.pow(base, x);
}
exp({ base: 2, x: 10 })     //= 1024
exp({ x: 10, base: null })  //= e^10
```

The code is almost the same as in Section 10.3.2. Note that the **`typeof`** operator is the standard way to perform type introspection in TypeScript, and the type of `args.base` is refined from the **`number|null`** to **`number`** in the true-branch. We assume the call sites have been rewritten in the code above. In this manner, named and optional arguments can be added to TypeScript as syntactic sugar.

Although we have discussed several alternatives to intersection and union types, we believe that if a language is designed from scratch, our approach is a good choice. Intersection and union types not only subsume multi-field record and nullable types but also provide a solid and elegant foundation for other advanced features, such as function over-

loading and heterogeneous data structures. The essay by Castagna [2023] is an excellent further reading on the beauty of programming with intersection and union types.

10.4 Formalization

In this section, we formalize the translation of named and optional arguments as an elaboration semantics. The target of elaboration is called λ_{iu}, and the source is called Uaena. We prove that the source language with named and optional arguments is type-safe via (1) the type soundness of the target calculus and (2) the type soundness of elaboration. All the theorems are mechanically proven using the Coq proof assistant.

10.4.1 The Target Calculus: λ_{iu}

λ_{iu} is an extension to the calculus in Chapter 5 of the dissertation by Rehman [2023] with **null**, single-field records, and let-in bindings. The addition of let-in bindings is not essential because they can be desugared into lambda abstractions and applications:

$$\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \quad \equiv \quad (\lambda x.\ e_2)\ e_1$$

However, we still have let-in bindings for the sake of readability, and this form of let-in bindings simplifies the rules of parameter elaboration (introduced later in Figure 10.5). Another difference is that the original calculus uses the locally nameless representation [Charguéraud 2012] while ours directly uses names for bound variables.

Our changes to Rehman's calculus are relatively trivial, and we do not touch the rules for intersection and union types. We will not discuss his design choices in this thesis, because our focus is on the type soundness with the addition of **Null** and record types. We have proven in Coq that these extensions preserve type soundness.

Syntax of λ_{iu}

$$\begin{array}{lll}
\text{Types} & A, B ::= & \top \mid \bot \mid \textbf{Null} \mid \mathbb{Z} \mid A \to B \mid \{\ell : A\} \mid A \wedge B \mid A \vee B \\
\text{Expressions} & e ::= & \{\} \mid \textbf{null} \mid n \mid x \mid \lambda x{:}A.\ e{:}B \mid e_1\ e_2 \mid \{\ell : A = e\} \mid e.\ell \\
& & \mid e_1\ ,\ e_2 \mid \textbf{switch}\ e_0\ \textbf{as}\ x\ \textbf{case}\ A \Rightarrow e_1\ \textbf{case}\ B \Rightarrow e_2 \mid \mathit{letin}\ e
\end{array}$$

The types include the top type $\top$, the bottom type $\bot$, the null type **Null**, the integer type $\mathbb{Z}$, function types $A \to B$, record types $\{\ell : A\}$, intersection types $A \wedge B$, and union types $A \vee B$. **Null** is a unit type that has only one value **null**.

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

$$\textsc{Sub-Null}\ \frac{}{\mathbf{Null} <: \mathbf{Null}} \qquad \textsc{Sub-Int}\ \frac{}{\mathbb{Z} <: \mathbb{Z}} \qquad \textsc{Sub-Arrow}\ \frac{A_2 <: A_1 \qquad B_1 <: B_2}{A_1 \to B_1 <: A_2 \to B_2} \qquad \textsc{Sub-Rcd}\ \frac{A <: B}{\{\ell : A\} <: \{\ell : B\}}$$

$$\textsc{Sub-And}\ \frac{A <: B \qquad A <: C}{A <: B \wedge C} \qquad \textsc{Sub-AndL}\ \frac{A <: C}{A \wedge B <: C} \qquad \textsc{Sub-AndR}\ \frac{B <: C}{A \wedge B <: C} \qquad \textsc{Sub-Or}\ \frac{A <: C \qquad B <: C}{A \vee B <: C}$$

$$\textsc{Sub-OrL}\ \frac{A <: B}{A <: B \vee C} \qquad \textsc{Sub-OrR}\ \frac{A <: C}{A <: B \vee C} \qquad \textsc{Sub-Top}\ \frac{}{A <: \top} \qquad \textsc{Sub-Bot}\ \frac{}{\bot <: A}$$

Figure 10.3: Subtyping of λ_{iu}.

The expressions include the empty record $\{\}$, the null value **null**, integer literals n, variables x, lambda abstractions $\lambda x : A.\ e : B$, function applications $e_1\ e_2$, record literals $\{\ell : A = e\}$, record projections $e.\ell$, merges $e_1\ ,\ e_2$, type-based switch expressions $\mathbf{switch}\ e_0\ \mathbf{as}\ x\ \mathbf{case}\ A \Rightarrow e_1\ \mathbf{case}\ B \Rightarrow e_2$, and let-in bindings $\mathit{letin}\ e$. The syntax of *letin* is as follows:

$$\mathit{letin} ::= \mathbf{let}\ x = e\ \mathbf{in} \mid \mathit{letin}_1 \circ \mathit{letin}_2 \mid \mathbf{id}$$

The composition of two let-in bindings is denoted by $\mathit{letin}_1 \circ \mathit{letin}_2$, and an empty binding is denoted by **id**.

Subtyping. Figure 10.3 shows the subtyping rules of λ_{iu}. The rules are standard for a type system with intersection and union types. Rule Sub-Top shows that the top type $\top$ is a supertype of any type, and rule Sub-Bot shows that the bottom type $\bot$ is a subtype of any type. Rules Sub-And, Sub-AndL, and Sub-AndR handle the subtyping for intersection types, while rules Sub-Or, Sub-OrL, and Sub-OrR are for union types. Rules Sub-Null and Sub-Rcd added by us are straightforward. We prove that the subtyping relation is reflexive and transitive.

Theorem 10.1 (Subtyping Reflexivity). $\forall A, A <: A$.

Theorem 10.2 (Subtyping Transitivity). *If* $A <: B$ *and* $B <: C$*, then* $A <: C$.

Typing contexts $\quad \Gamma ::= \cdot \mid \Gamma, x : A$

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

$$\textsc{Typ-Top}\ \frac{}{\Gamma \vdash \{\} : \top} \qquad \textsc{Typ-Null}\ \frac{}{\Gamma \vdash \mathbf{null} : \mathbf{Null}} \qquad \textsc{Typ-Int}\ \frac{}{\Gamma \vdash n : \mathbb{Z}} \qquad \textsc{Typ-Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\textsc{Typ-Abs}\ \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash (\lambda x{:}A.\ e{:}B) : A \to B} \qquad \textsc{Typ-App}\ \frac{\Gamma \vdash e_1 : A \to B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B} \qquad \textsc{Typ-Rcd}\ \frac{\Gamma \vdash e : A}{\Gamma \vdash \{\ell : A = e\} : \{\ell : A\}}$$

$$\textsc{Typ-Prj}\ \frac{\Gamma \vdash e : \{\ell : A\}}{\Gamma \vdash e.\ell : A} \qquad \textsc{Typ-Let}\ \frac{\Gamma \vdash \mathit{letin} \dashv \Gamma' \quad \Gamma' \vdash e : A}{\Gamma \vdash \mathit{letin}\ e : A} \qquad \textsc{Typ-Merge}\ \frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash e_1\ ,\ e_2 : A \wedge B}$$

$$\textsc{Typ-Switch}\ \frac{\begin{array}{c}\Gamma \vdash e_0 : A \vee B \\ \Gamma, x : A \vdash e_1 : C \\ \Gamma, x : B \vdash e_2 : C\end{array}}{\Gamma \vdash \mathbf{switch}\ e_0\ \mathbf{as}\ x\ \mathbf{case}\ A \Rightarrow e_1\ \mathbf{case}\ B \Rightarrow e_2 : C} \qquad \textsc{Typ-Sub}\ \frac{\Gamma \vdash e : A \qquad A <: B}{\Gamma \vdash e : B}$$

$\boxed{\Gamma \vdash \mathit{letin} \dashv \Gamma'}$ *(Let-in binding)*

$$\textsc{LB-Let}\ \frac{\Gamma \vdash e : A}{\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in} \dashv \Gamma, x : A} \qquad \textsc{LB-Comp}\ \frac{\Gamma \vdash \mathit{letin}_1 \dashv \Gamma' \quad \Gamma' \vdash \mathit{letin}_2 \dashv \Gamma''}{\Gamma \vdash \mathit{letin}_1 \circ \mathit{letin}_2 \dashv \Gamma''} \qquad \textsc{LB-Id}\ \frac{}{\Gamma \vdash \mathbf{id} \dashv \Gamma}$$

Figure 10.4: Typing of λ_{iu}.

Typing. Figure 10.4 shows the typing rules of λ_{iu}. The empty record $\{\}$ has the top type $\top$, as shown in rule Typ-Top. Rule Typ-Merge is the introduction rule for intersection types. Merging two functions is used for function overloading, and merging two records is used for record concatenation. Rule Typ-Switch is the elimination rule for union types. The type-based switch expression scrutinizes an expression having a union of the two scrutinizing types (i.e. $e_0 : A \vee B$). This premise ensures the exhaustiveness of the cases in the switch. The **as**-variable x is refined to type A in e_1 and to type B in e_2. Rules Typ-Null, Typ-Rcd, and Typ-Prj added by us are straightforward. Rule Typ-Let uses an auxiliary judgment $\Gamma \vdash \mathit{letin} \dashv \Gamma'$ to obtain the typing context for the body of the let-in binding. For example, if e_1 has type A, then **let** $x = e_1$ **in** e_2 adds $x : A$ to the typing context before type-checking e_2.

Dynamic semantics. We have a small-step operational semantics for λ_{iu}. The judgment $e \longrightarrow e'$ means that e reduces to e' in one step, and $e \longrightarrow^* e'$ is for multi-step reduction. We extend the original dynamic semantics by adding rules for records and projections. Similarly to the applicative dispatch for function applications in the original calculus, we add a relation called projective dispatch for record projections. For example, $(\{x = 1\}, \{y = 2\}).x$ reduces to $\{x = 1\}.x$ via projective dispatch to select the needed field.

Since the dynamic semantics of λ_{iu} is independent of the elaboration from Uaena to λ_{iu}, we omit the rules here but leave them in Appendix F. Note that the operational semantics is not commonplace in that it is type-directed and non-deterministic. Please refer to the dissertation by Rehman [2023] for detailed explanations.

Theorem 10.3 (Progress). *If* $\cdot \vdash e : A$, *then either* e *is a value or* $\exists e', e \longrightarrow e'$.

Theorem 10.4 (Preservation). *If* $\cdot \vdash e : A$ *and* $e \longrightarrow e'$, *then* $\cdot \vdash e' : A$.

Putting progress and preservation together, we conclude that λ_{iu} is type-sound: a well-typed term can never reach a stuck state.

Corollary 10.5 (Type Soundness). *If* $\cdot \vdash e : A$ *and* $e \longrightarrow^* e'$, *then either* e' *is a value or* $\exists e'', e' \longrightarrow e''$.

10.4.2 The Source Calculus: Uaena

Uaena (*Unnamed Arguments Extended with Named Arguments*) is a minimal calculus with named and optional arguments. Although the calculus is small, named arguments are supported as first-class values and can be passed to or returned by a function. Besides functions with named arguments, Uaena also supports normal functions with positional

arguments. The two kinds of functions are distinguished in the syntax, as seen in Ruby, Racket, OCaml, etc.

Syntax of Uaena

Types	$\mathcal{A}, \mathcal{B} ::= \mathbb{Z} \mid (\mathcal{A}) \to \mathcal{B} \mid \{\mathcal{P}\} \to \mathcal{B} \mid \{\mathcal{K}\}$
Named parameter types	$\mathcal{P} ::= \cdot \mid \mathcal{P};\ \ell : \mathcal{A} \mid \mathcal{P};\ \ell? : \mathcal{A}$
Named argument types	$\mathcal{K} ::= \cdot \mid \mathcal{K};\ \ell : \mathcal{A}$
Expressions	$\epsilon ::= n \mid x \mid \lambda(x{:}\mathcal{A}).\ \epsilon \mid \lambda\{\rho\}.\ \epsilon \mid \epsilon_1\ \epsilon_2 \mid \{\kappa\}$
Named parameters	$\rho ::= \cdot \mid \rho;\ \ell : \mathcal{A} \mid \rho;\ \ell = \epsilon$
Named arguments	$\kappa ::= \cdot \mid \kappa;\ \ell = \epsilon$

The types include the integer type $\mathbb{Z}$, normal function types $(\mathcal{A}) \to \mathcal{B}$, function types with named parameters $\{\mathcal{P}\} \to \mathcal{B}$, and (first-class) named argument types $\{\mathcal{K}\}$. The expressions include integer literals n, variables x, normal lambda abstractions $\lambda(x{:}\mathcal{A}).\ \epsilon$, lambda abstractions with named parameters $\lambda\{\rho\}.\ \epsilon$, function applications $\epsilon_1\ \epsilon_2$, and (first-class) named arguments $\{\kappa\}$.

A named parameter type $\mathcal{P}$ can be required ($\ell : \mathcal{A}$) or optional ($\ell? : \mathcal{A}$). If a named parameter is optional, its default value must be provided in the function definition. For example, $\lambda\{x : \mathbb{Z};\ y = 0\}.\ x + y$ has type $\{x : \mathbb{Z};\ y? : \mathbb{Z}\} \to \mathbb{Z}$. A function with named parameters can only be applied to named arguments, which are basically a list of key-value pairs. For example, the previous function can be applied to $\{x = 1;\ y = 2\}$ or $\{x = 1\}$ or a variable having a compatible type. The variable case demonstrates the first-class nature of named arguments in Uaena.

Careful readers may notice that a named argument type can also serve as the parameter of a normal function. This also demonstrates the first-class nature of named arguments. But note that a normal function that takes named arguments is different from a function with named parameters. Consider the following two functions, the former of which is a function with named parameters and the latter is a normal function:

$$\begin{aligned} (\lambda\{x : \mathbb{Z};\ y = 0\}.\ x + y) &\quad : \quad \{x : \mathbb{Z};\ y? : \mathbb{Z}\} \to \mathbb{Z} \\ (\lambda(args : \{x : \mathbb{Z};\ y : \mathbb{Z}\}).\ args) &\quad : \quad (\{x : \mathbb{Z};\ y : \mathbb{Z}\}) \to \{x : \mathbb{Z};\ y : \mathbb{Z}\} \end{aligned}$$

Although both functions can be applied to $\{x = 1;\ y = 2\}$, there are two main differences between them. First, optional parameters cannot be defined in a normal function. So we

Typing contexts $\quad \Delta ::= \cdot \mid \Delta,\, x : \mathcal{A}$

$\boxed{\Delta \vdash \epsilon : \mathcal{A} \rightsquigarrow e}$ *(Elaboration)*

$$\textsc{Ela-Int}\ \frac{}{\Delta \vdash n : \mathbb{Z} \rightsquigarrow n} \qquad \textsc{Ela-Var}\ \frac{x : \mathcal{A} \in \Delta}{\Delta \vdash x : \mathcal{A} \rightsquigarrow x}$$

$$\textsc{Ela-Abs}\ \frac{\Delta,\, x : \mathcal{A} \vdash \epsilon : \mathcal{B} \rightsquigarrow e}{\Delta \vdash \lambda(x{:}\mathcal{A}).\, \epsilon : (\mathcal{A}) \to \mathcal{B} \rightsquigarrow \lambda x{:}|\mathcal{A}|.\, e{:}|\mathcal{B}|} \qquad \textsc{Ela-App}\ \frac{\Delta \vdash \epsilon_1 : (\mathcal{A}) \to \mathcal{B} \rightsquigarrow e_1 \quad \Delta \vdash \epsilon_2 : \mathcal{A} \rightsquigarrow e_2}{\Delta \vdash \epsilon_1\, \epsilon_2 : \mathcal{B} \rightsquigarrow e_1\, e_2}$$

$$\textsc{Ela-NAbs}\ \frac{\Delta \vdash_x \rho : \mathcal{P} \rightsquigarrow \mathit{letin} \dashv \Delta' \quad \Delta' \vdash \epsilon : \mathcal{B} \rightsquigarrow e}{\Delta \vdash \lambda\{\rho\}.\, \epsilon : \{\mathcal{P}\} \to \mathcal{B} \rightsquigarrow \lambda x{:}|\mathcal{P}|.\, \mathit{letin}\, e{:}|\mathcal{B}|} \qquad \textsc{Ela-NApp}\ \frac{\Delta \vdash \epsilon_1 : \{\mathcal{P}\} \to \mathcal{B} \rightsquigarrow e_1 \quad \Delta \vdash \epsilon_2 : \{\mathcal{K}\} \rightsquigarrow e_2 \quad \Delta \vdash_{e_2} \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e_2'}{\Delta \vdash \epsilon_1\, \epsilon_2 : \mathcal{B} \rightsquigarrow e_1\, e_2'}$$

$$\textsc{Ela-NEmpty}\ \frac{}{\Delta \vdash \{\cdot\} : \{\cdot\} \rightsquigarrow \{\}} \qquad \textsc{Ela-NField}\ \frac{\Delta \vdash \{\kappa\} : \{\mathcal{K}\} \rightsquigarrow e' \quad \Delta \vdash \epsilon : \mathcal{A} \rightsquigarrow e}{\Delta \vdash \{\kappa;\, \ell = \epsilon\} : \{\mathcal{K};\, \ell : \mathcal{A}\} \rightsquigarrow e' \,,\, \{\ell : |\mathcal{A}| = e\}}$$

$\boxed{\Delta \vdash_x \rho : \mathcal{P} \rightsquigarrow \mathit{letin} \dashv \Delta'}$ *(Named parameter elaboration)*

$$\textsc{PEla-Empty}\ \frac{}{\Delta \vdash_x \cdot : \cdot \rightsquigarrow \mathbf{id} \dashv \Delta}$$

$$\textsc{PEla-Required}\ \frac{\Delta \vdash_x \rho : \mathcal{P} \rightsquigarrow \mathit{letin} \dashv \Delta'}{\Delta \vdash_x (\rho;\, \ell : \mathcal{A}) : (\mathcal{P};\, \ell : \mathcal{A}) \rightsquigarrow \mathit{letin} \circ \mathbf{let}\, \ell = x.\ell\, \mathbf{in} \dashv \Delta',\, \ell : \mathcal{A}}$$

$$\textsc{PEla-Optional}\ \frac{\Delta \vdash_x \rho : \mathcal{P} \rightsquigarrow \mathit{letin} \dashv \Delta' \quad \Delta' \vdash \epsilon : \mathcal{A} \rightsquigarrow e}{\begin{array}{l}\Delta \vdash_x (\rho;\, \ell = \epsilon) : (\mathcal{P};\, \ell? : \mathcal{A}) \\ \rightsquigarrow \mathit{letin} \circ \mathbf{let}\, \ell = \mathbf{switch}\, x.\ell\, \mathbf{as}\, y\, \mathbf{case}\, |\mathcal{A}| \Rightarrow y\, \mathbf{case}\, \mathbf{Null} \Rightarrow e\, \mathbf{in} \dashv \Delta',\, \ell : \mathcal{A}\end{array}}$$

Figure 10.5: Type-directed elaboration from Uaena to λ_{iu}.

cannot provide $y = 0$ as a default value in the second function. Second, x and y are not brought into the scope of the function body in a normal function. So the only accessible variable is *args* in the second function.

Elaboration. The type-directed elaboration from Uaena to λ_{iu} is defined at the top of Figure 10.5. $\Delta \vdash \epsilon : \mathcal{A} \rightsquigarrow e$ means that the source expression ϵ has type $\mathcal{A}$ and elaborates to the target expression e under the typing context Δ. Rules Ela-Abs and Ela-App for normal functions are straightforward. In rule Ela-NAbs for functions with named parameters, besides inferring the type of the function body ϵ and elaborating it to e, we generate let-bindings for the named parameters, which is delegated to the auxiliary judgment $\Delta \vdash_x \rho : \mathcal{P} \rightsquigarrow \mathit{letin} \dashv \Delta'$. In rule Ela-NApp, there is also an auxiliary judgment $\Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'$ that rewrites call sites according to the parameter and argument types. Rules Ela-NEmpty and Ela-NField are used to elaborate named arguments.

Named parameter elaboration. As shown at the bottom of Figure 10.5, $\Delta \vdash_x \rho : \mathcal{P} \rightsquigarrow \mathit{letin} \dashv \Delta'$ means that the named parameter ρ is inferred to have type $\mathcal{P}$ and elaborates to a series of let-in bindings *letin*, given that the named parameters correspond to the target bound variable x. In the meanwhile, the typing context Δ is extended with the types of the named parameters to form Δ'. Δ' is used for typing the body of the function with named parameters. Rule PEla-Required simply generates $\mathbf{let}\ \ell = x.\ell\ \mathbf{in}$, while rule PEla-Optional generates $\mathbf{let}\ \ell = \mathbf{switch}\ x.\ell\ \mathbf{as}\ y\ \mathbf{case}\ |\mathcal{A}| \Rightarrow y\ \mathbf{case}\ \mathbf{Null} \Rightarrow e\ \mathbf{in}$ to provide a default value e for the **Null** case.

Call site rewriting. As shown in Figure 10.6, $\Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'$ means that if the parameter type $\mathcal{P}$ is compatible with the argument type $\mathcal{K}$, the target expression e, which corresponds to the named arguments, will be rewritten to e'. The compatibility check is based on the parameter type $\mathcal{P}$. Rule PMat-Required handles the case where the argument is required, while rules PMat-Present and PMat-Absent handle the cases where the optional argument with a specific type is present and where the optional argument is absent, respectively. The remaining case, where the optional argument is present but associated with a wrong type, is prohibited and cannot elaborate to any term. We have two more auxiliary judgments $\mathcal{K} :: \ell \Rightarrow \mathcal{A}$ and $\mathcal{K} :: \ell \nRightarrow$ to indicate that the argument type $\mathcal{K}$ contains a field ℓ with type $\mathcal{A}$ or $\mathcal{K}$ does not contain ℓ.

Type translation. As we have informally mentioned in Section 10.3.1, we translate named parameters to intersection types and optional parameters to union types. The

$\boxed{\Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'}$ *(Call site rewriting)*

$$\textsc{PMat-Empty}\ \frac{}{\Delta \vdash_e \cdot \diamond \mathcal{K} \rightsquigarrow \{\}} \qquad \textsc{PMat-Required}\ \frac{\mathcal{K} :: \ell \Rightarrow \mathcal{A} \qquad \Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'}{\Delta \vdash_e (\mathcal{P}; \ell : \mathcal{A}) \diamond \mathcal{K} \rightsquigarrow e' \,,\, \{\ell : |\mathcal{A}| = e.\ell\}}$$

$$\textsc{PMat-Present}\ \frac{\mathcal{K} :: \ell \Rightarrow \mathcal{A} \qquad \Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'}{\Delta \vdash_e (\mathcal{P}; \ell? : \mathcal{A}) \diamond \mathcal{K} \rightsquigarrow e' \,,\, \{\ell : |\mathcal{A}| \vee \mathbf{Null} = e.\ell\}}$$

$$\textsc{PMat-Absent}\ \frac{\mathcal{K} :: \ell \Rightarrow\!\!\!\!\!\!\circ \qquad \Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'}{\Delta \vdash_e (\mathcal{P}; \ell? : \mathcal{A}) \diamond \mathcal{K} \rightsquigarrow e' \,,\, \{\ell : |\mathcal{A}| \vee \mathbf{Null} = \mathbf{null}\}}$$

$\boxed{\mathcal{K} :: \ell \Rightarrow \mathcal{A}}$ *(Successful lookup)*

$$\textsc{LU-Present}\ \frac{\mathcal{K} :: \ell \Rightarrow\!\!\!\!\!\!\circ}{(\mathcal{K}; \ell : \mathcal{A}) :: \ell \Rightarrow \mathcal{A}} \qquad \textsc{LU-Absent}\ \frac{\ell' \neq \ell \qquad \mathcal{K} :: \ell \Rightarrow \mathcal{A}}{(\mathcal{K}; \ell' : \mathcal{B}) :: \ell \Rightarrow \mathcal{A}}$$

$\boxed{\mathcal{K} :: \ell \Rightarrow\!\!\!\!\!\!\circ}$ *(Failed lookup)*

$$\textsc{LD-Empty}\ \frac{}{\cdot :: \ell \Rightarrow\!\!\!\!\!\!\circ} \qquad \textsc{LD-Absent}\ \frac{\ell' \neq \ell \qquad \mathcal{K} :: \ell \Rightarrow\!\!\!\!\!\!\circ}{(\mathcal{K}; \ell' : \mathcal{A}) :: \ell \Rightarrow\!\!\!\!\!\!\circ}$$

Figure 10.6: Type-directed call site rewriting in Uaena.

$\boxed{|\mathcal{A}|}$ Type translation

$$|\mathbb{Z}| \equiv \mathbb{Z} \qquad |(\mathcal{A}) \to \mathcal{B}| \equiv |\mathcal{A}| \to |\mathcal{B}| \qquad |\{\mathcal{P}\} \to \mathcal{B}| \equiv |\mathcal{P}| \to |\mathcal{B}| \qquad |\{\mathcal{K}\}| \equiv |\mathcal{K}|$$

$\boxed{|\mathcal{P}|}$ Parameter type translation

$$|\cdot| \equiv \top \qquad |\mathcal{P}; \ell : \mathcal{A}| \equiv |\mathcal{P}| \wedge \{\ell : |\mathcal{A}|\} \qquad |\mathcal{P}; \ell? : \mathcal{A}| \equiv |\mathcal{P}| \wedge \{\ell : |\mathcal{A}| \vee \mathbf{Null}\}$$

$\boxed{|\mathcal{K}|}$ Argument type translation

$$|\cdot| \equiv \top \qquad |\mathcal{K}; \ell : \mathcal{A}| \equiv |\mathcal{K}| \wedge \{\ell : |\mathcal{A}|\}$$

$\boxed{|\Delta|}$ Typing context translation

$$|\cdot| \equiv \cdot \qquad |\Delta, x : \mathcal{A}| \equiv |\Delta|, x : |\mathcal{A}|$$

Figure 10.7: Type translation from Uaena to λ_{iu}.

rules for $|\cdot|$ can be found in Figure 10.7. Having defined the translation, we can prove the soundness of call site rewriting and elaboration.

Theorem 10.6 (Soundness of Call Site Rewriting). *If* $\Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'$ *and* $|\Delta| \vdash e : |\mathcal{K}|$, *then* $|\Delta| \vdash e' : |\mathcal{P}|$.

Theorem 10.7 (Soundness of Elaboration). *If* $\Delta \vdash \epsilon : \mathcal{A} \rightsquigarrow e$, *then* $|\Delta| \vdash e : |\mathcal{A}|$.

With the two theorems above and the type soundness of λ_{iu}, we can conclude that Uaena is type-safe.

10.5 Discussion

In this section, we first discuss OCaml, the only language we know of that has well-studied support for named and optional arguments, though its mechanism does not go well with higher-order functions. Then we briefly show how named arguments are handled very differently in Scala and Racket. Finally, we illustrate how named arguments can be encoded as records in Haskell while not natively supported. We will also explain why all these approaches have drawbacks.

10.5.1 OCaml

OCaml did not support named arguments originally. Nevertheless, Garrigue [1994] conducted research on the label-selective λ-calculus and implemented it in OLabl [Garrigue

1999], which extends OCaml with labeled and optional arguments, among others. All features of OLabl were merged into OCaml 3, despite subtle differences [Garrigue 2001].

Here is an example of the exponential function defined in a labeled style:

```
let exp ?(base = Float.exp 1.0) x = base ** x
(* val exp : ?base:float → float → float *)
exp 10.0              (*= e^10. *)
exp 10.0 ~base:2.0    (*= 1024. *)
(exp 10.0) ~base:2.0  (* TypeError! *)
```

In the definition of `exp`, `base` is an optional labeled parameter while `x` is a positional parameter. Changing `x` into a second labeled parameter will trigger an unerasable-optional-argument warning because OCaml expects that there should be a positional parameter after all optional parameters. This expectation is at the heart of how OCaml resolves the ambiguity introduced by currying.

For example, consider the function application `exp 10.0`. Is it a partially applied function or a fully applied one using the default value of `base`? Both interpretations are possible, but OCaml considers it to be a full application because the trailing positional argument `x` is given. The presence of the positional argument is used to indicate that the optional arguments before it can be replaced by their default values. However, this design may confuse users since `(exp 10.0) ~base:2.0` will raise a type error but `exp 10.0 ~base:2.0` will not. Partial application does not lead to an equivalent program in such situations.

Option types. In OCaml, an optional argument is internally represented as an `option` type, which comprises two constructors: `None` and `Some`. Here is an equivalent definition for `exp`:

```
let exp ?(base : float option) x =
  let base = match base with
             | None → Float.exp 1.0
             | Some b → b in
  base ** x
(* val exp : ?base:float → float → float *)
exp 10.0            (*> exp 10.0 ~base:None *)
exp 10.0 ~base:2.0  (*> exp 10.0 ~base:(Some 2.0) *)
```

This encoding is similar to union types, but it depends on the `option` type in the standard library. Unfortunately, this specific kind of `option` is not a built-in type in many mainstream languages, especially in those languages that do not support algebraic data types.

Higher-order functions. A surprising gotcha in OCaml is that the commutativity breaks down when we pass a function with labeled arguments to another function. *Real World OCaml* [Madhavapeddy and Minsky 2022] gives the following example:

```
let apply1 f (fst,snd) = f ~fst ~snd
(* val apply1 : (fst:'a → snd:'b → 'c) → 'a * 'b → 'c *)
let apply2 f (fst,snd) = f ~snd ~fst
(* val apply2 : (snd:'a → fst:'b → 'c) → 'b * 'a → 'c *)
let divide ~fst ~snd = fst / snd
(* val divide : fst:int → snd:int → int *)
apply1 divide (48,3)  (*= 16 *)
apply2 divide (48,3)
(* TypeError: "divide" has type fst:int → snd:int → int
       but was expected of type snd:'a → fst:'b → 'c *)
```

Normally, the order of named arguments does not matter in OCaml, so it type-checks whether we call `divide ~fst ~snd` or `divide ~snd ~fst`. However, order matters when we pass `divide` to a higher-order function. That is why `apply1 divide` type-checks while `apply2 divide` does not. It turns out that the OCaml way of handling labeled arguments does not go well with other features like higher-order functions. Our approach scales better in this regard and the commutativity still holds in higher-order contexts via intersection subtyping.

In short, OCaml has a very powerful label-selective core calculus that reconciles commutativity and currying, but it is quite complicated and may hinder its integration with other language features. Another thing worth mentioning is that labeled arguments in OCaml are not first-class values, so they cannot be assigned to a variable or passed around by functions. In contrast to OCaml, our approach supports first-class named arguments and targets a minimal core calculus with intersection and union types, which is compatible with many popular languages like Python, Ruby, JavaScript, etc.

10.5.2 Scala

Rytz and Odersky [2010] described the design of named and default arguments in Scala. Like in Python, parameter names in a method definition are non-mandatory keywords in Scala, and thus every argument can be passed with or without keywords. Furthermore, the parameter names are not part of the public interface of a method. This design is partly due to the backward compatibility with earlier versions of Scala, so the addition of named arguments will not break any existing code. As a result of the conservative treatment, named arguments are not first-class values in Scala and cannot be defined in an anonymous

function. In short, named arguments are more like syntactic sugar in Scala and do not interact with the type system.

Below we show an example in Scala. In order to let the default value of `c` depend on `a` and `b`, we make the function partly curried:

```
def f(a: Int, b: Int)(c: Int = a+b) = c
f(b = 1+1, a = 1)()
```

The code will be translated to equivalent code without keywords or defaults:

```
def f(a: Int, b: Int)(c: Int) = c
def f$default$3(a: Int, b: Int): Int = a+b
{
  val x$1 = 1+1
  val x$2 = 1
  val x$3 = f$default$3(x$2, x$1)
  f(x$2, x$1)(x$3)
}
```

There are two things to note here. First, a new function `f$default$3` is generated for the default value of `c`, taking two parameters `a` and `b`. Second, the call site is translated to a series of variable assignments for each argument and a keyword-free call to `f` with arguments reordered. The whole call site is wrapped in a block to avoid polluting the namespace.

In conclusion, named arguments in Scala are handled in a very different way from OCaml and our approach. The Scala way is more syntactic than type-theoretic, so it is hard to do an apples-to-apples comparison with our approach.

10.5.3 Racket

Flatt and Barzilay [2009] introduced keyword and optional arguments into Racket, which was known as PLT Scheme at that time. A keyword is prefixed with `#:` in syntax and is implemented as a new built-in type in Racket. Keyword arguments are supported by replacing **`define`**, **`lambda`**, and the core application form with newly defined macros that recognize keyword-argument forms. Here is an example of a function `f` with three keyword arguments `a`, `b`, and `c`, among which `c` is optional and defaults to `a+b`:

```
(define (f #:a a #:b b #:c [c (+ a b)]) c)
(f #:b (+ 1 1) #:a 1)
```

The function call with keywords seems hard to implement because it just lists the function and arguments in juxtaposition. In fact, an application form in Racket implicitly calls `#%app`

in its lexical scope, so the support for keyword arguments is done by supplying an `#%app` macro. A new **`keyword-apply`** function is also defined to accept keyword arguments as first-class values. For example, we can rewrite the function call above as follows[4]:

```
(keyword-apply f '(#:a #:b) `(1 ,(+ 1 1)) '()) ; OK!
(keyword-apply f '(#:b #:a) `(,(+ 1 1) 1) '()) ; Contract violation!
```

Note that we need to separate keywords and corresponding arguments into two lists. The third list is for positional arguments, so it is empty in this case. We cannot list keywords in arbitrary order: a contract violation will be signaled unless the keywords are sorted in alphabetical order. In other words, commutativity is lost for first-class keyword arguments in Racket.

In Typed Racket [Tobin-Hochstadt et al. 2015], Racket's gradually typed sister language, `f` can be typed as `(→* (#:a Number #:b Number) (#:c Number) Number)`. The first list contains the required arguments (`#:a` and `#:b`), and the second list contains the optional ones (`#:c`). However, Typed Racket does not provide a typed version of **`keyword-apply`**, and it is unclear how to properly type it.

In conclusion, Racket supports keyword and optional arguments in a unique way via its powerful macro system. However, the support for first-class keyword arguments is very limited and cannot easily transfer to a type-safe setting.

10.5.4 Haskell

Unlike the aforementioned languages, Haskell does not support named arguments natively. However, the paradigm of *named arguments as records* has long existed in the Haskell community. Although we have to uncurry a function to have all parameters labeled in a record, it is clearer and more human-readable, especially when different parameters have the same type. For example, in the web server library *warp* [Snoyman et al. 2011], various server settings are bundled in the data type `Settings`, as shown in Figure 10.8a. It is obvious how named arguments correspond to record fields, but it needs some thought on how to encode default values for optional arguments. The simplest approach, also used by *warp*, is to define a record `defaultSettings`, as shown in Figure 10.8b. Users can update whatever fields they want to change while keeping others. For example, we update `settingsPort` and `settingsHost` while keeping the rest unchanged in Figure 10.8c. Finally, we call the library function `runSettings` with the updated `settings` to run a server.

Such an approach works fine here but still has two drawbacks. The first issue is the dependency on `defaultSettings`. It is awkward for users to look for a record containing

[4] `'` is `quote`, `` ` `` is `quasiquote`, and `,` is `unquote` in Racket.

```haskell
data Settings = Settings
  { settingsPort :: Port
  , settingsHost :: HostPreference
  , settingsTimeOut :: Int
  , ...
  }
```

(a) Record type.

```haskell
defaultSettings = Settings
  { settingsPort = 3000
  , settingsHost = "*4"
  , settingsTimeout = 30
  , ...
  }
```

(b) Default values.

```haskell
runSettings :: Settings → Application → IO ()
runSettings = ...

main :: IO ()
main = runSettings settings app
  where settings = defaultSettings { settingsPort = 4000, settingsHost = "*6" }
```

(c) Updating some settings before running a server application.

Figure 10.8: Named arguments as records in Haskell.

particular default values, especially when there are a few similar records in a library. A better solution is to change the parameter of `runSettings` from a complete `Settings` to a function that updates `Settings`:

```haskell
runSettings' :: (Settings → Settings) → Application → IO ()
runSettings' update = runSettings (update defaultSettings)

main :: IO ()
main = runSettings' update app
  where update settings = settings { settingsPort = 4000, settingsHost = "*6" }
```

With the new interface, users do not need to look for default values anymore. However, this design still has a second drawback: all arguments must have default values. Usually, we do not consider every argument to be optional. For example, we may want to require users to fill in `settingsPort`. A workaround employed by `SqlBackend` in the library *persistent* [Parsons et al. 2021] is to have another function that asks for required arguments and supplements default values for optional arguments:

```haskell
{-# language DuplicateRecordFields, RecordWildCards #-}

data ReqSettings = ReqSettings { settingsPort :: Port }

mkSettings :: ReqSettings → Settings
mkSettings ReqSettings {..} =
```

```
              Settings { settingsHost = "*4", settingsTimeout = 30, .. }
```

Best practice. Although `mkSettings` resolves the second issue, there is a regression concerning the first issue: users have to look for `mk*` functions now. Fortunately, we can harmonize both design patterns to develop a third approach:

```
{-# language DuplicateRecordFields, RecordWildCards #-}

data OptSettings = OptSettings { settingsHost :: HostPreference
                               , settingsTimeOut :: Int }

runSettings'' :: (OptSettings → Settings) → Application → IO ()
runSettings'' update = runSettings (update defaultSettings)
  where defaultSettings = OptSettings { settingsHost = "*4"
                                      , settingsTimeout = 30 }

main :: IO ()
main = runSettings'' update app
  where update OptSettings {..} =
                  Settings { settingsPort = 4000, settingsHost = "*6", .. }
```

This last approach is probably the best practice in Haskell, though it is already quite complicated and requires two GHC language extensions. Of course, there could be other approaches to encoding named and optional arguments in Haskell. Users could get confused about the various available design patterns. This is largely due to lack of language-level support. We believe it is better for a language to natively support named and optional arguments.

Sidenote. The design pattern of *named arguments as records* can also be found in other functional languages like Standard ML, Elm, and PureScript, just to name a few. It is worth mentioning that these languages have first-class support for record types, so no separate type declarations are needed like in Figure 10.8a. However, they still suffer from lack of native support for optional arguments.

10.6 Conclusion

The benefits from named arguments are twofold. On the one hand, argument keywords serve as extra documentation at the language level. On the other hand, they lay the foundation for supporting commutativity and optionality.

Named and optional arguments are widely supported in mainstream programming languages but are hardly formalized. Our approach is inspired by existing mechanisms in OCaml and Haskell but further considers the interaction between first-class named arguments and subtyping. Static type checkers for Python and Ruby both suffer from a type-safety issue in this regard. We show that λ_{iu} can serve as a type-safe core calculus with compact support for intersection and union types.

We hope that this chapter will call more attention to the foundation for named and optional arguments and inspire future language developers to consider potential type-safety issues more carefully in their designs.

Part VI

Epilogue

11 Related Work

11.1 Embedded Domain-Specific Languages

Modular embeddings. There are quite a few existing approaches to modular embeddings, which essentially exploit solutions to the *expression problem* [Wadler 1998] in existing languages. Such approaches include, for example, *tagless-final embeddings* [Carette et al. 2009] and *data types à la carte* [Swierstra 2008] in functional programming, *polymorphic embeddings* [Hofer et al. 2008] and *object algebras* [Oliveira and Cook 2012] in object-oriented languages, and solutions to the *expression compatibility problem* [Haeri and Keir 2019; Haeri and Schupp 2016], just to name a few. Section 5.3.6 already compares those approaches with compositional embeddings in detail. In essence, the lack of sufficient programming language support for modularity makes it hard to model modular dependencies in those approaches. Compositional embeddings offer an elegant solution to many issues by exploiting the support for compositional programming in the CP language.

Hybrid embeddings. Svenningsson and Axelsson [2015] propose combining shallow and deep embeddings by translating a shallowly embedded interface to a deeply embedded core language. In this way, language interfaces can be shallowly extended, and multiple interpretations are possible in the deep core. In addition, some features in the host language can be exploited in the shallow part, which is called *deep linguistic reuse* in Scala-Virtualized [Rompf et al. 2012]. Jovanović et al. [2014] further propose an automatic translation between shallow and deep embeddings using Scala macros instead of a manual translation. Their Yin-Yang framework targets lightweight modular staging [Rompf and Odersky 2010] as the deep embedding backend but completely conceals internal encodings from the users. A similar but slightly different approach called *implicit staging* is proposed by Scherr and Chiba [2014]. Implicit staging extracts an intermediate representation from a shallowly embedded DSL and reintegrates it after some transformations. Their research prototype works in Java through load-time reflection. Compared to such hybrid approaches, compositional embeddings do not require the use of two different em-

beddings and the translations between them while supporting most features from both shallow and deep embeddings.

Generative programming. As stated in Section 5.4, EXT performs a lightweight desugaring during parsing to generate compositionally embedded fragments in CP. This is an *ad-hoc* generative technique. Some generative approaches in other languages, such as Racket macros [Ballantyne et al. 2020] and Template Haskell [Sheard and Peyton Jones 2002], provide more flexible mechanisms compared to ours. Nevertheless, in this work, our goal is not to develop generative programming in CP. Instead, we try to keep the DSL as close to the underlying compositional embeddings as possible so that there is an obvious one-to-one mapping between EXT and CP. Rather than syntactic extensibility (i.e. the ability to extend the syntax of the source language), our focus is on semantic support for DSLs. Our approach is different from some generative frameworks like EVF [Zhang and Oliveira 2017] and Castor [Zhang and Oliveira 2020], which generate extensible visitors from annotations via meta-programming. Although they also enable modular and extensible programming language components, their code uses various non-standard annotations, whose semantics may be unclear to programmers. What is worse, type checking is delayed in the aforementioned generative frameworks so error messages are reported in terms of the generated code. This can be quite confusing without detailed knowledge of how the generated code works. In contrast, in our implementation, type checking is done in the source language, and error messages are reported in terms of the original EXT code.

Language workbenches. JetBrains MPS [Fowler 2005b], Xtext [Efftinge and Völter 2006], MontiCore [Krahn et al. 2010], and Spoofax [Kats and Visser 2010], just to name a few, are popular *language workbenches* [Fowler 2005c] that enable easier DSL development. Erdweg et al. [2015] have done a comprehensive evaluation about them. Notably, in addition to textual DSLs, JetBrains MPS supports tables, mathematical symbols, and diagrammatic notations, which significantly enhance user experience. Modern language workbenches also provide editor support for user-defined DSLs. Similar to compositional embeddings, they often allow language components to be modularly composed, but their focus is mainly on syntactic extensibility instead of semantic extensibility. In contrast to direct embeddings into host languages, external tools are required by language workbenches to generate DSL implementations from specifications. This is the essential difference from compositional embeddings, where composition is directly built in the programming language.

Document DSLs. There are quite a few existing DSLs for document authoring. *LaTeX* is one of the most commonly used document languages, and ExT's syntax is inspired by it. In contrast to ExT, LaTeX is an external DSL and does not have a static type system. LaTeX supports arbitrary computation, but it is not easy to write meta-programs over the AST of LaTeX. Moreover, LaTeX is not intended to render web documents, which is an important goal of ExT. *Wikitext* [MediaWiki 2003] is widely used across wiki websites, including Wikipedia and Fandom. Wikitext has some design limitations like the absence of type safety, data consistency, and general-purpose computation, which have been elaborated in Section 5.5.1. *Skribe* [Gallesio and Serrano 2005] is a document DSL built on top of the Scheme programming language and extends Scheme's syntax from S-expressions to *Sk-expressions* for easier text writing. Skribe makes use of Scheme macros and functions to construct documents. This idea is inherited and popularized by *Scribble* [Flatt et al. 2009] in the Racket community, and DrRacket offers very good tool support for Scribble. Regarding syntax, a new *@-notation* is designed for Scribble, which is desugared to normal S-expressions. ExT's syntax is similar to Scribble's but is more flexible. Based on the infrastructure provided by Racket, Scribble supports arbitrary computation and linguistic reuse like ExT. However, Scribble has a fixed document structure [Flatt 2009], and its rendering function performs conventional pattern matching on it. Thus, it is not easy to extend their core document constructs and functions without modifying the original source code. Furthermore, there is no static typing in Scribble, though it uses Racket's *contract* system to perform runtime type checking.

11.2 Multiple, Dynamic Inheritance and Virtual Classes

Multiple inheritance is a well-known troublemaker in OOP languages, bringing the diamond problem and method conflicts, among other issues. Alternative notions like *mixins* [Bracha and Cook 1990] and *traits* [Ducasse et al. 2006] are proposed to alleviate the issues. A core difference between mixins and traits is how they handle conflicts when the same method name occurs in multiple ancestors. Mixins resolve conflicts implicitly by linearization (e.g. C3 linearization [Barrett et al. 1996]). However, the implicit resolution of conflicts may conceal accidental conflicts and lead to subtle bugs. Traits, on the other hand, require the programmer to resolve conflicts explicitly. CP adopts the trait model and imposes the disjointness constraint on merging (and trait inheritance). Note that the disjointness constraint does not only consider the method names but also takes into account the types of the methods, so the methods with the same name but different return

types are considered disjoint and do not conflict with each other. By this means, CP tries to reach a balance between safety and flexibility.

Dynamic inheritance and first-class classes. While various forms of multiple inheritance are well studied and implemented in some popular languages, such as C++, Ruby, and Scala, dynamic inheritance is more challenging and involved, especially in terms of static typing. In the literature of OOP, dynamic inheritance is often discussed in a broader context of *first-class classes* [Strickland et al. 2013], where inherited classes can be determined at run time, among other dynamic features. There are only a few statically typed languages that support first-class classes. To the best of our knowledge, they are `gbeta` [Ernst 2000], TypeScript [Microsoft 2012], Typed Racket [Takikawa et al. 2012], Wyvern [Lee et al. 2015], and most recently, CP [Zhang et al. 2021]. As elaborated in Section 4.2, the most popular one, TypeScript, has significant type-safety issues when dealing with dynamic inheritance.

Typed Racket is gradually typed and uses row polymorphism to represent class types. Similarly to the disjointness constraints in CP, there are constraints on row variables to express absence, and thus the *inexact superclass problem* that TypeScript suffers from is resolved in Typed Racket. However, the absence constraint on a row variable only includes the method name but not the type, so the dynamically inherited class is more restricted than in CP. Moreover, Xie et al. [2020] formally prove that CP's disjoint polymorphism is more powerful than similar forms of row polymorphism. Furthermore, unlike Typed Racket, CP can model virtual classes and family polymorphism.

Wyvern is a language for design-driven assurance, and Lee et al. [2015] explored a foundational account of first-class classes based on tagging [Glew 1999]. Similarly to our formalization, they give an elaboration semantics of an OOP language. However, their theory is very different from ours, and they target a more sophisticated calculus with hierarchical tagging and dependent types. In contrast, our target language is a standard record calculus. Furthermore, their calculus cannot model multiple inheritance or family polymorphism, and their implementation is an interpreter rather than a compiler.

The `gbeta` language is the most interesting one and is the closest to CP because it supports dynamic multiple inheritance and family polymorphism. However, separate compilation was not supported at the time when Ernst [2000] wrote his dissertation because of some technical issues with the Mjølner BETA persistence support. If this factor is disregarded, separate compilation can still be accomplished, but at the cost of efficient attribute lookup. Since an object in `gbeta` may have more mixin instances at run time than what is statically known, and the mixin instances may occur in a different order, the offset of an attribute cannot be determined statically. As a result, `gbeta` has to perform a linear search

through super-mixins to look up inherited attributes. In contrast, attribute lookup in CP, even with dynamic inheritance, is more efficient, and no linear search is needed.

The notion of *patterns* in `gbeta` unifies classes and methods, and patterns can be composed using the combination operator '&', which is similar to the merge operator '$,$' in CP. Though dynamic multiple inheritance can be achieved using '&', only a subset of dynamic combinations is safe, where at least one of the classes being composed must be created by single inheritance [Ernst 2002]. Otherwise, the C3 linearization algorithm used by `gbeta` may fail at run time. In contrast, dynamic inheritance via '$,$' is completely type-safe, because CP utilizes disjointness to avoid conflicts, and no linearization is needed. This difference has been summarized earlier as *mixins versus traits*. Some other notable consequences of this difference are:

- '$,$' is commutative, while '&' is not;
- '$,$' supports mutual dependencies between traits, while '&' rejects such cycles.

Virtual classes and family polymorphism. Virtual classes [Madsen and Møller-Pedersen 1989], similarly to virtual methods, are nested classes that can be overridden in subclasses. Virtual classes enable family polymorphism [Ernst 2001], which can naturally solve the expression problem [Ernst 2004]. The idea of virtual classes was initially introduced in the BETA programming language [Madsen et al. 1993] and later generalized in `gbeta` [Ernst 2000]. CaesarJ [Aracic et al. 2006], an aspect-oriented programming language based on Java, also supports virtual classes but does not allow cross-family inheritance and dynamic inheritance. Newspeak [Bracha et al. 2010], a descendant of Smalltalk, combines virtual classes and first-class modules (i.e. instances of top-level classes) but is dynamically typed. The calculi Jx [Nystrom et al. 2004], J& [Nystrom et al. 2006], *vc* [Ernst et al. 2006], Tribe [Clarke et al. 2007], and .FJ [Saito et al. 2008], just to name a few, formalize virtual classes with static inheritance but do not support dynamic inheritance.

Zhang and Myers [2017] propose the Familia programming language that unites object-oriented polymorphism and parametric polymorphism by unifying interfaces and type classes. In Familia, a mechanism of family polymorphism based on *nested inheritance*, similarly to Jx [Nystrom et al. 2004], is also deployed. During compilation, a *linkage* is computed for every class, which consists of a self-reference, a dispatch table, and the linkages of its nested classes, among others. At the heart of the mechanism is *further binding* [Madsen et al. 1993]: rewiring self-references for nested classes. Further binding is realized in Familia by linkage concatenation between families. This process is similar to the nested trait composition in CP, but there is a significant distinction in terms of sep-

arate compilation. CP *only* needs type information of the imported modules at compile time, while Familia requires class linkages that contain some implementation details of the imported modules (e.g. method definitions) and copy these details from superclasses' linkages. In this sense, with linkages, Familia supports some degree of separate compilation, but not to the same extent as the CP compiler does. Moreover, since Familia does not support dynamic inheritance, their class hierarchies are determined statically. In contrast, CP supports dynamic trait composition, which brings extra flexibility.

More recently, Kravchuk-Kirilyuk et al. [2024] propose Persimmon, a functional programming language that features extensible variant types and extensible pattern matching. CP also supports them via compositional interfaces and method patterns. Persimmon additionally allows types to be members of a family, relying on the support for *relative path types* [Saito et al. 2008] in their core calculus. Internally, Persimmon makes use of *linkages* that are similar to those in Familia. An important limitation of their current design is that modular type checking and separate compilation are *not* supported for multi-file programs, while CP fully supports them.

11.3 Compilation of Inheritance

In his excellent survey on inheritance, Taivalsaari [1996] distinguishes two strategies for implementing inheritance: *delegation* and *concatenation*. Most prototype-based languages, such as Self [Ungar and Smith 1987] and JavaScript, implement inheritance via delegation, where an object contains a reference to its prototype (e.g. `__proto__` in JavaScript), and methods that are not found in the current object will be delegated to its ancestors in the prototype chain. In contrast, CP implements inheritance via concatenation (a.k.a. *merging* throughout the thesis), where a trait is self-contained and itself contains all the methods of its ancestors. Although some copying is involved, the concatenation strategy is more efficient than delegation in terms of method lookup.

To improve the performance of method lookup, newer implementations of the Self language cache all lookup results for a polymorphic call site in a *polymorphic inline cache* (PIC) [Hölzle et al. 1991]. The methods cached in a PIC will be inlined into the caller to further reduce the overhead of method calls. Since a PIC is empty until a method is called for the first time, dynamic recompilation is required to optimize the code at run time. Moreover, the presence of *dynamic inheritance* may lead to a full method lookup in Self [Chambers 1992]. Modern JavaScript engines, such as V8 used in Node.js, utilize similar PIC-based techniques to optimize method calls. Though the CP compiler does not

implement inlining at all, which is definitely a useful optimization, it is still efficient in terms of method lookup, and dynamic inheritance *never* causes a slower lookup.

Typical compilers for mainstream class-based languages, such as C++ and Java, add a *virtual method table* (vtable) [Driesen et al. 1995] to each object to avoid searching for methods in the inheritance hierarchy at run time. A vtable is basically an array of function pointers, associating each method name (and parameter types if overloaded) with its implementation. Similarly, CP compiles an object to a type-indexed record, which also associates each method name and type with the corresponding implementation, among other fields. What is more, CP allows for first-class classes (traits) and dynamic inheritance, which are not supported by most mainstream languages. This is one of the key differences of our work compared to other OOP language compilers.

Another significant difference from mainstream OOP languages is that our compilation of inheritance is based on the denotational model by Cook and Palsberg [1989]. In this model, classes (traits) are encoded as functions, and inheritance is essentially merging functions, which is illustrated in Section 4.3.1. That is why the source language of the compilation scheme (λ_{i}^{+}) does not contain any notion of classes or objects. Such encodings are common in the literature on foundations for statically typed OOP [Bruce 2002; Bruce et al. 1999; Pierce 2002], and they largely simplify the formalization of compilation and its metatheory.

Since objects are encoded as multi-field records, which are essentially merges of single-field records in λ_{i}^{+}, the compilation of objects is closely related to the elaboration of merges. Next, we discuss the elaboration of intersection types and the merge operator.

Elaboration of intersection types and the merge operator. Dunfield [2014] shows that *unrestricted* intersection types and a term-level merge operator [Reynolds 1997] can encode various features like overloading and multi-field records, and they can be elaborated into product types and pairs. However, her approach lacks the critical property of coherence, i.e. the property that ensures the result of a merge is unambiguous. In the follow-up work on *disjoint* intersection types [Oliveira et al. 2016], the merged components are required to be disjoint with each other to avoid the semantic ambiguity. Alpuim et al. [2017] added parametric polymorphism to the calculus. Bi et al. [2018, 2019] further enhanced the intersection subtyping with distributivity, enabling more novel features like nested composition and family polymorphism. In other words, only Bi et al.'s F_{i}^{+} calculus fully covers the topics mentioned in Section 4.3. All the aforementioned work employs elaboration semantics with standard λ-calculi serving as targets. They use nested pairs as the target of elaboration, and consequently, the time complexity of extracting a compo-

nent by type can degenerate to linear in the worst case. In addition, extensible records require fewer coercions than nested pairs because some different source terms compile to equivalent records. These differences from our CP compiler have been discussed in detail in Section 6.1. In short, they do not consider more efficient runtime representations or eliminating redundant coercions, nor do they have benchmarks to evaluate performance. Instead, their focus is on proving the type safety and coherence of the elaboration. Furthermore, none of the aforementioned work develops a language with separate compilation units.

The compilation of merges in our work has similarities to the compilation of *type-indexed rows* [Shields and Meijer 2001], where record labels are discarded and record fields are sorted by their types. However, the work on type-indexed rows does not consider subtyping, which eliminates many of the issues that we had to deal with. For instance, they do not need to apply coercions to ensure that information statically hidden by subtyping is also hidden at run time.

Compilation of extensible records. Ohori [1995] investigates a polymorphic record calculus and introduces an efficient type-directed compilation method for records. Following the type-inference stage, records are converted into vectors with explicit indexing. However, his records are not extensible, and his method has difficulties to handle subtyping. Subtyping for records frequently enables field hiding and reordering, rendering it impossible to determine a label's offset statically. Gaster and Jones [1996] propose a compilation technique for polymorphic extensible records that utilizes qualified types [Jones 1994]. During the compilation process to the target language, supplementary parameters are introduced to determine suitable offsets. This approach is integrated into Hugs, a well-known implementation of Haskell, as an extension. Their system is later generalized by type-indexed rows [Shields and Meijer 2001]. In summary, subtyping and record concatenation (or merges) pose significant challenges to the compilation of extensible records. Our work takes pragmatic considerations into account, including targeting widely used dynamic languages such as JavaScript. As a result, we rely on the primitive support of objects and object extension in our target language and do not delve into low-level representations of extensible records, for which a comprehensive summary can be found in the paper by Leijen [2005].

11.4 Record Calculi with Optional Arguments

Ohori [1995] discussed how to model optional arguments in the future work of his seminal paper on compiling a polymorphic record calculus. He proposed to extend a record calculus with optional-field selection ($e.\ell$? d) which behaves like $e.\ell$ if ℓ is present in the record e or evaluates to d otherwise. However, his proposal is subject to a similar type-safety issue as mypy. The static type of e can easily lose track of the optional field ℓ and fail to ensure that $e.\ell$ has the same type as d at run time. Since Ohori did not explicitly mention how to type-check optional-field selection, we cannot make any firm conclusion about the type safety of his proposal.

Osinski [2006] also discussed the support for optional arguments in Section 3.5 of his dissertation on compiling record concatenation. His approach is based on row polymorphism and makes use of a sort of predicate on rows: $row_1 \blacktriangleright row_2$, which means that row_1 consists of all the fields in row_2. With this predicate, a function has type $\forall \rho.\ row_o \blacktriangleright \rho \Rightarrow \{row_r, \rho\} \to \tau$ if the required and optional arguments are denoted by row_r and row_o, respectively. Roughly speaking, it means the parameter has a type between $\{row_r\}$ and $\{row_r, row_o\}$. At the term level, he introduced a compatible concatenation operator `|&|`, which allows overlapping fields with the same types and prefers the fields on the right-hand side when overlapping occurs. An example of their translation is as follows:

```
fun add { x, y = 0 } = ...
(* is translated to *)
fun add r = let r' = { y = 0 } |&| r in
            let x = r'.x in
            let y = r'.y in ...
```

His approach is free from the type-safety issue, though based on a more sophisticated row-polymorphic system. There are two sorts of predicates and three variants of record concatenation operators in his calculus, for example, demonstrating some sophistication of his calculus.

It is worth noting that neither Ohori's nor Osinski's calculus supports subtyping. This is a significant limitation since subtyping is a common feature in many popular languages, especially object-oriented languages.

12 Conclusion and Future Work

The contributions of this thesis are threefold. First, it investigates the application of compositional programming to embedded domain-specific languages and object-oriented programming. Second, it presents our design and implementation of the CP compiler, which efficiently compiles CP to JavaScript. Third, it explores the extension of CP with union types for named and optional arguments. We anticipate that the design principles and implementation strategies proposed in this thesis will inform the development of future programming languages that prioritize compositionality.

In the following sections, we discuss some future directions that are worth exploring.

12.1 Formalizing the Document DSL

EXT, introduced in Section 5.4, is a powerful document DSL that supports general-purpose computation. However, we only give an informal description of EXT without formalizing its semantics. Recently, Crichton and Krishnamurthi [2024] proposed a core calculus for documents and provided a formal semantics for it. It would be interesting to investigate the relationship between our EXT and their document calculus.

Crichton and Krishnamurthi classified documents into eight levels based on the document domain and expressiveness. According to their taxonomy, EXT corresponds to the article template literal calculus $\mathcal{D}^{\mathsf{Article}}_{\mathsf{TLit}}$, meaning that the documents generated by EXT are annotated trees of strings (called "articles"), and documents are constructed of article literals with interpolation, similar to Scribble in Racket.

To better understand the document calculus, we restate the definition of templates in $\mathcal{D}^{\mathsf{Article}}_{\mathsf{TLit}}$ below:

$$\begin{aligned}\mathsf{Template}^{\mathsf{Article}}_{\mathsf{TLit}}\ t &::= [p^*] \\ \mathsf{TPart}^{\mathsf{Article}}_{\mathsf{TLit}}\ p &::= s \mid e \mid \textbf{node}\ (s, [(s, e)^*], t)\end{aligned}$$

A template t is a list of template parts p, which can be a string literal s, an interpolated expression e, or an attributed tagged tree **node**. For example, a section can be represented

as **node** ("section", [("id", 0)], ["title"]), where an attribute called "id" is attached. The syntax of EXT is almost the same: an interpolated expression is denoted by \(e), and an attributed tagged tree is like \Section{id=0}[title].

Computation in EXT. Relating EXT to $\mathcal{D}^{\text{Article}}_{\text{TLit}}$ is not particularly interesting because the ability to perform general-purpose computation in EXT is not captured in $\mathcal{D}^{\text{Article}}_{\text{TLit}}$. $\mathcal{D}^{\text{Article}}_{\text{TProg}}$ extends $\mathcal{D}^{\text{Article}}_{\text{TLit}}$ with template programs, namely three new template parts: **set** $x = e$, **if** e **then** t_1 **else** t_2, and **foreach** e $\{x.\, t\}$. Although EXT does not support these constructs directly, we can simulate the latter two by implementing customized commands. In other words, **if**-expressions and **foreach**-loops can be encoded as special **node**s. A simplified code snippet for such encodings is shown below:

```
-- \If(cond)[then][else]
If (b: Bool) (t: Element) (e: Element) = trait => if b then t else e;
-- \Foreach@Type(array)(func)
Foreach A (xs: [A]) (f: A -> Element) = trait =>
  letrec go (i: Int): Element = if i == #xs then Str ""
                                else Comp (f (xs!!i)) (go (i+1)) in go 0;
...
fruits = [ "apple"; "banana"; "cherry" ];  -- \Set is not encodable yet
...
`\Itemize[\Foreach@String(fruits)(\(x: String) ->
    `\If(x == "apple")[][\Item[\(x)]]`
)]`
```

However, encoding the **set**-statement is much more challenging, because variable binding is difficult, if not impossible, to implement in the userland. Notwithstanding this minor mismatch, it is worth investigating how to build a formal correspondence between EXT and $\mathcal{D}^{\text{Article}}_{\text{TProg}}$ and show that EXT is as expressive as an article template program DSL.

12.2 Improving the CP Compiler

Side effects in top-like terms. Our formalization of λ^{+}_{i} follows the distributive subtyping rules proposed by Barendregt et al. [1983], which allows the top type $\top$ to be a

subtype of $\top \to \top$. Via transitivity and contravariance of functions, we can show that $\top$ is a subtype of any function type returning $\top$:

$$\dfrac{\top \;<:\; \top \to \top \qquad \dfrac{A <: \top \qquad \top <: \top}{\top \to \top \;<:\; A \to \top}\ \textsc{S-Fun}}{\top \;<:\; A \to \top}\ \textsc{S-Trans}$$

This relation can be understood as a generalization of the distributive subtyping rule for function types, where $n = 0$:

$$(A \to B_1) \,\&\, (A \to B_2) \,\&\, \cdots \,\&\, (A \to B_n) \;<:\; A \to (B_1 \,\&\, B_2 \,\&\, \cdots \,\&\, B_n)$$

Since $\top$ represents the 0-ary intersection, the relation above is simplified to $\top \;<:\; A \to \top$. The other direction $A \to \top \;<:\; \top$ also holds because any type is a subtype of the maximal type $\top$. This means that $A \to \top$ is equivalent to $\top$ and is called *top-like* (i.e. $⌉A \to \top⌈$) in Figure 7.4.

In our elaboration rules, all top-like terms are treated as $\top$ and elaborated to an empty record (see rules Ela-Top, Ela-TopAbs, and Ela-TopRcd). Moreover, the coercive subtyping rule S-Top in coerce a target term to an empty record if B is top-like in $A <: B$. As a result, side effects in top-like terms are erased during elaboration, which is not desired in imperative languages. For example, $(\lambda r.\ r := 1) : \mathbf{Ref}\ \mathbb{Z} \to \top$ is elaborated to $\{\}$, and the original function is erased. So the reference r will never be updated when the coerced function is applied.

One obvious solution is not to erase the top-like terms during the elaboration. For example, we can elaborate the previous expression to $\{|\mathbf{Ref}\ \mathbb{Z} \to \top| \Mapsto \lambda r.\ \epsilon\}$, assuming $r := 1$ is elaborated to ϵ. We also need to revise the rule A-Top to call the function rather than just return an empty record. However, this change breaks the current definition of equivalent types (informally in Section 6.3 and formally in Figure 7.6) because different top-like terms can have different representations now. Coercions between top-like terms cannot be eliminated because, for example, $\{|\top| \Mapsto \dots\}$ is different from $\{|\mathbf{Ref}\ \mathbb{Z} \to \top| \Mapsto \dots\}$. An immediate follow-up question is how to correctly coerce $\lambda r.\ r := 1$ to type $\top$ or $\{\ell : \top\}$, both of which are supertypes of $\mathbf{Ref}\ \mathbb{Z} \to \top$.

A first try is to keep the term intact while only changing the type indices. For example, $\{|\top| \Mapsto \lambda r.\ \epsilon\}$ and $\{|\{\ell : \top\}| \Mapsto \lambda r.\ \epsilon\}$ can be the results of the above-mentioned coercions. However, how to project the latter term by label ℓ is not clear then. Of course, we can do nothing and still keep the term intact, but consider another source term **let** r =

ref 0 **in** $\{\ell = (r := 1)\} : \{\ell : \top\}$. This term also has type $\{\ell : \top\}$, but we expect projecting it by ℓ will assign 1 to r instead of doing nothing. Therefore, there is no trivial way to revise the rule P-Top (and the rule A-Top similarly).

As discussed briefly above, we believe that it is challenging to redesign the relevant rules to support side effects in top-like terms, so we leave it for future work.

Reconciliation between eagerness and laziness. As observed by Fan et al. [2022], trait instantiation may diverge in CP if a call-by-value evaluation strategy is used. Several years ago, a similar observation was made by Bruce et al. [1999] in the context of object encodings. Fan et al. fixed the divergence issue by switching to a call-by-name evaluation strategy completely, which is inefficient in practice. That is why we turn to a hybrid strategy: only self-annotated trait fields are lazily evaluated, as shown in Figure 8.3c, and other constructs are eagerly evaluated. Our hybrid approach is implemented in the CP compiler and proves to be effective in practice, but it has not been formalized or well studied. It is worthwhile to investigate formal semantics with the hybrid evaluation strategy and see how it can be applied to other languages.

First, let us review why the trait instantiation does not terminate with the call-by-value strategy. The example used for Figure 8.3c corresponds to the following code using the fixpoint operator, and it reduces as follows:

```
   fix this. { x = this.y; y = 48 }
↪ { x = (fix this. { x = this.y; y = 48 }).y; y = 48 }
↪ { x = { x = (fix this. { x = this.y; y = 48 }).y; y = 48 }.y; y = 48 }
↪ ...
```

The evaluation diverges because the variable `this` is evaluated repeatedly, despite the fact that only `this.y` is needed. Therefore, we can conclude that the key to laziness is: do not evaluate anything until it is needed.

The solution employed in the CP compiler uses smart getters in JavaScript, which are essentially adding *thunks* to certain record fields. A natural direction is to formalize the semantics using the call-by-push-value model [Levy 2012]. For example, the previous example can be rewritten to the following form:

```
   fix this. { x = thunk this.y; y = 48 }
↪ { x = thunk (fix this. { x = this.y; y = 48 }).y; y = 48 }
```

The evaluation terminates because the thunk is already a value of type $U(F\ \mathbb{Z})$, meaning a thunk that returns an integer. To obtain the integer value from the `x` field, we have to force the thunk, while the `y` field can be directly accessed. This difference can be detected statically by their types: `x` has type $U(F\ \mathbb{Z})$ while `y` has type $\mathbb{Z}$. Therefore, we probably

Destinations $\quad dst ::= \mathbf{nil} \mid z$

$\boxed{\Gamma; dst \vdash e \Leftrightarrow A \rightsquigarrow \mathtt{J} \mid z}$ *(Type-directed compilation)*

J-Abs

$$\frac{\Gamma, x : A; z \vdash e \Leftarrow B \rightsquigarrow \mathtt{J} \mid z \qquad \Gamma, x : A; \mathbf{nil} \vdash e \Leftarrow B \rightsquigarrow \mathtt{J} \mid y}{\Gamma; z \vdash \lambda x{:}A.\ e{:}B \Rightarrow A \to B \rightsquigarrow \texttt{z["func_|B|_dst"] = (x, z) => \{ J \};} \texttt{z["func_|B|_nodst"] = (x) => \{ J; return y; \};} \mid z}$$

$\boxed{\Gamma; dst \vdash x : A \bullet y : B \rightsquigarrow \mathtt{J} \mid z : C}$ *(Function application)*

JA-ArrowEquiv

$$\frac{A \fallingdotseq C}{\Gamma; z \vdash x : A \to B \bullet y : C \rightsquigarrow \texttt{x["func_|B|_dst"](y, z);} \mid z : B}$$

JA-ArrowNil

$$\frac{A \fallingdotseq C}{\Gamma; \mathbf{nil} \vdash x : A \to B \bullet y : C \rightsquigarrow \texttt{var z = x["func_|B|_nodst"](y);} \mid z : B}$$

Figure 12.1: A variant of destination-passing style without optional destinations.

need different projection rules for different types of fields. We leave further exploration of this topic for future work.

A variant of destination-passing style without optional destinations. In Section 8.5, we mention that there are three forms of destinations: z, $y?$, and **nil**, where $y?$ represents an optional destination in the context of a function body. An alternative approach that avoids the case of optional destinations is to compile each function twice: one with a destination and the other without (see J-Abs in Figure 12.1). Correspondingly, we need to change the rules for function application to choose the appropriate version of a function statically (see JA-ArrowEquiv and JA-ArrowNil in Figure 12.1).

Although this approach doubles the compilation time of a function body and duplicates every function definition in compiled code, it reduces the overhead of dynamic checks for optional destinations at run time. A quantitative analysis is needed to compare the two approaches in terms of time and space complexity. We leave this evaluation for future work.

Representation of type indices. As discussed in Section 6.5, we use a shorter representation for function types in our implementation, which only includes the return type. However, there are some rare corner cases where this design causes trouble. Here is an example:

```
type A = (Int → Int&Bool) → Int;
type B = (Int → Int&String) → Int;
f (g: Int → Int) = g 0;
f' = f : A&B;
f'' = f' : A;
f'' (\(x:Int) → x,true)
```

Although the types of components of a merge are guaranteed to be disjoint, there is no rule guaranteeing components of an intersection type to be disjoint. Therefore, we can duplicate the function by casting `f` to type `A&B`. Since `A` and `B` are not disjoint and have the same return type, they would share the same type index `"func_int"`. We could not distinguish them by type indices, and the field for `f:B` would override that for `f:A` in our implementation. At first sight, it somehow makes sense because they are essentially the same function so we do not need to keep two copies. However, the trouble here is that their parameters have different type indices: `"func_(bool&int)"` for `A`'s parameter and `"func_(int&string)"` for `B`'s. This subtle difference leads to slightly different compilation results for `f:A` and `f:B`. The code above would go wrong as we do not distinguish them and misuse `f:B` as `f:A`.

The fix is easy. Our formalization in Chapter 7 uses a more precise representation for function types: both the parameter and return types are included in the type indices. Nevertheless, the issue is rare, and we have never encountered such a case in benchmarks. For the considerations of performance and code size, we still use the shorter type indices that only include return types in our implementation. It is worthwhile to investigate the trade-off between soundness and performance in the representation of type indices. For example, we can use bit sets to represent intersection types.

12.3 Mixing Named and Positional Arguments

As shown in Table 10.1, CP advocate distinctness between named and positional arguments. An interesting direction worth exploring is to drop distinctness. By this means, we can support omitting keywords if the order of arguments is not changed from how they are defined. However, mixing positional and named arguments arbitrarily can lead to confusion. For example, consider the following pseudo-Python function:

```
def f(x=1, y, z): ...
f(y=2, 3, 4)
```

It is not clear which parameters `3` and `4` in the function call correspond to, though `f(x=3, y=2, z=4)` could be a plausible interpretation. In fact, Python syntactically rejects the code for two reasons:

1. The parameters without default values (`y` and `z`) should not follow a parameter with a default value (`x=1`) in the function definition.
2. Positional arguments (`3` and `4`) should not follow a keyword argument (`y=2`) in the call site.

These two restrictions are reasonable and largely improve the code clarity with named and positional arguments mixed. Nevertheless, these restrictions are not necessarily the best because they are too strict. For example, Scala enforces neither restriction for more flexibility. Although the previous confusing code is also rejected by Scala, the following code is valid:

```
def f(a: Int, b: Int, c: Int = 3, d: Int, e: Int) = ...
f(1, d=4, b=2, 5)  //= f(a=1, b=2, c=3, d=4, e=5)
```

Although this code is clear in general, its counterpart in Python is invalid because `d` and `e` cannot follow `c=3` in the function definition and `5` cannot follow `b=2` in the call site. In contrast, Scala lifts these restrictions and enforces a weaker one. Unfortunately, neither the official documentation[1] nor the language specification[2] can explain why the code above is valid in Scala. According to either description, the code should be rejected. Here we try to give an intuitive description that is inferred from our experimental observations:

> *For any named argument $p_i = e_i$, positional arguments to the right ($e_j, j > i$) should correspond to formal parameters to the right of p_i in the function definition. The same applies to the positional arguments to the left.*

No matter how the restriction is formulated, and whether it is stricter or weaker, this is an important first step for the design of the source language with named and positional arguments mixed. We can give semantics to the source language by translating it to λ_{iu}, which

[1] https://docs.scala-lang.org/tour/named-arguments.html: *"Once the arguments are not in parameter order, reading from left to right, then the rest of the arguments must be named."*

[2] https://www.scala-lang.org/files/archive/spec/3.4/06-expressions.html#named-and-default-arguments: *"For every named argument $p_i = e_i$ which appears left of a positional argument in the argument list $e_1 \dots e_m$, the argument position i coincides with the position of parameter p_i in the parameter list of the applied method."*

is similar to what happens in Scala. The translation should be purely syntax-directed. By this means, we obtain a more flexible language without changing the type-theoretic foundation at all.

Part VII

Appendices

A Non-Modular Dependencies in Tagless-Final Embeddings

```
{-# LANGUAGE DuplicateRecordFields, NamedFieldPuns, OverloadedRecordDot,
    RecordWildCards #-}
```

Modular Interpretations in Tagless-Final Embeddings

```
data Vector = Vector { x :: Double, y :: Double } deriving Show
```

A tagless-final embedding defines region constructors in a type class, instead of a closed algebraic data type:

```
class RegionHudak repr where
  circle    :: Double → repr
  outside   :: repr → repr
  union     :: repr → repr → repr
  intersect :: repr → repr → repr
  translate :: Vector → repr → repr
```

`Size` can be modularly defined as an instance of the type class since it has no dependency:

```
newtype Size = S { size :: Int }

instance RegionHudak Size where
  circle      _ = S 1
  outside     a = S $ a.size + 1
  union     a b = S $ a.size + b.size + 1
  intersect a b = S $ a.size + b.size + 1
  translate _ a = S $ a.size + 1
```

But how about `Text`? As a first try, we might write:

```
newtype Text = T { text :: String }

instance RegionHudak Text where
```

```
   circle  r = T { text = "circular region of radius " ++ show r }
-- outside a = T { text = "outside a region of size " ++ show a.size }
-- ......
```

We will get a type error concerning `a.size` if we uncomment the line above, because a has type `Text` and thus does not contain a field named `size`. So the problem is that, once we need operations that have some dependencies on other operations, we get into trouble! But programs with dependencies are common in practice. This is a serious limitation.

A Non-Modular Workaround for Dependencies

A simple workaround is to pack the two operations together and duplicate the code of size calculation. We have already described this approach for shallow embeddings, and the same workaround works for tagless-final embeddings:

```
data SizeAndText = ST { size :: Int, text :: String }

instance RegionHudak SizeAndText where
  circle  r = ST { size = 1, text = "circular region of radius " ++ show r }
  outside a = ST { size = a.size + 1, text = "outside a region of size "
                                          ++ show a.size }
  union a b = ST { size, text = "union of two regions of size "
                              ++ show size ++ " in total" }
    where size = a.size + b.size + 1
  intersect a b = ST { size, text = "intersection of two regions of size "
                                 ++ show size ++ " in total" }
    where size = a.size + b.size + 1
  translate v a = ST { size, text = "translated region of size " ++ show size }
    where size = a.size + 1
```

However, this is not modular since we have to duplicate code for size calculation. If we have another operation that depends on `size`, we have to repeat the same code even again. This workaround is an *anti-pattern*, which violates basic principles of software engineering.

Modular Language Constructs

Of course, new region constructors can be modularly added in a tagless-final embedding:

```
class RegionHofer repr where
  univ  :: repr
  empty :: repr
  scale :: Vector → repr → repr
```

We have to resort to the workaround again when we encounter mutual recursion:

```
data UE = UE { isUniv :: Bool, isEmpty :: Bool }

instance RegionHudak UE where
  circle      _ = UE { isUniv = False,     isEmpty = False }
  outside     a = UE { isUniv = a.isEmpty, isEmpty = a.isUniv }
  union     a b = UE { isUniv = a.isUniv || b.isUniv
                     , isEmpty = a.isEmpty && b.isEmpty }
  intersect a b = UE { isUniv = a.isUniv && b.isUniv
                     , isEmpty = a.isEmpty || b.isEmpty }
  translate _ a = UE { isUniv = a.isUniv,  isEmpty = a.isEmpty }

instance RegionHofer UE where
  univ      = UE { isUniv = True,     isEmpty = False }
  empty     = UE { isUniv = False,    isEmpty = True }
  scale _ a = UE { isUniv = a.isUniv, isEmpty = a.isEmpty }
```

Use Case

We can use all constructors from `RegionHudak` and `RegionHofer` to create a region, as long as `UE` implements both type classes:

```
region :: UE
region = outside empty `union` circle 1

main :: IO ()
main = do putStrLn $ "Univ:  " ++ show region.isUniv
          putStrLn $ "Empty: " ++ show region.isEmpty
```

B Modular Dependencies in Tagless-Final Embeddings

```
{-# LANGUAGE ConstraintKinds, DataKinds, FlexibleInstances, GADTs,
    KindSignatures, MultiParamTypeClasses, RankNTypes, ScopedTypeVariables,
    TypeApplications, TypeOperators, UndecidableInstances #-}
```

Generic Definitions for Records

First of all, we implement a type-indexed record as a *typeful heterogeneous list* [Kiselyov et al. 2004]:

```
data Record :: [*] → * where
  Nil  :: Record '[]
  Cons :: a → Record as → Record (a ': as)
```

We also need a projection operation that finds an element by its type from the record:

```
class a `In` as where
  project :: Record as → a

instance {-# OVERLAPPING #-} a `In` (a ': as) where
  project (Cons x _) = x

instance {-# OVERLAPPING #-} a `In` as ⇒ a `In` (b ': as) where
  project (Cons _ xs) = project xs
```

Moreover, `All c s` is a data type whose term can be constructed only if all types in `s` implement the type class `c`:

```
data All c :: [*] → * where
  AllNil  :: All c '[]
  AllCons :: c a ⇒ All c as → All c (a ': as)
```

Region DSL Infrastructure

```
data Vector = Vector { x :: Double, y :: Double }
```

The same as before, we define region constructors in a type class:

```
class Region0 r where
  univ_   :: r
  empty_  :: r
  circle_ :: Double → r
```

Note that the constructors above are simpler because they do not have `r` in input positions. The encoding of other constructors is more ingenious because we need to consider how to inject dependencies. Here we employ `Record s` to encode the dependencies an interpretation relies on:

```
class Region0 r ⇒ Region1 s r where
  outside_   :: Record s → r
  union_     :: Record s → Record s → r
  intersect_ :: Record s → Record s → r
  translate_ :: Vector → Record s → r
  scale_     :: Vector → Record s → r
```

Furthermore, we create an auxiliary type class that constrains `s1` to satisfy the dependencies of all interpretations in `s2`:

```
class Region2 s1 s2 where
  modality :: All (Region1 s1) s2

instance Region2 s1 '[] where
  modality = AllNil

instance (Region1 s1 a, Region2 s1 as) ⇒ Region2 s1 (a ': as) where
  modality = AllCons modality
```

With the infrastructure above, we can define smart constructors for all kinds of regions. Each smart constructor returns a term of type `Record s` that composes all corresponding interpretations if `s` is self-contained (no interpretation in `s` has external dependencies):

```
univ :: forall s. Region2 s s ⇒ Record s
univ = univ' (modality @s @s)
  where univ' :: All (Region1 s1) s2 → Record s2
        univ' AllNil      = Nil
        univ' (AllCons m) = Cons univ_ (univ' m)
```

```
empty :: forall s. Region2 s s ⇒ Record s
empty = empty' (modality @s @s)
  where empty' :: All (Region1 s1) s2 → Record s2
        empty' AllNil      = Nil
        empty' (AllCons m) = Cons empty_ (empty' m)

circle :: forall s. Region2 s s ⇒ Double → Record s
circle = circle' (modality @s @s)
  where circle' :: All (Region1 s1) s2 → Double → Record s2
        circle' AllNil      _ = Nil
        circle' (AllCons m) r = Cons (circle_ r) (circle' m r)

outside :: forall s. Region2 s s ⇒ Record s → Record s
outside = outside' (modality @s @s)
  where outside' :: All (Region1 s1) s2 → Record s1 → Record s2
        outside' AllNil      _ = Nil
        outside' (AllCons m) a = Cons (outside_ a) (outside' m a)

union :: forall s. Region2 s s ⇒ Record s → Record s → Record s
union = union' (modality @s @s)
  where union' :: All (Region1 s1) s2 → Record s1 → Record s1 → Record s2
        union' AllNil      _ _ = Nil
        union' (AllCons m) a b = Cons (union_ a b) (union' m a b)

intersect :: forall s. Region2 s s ⇒ Record s → Record s → Record s
intersect = intersect' (modality @s @s)
  where intersect':: All (Region1 s1) s2 → Record s1 → Record s1 → Record s2
        intersect' AllNil      _ _ = Nil
        intersect' (AllCons m) a b = Cons (intersect_ a b) (intersect' m a b)

translate :: forall s. Region2 s s ⇒ Vector → Record s → Record s
translate = translate' (modality @s @s)
  where translate' :: All (Region1 s1) s2 → Vector → Record s1 → Record s2
        translate' AllNil      _ _ = Nil
        translate' (AllCons m) v a = Cons (translate_ v a) (translate' m v a)

scale :: forall s. Region2 s s ⇒ Vector → Record s → Record s
scale = scale' (modality @s @s)
  where scale' :: All (Region1 s1) s2 → Vector → Record s1 → Record s2
        scale' AllNil      _ _ = Nil
        scale' (AllCons m) v a = Cons (scale_ v a) (scale' m v a)
```

As shown above, there is a lot of boilerplate code for each language construct of the region DSL.

Tagless-Final Embeddings

Now we can write dependent interpretations in a tagless-final style. We take the more interesting example with mutual recursion for example:

```
newtype IsUniv  = U { isUniv  :: Bool }
newtype IsEmpty = E { isEmpty :: Bool }

instance Region0 IsUniv where
  circle_ _ = U False
  univ_     = U True
  empty_    = U False

instance (IsEmpty `In` s, IsUniv `In` s) ⇒ Region1 s IsUniv where
  outside_     a = U $ isEmpty (project a)
  union_     a b = U $ isUniv (project a) || isUniv (project b)
  intersect_ a b = U $ isUniv (project a) && isUniv (project b)
  translate_ _ a = U $ isUniv (project a)
  scale_     _ a = U $ isUniv (project a)

instance Region0 IsEmpty where
  circle_ _ = E False
  univ_     = E False
  empty_    = E True

instance (IsUniv `In` s, IsEmpty `In` s) ⇒ Region1 s IsEmpty where
  outside_     a = E $ isUniv (project a)
  union_     a b = E $ isEmpty (project a) && isEmpty (project b)
  intersect_ a b = E $ isEmpty (project a) || isEmpty (project b)
  translate_ _ a = E $ isEmpty (project a)
  scale_     _ a = E $ isEmpty (project a)
```

We can easily declare dependencies using type constraints, and explicit projections help us find appropriate interpretations in the type-indexed record.

Use Case

Since `'[IsUniv, IsEmpty]` is self-contained, we can use smart constructors to create a region:

```
region :: Record '[IsUniv, IsEmpty]
region = outside empty `union` circle 1

main :: IO ()
main = do let Cons u (Cons e Nil) = region
          putStrLn $ "Univ:  " ++ show (isUniv u)
          putStrLn $ "Empty: " ++ show (isEmpty e)
```

C Compilation Scheme from F_i^+ to JavaScript

The syntax of F_i^+ is defined below, where the main differences from λ_i^+ are the addition of parametric polymorphism and fixpoint expressions:

$$
\begin{array}{ll}
\text{Types} & A, B ::= \top \mid \bot \mid \mathbb{Z} \mid X \mid A \to B \mid \forall X{*}A.\ B \mid \{\ell : A\} \mid A \,\&\, B \\
\text{Expressions} & e ::= \{\} \mid n \mid x \mid \mathbf{fix}\, x{:}A.\ e \mid \lambda x{:}A.\ e{:}B \mid e_1\ e_2 \mid \Lambda X{*}A.\ e{:}B \mid e\,A \\
& \quad \mid \{\ell = e\} \mid e.\ell \mid e_1\ ,\ e_2 \mid e : A
\end{array}
$$

The compilation scheme we describe here directly generates JavaScript code instead of λ_r terms, which is closer to the actual implementation. We denote the generated JavaScript code by $\mathcal{J}$, which can be empty ($\varnothing$), concatenation of two pieces of code ($\mathcal{J}_1; \mathcal{J}_2$), or some predefined code that is listed in Figure C.1. There are some notations for type indices, which are actually implemented as strings in JavaScript (as discussed in Section 6.5). Destinations [Shaikhha et al. 2017] also play an important role in the compilation scheme, being part of the rules for type-directed compilation and distributive application. The key idea of destinations has been elaborated in Section 8.5.

$$
\begin{array}{ll}
\text{JavaScript code} & \mathcal{J} ::= \varnothing \mid \mathcal{J}_1; \mathcal{J}_2 \mid \mathsf{code} \\
\text{Type indices} & T ::= \mathbb{Z} \mid \overrightarrow{T} \mid T^{\forall} \mid \{\ell : T\} \mid T_1 \,\&\, T_2 \\
\text{Destinations} & dst ::= \mathbf{nil} \mid y? \mid z
\end{array}
$$

Type-Directed Compilation

Similarly to the elaboration in Chapter 7, the compilation process is type-directed. Besides the typing context Γ, there is also a destination variable dst that guides the code generation. A rule basically reads as: given a typing context Γ and a destination dst, the F_i^+ term e is checked/inferred to have type A and is compiled to variable z in JavaScript code $\mathcal{J}$. That is, after running $\mathcal{J}$, the result is stored in the JavaScript variable z.

```
/* J-Nil */
var z = {};
J;

/* J-Opt */
var z = y || {};
J;

/* J-Int */
z[T] = n;

/* J-IntOpt */
var z = n;
if (y) y[T] = n;

/* J-IntNil */
var z = n;

/* J-Var */
copy(z, x);

/* J-VarOpt */
if (y) copy(y, x);

/* J-Fix */
var x = z;
J;

/* J-Abs */
z[T] = (x, y) ⇒ {
  J;  return y0;
};

/* J-TAbs */
z[T] = (X, y) ⇒ {
  J;  return y0;
};

/* J-Rcd */
z.__defineGetter__(T, () ⇒ {
  J;
  delete this[T];
  return this[T] = y;
});

/* J-Def */
export var x = {};
J1;  J2;

/* JA-Nil */
var z = {};
J;

/* JA-Opt */
var z = y || {};
J;

/* JA-Arrow */
var y0 = {};
J1;  J2;

/* JA-ArrowEquiv */
x[T](y, z);

/* JA-ArrowOpt */
var z = x[T](y, z0);

/* JA-ArrowNil */
var z = x[T](y);

/* JA-All */
x[T](Ts, z);

/* JA-AllOpt */
var z = x[T](Ts, y);

/* JA-All */
var z = x[T](Ts);

/* JP-RcdEq */
var z = x[T];

/* JS0-Int */
J;
y = y[T];

/* JS0-Var */
J;
if (primitive(X)) y = y[T];

/* JS-Equiv */
copy(y, x);

/* JS-Bot */
y[T] = null;

/* JS-Int */
y[T] = x;

/* JS-IntAnd */
y[T] = x[T];

/* JS-Var */
copy(y, x);

/* JS-Arrow */
y[T2] = (x1, y2) ⇒ {
  var y1 = {};  J1;
  var x2 = x[T1](y1);
  y2 = y2 || {};
  J2;  return y2;
};

/* JS-All */
y[T2] = (X, y0) ⇒ {
  var x0 = x[T1](X);
  y0 = y0 || {};
  J;  return y0;
};

/* JS-Rcd */
y.__defineGetter__(T2, () ⇒ {
  var x0 = x[T1];
  var y0 = {};  J;
  delete this[T];
  return this[T] = y0;
});

/* JS-Split */
var y1 = {}; // if y1 != z
var y2 = {}; // if y2 != z
J1;  J2;  J3;

/* JM-Arrow */
z[T] = (p, y) ⇒ {
  y = y || {};
  var y1 = {}; // if y1 != y
  var y2 = {}; // if y2 != y
  x1[T1](p, y1);
  x2[T2](p, y2);
  J;  return y;
};

/* JM-All */
z[T] = (X, y) ⇒ {
  y = y || {};
  var y1 = {}; // if y1 != y
  var y2 = {}; // if y2 != y
  x1[T1](X, y1);
  x2[T2](X, y2);
  J;  return y;
};

/* JM-Rcd */
z.__defineGetter__(T, () ⇒ {
  var y = {};
  var y1 = {}; // if y1 != y
  var y2 = {}; // if y2 != y
  copy(y1, x1[T1]);
  copy(y2, x2[T2]);
  J;
  delete this[T];
  return this[T] = y;
});
```

Figure C.1: Predefined JavaScript code.

Rules J-Int and J-Var have three variants for different destinations, while rules J-App and J-TApp only have one version each but delegate to three variants of application for different destinations, which helps to generate more optimized JavaScript code. Examples illustrating variants of rule J-Var and rule JA-ArrowEquiv (via rule J-App) has been explained in Section 8.5. Rules J-IntOpt and J-IntNil are designed for the optimization of boxing/unboxing (see Section 8.4). Other rules assume that the destination is present and generate code accordingly. Rule J-Nil serves as the bridge from empty destinations to non-empty ones, while rule J-Opt is for optional ones.

$\boxed{\Gamma; dst \vdash e \Leftrightarrow A \rightsquigarrow \mathcal{J} \mid z}$ *(Type-directed compilation)*

J-Nil
$$\frac{\Gamma; z \vdash e \Leftrightarrow A \rightsquigarrow \mathcal{J} \mid z}{\Gamma; \mathbf{nil} \vdash e \Leftrightarrow A \rightsquigarrow \mathsf{code} \mid z}$$

J-Opt
$$\frac{\Gamma; z \vdash e \Leftrightarrow A \rightsquigarrow \mathcal{J} \mid z}{\Gamma; y? \vdash e \Leftrightarrow A \rightsquigarrow \mathsf{code} \mid z}$$

J-Top
$$\frac{}{\Gamma; z \vdash \{\} \Rightarrow \top \rightsquigarrow \varnothing \mid z}$$

J-Int
$$\frac{T = |\mathbb{Z}|}{\Gamma; z \vdash n \Rightarrow \mathbb{Z} \rightsquigarrow \mathsf{code} \mid z}$$

J-IntOpt
$$\frac{T = |\mathbb{Z}|}{\Gamma; y? \vdash n \Rightarrow \mathbb{Z} \rightsquigarrow \mathsf{code} \mid z}$$

J-IntNil
$$\frac{}{\Gamma; \mathbf{nil} \vdash n \Rightarrow \mathbb{Z} \rightsquigarrow \mathsf{code} \mid z}$$

J-Var
$$\frac{x : A \in \Gamma}{\Gamma; z \vdash x \Rightarrow A \rightsquigarrow \mathsf{code} \mid z}$$

J-VarOpt
$$\frac{x : A \in \Gamma}{\Gamma; y? \vdash x \Rightarrow A \rightsquigarrow \mathsf{code} \mid x}$$

J-VarNil
$$\frac{x : A \in \Gamma}{\Gamma; \mathbf{nil} \vdash x \Rightarrow A \rightsquigarrow \varnothing \mid x}$$

J-Fix
$$\frac{\Gamma, x : A; z \vdash e \Leftarrow A \rightsquigarrow \mathcal{J} \mid z}{\Gamma; z \vdash \mathbf{fix}\, x{:}A.\ e \Rightarrow A \rightsquigarrow \mathsf{code} \mid z}$$

J-TopAbs
$$\frac{\rceil B \lceil}{\Gamma; z \vdash \lambda x{:}A.\ e{:}B \Rightarrow A \to B \rightsquigarrow \varnothing \mid z}$$

J-Abs
$$\frac{T = |\overrightarrow{B}| \qquad \Gamma, x : A; y? \vdash e \Leftarrow B \rightsquigarrow \mathcal{J} \mid y_0}{\Gamma; z \vdash \lambda x{:}A.\ e{:}B \Rightarrow A \to B \rightsquigarrow \mathsf{code} \mid z}$$

J-App
$$\frac{\Gamma; \mathbf{nil} \vdash e_1 \Rightarrow A \rightsquigarrow \mathcal{J}_1 \mid x \qquad \Gamma; \mathbf{nil} \vdash e_2 \Rightarrow B \rightsquigarrow \mathcal{J}_2 \mid y \qquad \Gamma; dst \vdash x : A \bullet y : B \rightsquigarrow \mathcal{J}_3 \mid z : C}{\Gamma; dst \vdash e_1\, e_2 \Rightarrow C \rightsquigarrow \mathcal{J}_1; \mathcal{J}_2; \mathcal{J}_3 \mid z}$$

J-TopTAbs
$$\frac{\rceil B \lceil}{\Gamma; z \vdash \Lambda X{*}A.\ e{:}B \Rightarrow \forall X{*}A.\ B \rightsquigarrow \varnothing \mid z}$$

J-TAbs
$$\frac{T = |B|^{\forall} \qquad \Gamma, X * A; y? \vdash e \Leftarrow B \rightsquigarrow \mathcal{J}_2 \mid y_0}{\Gamma; z \vdash \Lambda X{*}A.\ e{:}B \Rightarrow \forall X{*}A.\ B \rightsquigarrow \mathsf{code} \mid z}$$

J-TApp
$$\frac{\Gamma; \mathbf{nil} \vdash e \Rightarrow B \rightsquigarrow \mathcal{J}_1 \mid y \qquad \Gamma; dst \vdash y : B \bullet A \rightsquigarrow \mathcal{J}_2 \mid z : C}{\Gamma; dst \vdash e\, A \Rightarrow C \rightsquigarrow \mathcal{J}_1; \mathcal{J}_2 \mid z}$$

J-TopRcd
$$\frac{\Gamma \vdash e \Rightarrow A \qquad \rceil A \lceil}{\Gamma; z \vdash \{\ell = e\} \Rightarrow \{\ell : A\} \rightsquigarrow \varnothing \mid z}$$

J-Rcd
$$\frac{T = \{\ell : |A|\} \qquad \Gamma; \mathbf{nil} \vdash e \Rightarrow A \rightsquigarrow \mathcal{J} \mid y}{\Gamma; z \vdash \{\ell = e\} \Rightarrow \{\ell : A\} \rightsquigarrow \mathsf{code} \mid z}$$

J-Proj
$$\frac{\Gamma; \mathbf{nil} \vdash e \Rightarrow A \rightsquigarrow \mathcal{J}_1 \mid y \qquad y : A \bullet \{\ell\} \rightsquigarrow \mathcal{J}_2 \mid z : B}{\Gamma; z \vdash e.\ell \Rightarrow B \rightsquigarrow \mathcal{J}_1; \mathcal{J}_2 \mid z}$$

J-Merge
$$\frac{\Gamma; z \vdash e_1 \Rightarrow A \rightsquigarrow \mathcal{J}_1 \mid z \qquad \Gamma; z \vdash e_2 \Rightarrow B \rightsquigarrow \mathcal{J}_2 \mid z \qquad \Gamma \vdash A * B}{\Gamma; z \vdash e_1 \,,\, e_2 \Rightarrow A \,\&\, B \rightsquigarrow \mathcal{J}_1; \mathcal{J}_2 \mid z}$$

J-Anno
$$\frac{\Gamma; dst \vdash e \Leftarrow A \rightsquigarrow \mathcal{J} \mid z}{\Gamma; dst \vdash e : A \Rightarrow A \rightsquigarrow \mathcal{J} \mid z}$$

J-Def
$$\frac{\Gamma; x \vdash e_1 \Rightarrow A \rightsquigarrow \mathcal{J}_1 \mid x \qquad \Gamma, x : A; z \vdash e_2 \Rightarrow B \rightsquigarrow \mathcal{J}_2 \mid z}{\Gamma; z \vdash x = e1; e_2 \Rightarrow B \rightsquigarrow \mathsf{code} \mid z}$$

J-Sub
$$\frac{\Gamma; \mathbf{nil} \vdash e \Rightarrow A \rightsquigarrow \mathcal{J}_1 \mid x \qquad x : A <: y : B \rightsquigarrow \mathcal{J}_2}{\Gamma; y \vdash e \Leftarrow B \rightsquigarrow \mathcal{J}_1; \mathcal{J}_2 \mid y}$$

J-SubEquiv
$$\frac{A \eqsim B \qquad \Gamma; dst \vdash e \Rightarrow A \rightsquigarrow \mathcal{J} \mid z}{\Gamma; dst \vdash e \Leftarrow B \rightsquigarrow \mathcal{J} \mid z}$$

Distributive Application and Projection

We have mentioned that function applications (and record projections) have to be specially handled because of distributive subtyping in F_{i}^+. To put it simply, we need to additionally consider the cases where functions (and records) have intersection types or top-like types. Similarly to previous rules, destinations also guide the code generation. A rule for function applications basically reads as: given a typing context Γ and a destination dst, applying the compiled function in x of type A to the compiled argument p yields variable z of type B in JavaScript code $\mathcal{J}$. Depending on whether the function is λ- or Λ-bound, the argument p can be either a value $(y : C)$ or a type (C).

$\boxed{\Gamma; dst \vdash x : A \bullet p \rightsquigarrow \mathcal{J} \mid z : B}$ *(Distributive application)*

JA-Nil
$$\frac{\Gamma; z \vdash x : A \bullet p \rightsquigarrow \mathcal{J} \mid z : B}{\Gamma; \mathbf{nil} \vdash x : A \bullet p \rightsquigarrow \mathsf{code} \mid z : B}$$

JA-Opt
$$\frac{\Gamma; z \vdash x : A \bullet p \rightsquigarrow \mathcal{J} \mid z : B}{\Gamma; y? \vdash x : A \bullet p \rightsquigarrow \mathsf{code} \mid z : B}$$

JA-Top
$$\frac{\rceil A \lceil}{\Gamma; z \vdash x : A \bullet p \rightsquigarrow \varnothing \mid z : \top}$$

JA-Arrow
$$\frac{T = |\overrightarrow{B}| \qquad y : C <: y_0 : A \rightsquigarrow \mathcal{J}_1 \qquad \Gamma; dst \vdash x : A \to B \bullet y_0 : A \rightsquigarrow \mathcal{J}_2 \mid z : B}{\Gamma; dst \vdash x : A \to B \bullet y : C \rightsquigarrow \mathsf{code} \mid z : B}$$

JA-ArrowEquiv
$$\frac{A \coloneqq C \qquad T = |\overrightarrow{B}|}{\Gamma; z \vdash x : A \to B \bullet y : C \rightsquigarrow \mathsf{code} \mid z : B}$$

JA-ArrowOpt
$$\frac{A \coloneqq C \qquad T = |\overrightarrow{B}|}{\Gamma; z_0? \vdash x : A \to B \bullet y : C \rightsquigarrow \mathsf{code} \mid z : B}$$

JA-ArrowNil
$$\frac{A \coloneqq C \qquad T = |\overrightarrow{B}|}{\Gamma; \mathbf{nil} \vdash x : A \to B \bullet y : C \rightsquigarrow \mathsf{code} \mid z : B}$$

JA-All
$$\frac{\Gamma \vdash A * C \qquad T = |B|^{\forall} \qquad Ts = \mathbf{itoa} \mid C \mid}{\Gamma; z \vdash x : \forall X{*}A.\ B \bullet C \rightsquigarrow \mathsf{code} \mid z : B[X \mapsto C]}$$

JA-AllOpt
$$\frac{\Gamma \vdash A * C \qquad T = |B|^{\forall} \qquad Ts = \mathbf{itoa} \mid C \mid}{\Gamma; y? \vdash x : \forall X{*}A.\ B \bullet C \rightsquigarrow \mathsf{code} \mid z : B[X \mapsto C]}$$

JA-AllNil
$$\frac{\Gamma \vdash A * C \qquad T = |B|^{\forall} \qquad Ts = \mathbf{itoa} \mid C \mid}{\Gamma; \mathbf{nil} \vdash x : \forall X{*}A.\ B \bullet C \rightsquigarrow \mathsf{code} \mid z : B[X \mapsto C]}$$

JA-And
$$\frac{\Gamma; z \vdash x : A \bullet p \rightsquigarrow \mathcal{J}_1 \mid z : A' \qquad \Gamma; z \vdash x : B \bullet p \rightsquigarrow \mathcal{J}_2 \mid z : B'}{\Gamma; z \vdash x : A \,\&\, B \bullet p \rightsquigarrow \mathcal{J}_1; \mathcal{J}_2 \mid z : A' \,\&\, B'}$$

As explained in Section 8.4, the rules for record projections are separated to reduce the number of coercions and improve the performance of generated JavaScript code, although

they were combined with the rules for function applications in the latest formalization of F_i^+ by Fan et al. [2022]. A rule for record projections basically reads as: projecting the compiled records in x of type A by label ℓ yields variable z of type B in JavaScript code $\mathcal{J}$.

$\boxed{x : A \bullet \{\ell\} \rightsquigarrow \mathcal{J} \mid z : B}$ *(Distributive projection)*

JP-Top
$$\frac{\rceil A \lceil}{x : A \bullet \{\ell\} \rightsquigarrow \varnothing \mid z : \top}$$

JP-RcdEq
$$\frac{T = \{\ell : |A|\}}{x : \{\ell : A\} \bullet \{\ell\} \rightsquigarrow \mathsf{code} \mid z : A}$$

JP-RcdNeq
$$\frac{\ell_1 \neq \ell_2 \qquad T = \{\ell : |A|\}}{x : \{\ell_1 : A\} \bullet \{\ell_2\} \rightsquigarrow \varnothing \mid z : \top}$$

JP-And
$$\frac{x : A \bullet \{\ell\} \rightsquigarrow \mathcal{J}_1 \mid z : A' \qquad x : B \bullet \{\ell\} \rightsquigarrow \mathcal{J}_2 \mid z : B'}{x : A \,\&\, B \bullet \{\ell\} \rightsquigarrow \mathcal{J}_1; \mathcal{J}_2 \mid z : A' \,\&\, B'}$$

Coercive Subtyping

In rule J-Sub, we check an expression of type A against its supertype B. Since the two types may correspond to compiled objects of different shapes, a coercion has to be inserted for each subtyping check. Such a form of subtyping is called *coercive* subtyping [Luo et al. 2013], in contrast to *inclusive* subtyping. A rule for coercive subtyping basically reads as: to upcast a compiled object x of type A to a compiled object y of type B, we need to insert a coercion in JavaScript code $\mathcal{J}$. The umbrella rule has three variants because of the optimization of boxing/unboxing (see Section 8.4).

$\boxed{x : A <: y : B \rightsquigarrow \mathcal{J}}$ *(Coercive subtyping)*

JS0-Sub
$$\frac{x : A <:^+ y : B \rightsquigarrow \mathcal{J}}{x : A <: y : B \rightsquigarrow \mathcal{J}}$$

JS0-Int
$$\frac{T = |\mathbb{Z}| \qquad x : A <:^+ y : \mathbb{Z} \rightsquigarrow \mathcal{J}}{x : A <: y : \mathbb{Z} \rightsquigarrow \mathsf{code}}$$

JS0-Var
$$\frac{T = |X| \qquad x : A <:^+ y : X \rightsquigarrow \mathcal{J}}{x : A <: y : X \rightsquigarrow \mathsf{code}}$$

As explained in Section 8.4, we add an extra flag to help optimize coercions for subtyping between equivalent types: $<:^+$ indicates that the optimization rule JS-Equiv can apply, while $<:^-$ not.

$\boxed{x : A <:^{\pm} y : B \rightsquigarrow \mathcal{J}}$ *(Coercive subtyping)*

JS-Equiv
$$\frac{A \fallingdotseq B}{x : A <:^{+} y : B \rightsquigarrow \mathsf{code}}$$

JS-Top
$$\frac{\lceil B \rceil}{x : A <:^{\pm} y : B \rightsquigarrow \varnothing}$$

JS-Bot
$$\frac{T = |A|}{x : \bot <:^{\pm} y : A \rightsquigarrow \mathsf{code}}$$

JS-Int
$$\frac{}{x : \mathbb{Z} <:^{+} y : \mathbb{Z} \rightsquigarrow \mathsf{code}}$$

JS-IntAnd
$$\frac{T = |\mathbb{Z}|}{x : \mathbb{Z} <:^{-} y : \mathbb{Z} \rightsquigarrow \mathsf{code}}$$

JS-Var
$$\frac{}{x : X <:^{\pm} y : X \rightsquigarrow \mathsf{code}}$$

JS-Arrow
$$\frac{\begin{array}{c} T_1 = |\overrightarrow{A_2}| \qquad T_2 = |\overrightarrow{B_2}| \\ Ts = \mathbf{itoa} \mid A_1 \mid \\ x_1 : B_1 <: y_1 : A_1 \rightsquigarrow \mathcal{J}_1 \\ x_2 : A_2 <: y_2 : B_2 \rightsquigarrow \mathcal{J}_2 \end{array}}{x : A_1 \to A_2 <:^{\pm} y : B_1 \to B_2 \rightsquigarrow \mathsf{code}}$$

JS-All
$$\frac{\begin{array}{c} T_1 = |A_2|^{\forall} \\ T_2 = |B_2|^{\forall} \qquad B_1 <: A_1 \\ x_0 : A_2 <: y_0 : B_2 \rightsquigarrow \mathcal{J} \end{array}}{x : \forall X{*}A_1.\ A_2 <:^{\pm} y : \forall X{*}B_1.\ B_2 \rightsquigarrow \mathsf{code}}$$

JS-Rcd
$$\frac{\begin{array}{c} T_1 = \{\ell : |A|\} \\ T_2 = \{\ell : |B|\} \\ x_0 : A <: y_0 : B \rightsquigarrow \mathcal{J} \end{array}}{x : \{\ell : A\} <:^{\pm} y : \{\ell : B\} \rightsquigarrow \mathsf{code}}$$

JS-Split
$$\frac{\begin{array}{c} B_1 \lhd B \rhd B_2 \\ y_1 : B_1 \rhd z : B \lhd y_2 : B_2 \rightsquigarrow \mathcal{J}_3 \\ x : A <:^{\pm} y_1 : B_1 \rightsquigarrow \mathcal{J}_1 \\ x : A <:^{\pm} y_2 : B_2 \rightsquigarrow \mathcal{J}_2 \end{array}}{x : A <:^{\pm} z : B \rightsquigarrow \mathsf{code}}$$

JS-AndL
$$\frac{x : A <:^{-} y : C \rightsquigarrow \mathcal{J}}{x : A \,\&\, B <:^{\pm} y : C \rightsquigarrow \mathcal{J}}$$

JS-AndR
$$\frac{x : B <:^{-} y : C \rightsquigarrow \mathcal{J}}{x : A \,\&\, B <:^{\pm} y : C \rightsquigarrow \mathcal{J}}$$

There are some auxiliary rules called *coercive merging* for rule JS-Split. These rules mean that if the splitting relation $A \lhd C \rhd B$ holds, we can merge the compiled objects x of type A and y of type B back into a single object z of type C in JavaScript code $\mathcal{J}$. Such merging is necessary after splitting the supertype distributively. For example, consider the following derivation of subtyping:

$$\frac{\begin{array}{c} \top \to \mathbf{Int} \lhd \top \to \mathbf{Int} \,\&\, \mathbf{Bool} \rhd \top \to \mathbf{Bool} \\ \top \to \mathbf{Int} \,\&\, \mathbf{String} \,\&\, \mathbf{Bool} <: \top \to \mathbf{Int} \qquad \top \to \mathbf{Int} \,\&\, \mathbf{String} \,\&\, \mathbf{Bool} <: \top \to \mathbf{Bool} \end{array}}{\top \to \mathbf{Int} \,\&\, \mathbf{String} \,\&\, \mathbf{Bool} <: \top \to \mathbf{Int} \,\&\, \mathbf{Bool}}$$

After splitting, the compiled object would have two fields with labels `"fun_int"` and `"fun_bool"`, but we expect only one field with label `"fun_(int&bool)"`. Rule JM-Arrow handles this case and merge the two fields back into one.

The notation may be misleading, but note that here only the variable name z is given (i.e. input) while variable names x and y are generated by the rules (i.e. output). This is because rule JM-And reuses the variable name z to also serve as x and y, which makes the caller perform more efficient in-place updates. Not to make it more confusing but we have to emphasize that the discussion is only about the variable *name* rather than the contents of the variable. Having a closer look at rule JS-Split will help to better understand our design. Below the judgment of coercive merging, the generated variable names (y_1 and y_2 in the case) are used to generate the coercions (in $\mathcal{J}_1$ and $\mathcal{J}_2$). The coercions are actually executed before the coercive merging (in $\mathcal{J}_3$) in generated JavaScript. To avoid y_1 and y_2 from being initialized more than once, some extra checks are performed when generating JavaScript code for rules JS-Split, JM-Arrow, JM-All, and JM-Rcd.

$\boxed{x : A \ \triangleright \ z : C \ \triangleleft \ y : B \rightsquigarrow \mathcal{J}}$ *(Coercive merging)*

JM-And

$$\frac{}{z : A \ \triangleright \ z : A \,\&\, B \ \triangleleft \ z : B \rightsquigarrow \varnothing}$$

JM-Arrow

$$\frac{\begin{array}{c} T = |\overrightarrow{B}| \\ T_1 = |\overrightarrow{B_1}| \qquad T_2 = |\overrightarrow{B_2}| \\ y_1 : B_1 \ \triangleright \ y : B \ \triangleleft \ y_2 : B_2 \rightsquigarrow \mathcal{J} \end{array}}{x_1 : A \to B_1 \ \triangleright \ z : A \to B \ \triangleleft \ x_2 : A \to B_2 \rightsquigarrow \mathsf{code}}$$

JM-All

$$\frac{\begin{array}{c} T = |B|^{\forall} \\ T_1 = |B_1|^{\forall} \qquad T_2 = |B_2|^{\forall} \\ y_1 : B_1 \ \triangleright \ y : B \ \triangleleft \ y_2 : B_2 \rightsquigarrow \mathcal{J} \end{array}}{x_1 : \forall X{*}A.\ B_1 \ \triangleright \ z : \forall X{*}A.\ B \ \triangleleft \ x_2 : \forall X{*}A.\ B_2 \rightsquigarrow \mathsf{code}}$$

JM-Rcd

$$\frac{\begin{array}{c} T = \{\ell : |A|\} \\ T_1 = \{\ell : |A_1|\} \\ T_2 = \{\ell : |A_2|\} \\ y_1 : A_1 \ \triangleright \ y : A \ \triangleleft \ y_2 : A_2 \rightsquigarrow \mathcal{J} \end{array}}{x_1 : \{\ell : A_1\} \ \triangleright \ z : \{\ell : A\} \ \triangleleft \ x_2 : \{\ell : A_2\} \rightsquigarrow \mathsf{code}}$$

D Type-Safety Issue in Ruby with Steep

```ruby
# rbs_inline: enabled

class App
  # @rbs host: String
  # @rbs port: Integer
  # @rbs debug: bool
  def run(host:, port:, debug: false)
    if debug != true && debug != false
      raise "Argument debug is not Boolean!"
    end
  end
end
app = App.new

# @type var args0: { host: String, port: Integer, debug: bool }
args0 = { host: "0.0.0.0", port: 80, debug: true }
app.run(**args0) # OK!

# @type var args1: { host: String, port: Integer }
args1 = { host: "0.0.0.0", port: 80 }
app.run(**args1) # OK!

# @type var args2: { host: String, port: Integer, debug: String }
args2 = { host: "0.0.0.0", port: 80, debug: "Oops!" }
app.run(**args2) # TypeError: ArgumentTypeMismatch!

class App
  # @rbs args: { host: String, port: Integer, debug: String }
  # @rbs return: { host: String, port: Integer }
  def f(args) = args
end
```

```
# @type var args3: { host: String, port: Integer }
args3 = app.f(args2)
app.run(**args3) # Type-checks in Steep, but has a runtime error:
                 # Argument debug is not Boolean!
```

E Type-Safety Issue in Ruby with Sorbet

```ruby
# typed: true
require "sorbet-runtime"

class App
  extend T::Sig

  sig {params(host: String, port: Integer, debug: T::Boolean).void}
  def run(host:, port:, debug: false)
    if debug != true && debug != false
      raise "Argument debug is not Boolean!"
    end
  end
end
app = App.new

args0 = { host: "0.0.0.0", port: 80, debug: true }
app.run(**args0) # OK!

args1 = { host: "0.0.0.0", port: 80 }
app.run(**args1) # OK!

args2 = { host: "0.0.0.0", port: 80, debug: "Oops!" }
app.run(**args2) # TypeError: Expected T::Boolean
                 # but found String("Oops!") for argument debug!
class App
  sig do
    params(args: { host: String, port: Integer, debug: String })
    .returns({ host: String, port: Integer })
  end
  def f(args) = args
end
```

```
args3 = app.f(args2)
app.run(**args3)
# This call passes Sorbet's static type checking,
# but the Sorbet runtime raises a dynamic type error in App#f:
# Return value expected type {host: String, port: Integer},
# but got type {host: String, port: Integer, debug: String}!
```

F Dynamic Semantics of λ_{iu}

Values	$v ::= \{\} \mid \mathbf{null} \mid n \mid \lambda x{:}A.\ e{:}B \mid \{\ell : A = v\} \mid v_1 \mathbin{,} v_2$
Evaluation contexts	$E ::= [\,\cdot\,] \mid E\,e \mid v\,E \mid \{\ell : A = E\} \mid E.\ell \mid E \mathbin{,} e \mid v \mathbin{,} E$
	$\mid \mathbf{switch}\,E\,\mathbf{as}\,x\,\mathbf{case}\,A \Rightarrow e_1\,\mathbf{case}\,B \Rightarrow e_2 \mid E : A$

$\boxed{e \longrightarrow e'}$ *(Small-step operational semantics)*

Step-App
$$\frac{v \longrightarrow_A v'}{(\lambda x{:}A.\ e{:}B)\ v \longrightarrow ([v'/x]\ e) : B}$$

Step-Prj
$$\frac{v \longrightarrow_A v'}{\{\ell : A = v\}.\ell \longrightarrow v'}$$

Step-AppDispatch
$$\frac{v_1 \mathbin{,} v_2 \bullet v \longrightarrow e'}{(v_1 \mathbin{,} v_2)\ v \longrightarrow e'}$$

Step-PrjDispatch
$$\frac{v_1 \mathbin{,} v_2 \bullet \ell \longrightarrow e'}{(v_1 \mathbin{,} v_2).\ell \longrightarrow e'}$$

Step-SwitchL
$$\frac{v \longrightarrow_A v'}{\mathbf{switch}\ v\ \mathbf{as}\ x\ \mathbf{case}\ A \Rightarrow e_1\ \mathbf{case}\ B \Rightarrow e_2 \longrightarrow [v'/x]\ e_1}$$

Step-Anno
$$\frac{v \longrightarrow_A v'}{v : A \longrightarrow v'}$$

Step-SwitchR
$$\frac{v \longrightarrow_B v'}{\mathbf{switch}\ v\ \mathbf{as}\ x\ \mathbf{case}\ A \Rightarrow e_1\ \mathbf{case}\ B \Rightarrow e_2 \longrightarrow [v'/x]\ e_2}$$

Step-Ctx
$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

Ordinary types $\quad A^\circ, B^\circ ::= \mathbf{Null} \mid \mathbb{Z} \mid A \to B \mid \{\ell : A\}$

$\boxed{v \longrightarrow_A v'}$ *(Type casting)*

Cast-Top
$$\frac{}{v \longrightarrow_\top \{\}}$$

Cast-Null
$$\frac{}{\mathbf{null} \longrightarrow_{\mathbf{Null}} \mathbf{null}}$$

Cast-Int
$$\frac{}{n \longrightarrow_{\mathbb{Z}} n}$$

Cast-Arrow
$$\frac{A_1 \to B_1 <: A_2 \to B_2}{\lambda x{:}A_1.\ e{:}B_1 \longrightarrow_{A_2 \to B_2} \lambda x{:}A_1.\ e{:}B_2}$$

Cast-Rcd
$$\frac{\{\ell : A\} <: \{\ell : B\}}{\{\ell : A = v\} \longrightarrow_{\{\ell:B\}} \{\ell : B = v\}}$$

Cast-Merge
$$\frac{v \longrightarrow_A v_1 \quad v \longrightarrow_B v_2}{v \longrightarrow_{A \wedge B} v_1 \,,\, v_2}$$

Cast-MergeL
$$\frac{v_1 \longrightarrow_{A^\circ} v_1'}{v_1 \,,\, v_2 \longrightarrow_{A^\circ} v_1'}$$

Cast-MergeR
$$\frac{v_2 \longrightarrow_{B^\circ} v_2'}{v_1 \,,\, v_2 \longrightarrow_{B^\circ} v_2'}$$

Cast-OrL
$$\frac{v \longrightarrow_A v'}{v \longrightarrow_{A \vee B} v'}$$

Cast-OrR
$$\frac{v \longrightarrow_B v'}{v \longrightarrow_{A \vee B} v'}$$

$\boxed{v_1 \,,\, v_2 \bullet v \longrightarrow e}$ *(Applicative dispatch)*

AD-Left
$$\frac{\lfloor v \rfloor <: \lfloor v_1 \rfloor^\lambda \quad \neg(\lfloor v \rfloor <: \lfloor v_2 \rfloor^\lambda)}{v_1 \,,\, v_2 \bullet v \longrightarrow v_1\ v}$$

AD-Right
$$\frac{\lfloor v \rfloor <: \lfloor v_2 \rfloor^\lambda \quad \neg(\lfloor v \rfloor <: \lfloor v_1 \rfloor^\lambda)}{v_1 \,,\, v_2 \bullet v \longrightarrow v_2\ v}$$

AD-Both
$$\frac{\lfloor v \rfloor <: \lfloor v_1 \rfloor^\lambda \quad \lfloor v \rfloor <: \lfloor v_2 \rfloor^\lambda}{v_1 \,,\, v_2 \bullet v \longrightarrow v_1\ v \,,\, v_2\ v}$$

$\boxed{v_1 \,,\, v_2 \bullet \ell \longrightarrow e}$ *(Projective dispatch)*

PD-Left
$$\frac{\lfloor v_1 \rfloor <: \{\ell : \top\} \quad \neg(\lfloor v_2 \rfloor <: \{\ell : \top\})}{v_1 \,,\, v_2 \bullet \ell \longrightarrow v_1.\ell}$$

PD-Right
$$\frac{\lfloor v_2 \rfloor <: \{\ell : \top\} \quad \neg(\lfloor v_1 \rfloor <: \{\ell : \top\})}{v_1 \,,\, v_2 \bullet \ell \longrightarrow v_2.\ell}$$

PD-Both
$$\frac{\lfloor v_1 \rfloor <: \{\ell : \top\} \quad \lfloor v_2 \rfloor <: \{\ell : \top\}}{v_1 \,,\, v_2 \bullet \ell \longrightarrow v_1.\ell \,,\, v_2.\ell}$$

$\boxed{\lfloor v \rfloor}$ Dynamic type

$$\lfloor \{\} \rfloor \equiv \top \qquad \lfloor \mathbf{null} \rfloor \equiv \mathbf{Null} \qquad \lfloor n \rfloor \equiv \mathbb{Z} \qquad \lfloor \lambda x{:}A.\ e{:}B \rfloor \equiv A \to B$$

$$\lfloor \{\ell : A = v\} \rfloor \equiv \{\ell : A\} \qquad \lfloor v_1 \, , \, v_2 \rfloor \equiv \lfloor v_1 \rfloor \wedge \lfloor v_2 \rfloor$$

$\boxed{\lfloor v \rfloor^{\lambda}}$ Input type

$$\lfloor \lambda x{:}A.\ e{:}B \rfloor^{\lambda} \equiv A \qquad \lfloor v_1 \, , \, v_2 \rfloor^{\lambda} \equiv \lfloor v_1 \rfloor^{\lambda} \vee \lfloor v_2 \rfloor^{\lambda} \qquad \lfloor \ldots \rfloor^{\lambda} \equiv \bot$$

Bibliography

Dean Allen. 2002. Textile Markup Language Documentation. https://textile-lang.com [cited on page 85]

João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. 2017. Disjoint Polymorphism. In *ESOP*. https://doi.org/10.1007/978-3-662-54434-1_1 [cited on pages 20, 25, 124, and 195]

Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013a. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer. https://doi.org/10.1007/978-3-642-37521-7 [cited on page 76]

Sven Apel and Christian Kästner. 2009. An Overview of Feature-Oriented Software Development. *J. Object Technol.* 8, 5 (2009). https://doi.org/10.5381/jot.2009.8.5.c5 [cited on page 143]

Sven Apel, Christian Kästner, and Christian Lengauer. 2013b. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Trans. Software Eng.* 39, 1 (2013). https://doi.org/10.1109/TSE.2011.120 [cited on page 144]

Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. 2005. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *GPCE*. https://doi.org/10.1007/11561347_10 [cited on page 144]

Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. 2006. An overview of CaesarJ. *Trans. Aspect Oriented Softw. Dev.* 1 (2006). https://doi.org/10.1007/11687061_5 [cited on pages 44 and 193]

Michael Ballantyne, Alexis King, and Matthias Felleisen. 2020. Macros for Domain-Specific Languages. In *OOPSLA*. https://doi.org/10.1145/3428297 [cited on pages 84 and 190]

Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. 1995. Intersection and Union Types: Syntax and Semantics. *Inf. Comput.* 119, 2 (1995). https://doi.org/10.1006/inco.1995.1086 [cited on pages 21, 22, 160, and 164]

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A Filter Lambda Model and the Completeness of Type Assignment. *J. Symb. Log.* 48, 4 (1983). https://doi.org/10.2307/2273659 [cited on pages 15, 17, 59, 123, and 200]

Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. 1996. A Monotonic Superclass Linearization for Dylan. In *OOPSLA*. https://doi.org/10.1145/236337.236343 [cited on page 191]

Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. 2004. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.* 30, 6 (2004). https://doi.org/10.1109/TSE.2004.23 [cited on page 144]

Xuan Bi and Bruno C. d. S. Oliveira. 2018. Typed First-Class Traits. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2018.9 [cited on pages 24, 25, and 73]

Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2018. The Essence of Nested Composition. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2018.22 [cited on pages 6, 9, 16, 25, 59, 76, 109, 110, 114, 124, 125, 129, and 195]

Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. Distributive Disjoint Polymorphism for Compositional Programming. In *ESOP*. https://doi.org/10.1007/978-3-030-17184-1_14 [cited on pages 25, 38, 109, 124, 145, and 195]

Dariusz Biernacki and Piotr Polesiuk. 2015. Logical Relations for Coherence of Effect Subtyping. In *TLCA*. https://doi.org/10.4230/LIPIcs.TLCA.2015.107 [cited on page 129]

Viviana Bono and Mariangiola Dezani-Ciancaglini. 2020. A Tale of Intersection Types. In *LICS*. https://doi.org/10.1145/3373718.3394733 [cited on page 13]

Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. 1992. Experience with Embedding Hardware Description Languages in HOL. In *TPCD*. [cited on pages 5, 63, and 65]

Gilad Bracha and William Cook. 1990. Mixin-based Inheritance. In *OOPSLA/ECOOP*. https://doi.org/10.1145/97945.97982 [cited on pages 23, 42, and 191]

Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. 2010. Modules as Objects in Newspeak. In *ECOOP*. https://doi.org/10.1007/978-3-642-14107-2_20 [cited on page 193]

Frederik P. Brooks, Jr. 1987. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer* 20, 4 (1987). https://doi.org/10.1109/MC.1987.1663532 [cited on page 3]

Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. 1995. On Binary Methods. *Theory Pract. Object Sys.* 1, 3 (1995). https://doi.org/10.1002/j.1096-9942.1995.tb00019.x [cited on page 47]

Kim B. Bruce. 2002. *Foundations of Object-Oriented Languages: Types and Semantics.* MIT Press. [cited on page 195]

Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. 1999. Comparing Object Encodings. *Inf. Comput.* 155, 1-2 (1999). https://doi.org/10.1006/inco.1999.2829 [cited on pages 195 and 202]

Luca Cardelli and John C. Mitchell. 1991. Operations on Records. *Math. Struct. Comput. Sci.* 1, 1 (1991). https://doi.org/10.1017/S0960129500000049 [cited on page 19]

Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.* 17, 4 (1985). https://doi.org/10.1145/6041.6042 [cited on page 20]

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *J. Funct. Program.* 19, 5 (2009). https://doi.org/10.1017/S0956796809007205 [cited on pages 64, 69, 80, and 189]

Giuseppe Castagna. 2023. Programming with Union, Intersection, and Negation Types. In *The French School of Programming*. Chapter 12. https://doi.org/10.1007/978-3-031-34518-0_12 [cited on pages 13 and 170]

Giuseppe Castagna, Guillaume Duboc, and José Valim. 2023. The Design Principles of the Elixir Type System. *Art Sci. Eng. Program.* 8, 2 (2023). https://doi.org/10.22152/programming-journal.org/2024/8/4 [cited on page 13]

Giuseppe Castagna, Victor Lanvin, Mickaël Laurent, and Kim Nguyen. 2022. Revisiting Occurrence Typing. *Sci. Comput. Program.* 217 (2022). https://doi.org/10.1016/j.scico.2022.102781 [cited on page 21]

Craig Chambers. 1992. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* Ph. D. Dissertation. Stanford University. [cited on page 194]

Arthur Charguéraud. 2012. The Locally Nameless Representation. *J. Autom. Reason.* 49, 3 (2012). https://doi.org/10.1007/s10817-011-9225-2 [cited on page 170]

Alonzo Church. 1941. *The Calculi of Lambda-Conversion.* Number 6 in Annals of Mathematics Studies. Princeton University Press. [cited on page 157]

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP*. https://doi.org/10.1145/351240.351266 [cited on page 63]

Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. 2007. Tribe: A Simple Virtual Class Calculus. In *AOSD*. https://doi.org/10.1145/1218563.1218578 [cited on pages 43, 44, 53, and 193]

William Cook and Jens Palsberg. 1989. A Denotational Semantics of Inheritance and Its Correctness. In *OOPSLA*. https://doi.org/10.1145/74878.74922 [cited on pages 55 and 195]

William R. Cook, Walter L. Hill, and Peter S. Canning. 1990. Inheritance Is Not Subtyping. In *POPL*. https://doi.org/10.1145/96709.96721 [cited on pages 42 and 47]

Brad Cox. 1995. No Silver Bullet Revisited. *American Programmer Journal* (1995). [cited on page 3]

Will Crichton and Shriram Krishnamurthi. 2024. A Core Calculus for Documents: Or, Lambda: The Ultimate Document. In *POPL*. https://doi.org/10.1145/3632865 [cited on page 199]

Gael Curry, Larry Baer, Daniel Lipkie, and Bruce Lee. 1982. Traits: An Approach to Multiple-Inheritance Subclassing. In *OIS*. https://doi.org/10.1145/800210.806468 [cited on page 22]

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (1991). https://doi.org/10.1145/115372.115320 [cited on page 133]

Rowan Davies and Frank Pfenning. 2000. Intersection Types and Computational Effects. In *ICFP*. https://doi.org/10.1145/351240.351259 [cited on page 16]

N. G. de Bruijn. 1972. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972). https://doi.org/10.1016/1385-7258(72)90034-0 [cited on page 134]

Mariangiola Dezani-Ciancaglini, Furio Honsell, and Yoko Motohama. 2005. Compositional Characterisations of Lambda-terms using Intersection Types. *Theor. Comput. Sci.* 340, 3 (2005). https://doi.org/10.1016/j.tcs.2005.03.011 [cited on page 14]

Karel Driesen, Urs Hölzle, and Jan Vitek. 1995. Message Dispatch on Pipelined Processors. In *ECOOP*. https://doi.org/10.1007/3-540-49538-X_13 [cited on page 195]

Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. 2006. Traits: A Mechanism for Fine-Grained Reuse. *ACM Trans. Program. Lang. Syst.* 28, 2 (2006). https://doi.org/10.1145/1119479.1119483 [cited on pages 4, 23, 43, 73, and 191]

Jana Dunfield. 2014. Elaborating Intersection and Union Types. *J. Funct. Program.* 24, 2–3 (2014). https://doi.org/10.1017/S0956796813000270 [cited on pages 13, 17, 18, 74, 101, 102, 150, 151, 160, 164, and 195]

Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5 (2021). https://doi.org/10.1145/3450952 [cited on pages 16 and 118]

ECMA. 1999. ECMA-262 3rd Edition, ECMAScript Language Specification. https://ecma-international.org/wp-content/uploads/ECMA-262_3rd_edition_december_1999.pdf [cited on page 153]

ECMA. 2015. ECMA-262 6th Edition, ECMAScript 2015 Language Specification. https://262.ecma-international.org/6.0/ [cited on page 153]

Sven Efftinge and Markus Völter. 2006. *oAW xText: A Framework for Textual DSLs.* Technical Report. [cited on page 190]

Burak Emir, Martin Odersky, and John Williams. 2007. Matching Objects with Patterns. In *ECOOP*. https://doi.org/10.1007/978-3-540-73589-2_14 [cited on page 80]

Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido

Wachsmuth, and Jimi van der Woning. 2015. Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future. *Comput. Lang. Syst. Struct.* 44 (2015). https://doi.org/10.1016/j.cl.2015.08.007 [cited on page 190]

Erik Ernst. 2000. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance.* Ph. D. Dissertation. Aarhus University. https://doi.org/10.7146/dpb.v29i549.7654 [cited on pages 44, 45, 144, 192, and 193]

Erik Ernst. 2001. Family Polymorphism. In *ECOOP*. https://doi.org/10.1007/3-540-45337-7_17 [cited on pages 6, 16, 43, 53, 76, and 193]

Erik Ernst. 2002. Safe Dynamic Multiple Inheritance. *Nordic J. Comput.* 9, 3 (2002). [cited on pages 44 and 193]

Erik Ernst. 2004. The Expression Problem, Scandinavian Style. In *MASPEGHI@ECOOP*. https://doi.org/10.1007/978-3-540-30554-5_11 [cited on pages 6, 16, 53, 144, and 193]

Erik Ernst, Klaus Ostermann, and William R. Cook. 2006. A Virtual Class Calculus. In *POPL*. https://doi.org/10.1145/1111037.1111062 [cited on pages 43, 53, 74, and 193]

Andong Fan, Xuejing Huang, Han Xu, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2022. Direct Foundations for Compositional Programming. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2022.18 [cited on pages 38, 109, 124, 136, 138, 145, 202, and 224]

Matthew Flatt. 2009. Low-Level Scribble API: Structures And Processing. https://docs.racket-lang.org/scribble/core.html [cited on pages 86 and 191]

Matthew Flatt and Eli Barzilay. 2009. Keyword and Optional Arguments in PLT Scheme. In *Scheme*. [cited on pages 7, 160, and 181]

Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. 2009. Scribble: Closing the Book on Ad Hoc Documentation Tools. In *ICFP*. https://doi.org/10.1145/1596550.1596569 [cited on pages 84 and 191]

Martin Fowler. 2005a. Fluent Interface. https://martinfowler.com/bliki/FluentInterface.html [cited on page 5]

Martin Fowler. 2005b. A Language Workbench in Action - MPS. https://martinfowler.com/articles/mpsAgree.html [cited on page 190]

Martin Fowler. 2005c. Language Workbenches: The Killer-App for Domain Specific Languages? https://martinfowler.com/articles/languageWorkbench.html [cited on page 190]

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic Subtyping: Dealing Set-Theoretically with Function, Union, Intersection, and Negation Types. *J. ACM* 55, 4 (2008). https://doi.org/10.1145/1391289.1391293 [cited on pages 13, 160, 164, and 165]

Erick Gallesio and Manuel Serrano. 2005. Skribe: A Functional Authoring Language. *J. Funct. Program.* 15, 5 (2005). https://doi.org/10.1017/S0956796805005575 [cited on page 191]

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. [cited on pages 96, 97, and 98]

Jacques Garrigue. 1994. *Label-Selective Lambda-Calculi and Transformation Calculi.* Ph. D. Dissertation. University of Tokyo. [cited on pages 160 and 178]

Jacques Garrigue. 1999. Objective Label Trilogy. http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/ [cited on page 178]

Jacques Garrigue. 2001. Labeled and optional arguments for Objective Caml. In *JSSST SIG-PPL.* [cited on pages 7, 160, and 179]

Benedict R. Gaster and Mark P. Jones. 1996. *A Polymorphic Type System for Extensible Records and Variants.* Technical Report NOTTCS-TR-96-3. University of Nottingham. [cited on page 196]

Debasish Ghosh. 2010. *DSLs in Action.* Manning. [cited on page 5]

Jeremy Gibbons and Nicolas Wu. 2014. Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl). In *ICFP.* https://doi.org/10.1145/2628136.2628138 [cited on page 63]

Andy Gill. 2009. Type-Safe Observable Sharing in Haskell. In *Haskell@ICFP.* https://doi.org/10.1145/1596638.1596653 [cited on page 68]

Neal Glew. 1999. Type Dispatch for Named Hierarchical Types. In *ICFP.* https://doi.org/10.1145/317636.317797 [cited on page 192]

David Goodger. 2002. reStructuredText: Markup Syntax and Parser Component of Docutils. https://docutils.sourceforge.io/rst.html [cited on page 85]

Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernando Toninho, Philip Wadler, and Nobuko Yoshida. 2020. Featherweight Go. In *OOPSLA*. https://doi.org/10.1145/3428217 [cited on page 135]

Seyed Hossein Haeri and Paul Keir. 2019. Solving the Expression Problem in C++, à la LMS. In *ICTAC*. https://doi.org/10.1007/978-3-030-32505-3_20 [cited on page 189]

Seyed Hossein Haeri and Sibylle Schupp. 2016. Expression Compatibility Problem. In *SCSS*. https://doi.org/10.29007/xlbn [cited on page 189]

Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. 2008. Dynamic Software Product Lines. *Computer* 41, 4 (2008). https://doi.org/10.1109/MC.2008.123 [cited on page 60]

J. Roger Hindley. 1983. Coppo-Dezani Types do not Correspond to Propositional Logic. *Theor. Comput. Sci.* 28, 1-2 (1983). https://doi.org/10.1016/0304-3975(83)90074-9 [cited on page 17]

Ralf Hinze. 2006. Generics for the Masses. *J. Funct. Program.* 16, 4–5 (2006). https://doi.org/10.1017/S0956796806006022 [cited on page 64]

Christian Hofer and Klaus Ostermann. 2010. Modular Domain-Specific Language Components in Scala. In *GPCE*. https://doi.org/10.1145/1868294.1868307 [cited on page 82]

Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic Embedding of DSLs. In *GPCE*. https://doi.org/10.1145/1449913.1449935 [cited on pages 64, 65, 69, 71, 74, 80, 82, and 189]

Xuejing Huang and Bruno C. d. S. Oliveira. 2021. Distributing Intersection and Union Types with Splits and Duality (Functional Pearl). In *ICFP*. https://doi.org/10.1145/3473594 [cited on page 22]

Xuejing Huang, Jinxu Zhao, and Bruno C. d. S. Oliveira. 2021. Taming the Merge Operator. *J. Funct. Program.* 31 (2021). https://doi.org/10.1017/S0956796821000186 [cited on pages 9, 38, 109, 114, 115, 123, 124, and 125]

Paul Hudak. 1998. Modular Domain Specific Languages and Tools. In *ICSR*. https://doi.org/10.1109/ICSR.1998.685738 [cited on pages 65, 73, and 152]

John Hughes. 1995. The Design of a Pretty-printing Library. In *AFP*. https://doi.org/10.1007/3-540-59451-5_3 [cited on page 63]

Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP*. https://doi.org/10.1007/BFb0057013 [cited on page 194]

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (2001). https://doi.org/10.1145/503502.503505 [cited on page 135]

Mark P. Jones. 1994. A Theory of Qualified Types. *Sci. Comput. Program.* 22, 3 (1994). https://doi.org/10.1016/0167-6423(94)00005-0 [cited on page 196]

Vojin Jovanović, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. 2014. Yin-Yang: Concealing the Deep Embedding of DSLs. In *GPCE*. https://doi.org/10.1145/2658761.2658771 [cited on pages 64, 67, 80, 83, and 189]

Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *OOPSLA*. https://doi.org/10.1145/1869459.1869497 [cited on page 190]

Oleg Kiselyov. 2010. Typed Tagless Final Interpreters. In *SSGIP*. https://doi.org/10.1007/978-3-642-32202-0_3 [cited on pages 64, 69, 71, 72, 80, 82, and 83]

Oleg Kiselyov. 2011. Implementing Explicit and Finding Implicit Sharing in Embedded DSLs. In *DSL*. https://doi.org/10.4204/EPTCS.66.11 [cited on pages 68 and 83]

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly Typed Heterogeneous Collections. In *Haskell@ICFP*. https://doi.org/10.1145/1017472.1017488 [cited on page 213]

Holger Krahn, Bernhard Rumpe, and Steven Völkel. 2010. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *Int. J. Softw. Tools Technol. Transf.* 12, 5 (2010). https://doi.org/10.1007/s10009-010-0142-1 [cited on page 190]

Anastasiya Kravchuk-Kirilyuk, Gary Feng, Jonas Iskander, Yizhou Zhang, and Nada Amin. 2024. Persimmon: Nested Family Polymorphism with Extensible Variant Types. In *OOPSLA*. https://doi.org/10.1145/3649836 [cited on page 194]

Shriram Krishnamurthi. 2001. *Linguistic Reuse*. Ph. D. Dissertation. Rice University. [cited on pages 63 and 67]

Christian Kästner, Sven Apel, and Klaus Ostermann. 2011. The Road to Feature Modularity?. In *FOSD@SPLC*. https://doi.org/10.1145/2019136.2019142 [cited on page 143]

Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin. 2015. A Theory of Tagged Objects. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2015.174 [cited on pages 4, 44, 45, and 192]

Jukka Lehtosalo et al. 2012. mypy: Optional static typing for Python. https://www.mypy-lang.org [cited on pages 160, 162, and 169]

Daan Leijen. 2004. *First-Class Labels for Extensible Rows*. Technical Report UU-CS-2004-051. Utrecht University. [cited on pages 108 and 135]

Daan Leijen. 2005. Extensible Records with Scoped Labels. In *TFP*. [cited on pages 109 and 196]

Daan Leijen and Erik Meijer. 2001. *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Technical Report UU-CS-2001-35. Utrecht University. [cited on pages 5 and 63]

Paul Blain Levy. 2012. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Semantics Structures in Computation, Vol. 2. Springer. https://doi.org/10.1007/978-94-007-0954-6 [cited on page 202]

Roberto E. Lopez-Herrejon, Don Batory, and William Cook. 2005. Evaluating Support for Features in Advanced Modularization Technologies. In *ECOOP*. https://doi.org/10.1007/11531142_8 [cited on page 60]

Zhaohui Luo, Sergei Soloviev, and Tao Xue. 2013. Coercive Subtyping: Theory and Implementation. *Inf. Comput.* 223 (2013). https://doi.org/10.1016/j.ic.2012.10.020 [cited on pages 7, 105, 167, and 224]

John MacFarlane. 2014. CommonMark Spec. https://spec.commonmark.org [cited on page 85]

Anil Madhavapeddy and Yaron Minsky. 2022. *Real World OCaml: Functional Programming for the Masses*. Cambridge University Press. https://doi.org/10.1017/9781009129220 [cited on page 180]

Ole Lehrmann Madsen and Birger Møller-Pedersen. 1989. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *OOPSLA*. https://doi.org/10.1145/74877.74919 [cited on pages 43, 53, 74, and 193]

Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. 1993. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley. [cited on pages 53 and 193]

Koar Marntirosian, Tom Schrijvers, Bruno C. d. S. Oliveira, and Georgios Karachalias. 2020. Resolution as Intersection Subtyping via Modus Ponens. In *OOPSLA*. https://doi.org/10.1145/3428274 [cited on page 56]

Soutaro Matsumoto et al. 2018. Steep: Gradual typing for Ruby. https://github.com/soutaro/steep [cited on pages 160, 163, and 169]

MediaWiki. 2003. Wikitext. https://www.mediawiki.org/wiki/Wikitext [cited on pages 85 and 191]

MediaWiki. 2011. Parsoid. https://www.mediawiki.org/wiki/Parsoid [cited on page 85]

Microsoft. 2012. TypeScript: JavaScript with syntax for types. https://www.typescriptlang.org [cited on pages 45, 169, and 192]

Leonid Mikhajlov and Emil Sekerinski. 1998. A Study of The Fragile Base Class Problem. In *ECOOP*. https://doi.org/10.1007/BFb0054099 [cited on page 45]

Abel Nieto, Yaoyu Zhao, Ondřej Lhoták, Angela Chang, and Justin Pu. 2020. Scala with Explicit Nulls. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2020.25 [cited on pages 21 and 164]

Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. 2004. Scalable Extensibility via Nested Inheritance. In *OOPSLA*. https://doi.org/10.1145/1028976.1028986 [cited on page 193]

Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. 2006. J&: Nested Intersection for Scalable Software Composition. In *OOPSLA*. https://doi.org/10.1145/1167473.1167476 [cited on pages 44 and 193]

Atsushi Ohori. 1995. A Polymorphic Record Calculus and Its Compilation. *ACM Trans. Program. Lang. Syst.* 17, 6 (1995). https://doi.org/10.1145/218570.218572 [cited on pages 196 and 197]

Bruno C. d. S. Oliveira. 2009. Modular Visitor Components: A Practical Solution to the Expression Families Problem. In *ECOOP*. https://doi.org/10.1007/978-3-642-03013-0_13 [cited on page 64]

Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses: Practical Extensibility with Object Algebras. In *ECOOP*. https://doi.org/10.1007/978-3-642-31057-7_2 [cited on pages 64, 80, and 189]

Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. 2006. Extensible and Modular Generics for the Masses. In *TFP*. [cited on page 64]

Bruno C. d. S. Oliveira and Andres Löh. 2013. Abstract Syntax Graphs for Domain Specific Languages. In *PEPM@POPL*. https://doi.org/10.1145/2426890.2426909 [cited on page 68]

Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint Intersection Types. In *ICFP*. https://doi.org/10.1145/2951913.2951945 [cited on pages 18, 19, 25, 101, 104, 150, and 195]

Edward Osinski. 2006. *A Polymorphic Type System and Compilation Scheme for Record Concatenation*. Ph. D. Dissertation. New York University. [cited on page 197]

Matt Parsons et al. 2021. persistent: Type-safe, multi-backend data serialization. https://hackage.haskell.org/package/persistent [cited on page 183]

Benjamin C. Pierce. 1991. *Programming with Intersection Types and Bounded Polymorphism*. Ph. D. Dissertation. Carnegie Mellon University. [cited on page 13]

Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press. [cited on page 195]

Benjamin C. Pierce et al. 2008. Programming Language Foundations. In *Software Foundations*. https://softwarefoundations.cis.upenn.edu/plf-current/ [cited on page 168]

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000). https://doi.org/10.1145/345099.345100 [cited on pages 16 and 118]

Christian Prehofer. 1997. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP*. https://doi.org/10.1007/BFb0053389 [cited on page 143]

Baber Rehman. 2023. *A Blend of Intersection Types and Union Types*. Ph. D. Dissertation. The University of Hong Kong. [cited on pages 9, 21, 22, 25, 160, 164, 165, 170, and 173]

John C. Reynolds. 1997. Design of the Programming Language Forsythe. In *Algol-like Languages*. Vol. 1. Chapter 8. https://doi.org/10.1007/978-1-4612-4118-8_9 [cited on pages 17, 22, 164, and 195]

Chris Richardson. 2018. *Microservices Patterns*. Manning. [cited on page 3]

Roblox. 2019. Luau: A fast, small, safe, gradually typed embeddable scripting language derived from Lua. https://luau.org [cited on page 169]

Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. 2012. Scala-Virtualized: Linguistic Reuse for Deep Embeddings. *High. Order Symb. Comput.* 25, 1 (2012). https://doi.org/10.1007/s10990-013-9096-9 [cited on pages 63, 64, 67, 80, and 189]

Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *GPCE*. https://doi.org/10.1145/1868294.1868314 [cited on page 189]

Lukas Rytz and Martin Odersky. 2010. Named and default arguments for polymorphic object-oriented languages: A discussion on the design implemented in the Scala language. In *SAC*. https://doi.org/10.1145/1774088.1774529 [cited on pages 7, 160, and 180]

Chieri Saito, Atsushi Igarashi, and Mirko Viroli. 2008. Lightweight Family Polymorphism. *J. Funct. Program.* 18, 3 (2008). https://doi.org/10.1017/S0956796807006405 [cited on pages 43, 53, 193, and 194]

Maximilian Scherr and Shigeru Chiba. 2014. Implicit Staging of EDSL Expressions: A Bridge between Shallow and Deep Embedding. In *ECOOP*. https://doi.org/10.1007/978-3-662-44202-9_16 [cited on pages 63, 83, and 189]

Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. 2017. Destination-Passing Style for Efficient Memory Management. In *FHPC@ICFP*. https://doi.org/10.1145/3122948.3122949 [cited on pages 108, 133, and 219]

Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. In *Haskell@ICFP*. https://doi.org/10.1145/581690.581691 [cited on pages 84 and 190]

Mark Shields and Erik Meijer. 2001. Type-Indexed Rows. In *POPL*. https://doi.org/10.1145/360204.360230 [cited on pages 104 and 196]

Michael Snoyman et al. 2011. warp: A fast, light-weight web server for WAI applications. https://hackage.haskell.org/package/warp [cited on page 182]

T. Stephen Strickland, Christos Dimoulas, Asumu Takikawa, and Matthias Felleisen. 2013. Contracts for First-Class Classes. *ACM Trans. Program. Lang. Syst.* 35, 3 (2013). https://doi.org/10.1145/2518189 [cited on page 192]

Stripe. 2019. Sorbet: A static type checker for Ruby. https://sorbet.org [cited on pages 160 and 163]

Josef Svenningsson and Emil Axelsson. 2015. Combining Deep and Shallow Embedding of Domain-Specific Languages. *Comput. Lang. Syst. Struct.* 44 (2015). https://doi.org/10.1016/j.cl.2015.07.003 [cited on pages 64, 65, 67, 80, and 189]

Wouter Swierstra. 2008. Data Types à la Carte (Functional Pearl). *J. Funct. Program.* 18, 4 (2008). https://doi.org/10.1017/S0956796808006758 [cited on page 189]

W. W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.* 32, 2 (1967). https://doi.org/10.2307/2271658 [cited on page 129]

Antero Taivalsaari. 1996. On the Notion of Inheritance. *ACM Comput. Surv.* 28, 3 (1996). https://doi.org/10.1145/243439.243441 [cited on page 194]

Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Gradual Typing for First-Class Classes. In *OOPSLA*. https://doi.org/10.1145/2384616.2384674 [cited on pages 4, 44, 45, and 192]

Sam Tobin-Hochstadt, Vincent St-Amour, Eric Dobson, and Asumu Takikawa. 2015. *The Typed Racket Guide*. https://docs.racket-lang.org/ts-guide/ [cited on page 182]

David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. 1991. Organizing Programs Without Classes. *LISP Symb. Comput.* 4, 3 (1991). https://doi.org/10.1007/BF01806107 [cited on page 23]

David Ungar and Randall B. Smith. 1987. Self: The Power of Simplicity. In *OOPSLA*. https://doi.org/10.1145/38765.38828 [cited on page 194]

Paweł Urzyczyn. 1999. The Emptiness Problem for Intersection Types. *J. Symb. Log.* 64, 3 (1999). https://doi.org/10.2307/2586625 [cited on page 14]

A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. 1975. Revised Report on the Algorithmic Language ALGOL 68. *Acta Informatica* 5, 1–3 (1975). https://doi.org/10.1007/BF00265077 [cited on page 165]

Philip Wadler. 1998. The Expression Problem. Posted on the Java Genericity mailing list. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt [cited on pages 4, 16, 53, 55, 64, 68, 82, and 189]

Philip Wadler. 2003. A Prettier Printer. In *The Fun of Programming*. Chapter 11. https://doi.org/10.1007/978-1-349-91518-7_11 [cited on page 63]

Philip Wadler. 2015. Propositions as Types. *Commun. ACM* 58, 12 (2015). https://doi.org/10.1145/2699407 [cited on page 15]

Andrew K. Wright. 1995. Simple imperative polymorphism. *LISP Symb. Comput.* 8, 4 (1995). https://doi.org/10.1007/BF01018828 [cited on page 16]

Ningning Xie, Bruno C. d. S. Oliveira, Xuan Bi, and Tom Schrijvers. 2020. Row and Bounded Polymorphism via Disjoint Polymorphism. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2020.27 [cited on pages 20, 25, 124, and 192]

Han Xu, Xuejing Huang, and Bruno C. d. S. Oliveira. 2023. Making a Type Difference: Subtraction on Intersection Types as Generalized Record Operations. In *POPL*. https://doi.org/10.1145/3571224 [cited on page 25]

Wenjia Ye, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2024. Imperative Compositional Programming: Type Sound Distributive Intersection Subtyping with References via Bidirectional Typing. In *OOPSLA*. https://doi.org/10.1145/3689782 [cited on pages 16, 17, and 25]

Matthias Zenger and Martin Odersky. 2005. Independently Extensible Solutions to the Expression Problem. In *FOOL@POPL*. [cited on page 55]

Weixin Zhang and Bruno C. d. S. Oliveira. 2017. EVF: An Extensible and Expressive Visitor Framework for Programming Language Reuse. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2017.29 [cited on pages 82 and 190]

Weixin Zhang and Bruno C. d. S. Oliveira. 2019. Shallow EDSLs and Object-Oriented Programming: Beyond Simple Compositionality. *Art Sci. Eng. Program.* 3, 3 (2019). https://doi.org/10.22152/programming-journal.org/2019/3/10 [cited on page 82]

Weixin Zhang and Bruno C. d. S. Oliveira. 2020. Castor: Programming with Extensible Generative Visitors. *Sci. Comput. Program.* 193 (2020). https://doi.org/10.1016/j.scico.2020.102449 [cited on pages 79, 82, and 190]

Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2021. Compositional Programming. *ACM Trans. Program. Lang. Syst.* 43, 3 (2021). https://doi.org/10.1145/3460228 [cited on pages 3, 4, 25, 38, 72, 83, 145, and 192]

Yizhou Zhang and Andrew C. Myers. 2017. Familia: Unifying Interfaces, Type Classes, and Family Polymorphism. In *OOPSLA*. https://doi.org/10.1145/3133894 [cited on pages 43, 53, and 193]

Yaoda Zhou, Bruno C. d. S. Oliveira, and Andong Fan. 2022. A Calculus with Recursive Types, Record Concatenation and Subtyping. In *APLAS*. https://doi.org/10.1007/978-3-031-21037-2_9 [cited on page 25]