Disjoint Intersection Types: Theory and Practice

by

Xuan Bi

(Temporary Binding for Examination Purposes)

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

August 2018

Abstract of thesis entitled

"Disjoint Intersection Types: Theory and Practice"

Submitted by

Xuan Bi

for the degree of Doctor of Philosophy
at The University of Hong Kong
in August 2018

Programs are hard to write. It was so 50 years ago at the time of the so-called software crisis; it still remains so nowadays. Over the years, we have learned—the hard way—that software should be constructed in a *modular* way, i.e., as a network of smaller and loosely connected modules. To facilitate writing modular code, researchers and software practitioners have developed new methodologies; new programming paradigms; stronger type systems; as well as better tooling support. Still, this is not enough to cope with today's needs. Several reasons have been raised for the lack of satisfactory solutions, but one that is constantly pointed out is the inadequacy of existing programming languages for the construction of modular software.

This thesis investigates *disjoint intersection types*, a variant of intersection types. Disjoint intersections types have great potential to serve as a foundation for powerful, flexible and yet type-safe and easy to reason OO languages, suitable for writing modular software. On the theoretical side, this thesis shows how to significantly increase the expressiveness of disjoint intersection types by adding support for *nested composition*, along with *parametric polymorphism*. Nested composition extends inheritance to work on a whole family of classes, enabling high degrees of modularity and code reuse. The combination with parametric polymorphism further improves the state-of-art encodings of extensible designs. However, the extension with nested composition and parametric polymorphism is challenging, for two different reasons. Firstly, the subtyping relation that supports these features is non-trivial. Secondly, the syntactic method used to prove coherence for previous calculi with disjoint intersection types is too inflexible. This thesis addresses the first problem by adapting and extending the well-known BCD subtyping with records, universal quantification and coercions. To address the second problem, this thesis proposes a powerful proof method to establish coherence. Hence, this thesis puts disjoint intersection types on a solid footing by thoroughly exploring their meta-theoretical properties.

On the pragmatic side, this thesis proposes a new language design with support for *first-class traits*, *dynamic inheritance* and nested composition. First-class traits allow two objects of statically unknown types to be composed without conflicts. Dynamic inheritance allows a class to inherit from other classes at *run time*. To address the challenges of typing first-class traits and detecting conflicts statically, this thesis shows how to model source language constructs for first-class traits and dynamic inheritance by leveraging the fine-grained expressiveness of disjoint intersection types. To illustrate the applicability of the new design, this thesis conducts a case study that modularizes programming language features using a highly modular form of VISITORS.

All the results and metatheory presented (unless otherwise indicated) in this thesis are mechanized in Coq in order to show the rigorousness of the approach. This thesis unifies ideas that are seemingly unrelated but powerful on their own—dynamic inheritance, first-class traits, nested composition—by a lightweight mechanism, thus providing new insights into software modularity and extensibility.

An abstract of exactly 500 words

To my beloved parents

Declaration

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

. . . .

Xuan Bi
August 2018

Acknowledgments

First and foremost, I would like to thank my advisor Dr. Bruno C. d. S. Oliveira for his continued support and mentorship throughout my studies. It is my honor to be one of his first disciples. Back in 2014 when I was in my final year at ZJU, I wasn't sure if I wanted to pursue a PhD, let alone a PhD in Programming Languages! My fellow classmates were all considering applying for PhD in AI, computer vision, etc; there were no courses on the theories of programming languages in the computer science curriculum in most (if not all) universities in Mainland China. My first encounter with this field was due to a Coursera course I happened to enroll in. Immediately I got fascinated and wanted to dig more. Looking back, it was such a great timing that Bruno just landed his faculty position as an assistant professor at HKU, and that I just finished my undergraduate studies. Our initial conversation went very well, and shortly I became his first PhD student. For the first few months, I was quite "on the right track". I recall a conversation with my colleague (Haoyuan, is that you?) that Bruno referred me as the "push model" because I always brought lots of questions in our weekly meetings and pushed him to give me more tasks. It was not long before I lost myself, had difficulty making any progress, and that the weekly meetings with Bruno soon became nightmares to me. Bruno, thank you for not giving me up, encouraging me to keep reading more papers, keep trying different ideas no matter how hopeless they may seem, and especially for not reminding me of the possibility that I may not be able to finish PhD on time! All in all, I could not wish for a better PhD advisor.

I am incredibly fortunate to have met Prof. Tom Schrijvers from KU Leuven, talked to him about my work and found a topic of mutual interest, which led to one of the key publications for this PhD thesis. I learned a lot from you! I would also like to thank my co-advisor Prof. T.H. Tse for his valuable suggestions and guidelines. He helped revise my papers and this thesis, continuously nudging me to make my writing clearer. His feedback is nothing short of inspiring, which I greatly enjoyed. During the years I spent at HKU, I had the opportunity to collaborate with a number of excellent fellow researchers, to name a few (in no particular order): Tomas Tauber, Zhiyuan Shi, Weixin Zhang, Huang Li, Yanpeng Yang, Ningning Xie, Yanlin Wang and Haoyuan Zhang. Especially my coauthor Ningning Xie, who, during my low days, offered me to collaborate on a project of her. This collaboration kept me busy for

a while, occupying the time I would otherwise spend on indulging in the gloomy outlook of my research life. Our fruitful collaboration is still going on as of this writing.

Last but not least, I would love to thank my family for being extremely supportive throughout my studies. I made it through thanks to all of you.

Contents

Contents

Contents

List of Figures

List of Figures

LIST OF TABLES

Part I

Prologue

1   Introduction

This thesis investigates disjoint intersection types—a variant of intersection types—focusing on its theoretical foundation and applications in the context of object-oriented programming. The results are three new typed calculi, the first two being core calculi and the last one a source calculus, combining the power of parametric polymorphism, a rich subtyping relation with the fine-grained expressiveness of disjoint intersection types. The key contribution of the thesis is that it unifies ideas that are seemingly unrelated but powerful on their own in object-oriented programming—dynamic inheritance, first-class traits, family polymorphism, extensible design patterns—by a single lightweight mechanism, thus providing new insights into software modularity and extensibility.

1.1   Motivation

Programs are hard to write. It was so 50 years ago at the time of the so-called *software crisis* [Naur and Randell 1969]; it still remains so nowadays, as the software we use daily is getting more and more complex and harder to maintain. Over the years, we have learned—the hard way—that software should be constructed in a *modular* way, i.e., as a network of smaller and loosely connected modules. To facilitate writing modular code, researchers and software practitioners have developed new methodologies; new programming paradigms; more expressive type systems; as well as better tooling support. Still, this is not enough to cope with today's needs. We will mention some limitations of existing mainstream languages on supporting modular programming shortly. But before that, let us identify the following well-established requirements for construction of modular software:

1. **Extensibility in both dimensions:** Extensions may require new variants to the datatype and new operations on the datatype.

2. **Strong static type safety:** Extensions cannot cause run-time type errors.

3. **No modification or duplication:** Existing code must not be modified nor duplicated.

4. **Separate compilation and type-checking:** Safety checks or compilation steps must not be deferred until linking or at run time.

5. **Independent extensibility:** Independently developed extensions should be composable so that they can be used jointly.

6. **Scalability:** Extension should be scalable. The amount of code needed should be proportional to the functionality added.

7. **Non-destructive extension:** The base system should still be available for use within the extended system.

The first four of these requirements correspond to Wadler's expression problem [Wadler 1998]. Zenger and Odersky [2005] added the 5th requirement. The last two requirements were proposed by Nystrom et al. [2006]. Scalability (6th) is often but not necessarily satisfied by separate compilation; it is important for extending large software. Non-destructive extension (7th) is an important requirement for legacy and performance reasons: it enables clients of the extended system to reuse code and data of the base system, allowing some interoperability between new functionality and legacy code. To address the requirements, many solutions have been proposed over the years (for example, see Oliveira [2009]; Oliveira and Cook [2012]; Swierstra [2008]; Wang and Oliveira [2016]; Zenger and Odersky [2005], to cite a few). They differ considerably in the language context with varying degrees of extensibility they offer, as well as the limitations they impose. Building on the previous solutions, this thesis proposes a lightweight language design that addresses all of these requirements.

Various programming language features support modular programming, with varying degrees of limitations. Functional languages, notably ML and OCaml, use module systems [MacQueen 1984] for flexible program construction. In particular, ML "functors"—which are functions over modules—allow one to develop and compile a module independently from the modules on which it depends. One functor can then be instantiated with multiple different modules during the execution of the program, enabling a powerful form of code reuse. One prominent weakness of ML modules (at least in current module implementations) is that they cannot be defined recursively, that is, mutually recursive functions and datatypes must be written in the same module, even though they may belong conceptually to different modules. Another limitation is that modules form a separate, higher-order functional language on top of the core and therefore ML is actually two languages in one. Moreover, module systems usually put more emphasis on supporting data abstraction, which adds considerable complexity to languages adopting module systems as the primary way to construct modular programs.

Object-oriented languages, on the other hand, use classes and inheritance as primary mechanisms to support code extensibility and reuse. Single inheritance found in mainstream

object-oriented languages (such as Java or C++) is perhaps the most well-known and well-studied mechanism. However, programmers have long realized that single inheritance is not flexible enough when it comes to structuring a class hierarchy: it works for small and simple extensions, but does not work well for larger software systems such as compilers and operating systems. There has been great interest in the past several years in mechanisms for providing greater extensibility in object-oriented languages. Of particular relevance to the subject of this thesis are three powerful linguistic mechanisms for software extensibility, providing increasing order of flexibility, as well as complexity: first-class classes [Takikawa et al. 2012], (first-class) mixins/traits [Bracha and Cook 1990; Schärli et al. 2003], and family polymorphism [Ernst 2001], as we will briefly discuss below.

1.1.1 First-Class Classes

Many dynamically typed languages (including JavaScript, Ruby, Python or Racket) support *first-class classes*. In those languages classes are first-class values and, like any other values, they can be passed as an argument, or returned from a function. Furthermore, first-class classes support *dynamic inheritance*: i.e., they can inherit from other classes at *run time*, enabling programmers to abstract over the inheritance hierarchy. In contrast, most statically typed languages do not have first-class classes and dynamic inheritance. While all statically typed object-oriented languages allow first-class *objects* (i.e., objects can be passed as arguments and returned as results), the same is not true for classes. Classes in languages such as Scala, Java or C++ are typically a second-class construct, and the inheritance hierarchy is *statically determined*.

Despite the popularity and expressive power of first-class classes in dynamically typed languages, there is surprisingly little work on typing of first-class classes. First-class classes and dynamic inheritance pose well-known difficulties in terms of typing. For example, in his thesis, Bracha [1992] comments several times on the difficulties of typing dynamic inheritance and first-class mixins, and proposes the restriction to static inheritance that is also common in statically typed languages. One of the motivations for this thesis is to propose a type discipline that can encode first-class classes. Moreover, we push this one step further: for the first time, this thesis shows how to encode *first-class traits* in a statically typed setting. But first things first, let us briefly explain what are traits, and the related concept "mixins".

1.1.2 (First-Class) Mixins and Traits

As remarked earlier, single inheritance is inadequate and inflexible to write large software. To overcome this limitation, multiple inheritance was proposed as a generalization of single

inheritance. However, multiple inheritance is renowned for being tricky to get right, largely because of the possible ambiguity issues that arise when conflicting features are inherited along different paths. Mixins [Bracha and Cook 1990] provide a simple mechanism for multiple inheritance without the ambiguity issue. A mixin is a class declaration parameterized over a superclass, able to extend a variety of parent classes with the same set of features. Mixins are composed *linearly*, and that methods defined in mixins appearing later override all the identically named methods of earlier mixins. Because of the linear order of composition, a class may not be able to access a member of a given super-mixin because the member is overridden by another mixin.

Traits [Ducasse et al. 2006; Schärli et al. 2003] are an alternative to mixins, and other models of multiple inheritance. The key difference between traits and mixins lies on the treatment of conflicts when composing multiple traits/mixins. Mixins adopt an *implicit* resolution strategy for conflicts, where the compiler automatically picks one implementation in case of conflicts. Traits, on the other hand, employ an *explicit* resolution strategy, where the compositions with conflicts are rejected, and the conflicts are explicitly resolved by programmers. Schärli et al. [2003] make a good case for the advantages of the trait model. In particular, traits avoid bugs that could arise from accidental conflicts that were not noticed by programmers. With the mixin model, such conflicts would be silently resolved, possibly resulting in unexpected run-time behavior due to a wrong method implementation choice. From a modularity point of view, the trait model also ensures that composition is *commutative*, thus the order of composition is irrelevant and does not affect the semantics. Bracha [1992] claims that "*The only modular solution is to treat the name collisions as errors...*", strengthening the case for the use of a trait model of composition. Otherwise, if the semantics is affected by the order of composition (like in the mixin model), global knowledge about the full inheritance graph is required to determine which implementations are chosen.

Mixins and traits as found in most statically typed languages/calculi are typically a second-class construct. Promoting mixins/traits to first-class citizens adds considerable expressiveness and flexibility in terms of software extensibility, as will be illustrated throughout this thesis. Only recently some progress has been made in statically typing first-class classes and dynamic inheritance [Lee et al. 2015; Takikawa et al. 2012]. However, prior to this thesis, we did not know how to model *typed first-class traits*. A key challenge, compared to models with first-class classes or mixins, is how to detect conflicts at compile time even when *not* knowing all components being composed statically. This is important because in the setting with dynamic inheritance and polymorphism, the possibility of accidental conflicts caused by programmers is extremely high.

1.1.3 Family Polymorphism and Nested Composition

The last mechanism—also the most powerful and complex one—is *family polymorphism*. In family polymorphism [Ernst 2001], inheritance is extended to work on a *whole family of classes*, rather than just a single class. This enables high degrees of modularity and code reuse, enabling simple solutions to hard programming language problems, like the expression problem [Wadler 1998]. An essential feature of family polymorphism is *nested composition* [Corradi et al. 2012; Ernst et al. 2006; Nystrom et al. 2004], which allows the automatic inheritance/composition of nested (or inner) classes when the enclosing classes are composed. Nystrom et al. [2004] call this *scalable extensibility*: "the ability to extend a body of code while writing new code proportional to the differences in functionality".

Not many mechanisms that support family polymorphism are available in existing mainstream languages. The Cake pattern [Odersky and Zenger 2005; Zenger and Odersky 2005] in Scala provides some form of family polymorphism. In order to model this modest form of family polymorphism, this pattern uses *virtual types*, *self types*, *path-dependent types* and *static mixin composition*. Even with so many sophisticated features, composition of families is still quite heavyweight and manual. The problem is due to the lack of *deep* mixin composition. Though solutions do exist [Oliveira et al. 2013], they usually require low-level type-unsafe programming features such as dynamic proxies, reflection or other meta-programming techniques. It is known that designing a sound type system that fully supports family polymorphism and nested composition is notoriously hard; there are only a few, quite sophisticated, research languages that manage this [Clarke et al. 2007; Ernst et al. 2006; Nystrom et al. 2004; Saito et al. 2007]. But those mechanisms usually focus on getting a relatively complex Java-like language with support for family polymorphism. Instead, one of the motivations for the work presented in this thesis is to come up with a *minimal* calculus that supports nested composition.

1.2 Our Proposed Solution

This thesis sets out to explore an alternative object-oriented language design that makes it easy and safe to extend and compose existing code on the language level. More specifically, we seek to rein in ideas that are seemingly unrelated but powerful in object-oriented programming—dynamic inheritance, first-class traits, family polymorphism—under a simple unifying mechanism: they are but different manifestations of a single underlying type discipline: *disjoint intersection types*. Through a serious of examples and rigorous analysis in this thesis, we hope to convince readers that disjoint intersection types are a feasible semantic tool to facilitate code reuse and modularity. In particular, for family polymorphism, we

show that the combination of the *merge operator* and a rich subtyping relation captures the essence of nested composition; for traits, we show that with the merge operator and disjoint intersection types, we are able to express *typed first-class traits*. Combined with parametric polymorphism, we can further express a very dynamic form of mixin-style compositions, enabling programmers to write highly modular and reusable software components.

So what are disjoint intersecting types? Here only highlights are given—more details are to be delivered in later chapters.

1.2.1 Disjoint Intersection Types

One recurring theme of this thesis are *intersection types* (usually written $A \& B$). Intersection types [Coppo and Dezani-Ciancaglini 1978; Pottinger 1980] have a long history in programming languages. They were originally introduced to characterize exactly all strongly normalizing lambda terms. Since then, starting with Reynolds's work on Forsythe [Reynolds 1988], they have also been employed to express useful programming language constructs, such as key aspects of multiple inheritance [Compagnoni and Pierce 1996] in object-oriented programming. One notable example is the Scala language [Odersky et al. 2004] and its DOT calculus [Amin et al. 2012], which make fundamental use of intersection types to express a class/trait that extends multiple other traits. Other modern programming languages, such as TypeScript [Microsoft 2012], Flow [Facebook 2014] and Ceylon [Redhat 2011], also adopt some form of intersection types.

Intersection types come in different varieties in the literature. A far more common form of intersection types are the so-called *refinement types* [Davies and Pfenning 2000; Dunfield and Pfenning 2003; Freeman and Pfenning 1991]. Refinement types restrict the formation of intersection types so that the two types in an intersection are refinements of the same simple (unrefined) type. Refinement intersections increase only the expressiveness of types (more precise properties can be checked) and not of terms. For this reason, Dunfield [2014] argues that refinement intersections are unsuited for encoding various useful language features that require the *merge operator* (or an equivalent term-level operator).

Unrestricted intersection types with a term-level "merge" operator as an *explicit* introduction form increase the expressiveness of the term language. This operator was introduced by Reynolds in Forsythe [Reynolds 1988] and adopted by a few other calculi [Alpuim et al. 2017; Castagna et al. 1992; Dunfield 2014; Oliveira et al. 2016]. Unfortunately, while the merge operator is powerful, it also makes it hard to get a *coherent* [Reynolds 1991] (or unambiguous) semantics. As a first approximation, a semantics is said to be coherent if a valid program has exactly *one* meaning (i.e., one value when run). Unrestricted uses of the merge operator can be ambiguous, leading to an incoherent semantics where the same program can evaluate

to different values. We shall come back to this form of intersection types in more details in Section 2.1.

Recently, Oliveira et al. [2016] proposed λ_i: a calculus with a variant of intersection types called *disjoint intersection types*. Calculi with disjoint intersection types also feature the merge operator, with restrictions that all expressions in a merge operator must have disjoint types and all well-formed intersections are also disjoint. With the disjointness restrictions, λ_i is proved to be coherent. As shown by Alpuim et al. [2017], calculi with disjoint intersection types are very expressive and can be used to statically type-check JavaScript-style programs using mixins. Yet they retain both type safety and coherence. While coherence may seem at first of mostly theoretical relevance, it turns out to be very relevant for object-oriented programming. As remarked earlier, a key issue for multiple inheritance is *ambiguity* caused by conflicting features inherited from different parents. Disjoint intersection types enforce that the types of parents are disjoint and thus that no conflicts exist. Any violations are statically detected and can be manually resolved by the programmer (for example by dropping one of the conflicting field/methods from one of the parents). This is very similar to existing trait models [Ducasse et al. 2006; Schärli et al. 2003]. Therefore in an object-oriented language modeled on top of disjoint intersection types, coherence implies that no ambiguity arises from multiple inheritance. This makes reasoning a lot simpler.

The main goal of this thesis is to significantly increase the expressiveness of disjoint intersection types by extending the simple forms of multiple inheritance/composition supported by previous work [Alpuim et al. 2017; Oliveira et al. 2016] into a more powerful form supporting nested composition and parametric polymorphism. On the pragmatic side, the outcome is a programming language with support for first-class traits, dynamic inheritance and nested composition. On the theoretical side, we put disjoint intersection types on a solid footing by thoroughly exploring their meta-theoretical properties.

1.3 Contributions

In this thesis, we present three new typed calculi, starting from a simple calculus with disjoint intersection types, then adding parametric polymorphism and finally ending up with a relatively sophisticated object-oriented language with support for first-class traits, dynamic inheritance and nested composition.

The λ_i^+ calculus.   The first one, named λ_i^+, is a simple calculus with records and disjoint intersection types that supports *nested composition*. The essential novelty of λ_i^+ is the adoption of the Barendregt, Coppo and Dezani (BCD) subtyping [Barendregt et al. 1983], which

includes distributivity rules between function/record types and intersection types. These rules are the delta that enables extending the simple forms of multiple inheritance/composition supported by previous work [Oliveira et al. 2016] into a more powerful form supporting nested composition. The incorporation of BCD subtyping is highly challenging for two different reasons. The first difficulty is how to preserve coherence. Although previous work on disjoint intersection types proposes a solution to coherence, the solution imposes several ad-hoc restrictions to guarantee the uniqueness of the elaboration and thus allows for a simple syntactic proof. However such restrictions make it hard or impossible to adapt the proof to extensions of the calculus with distributivity rules. To deal with coherence, we employ a more semantic proof method based on *logical relations* [Plotkin 1973; Statman 1985; Tait 1967] that we call the *canonicity* relation. The second difficulty is that BCD subtyping is non-algorithmic: the presence of a transitivity axiom in the rules makes it hard to get an algorithmic version. To address it, we adapt and extend Pierce's decision procedure [Pierce 1989] (closely related to BCD) with subtyping of records and coercions, and propose an equivalent algorithmic subtyping relation.

THE F_i^+ CALCULUS. The second one, named F_i^+, is a polymorphic calculus with disjoint intersection types. F_i^+ is essentially λ_i^+ enriched with a variant of parametric polymorphism called disjoint polymorphism [Alpuim et al. 2017]. The addition of parametric polymorphism greatly increases the expressiveness of λ_i^+: F_i^+ improves upon the finally tagless [Carette et al. 2009] and object algebra [Oliveira and Cook 2012] approaches and support advanced compositional designs, and enables the development of highly modular and reusable programs. F_i^+ is a generalization and extension of the F_i calculus [Alpuim et al. 2017], which proposed the idea of *disjoint polymorphism*. The main novelty of F_i^+ is a novel subtyping algorithm with distributivity laws. Distributivity plays a fundamental role in improving compositional designs, by enabling the automatic composition of multiple operations/interpretations. The main technical challenge is the proof of coherence as impredicativity makes it hard to develop a well-founded logical relation for coherence. However, by restricting the system to predicative instantiations only we are able to develop a suitable logical relation and show coherence. Besides coherence, we show several other important meta-theoretical results, such as type-safety, sound and complete algorithmic subtyping, and decidability of the type system. Remarkably, unlike $F_{<:}$'s *bounded polymorphism* [Cardelli and Wegner 1985], disjoint polymorphism in F_i^+ supports decidable type-checking.

TYPED FIRST-CLASS TRAITS. Lastly we present the design of SEDEL: a polymorphic language with *first-class traits*, supporting *parametric polymorphism*, *dynamic inheritance* as well

as conventional object-oriented features such as *dynamic dispatching* and *abstract methods*. Traits pose additional challenges when compared to models with first-class classes or mixins, because method conflicts should be detected *statically*, even in the presence of features such as dynamic inheritance and parametric polymorphism. To address the challenges of typing first-class traits and detecting conflicts statically, SEDEL adopts the well-established approach of elaborating high-level language constructs to a low-level core calculus. The main contribution of SEDEL is to show how to model source language constructs for first-class traits and dynamic inheritance. The work on λ_i^+ and F_i^+ aimed at core record calculi, and omits important features for practical object-oriented languages, including (dynamic) inheritance, dynamic dispatching and abstract methods. Based on Cook and Palsberg's work on the denotational semantics for inheritance [Cook and Palsberg 1989], we show how to design a source language that is elaborated into F_i^+. SEDEL's elaboration into F_i^+ is proved to be both type-safe and coherent. Coherence ensures that the semantics of SEDEL is unambiguous. In particular this property is useful to ensure that programs using traits are free of conflicts/ambiguities (even when the types of the object parts being composed are not fully statically know). We illustrate the applicability of SEDEL with several example uses for first-class traits. Furthermore, we conduct a case study that modularizes programming language interpreters using a highly modular form of VISITORS [Oliveira 2009; Torgersen 2004].

In summary the contributions of this thesis are:

- We present λ_i^+, a calculus with disjoint intersection types that features both *BCD-style subtyping* and *the merge operator*. This calculus is both type-safe and coherent, and supports *nested composition*.

- We present F_i^+, a polymorphic calculus with disjoint intersection types. F_i^+ is incorporated with a BCD-like subtyping relation extended with disjoint polymorphism. F_i^+ is both type-safe and coherent, and supports nested composition.

- We present SEDEL, an object-oriented language design that supports *typed first-class traits*, dynamic inheritance, as well as standard object-oriented features such as dynamic dispatching and abstract methods. We show how the semantics of SEDEL can be defined by elaboration into F_i^+.

- A more flexible notion of disjoint intersection types where only merges need to be checked for disjointness. This removes the need for enforcing disjointness for all well-formed types, making calculi with disjoint intersections more easily extensible.

- The canonicity relation: a powerful proof method for establishing coherence of calculi with disjoint intersection types, BCD-like subtyping and polymorphism.

- A comprehensive Coq mechanization of all metatheory, including type safety, coherence, algorithmic soundness and completeness, etc.[1] This has notably revealed several missing lemmas and oversights in Pierce's manual proof of BCD's algorithmic subtyping [Pierce 1989]. As a by-product, we obtain the first mechanically verified BCD-style subtyping algorithm with coercions.

- A full-blown implementation of SEDEL; it runs and type-checks all the examples in this thesis. We also conduct a case study, which shows that support for composition of object algebras [Oliveira and Cook 2012] is greatly improved in SEDEL. Using such improved design patterns we re-code the interpreters from an undergraduate textbook on programming languages [Cook 2013] in a modular way. The implementation, Coq formalization and all code presented in this thesis are available at `https://github.com/bixuanzju/phd-thesis-artifact`.

1.4 ORGANIZATION

We begin with some background in the main topics of this thesis in Chapter 2 in order to keep this thesis as self-contained as possible and also to put our methods and contributions into context. The structure of the technical content in the thesis is divided into three parts:

Part II: Chapters 3 and 4 formally define the type systems of λ_i^+ and F_i^+, respectively. We first give the syntax and semantics of the two calculi. The semantics is defined in two parts. The "target" languages are two standard type systems (simply-typed lambda calculus and System F, respectively) that do not have intersection types, the merge operator or subtyping. The "source" languages, defined by translation into the target languages, contain intersection types, the merge operator and subtyping. We then prove some basic properties such as type safety of the elaboration, soundness and completeness of the algorithmic subtyping, etc.

Part III: Chapters 5 and 6 explore the issue of coherence. In Chapter 5 we first propose a semantically founded definition of coherence. We then propose a proof method called the canonicity relation to establish coherence of λ_i^+. In Chapter 6 we follow the same technique in Chapter 5 but encounter a severe issue of impredicativity. We impose a predicativity restriction and adapt the canonicity relation to establish coherence of F_i^+.

---

[1] For convenience, whenever possible, definitions, lemmas and theorems have hyperlinks (click ☞) to their Coq counterparts. Also since F_i^+ completely subsumes λ_i^+, to save work, for λ_i^+ metatheory we provide cross references to the corresponding F_i^+ Coq definitions, instead.

Part IV: In Chapter 7 we present the syntax and semantics of SEDEL. In particular we show how to elaborate source-level constructs for first-class traits into expressions of F_i^+. In Chapter 8 we conduct a case study of modularizing programming language features using a highly modular form of VISITORS.

Chapter 9 reviews related work, Chapter 10 discusses future work and Chapter 11 concludes.

This thesis is largely based on two publications [Bi and Oliveira 2018; Bi et al. 2018] by the author and one draft [Bi et al. 2019], currently under review as of this writing. In comparison to the original publications, this thesis contains a more in-depth and consistent treatment of disjoint intersection types.

Chapters 3 and 5: Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2018. "The Essence of Nested Composition". In *European Conference on Object-Oriented Programming (ECOOP)*.

Chapters 4 and 6: Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. "Distributive Disjoint Polymorphism for Compositional Programming". Submitted to *European Symposium on Programming (ESOP)*.

Chapters 7 and 8: Xuan Bi and Bruno C. d. S. Oliveira. 2018. "Typed First-Class Traits". In *European Conference on Object-Oriented Programming (ECOOP)*.

This thesis assumes familiarity with basic knowledge of programming language theory and object-oriented programming. We recommend Pierce's excellent textbook on programming languages [Pierce 2002] for a general introduction.

2   Background

This chapter sets the stage for the three typed calculi we are going to present in later chapters by expanding upon some relevant topics from the introduction. In Section 2.1 we start with the traditional formulation of intersection types, followed by an introduction of the merge operator and the issue of coherence. We then review the λ_i calculus [Oliveira et al. 2016], the first calculus featuring disjoint intersection types, and briefly discuss how disjointness achieves coherence. In Section 2.2 we review the concepts of mixins and traits, their drawbacks and strengths. In Section 2.3 we introduce family polymorphism by means of presenting Ernst's elegant solution [Ernst 2004] to the expression problem. Section 2.4 reviews the denotational model of inheritance. Finally in Section 2.5 we give a simple introduction to program equivalence and logical relations.

2.1   Intersection Types

Intersection types in the pure lambda calculus were developed in the late 1970s by Coppo and Dezani-Ciancaglini [1978], and independently by Pottinger [1980]. The original motivation for intersection types was to devise a type-assignment system à la Curry [Curry and Feys 1958] that satisfies the following two properties:

1. The typing of a term should be preserved under β-conversion. (Under Curry's system, β-reduction preserves types but β-expansion, in general, does not.)

2. Every (strongly) normalizable term has a meaningful type. (We refer the reader to their paper for a precise definition of "meaningful".)

The idea of intersection types is remarkably simple and natural. From the set-theoretic perspective, an intersection type $A \& B$ for every pair of types A and B is thought of as containing all the elements of A that are also elements of B; from the type-theoretic point of view, $A \& B$ is a subtype of A, as well as of B; from the order-theoretic point of view, $A \& B$ is a greatest lower bound of A and B.[1] In the literature of object-oriented programming, intersection types are long known to model *multiple inheritance* [Compagnoni and Pierce 1996].

---

[1] Note that we say "a" rather than "the" because greatest lower bounds are not unique, but they are all "equal" to $A \& B$ in a sense that will be made precise in Section 10.1.

The intuition is that if we read the subtyping $A <: B$ as "A is a subclass of B", then $A \& B$ is a "name" of a class with all the common properties of A and B. Of course, this analog is not exact, in the same sense that inheritance is not subtyping [Cook et al. 1989]. But it is intuitively appealing, and as we will see, can be made more precise in a sufficiently enriched calculus based on intersection types. More pragmatically, many programming languages, such as Scala, TypeScript, Flow and Ceylon adopt some form of intersection types. For example, in Scala we can express a class A implements *both* B and C by the following declaration:

    class A **extends** B **with** C

where B **with** C denotes an intersection type between B and C.

INTERSECTION SUBTYPING.   Three subtyping rules capture the order-theoretic properties of intersection types:

S-INTERL

$$\overline{\phantom{A \& B <: A}}$$
$$A \& B <: A$$

S-INTERR

$$\overline{\phantom{A \& B <: B}}$$
$$A \& B <: B$$

S-INTER
$$\dfrac{C <: A \qquad C <: B}{C <: A \& B}$$

Two nice consequences follow:

1. The top type $\top$ can be regarded as the 0-ary form of intersection. It is a maximum element of the subtyping ordering, i.e., $A <: \top$ for every type A.

2. Multi-field record types can be thought of as an intersection of single-field record types. Thus, instead of

$$\{l_1 : A_1, \ldots, l_n : A_n\}$$

we can write

$$\{l_1 : A_1\} \& \ldots \& \{l_n : A_n\}$$

Note that the width and depth subtyping of records become a consequence of intersection subtyping.

DISTRIBUTIVITY RULES.   Two additional subtyping rules are usually found in the literature of intersection types (e.g., see Barendregt et al. [1983]; Reynolds [1988]). The first one captures the relation between intersections and function spaces, allowing intersections to distribute over the right-hand side of $\rightarrow$'s:

S-DISTARR

$$\overline{\phantom{(A_1 \rightarrow A_2) \& (A_1 \rightarrow A_3) <: A_1 \rightarrow A_2 \& A_3}}$$
$$(A_1 \rightarrow A_2) \& (A_1 \rightarrow A_3) <: A_1 \rightarrow A_2 \& A_3$$

Note that the other direction is also derivable (cf. Section 3.3). The second rule captures the relation between intersections and (singleton) records, allowing intersections to distribute over record labels:

S-ᴅɪsᴛRᴄᴅ

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

These two rules, though intuitively reasonable, will have a strong effect on both syntactic and semantic properties of the language. For example, rule S-ᴅɪsᴛAʀʀ implies that $\top <: A \to \top$ for any A; and rule S-ᴅɪsᴛRᴄᴅ implies that $\top <: \{l : \top\}$.

Iɴᴛᴇʀsᴇᴄᴛɪᴏɴ ᴛʏᴘɪɴɢ.   The introduction rule of intersection types says that a term E can be given type $A \,\&\, B$ if it inhabits both A and B:

ɪɴᴛᴇʀI

$$\frac{E : A_1 \qquad E : A_2}{E : A_1 \,\&\, A_2}$$

The corresponding elimination rule allows us to derive, given a derivation of $E : A_1 \,\&\, A_2$, that $E : A_1$ and $E : A_2$. But this already follows from intersection subtyping and the subsumption rule; so we need not to add the elimination rule explicitly to the calculus.

2.1.1 Tʜᴇ Mᴇʀɢᴇ Oᴘᴇʀᴀᴛᴏʀ

Intersection types were first incorporated into a practical programming language "Forsythe" by Reynolds [1988, 1997], who used them to encode features such as operator overloading by means of a "merge" operator $p_1, , p_2$—"a construction for intersecting or 'merging' meanings" [Reynolds 1997, p. 24]. (Reynolds actually used single comma p_1, p_2, but here we follow Dunfield by using double commas for consistency.) Reynolds demonstrated the power of the merge operator by developing an encoding of records by using intersection types; similar ideas also appear in Castagna et al. [1992]. The idea is to have only single-field records with the introduction form $\{l = E\}$ of type $\{l : A\}$ and the elimination form $E.l$ (record projection). Thus instead of

$$\{l_1 = E_1, \dots, l_n = E_n\}$$

we can write

$$\{l_1 = E_1\} \, , , \, \dots \, , , \, \{l_n = E_n\}$$

which plays nicely with the syntactic sugar of multi-field record types as an intersection of single-field record types.

Recently, Dunfield [2014] developed a method for elaborating intersections and unions into products and sums. Central to his system is a source-level *merge operator* $E_1 ,, E_2$, reminiscent of Forsythe [Reynolds 1997], which embodies several computationally distinct terms, and can be checked against various parts of an intersection type. In his system, the introduction form of intersection types is still rule INTERI, and two additional rules for the merge operator are added:

$$\text{MERGEL} \quad \frac{E_1 : A}{E_1 ,, E_2 : A} \qquad\qquad \text{MERGER} \quad \frac{E_2 : A}{E_1 ,, E_2 : A}$$

In other words, a merge expression can choose to type one subterm and ignore the other. In combination of rule INTERI, they allow to type check two distinct implementations E_1 and E_2 with completely different types A_1 and A_2 of the intersection. For example, let $E_1 = \lambda x.\, x$ and $E_2 = 1$, then the type $(\mathsf{Int} \to \mathsf{Int}) \,\&\, \mathsf{Int}$ is inhabited by $E_1 ,, E_2$:

$$\cfrac{\cfrac{E_1 : \mathsf{Int} \to \mathsf{Int}}{E_1 ,, E_2 : \mathsf{Int} \to \mathsf{Int}}\ \text{MERGEL} \qquad \cfrac{E_2 : \mathsf{Int}}{E_1 ,, E_2 : \mathsf{Int}}\ \text{MERGER}}{E_1 ,, E_2 : (\mathsf{Int} \to \mathsf{Int}) \,\&\, \mathsf{Int}}\ \text{INTERI}$$

Dunfield also developed elaboration typing rules which, given a source expression with unrestricted intersections and unions, type-check and transform the program into an ordinary λ-calculus term with sums and products. For example, the expression $(\lambda x.\, x) ,, 1$ elaborates to a pair $\langle \lambda x.\, x, 1 \rangle$. As usual, his system does not have explicit source-level intersection eliminations; elaboration puts all needed projections into the target program. For instance, the same expression $(\lambda x.\, x) ,, 1$, when checked against Int, elaborates to $\pi_2 \langle \lambda x.\, x, 1 \rangle$. The type-directed elaboration is elegant, type-safe, and serves as the original foundation for calculi with disjoint intersection types.

2.1.2 (In)Coherence

While Dunfield's system is simple and powerful, it has serious usability issues. More specifically, it lacks the theoretically and practically important property of *coherence* [Reynolds 1991]: the meaning of a target program depends on the choice of elaboration typing derivation. For example, the expression $1 ,, \mathsf{true}$ has type $\mathsf{Int} \,\&\, \mathsf{Bool}$. It can be used either as an

integer or a Boolean, the result is always clear (1 or true). However, when two types have overlapping components, it is not at all clear which value to pick. For example, the expression $1\,,,2$ (when checked against Int) could elaborate to either 1 or 2, depending on the particular choice in the implementation. Dunfield [2014] had a workaround by trying the left part 1 first. It is equally acceptable that one can opt to choose the right part 2. But neither is satisfying from a theoretical point of view.

To recover a coherent semantics, one could limit the merges according to their surface syntax, as Reynolds did in Forsythe. But as Dunfield [2014] pointed out, "crafting an appropriate syntactic restriction depends on details of the type system, which is not robust as the type system is extended". Another simple idea would be to require all types in an intersection be *distinct*. This works fine for simple types such as Int and Bool: Int & Bool is clearly a good intersection. But it is less clear as to what constituents a "good" (read unambiguous) intersection type in general. A moment of thoughts leads to the following principle: good intersection types are defined in terms of the subtyping relation. After all, it is the subtyping relation that defines the behavior of intersection types. A first attempt would be to require that two types *A* and *B* can form an intersection if both types are *not* subtype of each other. At first glance, this seems to be a reasonable definition because it rules out the problematic merge $1\,,,2$. However, it is still not enough. Consider the following expression (taken from Oliveira et al. [2016]):

$$(1\,,,\,``c") \,,, (2\,,,\,\mathsf{true})$$

The first component $(1\,,,\,``c")$ has type Int & String and the second component $(2\,,,\,\mathsf{true})$ has type Int & Bool. It is clear that neither of the two is a subtype of the other. However, extracting an integer from the above expression is ambiguous (1 or 2).

When moving to richer types, it is even less clear how to deal with, for example, intersections of higher-order functions. Consider the following intersection types (again taken from Oliveira et al. [2016]):

1. $(\mathsf{Int} \to \mathsf{Int}) \,\&\, (\mathsf{String} \to \mathsf{String})$

2. $(\mathsf{String} \to \mathsf{Int}) \,\&\, (\mathsf{String} \to \mathsf{String})$

3. $(\mathsf{Int} \to \mathsf{String}) \,\&\, (\mathsf{String} \to \mathsf{String})$

We can ask which of those intersection types are qualified as good. It seems reasonable to expect the first one is good, since both the domain and range types are different. But the other two are not that obvious to see. Clearly a formal notion of well-behaved intersection types are called for!

The issue of coherence is addressed (with some compromises) by Oliveira et al. [2016] with the notion of *disjointness*, as we will discuss next.

2.1.3 Disjoint Intersection Types

Disjoint intersection types, first introduced in the λ_i calculus [Oliveira et al. 2016] provide a remedy for the coherence problem, by imposing restrictions on the uses of merges and on the formation of intersection types. The syntax of λ_i is shown below:

$$\text{Types} \quad A ::= \mathsf{Int} \mid A_1 \rightarrow A_2 \mid A_1 \, \& \, A_2$$
$$\text{Terms} \quad E ::= i \mid x \mid \lambda x. E \mid E_1 \, E_2 \mid E_1 \,,, E_2 \mid E : A$$

Its full (bidirectional) type system is shown in Fig. 2.1. (It is probably best to skim this figure on first reading. We will explain and contrast this figure in later chapters.) Central to their system is the notion of *disjointness*. As a first approximation, for two types A and B to be disjoint (written $A * B$), they must not have any sub-components sharing the same type. In a type system without $\top$, this can be ensured by the following specification:

Definition 1 (Simple disjointness). $A * B \triangleq \not\exists C. \, A <: C \land B <: C$

The disjointness judgment appears in the well-formedness of intersection types (rule WF-AND) and the typing rule of merges (rule TI-MERGE). Rule WF-AND—the well-formedness of intersections—enforces that only disjoint types can form an intersection type: so $\mathsf{Int} \, \& \, \mathsf{Bool}$ is well-formed but $\mathsf{Int} \, \& \, \mathsf{Int}$ is not. Rule TI-MERGE—the typing rule for merges—prevents problematic merges such as $1 \,,, 2$ (because Int and Int are not disjoint), while accepting unambiguous merges such as $1 \,,, \mathsf{true}$.

Remark. Note that the introduction form for disjoint intersection types (rule TI-MERGE) is not as expressive as rule INTERI. For instance, rule INTERI entails the following derivation:

$$\frac{\lambda x. x : \mathsf{Int} \rightarrow \mathsf{Int} \qquad \lambda x. x : \mathsf{Char} \rightarrow \mathsf{Char}}{\lambda x.x : (\mathsf{Int} \rightarrow \mathsf{Int}) \, \& \, (\mathsf{Char} \rightarrow \mathsf{Char})}$$

which is impossible to express in λ_i.

To ensure that subtyping produces unique coercions, λ_i employs the notion of *ordinary types* [Davies and Pfenning 2000]—those that are not intersection types—and use the judgment "A ordinary" in rules SI-ANDL and SI-ANDR. Ordinary types and disjointness are sufficient to ensure a coherent semantics of a type system without $\top$.

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

Si-int

$$\overline{\mathsf{Int} <: \mathsf{Int} \rightsquigarrow \lambda x.\, x}$$

Si-top

$$\overline{A <: \top \rightsquigarrow \lambda x.\, \langle\rangle}$$

Si-arr

$$\frac{B_1 <: A_1 \rightsquigarrow e_1 \qquad A_2 <: B_2 \rightsquigarrow e_2}{A_1 \to A_2 <: B_1 \to B_2 \rightsquigarrow \lambda f.\, \lambda x.\, e_2\,(f\,(e_1\,x))}$$

Si-and

$$\frac{A_1 <: A_2 \rightsquigarrow e_1 \qquad A_1 <: A_3 \rightsquigarrow e_2}{A_1 <: A_2 \,\&\, A_3 \rightsquigarrow \lambda x.\, \langle e_1\,x, e_2\,x \rangle}$$

Si-andL

$$\frac{A_1 <: A_3 \rightsquigarrow e \qquad A_3 \text{ ordinary}}{A_1 \,\&\, A_2 <: A_3 \rightsquigarrow \lambda x.\, [\![A_3]\!]_{e\,(\pi_1\,x)}}$$

Si-andR

$$\frac{A_2 <: A_3 \rightsquigarrow e \qquad A_3 \text{ ordinary}}{A_1 \,\&\, A_2 <: A_3 \rightsquigarrow \lambda x.\, [\![A_3]\!]_{e\,(\pi_2\,x)}}$$

$\boxed{\Gamma \vdash A}$ *(Well-formedness of types)*

wf-int

$$\overline{\Gamma \vdash \mathsf{Int}}$$

wf-top

$$\overline{\Gamma \vdash \top}$$

wf-and

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B \qquad A * B}{\Gamma \vdash A \,\&\, B}$$

wf-arr

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B}$$

$\boxed{\Gamma \vdash E \Rightarrow A}$ *(Inference)*

Ti-top

$$\overline{\Gamma \vdash \top \Rightarrow \top \rightsquigarrow \langle\rangle}$$

Ti-lit

$$\overline{\Gamma \vdash i \Rightarrow \mathsf{Int} \rightsquigarrow i}$$

Ti-var

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A \rightsquigarrow x}$$

Ti-app

$$\frac{\Gamma \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e_1 \qquad \Gamma \vdash E_2 \Leftarrow A_1 \rightsquigarrow e_2}{\Gamma \vdash E_1\,E_2 \Rightarrow A_2 \rightsquigarrow e_1\,e_2}$$

Ti-anno

$$\frac{\Gamma \vdash E \Leftarrow A \rightsquigarrow e}{\Gamma \vdash E : A \Rightarrow A \rightsquigarrow e}$$

Ti-merge

$$\frac{\Gamma \vdash E_1 \Rightarrow A_1 \rightsquigarrow e_1 \quad \begin{array}{cc} \Gamma \vdash E_2 \Rightarrow A_2 \rightsquigarrow e_2 & A_1 * A_2 \end{array}}{\Gamma \vdash E_1\,,,\,E_2 \Rightarrow A_1 \,\&\, A_2 \rightsquigarrow \langle e_1, e_2 \rangle}$$

$\boxed{\Gamma \vdash E \Leftarrow A}$ *(Checking)*

Ti-abs

$$\frac{\Gamma \vdash A \qquad \Gamma, x : A \vdash E \Leftarrow B \rightsquigarrow e}{\Gamma \vdash \lambda x.\, E \Leftarrow A \to B \rightsquigarrow \lambda x.\, e}$$

Ti-sub

$$\frac{\Gamma \vdash B \qquad \Gamma \vdash E \Rightarrow A \rightsquigarrow e \qquad A <: B \rightsquigarrow e'}{\Gamma \vdash E \Leftarrow B \rightsquigarrow e'\,e}$$

Figure 2.1: Type system of λ_i

⊤ brings extra complications, because Definition 1 does not hold anymore (⊤ is trivially a supertype of every type). To address this problem, the notion of *top-like types* was introduced, which are those types that behave like ⊤ (such as ⊤ & ⊤, ⊤ & ⊤ & ⊤, ...), and captured by a predicate ⌉ · ⌈. An important observation is that any coercions for top-like types are unique, *even if multiple derivations exist*. The meta-function ⟦·⟧ used in rules Sɪ-ᴀɴᴅL and Sɪ-ᴀɴᴅR defines coercions for top-like types. With top-like types, Definition 1 is refined to account for ⊤, as shown in Definition 2.

Definition 2 (⊤-Disjointness). $A * B \triangleq \neg\lceil A \rceil \wedge \neg\lceil B \rceil \wedge (\forall C.\ A <: C \wedge B <: C \Longrightarrow \rceil C \lceil)$

However, a careful analysis of Definition 2 shows that intersection types such as ⊤ & ⊤ and ⊤ & Int are not well-formed because their constituent types are not disjoint. This is one of the limitations in λ_i, since "a merge of two ⊤-types will always return the same value regardless of which component of the merge is chosen" [Alpuim et al. 2017]. In other words, ⊤ is always disjoint to every other type. This restriction was later lifted in the F_i calculus of Alpuim et al. [2017] by a set of inference rules, but whether a corresponding specification of disjointness exists or not was not known at that time. We will present a suitable specification in Section 10.1.

Combined with bidirectional type-checking, Oliveira et al. [2016] formalized the λ_i calculus in Coq and prove that there is at most one elaboration derivation for any expression, and as a consequence, there is only one possible target program and thus coherence follows trivially. We refer the reader to their paper for a detailed account of λ_i.

2.1.4 Dɪsᴊᴏɪɴᴛ Pᴏʟʏᴍᴏʀᴘʜɪsᴍ

Disjoint polymorphism, first proposed by Alpuim et al. [2017] in the F_i calculus, is a more advanced mechanism to combine disjoint intersection types with parametric polymorphism. The combination allows objects with statically unknown types to be composed without conflicts. To understand the usefulness of disjointness polymorphism, consider the following program (adapted from Alpuim et al. [2017]):

```
mergeBad X (x : X) : X & Int = x ,, 2;
```

mergeBad takes an argument x of type X (which is itself a type variable), and merges it with 2. However, if we were to allow such definition, we could easily create an example where incoherence occurs again:

```
(mergeBad Int 1) : Int -- 1 or 2
```

This is essentially the same problem of allowing 1 ,, 2, which as we discussed will cause ambiguity. For λ_i, we know the concrete type for each variable and thus disjointness checking can

help avoid this problematic expression. However, with parametric polymorphism, a variable could have any types, including those that are already in the intersection. So a question to ask is to decide under what conditions a type variable is disjoint with, say, **Int**. This is where *disjointness constraints* come into stage. The key idea is that since we do not know *a priori* what is the type with which a type variable can be instantiated, we can restrict the set of types it can be instantiated to. Let us rewrite the above program as follows:

```
mergeGood [X * Int] (x : X) : X & Int = x ,, 2;
```

The only change is the notation [X * **Int**], where the left-side of * denotes the type variable being declared, and the right-side denotes the disjointness constraint(s). Here the disjointness constraint (**Int**) effectively states that the type variable X can be instantiated to any types disjoint with **Int**. For instance, the expression mergeGood **Bool** True type checks but the expression mergeGood **Int** 1 is rejected because **Int** (the type argument) is not disjoint with **Int** (the disjointness constraint). Moreover, we can express multiple constraints using intersection types, for example,

```
mergeThree [X * Int & Bool] (x : X) : X & Int & Bool = x ,, 2 ,, True;
```

Here the type variable X can only be instantiated to types disjoint with both **Int** and **Bool**.

In essence disjoint intersection types and disjoint polymorphism retain most of the expressive power of the merge operator. For example, as noted by Alpuim et al. [2017], they can be used to model powerful forms of extensible records. However, forcing every intersection types to be disjoint is unnecessarily restrictive. For instance, 1 : Int & Int is undoubtedly unambiguous, but is rejected by λ_i and F_i. Another issue is that because of the restriction, F_i lacks a general substitution lemma; only a restricted form applies, which greatly complicates the metatheory. Our starting point in this thesis is to lift this restriction and makes room for more expressiveness for calculi with disjoint intersection types.

2.2 Mixins and Traits

Programmers have long realized that single inheritance is not flexible enough when it comes to structuring a class hierarchy. For example, consider two classes in different branches of the inheritance hierarchy, and assume that they share features not inherited from their (unique) common parent. Attempting to share the implementation of the common features may lead to putting the common methods *too high* in the hierarchy (i.e., they are forced into their common parent), and these methods will be inherited by other classes in the same hierarchy, which may not be desirable. On the other hand, putting those methods in a *lower* position

(a) The diamond problem   (b) Mixin composition
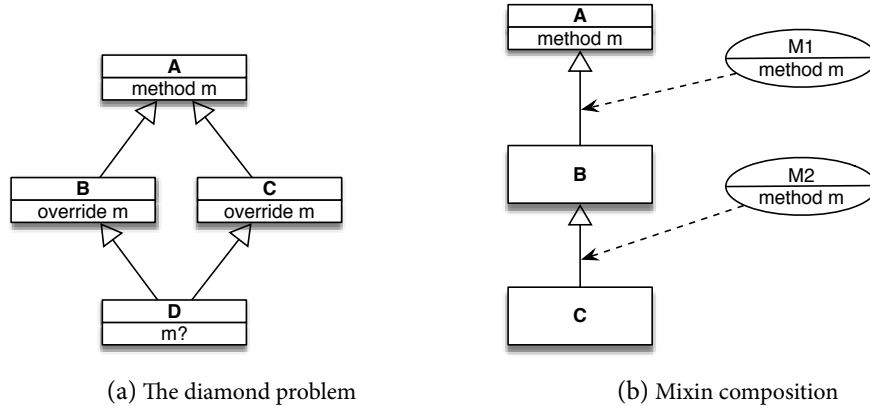
Figure 2.2: Multiple inheritance and mixins

results in code duplication. To overcome this limitation, *multiple inheritance* was proposed as a generalization of single inheritance. However, as Steve Cook [Cook 1987] put it:

> *"Multiple inheritance is good, but there is no good way to do it."*

One of the problems in multiple inheritance is the ambiguity issue that arises when conflicting features are inherited along different paths. A classic situation is the *diamond problem* [Bracha and Cook 1990] where a class inherits from two parent classes that have a common superclass, as depicted by Fig. 2.2a.

Mixins and traits are two well-studied mechanisms to provide some form of multiple inheritance. Mixins [Bracha and Cook 1990] provide a simple mechanism for multiple inheritance without the ambiguity issue. A mixin is a subclass declaration parameterized over a superclass. Or simply put, a mixin can be treated as a function from classes to classes. Thus the same mixin can be used to extend a variety of parent classes with the same set of features. Figure 2.2b shows a typical class hierarchy when using mixins. In the mixin model, a class can inherit from another class by means of single inheritance as usual. Apart from that, it can also have several mixins applied *one at a time*. Let us take a close look at Fig. 2.2b. Both mixins M1 and M2 contain a method m, a question arises as to which one is inherited in the class C. The answer is m from the mixin M2. This is because mixin composition is *linear*: methods defined in mixins appearing later override all the identically named methods of earlier mixins. While this simple mechanism does avoid conflicts, it also lead to other problems. For example, though we can obtain the method m from the mixin M1 by switching the order of M1 and M2, no suitable order of composition exists to obtain m from the superclass A.

In response to the problems in the then compositional models, Schärli et al. [2003] proposed a mechanism called *traits* as a better way to foster code reuse in object-oriented pro-

(a) A simple trait   (b) Trait composition with conflicts
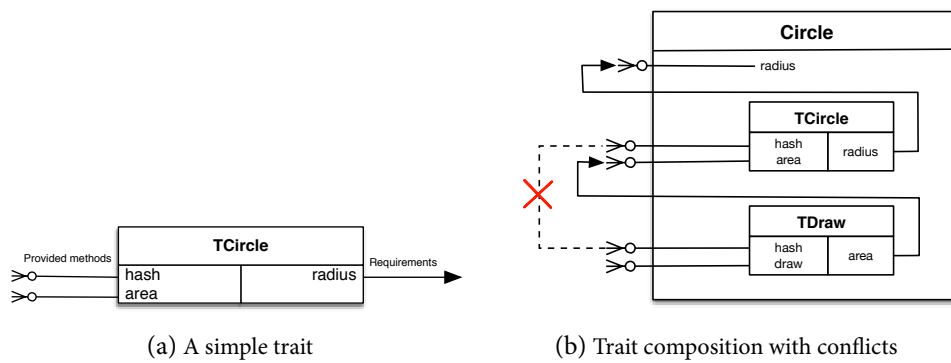
Figure 2.3: Traits and conflicts

grams. A trait is essentially *a set of pure methods*, divorced from any class hierarchy. A trait *provides* a set of methods to implement the behavior, and it may also specify a set of *required methods* that parameterize the provided behavior. Figure 2.3a shows a simple trait `TCircle`, which provides two methods `hash` and `area`, and requires a method `radius`. A class is then constructed by inheriting from a superclass and incorporating a collection of traits, as shown in Fig. 2.3b. Notice that there is a conflicting method `hash` that is provided by both `TCircle` and `TDraw`. This is where the trait model is very different from the mixin model. Unlike mixins that force a linear order in their composition, traits can be composed in arbitrary order, and as a consequence, conflicting methods must be resolved *explicitly*, either by overriding the conflicting methods, or by excluding a method from all but one trait. Schärli et al. [2003] discuss several other issues with mixins, which can be improved by traits. We refer to their paper for a detailed account of traits.

2.3 Family Polymorphism and Nested Composition

Family polymorphism [Ernst 2001] is the ability to simultaneously refine a family of related classes through inheritance. This is motivated by a need to not only refine individual classes, but also to preserve and refine their mutual relationships. Nystrom et al. [2004] call this *scalable extensibility*: "the ability to extend a body of code while writing new code proportional to the differences in functionality". A well-studied mechanism to achieve family inheritance is *nested inheritance* [Nystrom et al. 2004]. Nested inheritance combines two aspects. Firstly, a class can have nested class members; the outer class is then a family of (inner) classes. Secondly, when one family extends another, it inherits (and can override) all the class members, as well as the relationships within the family between the class members. However, the members of the new family do not become subtypes of those in the parent family.

THE EXPRESSION PROBLEM.   Ernst [2004] illustrates the benefits of nested inheritance for modularity and extensibility with one of the most elegant and concise solutions to the *expression problem* [Wadler 1998]. The expression problem, as surveyed by Torgersen [2004], is to answer the question:

> *"To which degree can your application be structured in such a way that both the data model and the set of virtual operations over it can be extended without the need to modify existing code, without the need for code repetition and without run-time type errors."*

The expression problem is concerned with two-dimensional extensions: (1) adding new variants to the datatype; (2) and adding new operations on the datatype. Depending on the programming style used in the code, it is usually straightforward to add either new variants or new operations. For example, in an object-oriented language such as Java where an abstract datatype is represented by means of classes whose methods are the operations on the datatype, it is easy to extend the set of variants by writing another class. On the other hand, in a functional language such as Haskell where the abstract datatype is modeled by means of algebraic datatypes with a set of pattern matching functions as the operations, then it is easy to add new operations by writing new pattern matching functions. In either case, it is much harder to perform both extensions in the *same* language.

THE EXPRESSION PROBLEM, SCANDINAVIAN STYLE.   Nowadays we know many solutions to the expression problem (for example, see Oliveira [2009]; Oliveira and Cook [2012]; Swierstra [2008]; Wang and Oliveira [2016]; Zenger and Odersky [2005], to cite a few). Among all of them, Ernst's solution is perhaps one of the most elegant solutions out there. Ernst solves the expression problem in the gbeta language [Ernst 2000], which he adorns with a Java-like syntax for presentation purposes, for a small abstract syntax tree (AST) example. His starting point is the code shown in Fig. 2.4a. The outer class `Lang` contains a family of related AST classes: the common superclass `Exp` and two classes, `Lit` for literals and `Add` for addition. The AST comes equipped with one operation, `toString`, which is implemented by both cases. Notice that all the inner classes are *virtual*, in the same sense of virtual methods, which means that they may be redefined in subclasses of the enclosing class.

ADDING A NEW OPERATION.   One way to extend the family is to add an additional evaluation operation, as shown in the top half of Fig. 2.4b. This is done by subclassing the `Lang` class and refining all the contained classes by implementing the additional `eval` method. The semantics of the keyword **refine** is that the virtual class is constrained to be a subclass

```
class Lang {                          // Adding a new operation
  virtual class Exp {                 class LangEval extends Lang {
    String toString() {}                refine class Exp {
  }                                       int eval() {}
  virtual class Lit extends Exp {       }
    int value;                          refine class Lit {
    Lit(int value) {                      int eval { return value; }
      this.value = value;               }
    }                                   refine class Add {
    String toString() {                   int eval { return
      return value;                         left.eval() + right.eval();
    }                                     }
  }                                     }
  virtual class Add extends Exp {     }
    Exp left,right;                   // Adding a new case
    Add(Exp left, Exp right) {        class LangNeg extends Lang {
      this.left = left;                 virtual class Neg extends Exp {
      this.right = right;                 Neg(Exp exp) { this.exp = exp; }
    }                                     String toString() {
    String toString() {                     return "-(" + exp + ")";
      return left + "+" + right;          }
    }                                     Exp exp;
  }                                     }
}                                     }
```

(a) Base family: the language `Lang`   (b) Extending in two dimensions

Figure 2.4: The expression problem, Scandinavian Style

of the new declaration. In other words, `Exp`, `Lit` and `Add` are all extended with the `eval` method. Note that the inheritance between, e.g., `Lang.Exp` and `Lang.Lit` is transferred to `LangEval.Exp` and `LangEval.Lit`. Similarly, the `Lang.Exp` type of the `left` and `right` fields in `Lang.Add` is automatically refined to `LangEval.Exp` in `LangEval.Add`.

ADDING A NEW CASE. A second dimension to extend the family is to add a case for negation, shown in the bottom half of Fig. 2.4b. This is similarly achieved by subclassing `Lang`, and now adding a new contained virtual class `Neg` that represents the unary negation operator. Note that `Neg` is declared to be a subclass of `Exp`, which means that the extension to `Exp` will also be added to `Neg`.

COMBINING BOTH EXTENSIONS. Finally, the two extensions are naturally combined by means of multiple inheritance, closing the diamond. (Ernst uses the symbol ⊕ to play the role of "intersecting" two classes.)

```
class LangNegEval extends LangEval ⊕ LangNeg {
  refine class Neg {
    int eval() { return -exp.eval(); }
  }
}
```

The only effort required is to implement the one missing operation case, evaluation of negated expressions.

2.4 Functional Object Encodings

Cook and Palsberg [1989] developed a method for modeling inheritance in the presence of self-reference, based on the fixed-point semantics of recursive definitions. In their model, the interpretation of inheritance is taken as a mechanism of *incremental programming*, where new programs are developed by specifying the *modification*—i.e., how they differ from existing ones; self-reference in the original definition must be changed to refer to the modified definition.

We use an example to illustrate the encodings of classes and objects. First we define a class of points. Points have two components x and y to specify their locations. The `dist` method computes their (Euclidean) distance from the origin. The following is a Scala class `Point`:

```
class Point(x : Int, y : Int) {
  def dist() = sqrt(square(this.x) + square(this.y))
}
```

In a purely functional setting, objects are modeled as records whose fields represent methods. The class `Point` is then modeled as a *generator* `PointGen(a, b)`, defined as follows:

```
PointGen(a, b) = λthis.
  { x = a
  , y = b
  , dist() = sqrt(square(this.x) + square(this.y))
  }
```

Notice that the keyword **this** is modeled as a formal parameter of the function. Formally speaking, a function intended to specify a fixed point whose formal parameter represents self-reference is called a generator.

A point $(3, 4)$ is created by taking a fixed point of `PointGen(3, 4)` using a *lazy recursive* let binding:

```
p = letrec this = PointGen(3, 4, this) in this
```

and method invocation on objects is simply record projections: `p.dist()` evaluates to 5 as expected.

MODELING INHERITANCE.   Inheritance allows a new class to be defined by adding or replacing methods in an existing class. We illustrate this by defining another class `Circle`:

```
class Circle(x : Int, y : Int, radius : Int) extends Point(x, y) {
  override def dist() = abs(super.dist() - this.radius)
}
```

The class `Circle` inherits from `Point` and redefines the method `dist` to mean the closest distance from the circle to the origin. It reuses the original `dist` method in the body. To correctly model inheritance, there are three aspects to note: (1) the addition or replacement of methods, (2) the redirection of **this** in the original generator to the modified methods, (3) and the binding of **super** to refer to the original methods.

Inheritance is modeled as a function that takes a generator and returns a new generator. Such functions are called *wrappers*. Below we give a wrapper for the subclass `Circle`:

```
CircleWrapper(a, b, r) = λthis. λsuper.
  { radius = r
  , dist() = abs(super.dist() - this.radius)
  }
```

`CircleWrapper(a, b, r)` is defined as a function of two arguments, one representing **this** and the other representing **super**.

The generator for the class `Circle` can now be defined by applying `CircleWrapper` to `PointGen` as follows:

```
CircleGen(a, b, r) = λthis.
  let super = PointGen(a, b, this)
  in (CircleWrapper(a, b, r, this) super) ⊕ super
```

That is, a wrapper works by first distributing **this** to both the wrapper and the original generator. Then the modification is applied to the original record definition to produce a modification record. Note that at this stage, the binding of **this** correctly refers to the modification, while the binding of **super** refers to the original record. Finally the modification record is combined with the original record using ⊕. (`M` ⊕ `N` is defined in a way such that any method defined in `M` replaces the corresponding method defined in `N`.)

2.5 Program Equivalence and Logical Relations

Proving equivalence of programs is important for a variety of settings, e.g., verifying the correctness of compiler optimization and other program transformations, establishing the property that program behavior is independent of the representation of an abstract type. The latter—so-called the property of *representation independence*—is particularly relevant for programmers and clients in the sense that a client will not be able to tell a difference if one implementation is swapped by another, as long as they all adhere to the same interface.

Program equivalence is generally defined in terms of *contextual equivalence*. The intuition is that two programs are equivalent if we *cannot* tell them apart in any context. More formally, we introduce the notion of *expression contexts*. An expression context $\mathcal{D}$ is a term with a single hole $[\cdot]$ (possibly under some binders) in it. Take the simply-typed lambda calculus (STLC) for example, the syntax of expression contexts is as follows:

$$\text{Contexts} \quad \mathcal{D} ::= [\cdot] \mid \lambda x.\, \mathcal{D} \mid \mathcal{D}\, e \mid e\, \mathcal{D}$$

The only operation of expression contexts is *replacement*, which is the process of filling a hole in an expression context $\mathcal{D}$ with an expression e, written $\mathcal{D}\{e\}$. An important point is that replacement *is not* substitution, that is, the free variables of e that are exposed by $\mathcal{D}$ are captured by replacement. The static semantics of STLC is extended to expression contexts by defining the typing judgment

$$\mathcal{D} : (\Psi \vdash \tau) \mapsto (\Psi' \vdash \tau')$$

where $(\Psi \vdash \tau)$ indicates the type of the hole. This judgment is inductively defined so that if $\Psi \vdash e : \tau$, then $\Psi' \vdash \mathcal{D}\{e\} : \tau'$.

Contextual equivalence. We define a *complete program* to mean any closed term of type Int. The following two definitions capture the notion of *contextual equivalence*:

Definition 3 (Kleene Equality $\simeq$). Two complete programs, e and e', are Kleene equal, written $e \simeq e'$, if there exists i such that $e \longrightarrow^* i$ and $e' \longrightarrow^* i$.

Definition 4 (Contextual Equivalence $\simeq_{ctx}$).

$$\Psi \vdash e_1 \simeq_{ctx} e_2 : \tau \triangleq \Psi \vdash e_1 : \tau \wedge \Psi \vdash e_2 : \tau \wedge$$
$$(\forall \mathcal{D}.\ \mathcal{D} : (\Psi \vdash \tau) \mapsto (\bullet \vdash \text{Int}) \implies \mathcal{D}\{e_1\} \simeq \mathcal{D}\{e_2\})$$

In other words, for all possible experiments $\mathcal{D}$, the outcome of an experiment on e_1 is the same as the outcome on e_2 (i.e., $\mathcal{D}\{e_1\} \simeq \mathcal{D}\{e_2\}$), which is an equivalence relation.

LOGICAL RELATIONS.   Unfortunately, directly proving contextual equivalence is very difficult in general (if not possible at all), since it involves quantification over *all* possible contexts. There has been much work on finding tractable techniques for proving contextual equivalence, many of which are based on the proof method called *logical relations* [Plotkin 1973; Statman 1985; Tait 1967].

In a nutshell, logical relations specify relations over well-typed terms via a structural induction on the syntax of types. For instance, logically related functions, when taken logically related arguments, return logically related results. For STLC, the logical relation is a family of relations $(v_1, v_2) \in \mathcal{V}[\![\tau]\!]$ between closed values of type τ. It is inductively defined on the structure of τ as follows:

$$
\begin{aligned}
(v_1, v_2) \in \mathcal{V}[\![\mathsf{Int}]\!] &\triangleq \exists i.\, v_1 = v_2 = i \\
(v_1, v_2) \in \mathcal{V}[\![\tau_1 \to \tau_2]\!] &\triangleq \forall (v_1', v_2') \in \mathcal{V}[\![\tau_1]\!].\, (v_1\, v_1', v_2\, v_2') \in \mathcal{E}[\![\tau_2]\!] \\
(e_1, e_2) \in \mathcal{E}[\![\tau]\!] &\triangleq \exists v_1, v_2.\, e_1 \longrightarrow^* v_1 \wedge e_2 \longrightarrow^* v_2 \wedge (v_1, v_2) \in \mathcal{V}[\![\tau]\!]
\end{aligned}
$$

That is, two integers are related if they are the same integer. Two functions v_1 and v_2 are related at the type $\tau_1 \to \tau_2$ if given two arguments v_1' and v_2' related at the domain type τ_1, the functions applied to the arguments are related expressions at the codomain type τ_2.

LOGICAL AND CONTEXTUAL EQUIVALENCE COINCIDE.   The usefulness of the logical relation lies in the fact that it characterize *exactly* contextual equivalence—i.e., logical and contextual equivalence coincide for STLC:

Proposition 2.1. *For closed expression $e : \tau$ and $e' : \tau$, $(e, e') \in \mathcal{E}[\![\tau]\!]$ iff $\bullet \vdash e \simeq_{ctx} e' : \tau$.*

The proofs proceeds by generalizing to open terms, which will be explained in more details in Chapter 5. The above proposition licenses a common approach of proving properties involving contextual equivalence: we first prove a related property using logical relations, and then transfer it back to the one involving contextual equivalence.

Part II

Typed Calculi

3 Semantics of the λ_i^+ Calculus

This chapter presents λ_i^+,[1] a calculus based on λ_i [Oliveira et al. 2016] that features unrestricted intersections, BCD-style subtyping and a merge operator, which we believe captures the essence of nested composition. We illustrate this by presenting a solution to the expression problem based on family polymorphism. We then discuss the algorithmic aspects of λ_i^+. The coherence property of λ_i^+ is discussed in Chapter 5.

3.1 Introduction

λ_i^+ is a simple calculus with records and disjoint intersection types that supports *nested composition*. Nested composition enables encoding simple forms of family polymorphism. More complex forms of family polymorphism, involving binary methods [Bruce et al. 1996] and mutable state are not yet supported, but are interesting avenues for future work. Nevertheless, in λ_i^+, it is possible, for example, to encode Ernst's elegant family-polymorphism solution to the expression problem. Compared to λ_i the essential novelty of λ_i^+ are distributivity rules between function/record types and intersection types. These rules are the delta that enables extending the simple forms of multiple inheritance/composition supported by λ_i into a more powerful form supporting nested composition. The distributivity rule between function types and intersections is common in calculi with intersection types aimed at capturing the set of all strongly normalizable terms, and was first proposed by Barendregt et al. [1983]. However the distributivity rule is not common in calculi or languages with intersection types aimed at programming. For example the rules employed in languages that support intersection types (such as Scala, TypeScript, Flow or Ceylon) lack distributivity rules. Moreover distributivity is also missing from several calculi with a merge operator. This includes all calculi with disjoint intersection types [Alpuim et al. 2017; Oliveira et al. 2016] and Dunfield's work on elaborating intersection types [Dunfield 2014], which was the original foundation for λ_i. A possible reason for this omission in the past is that distributivity adds substantial complexity (both algorithmically and meta-theoretically), without having

---

[1]It was also called NeColus in the original publication [Bi et al. 2018]. Also the "+" symbol stands for two extra features compared to λ_i: *BCD subtyping* and *unrestricted intersections*.

any obvious practical applications. This chapter shows how to deal with the complications of BCD subtyping, while identifying a major reason to include it in a programming language: BCD enables nested composition and subtyping, which is of significant practical interest.

λ_i^+ differs significantly from previous BCD-based calculi in that it has to deal with the possibility of incoherence, introduced by the merge operator. Incoherence is a non-issue in the previous BCD-based calculi because they do not feature this merge operator or any other source of incoherence. Although previous work on disjoint intersection types proposes a solution to coherence, the solution imposes several ad-hoc restrictions (cf. Section 3.5) to guarantee the uniqueness of the elaboration and thus allows for a simple syntactic proof of coherence. Most importantly, it makes it hard or impossible to adapt the proof to extensions of the calculus, such as the new subtyping rules required by the BCD system. We shall return to this point in Chapter 5.

3.2 λ_i^+ by Examples

This section illustrates λ_i^+ with an encoding of a family polymorphism solution to the expression problem, and informally presents its salient features.

3.2.1 The Expression Problem, λ_i^+ Style

The λ_i^+ calculus allows us to solve the expression problem in a way that is very similar to Ernst's gbeta solution in Section 2.3. However, the underlying mechanisms of λ_i^+ are quite different from those of gbeta. In particular, λ_i^+ features a structural type system in which we can model objects with records, and object types with record types. For instance, we model the interface of Lang.Exp with the singleton record type { print : **String** }. For the sake of conciseness, we use **type** aliases to abbreviate types.

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

Similarly, we capture the interface of the Lang family in a record, with one field for each case's constructor.

```
type Lang = {
  lit : Int → IPrint,
  add : IPrint → IPrint → IPrint
};
```

Here is the implementation of Lang.

```
implLang : Lang = {
  lit (value : Int) = {
```

```
    print = value.toString
  },
  add (left : IPrint) (right : IPrint) = {
    print = left.print ++ "+" ++ right.print
  }
};
```

We assume several primitive types: fixed width integers **Int**, `Double` for numeric operations and **String** for text manipulation. A λ_i^+ program consists of a collection of definitions and declarations, separated by semicolon ;.

ADDING EVALUATION. We obtain `IPrint & IEval`, which is the corresponding type for `LangEval.Exp`, by intersecting `IPrint` with `IEval` where

```
type IEval = { eval : Int };
```

The type for `LangEval` is then

```
type LangEval = {
  lit : Int → IPrint & IEval,
  add : IPrint & IEval → IPrint & IEval → IPrint & IEval
};
```

We obtain an implementation for `LangEval` by merging the existing `Lang` implementation `implLang` with the new evaluation functionality `implEval` using the merge operator , ,.

```
implEval = {
  lit (value : Int) = {
    eval = value
  },
  add (left : IEval) (right : IEval) = {
    eval = left.eval + right.eval
  }
};
implLangEval : LangEval = implLang ,, implEval;
```

ADDING NEGATION. Adding negation to `Lang` works similarly.

```
type NegPrint = { neg : IPrint → IPrint };
type LangNeg   = Lang & NegPrint;

implNegPrint : NegPrint = {
  neg (exp : IPrint) = {
```

```
    print = "-" ++ exp.print
  }
};
implLangNeg : LangNeg = implLang ,, implNegPrint;
```

PUTTING EVERYTHING TOGETHER.   Finally, we can combine the two extensions and provide the missing implementation of evaluation for the negation case.

```
type NegEval = { neg : IEval → IEval};
implNegEval : NegEval = {
  neg (exp : IEval) = {
    eval = 0 - exp.eval
  }
};
```

```
type NegEvalExt  = { neg : IPrint & IEval → IPrint & IEval };
type LangNegEval = LangEval & NegEvalExt;
implLangNegEval : LangNegEval =
  implLangEval ,, implNegPrint ,, implNegEval;
```

We can test `implLangNegEval` by creating an object that represents $-2 + 3$, which is able to print and evaluate at the same time.

```
fac  = implLangNegEval;
e    = fac.add (fac.neg (fac.lit 2)) (fac.lit 3);
main = e.print ++ " = " ++ e.eval.toString -- Output: "-2+3 = 1"
```

MULTI-FIELD RECORDS.   Recall that in Section 2.1, we show how to model multi-field records by single-field records. Thus λ_i^+ does not have multi-field record types built in. They are merely syntactic sugar for intersections of single-field record types. Hence, the following is an equivalent definition of `Lang`:

```
type Lang = {lit : Int → IPrint} & {add : IPrint → IPrint → IPrint};
```

Similarly, the multi-field record expression in the definition of `implLang` is syntactic sugar for the explicit merge of two single-field records.

```
implLang : Lang = { lit = ... } ,, { add = ... };
```

SUBTYPING.   A distinctive difference compared to gbeta is that many more λ_i^+ types are related through subtyping. Indeed, gbeta is unnecessarily conservative [Ernst 2003]: none

of the families is related through subtyping, nor is any of the class members of one family related to any of the class members in another family. For instance, `LangEval` is not a subtype of `Lang`, nor is `LangNeg.Lit` a subtype of `Lang.Lit`.

In contrast, subtyping in λ_i^+ is much more nuanced and depends entirely on the structure of types. The primary source of subtyping are intersection types: any intersection type is a subtype of its components. For instance, `IPrint & IEval` is a subtype of both `IPrint` and `IEval`. Similarly `LangNeg = Lang & NegPrint` is a subtype of `Lang`. Compare this to gbeta where `LangEval.Expr` is not a subtype of `Lang.Expr`, nor is the family `LangNeg` a subtype of the family `Lang`.

However, gbeta and λ_i^+ agree that `LangEval` is not a subtype of `Lang`. The λ_i^+-side of this may seem contradictory at first, as we have seen that intersection types arise from the use of the merge operator. We have created an implementation for `LangEval` with `implLang ,, implEval` where `implLang` has type `Lang`, which suggests that `LangEval` is a subtype of `Lang`. Yet, there is a flaw in our reasoning: strictly speaking, `implLang ,, implEval` is not of type `LangEval` but instead of type `Lang & EvalExt`, where `EvalExt` is the type of `implEval`:

```
type EvalExt = { lit : Int → IEval, add : IEval → IEval → IEval };
```

Nevertheless, the definition of `implLangEval` is valid because `Lang & EvalExt` is a subtype of `LangEval`. Indeed, if we consider for the sake of simplicity only the `lit` field, we have that (**Int** → `IPrint`) & (**Int** → `IEval`) is a subtype of **Int** → `IPrint & IEval`. This follows from a standard subtyping axiom for distributivity of functions and intersections in the BCD system inherited by λ_i^+. In conclusion, `Lang & EvalExt` is a subtype of both `Lang` and of `LangEval`. However, neither of the latter two types is a subtype of the other. Indeed, `LangEval` is not a subtype of `Lang` as the type of `add` is not covariantly refined and thus admitting the subtyping is unsound. For the same reason `Lang` is not a subtype of `LangEval`.

A summary of the various relationships between the language components is shown in Fig. 3.1. Admittedly, the figure looks quite complex because our calculus has a structural type system (as often more foundational calculi do) where more types are related through subtyping, whereas mainstream object-oriented languages have nominal type systems.

STAND-ALONE EXTENSIONS.   Unlike in gbeta and other class-based inheritance systems, in λ_i^+ the extension `implEval` is not tied to `LangEval`. In that sense, it resembles trait and mixin systems that can apply the same extension to different classes. However, unlike those systems, `implEval` can also exist as a value on its own, i.e., it is not an extension per se.
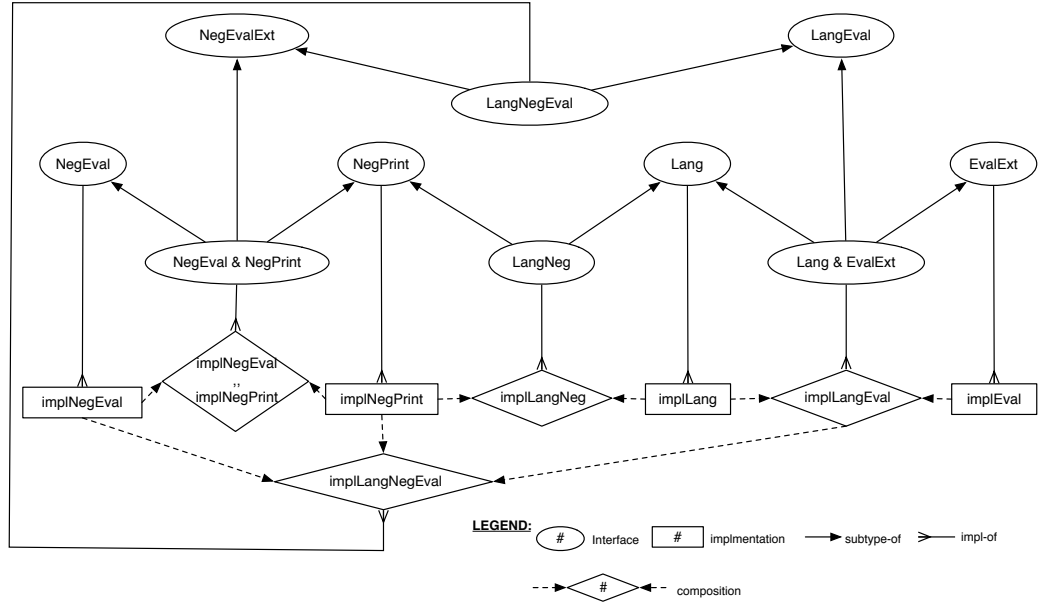
Figure 3.1: Summary of the relationships between language components

3.3 Syntax and Semantics of λ_i^+

In this section we formally present the syntax and semantics of λ_i^+. Compared to previous work [Alpuim et al. 2017; Oliveira et al. 2016], λ_i^+ has a more powerful subtyping relation. The new subtyping relation is inspired by BCD subtyping, but with two noteworthy differences: subtyping is coercive (in contrast to traditional formulations of BCD); and it is extended with records. We also have a new target language with explicit coercions inspired by the coercion calculus of Henglein [1994]. A full technical comparison between λ_i^+ and λ_i can be found in Section 3.5.

3.3.1 Syntax

Figure 3.2 shows the syntax of λ_i^+. For brevity of the meta-theoretic study, we do not consider primitive operations on primitive types. They can be easily added to the language, and our prototype implementation is indeed equipped with common primitive types and their operations. Metavariables A, B, C range over types. Types include integer types Int, a top type $\top$, function types $A \rightarrow B$, intersection types $A \,\&\, B$, and single-field record types $\{l : A\}$. Metavariable E ranges over expressions. Expressions include variables x, literals i, a canonical top value $\top$, lambda abstractions $\lambda x.\, E$, applications $E_1\, E_2$, merges $E_1\, , , E_2$, annotated terms $E : A$, single-field records $\{l = E\}$, and record projections $E.l$.

Types	A, B, C	$::=$	$\mathsf{Int} \mid \top \mid A \to B \mid A \,\&\, B \mid \{l : A\}$
Expressions	E	$::=$	$x \mid i \mid \top \mid \lambda x.\, E \mid E_1\, E_2 \mid E_1 \,,,\, E_2 \mid E : A \mid \{l = E\} \mid E.l$
Term contexts	Γ	$::=$	$\bullet \mid \Gamma, x : A$

Figure 3.2: Syntax of λ_i^+

3.3.2 Declarative Subtyping

Figure 3.3 presents the subtyping relation. We ignore the highlighted parts, and explain them later in Section 3.4.

BCD-style subtyping. The subtyping relation is essentially BCD subtyping [Barendregt et al. 1983], extended with subtyping for single-field records. The top type is a supertype of all types (rule S-top). Rules S-andL, S-andR, and S-and for intersection types axiomatize that $A \,\&\, B$ is the greatest lower bound of A and B. Rules S-arr and S-rcd for function and record subtyping are standard. Rule S-distArr is perhaps the most interesting rule. This, so-called "distributivity" rule, describes the interaction between the subtyping relations for function types and those for intersection types. Note that the other direction $A_1 \to A_2 \,\&\, A_3 \;<:\; (A_1 \to A_2) \,\&\, (A_1 \to A_3)$ and the contravariant distribution $(A_1 \to A_2) \,\&\, (A_3 \to A_2) \;<:\; A_1 \,\&\, A_3 \to A_2$ are both derivable from the existing subtyping rules, as shown below:

$$
\dfrac{\dfrac{A_1 <: A_1 \qquad A_2 \,\&\, A_3 <: A_2}{A_1 \to A_2 \,\&\, A_3 <: A_1 \to A_2}\;\text{S-arr} \qquad \dfrac{A_1 <: A_1 \qquad A_2 \,\&\, A_3 <: A_3}{A_1 \to A_2 \,\&\, A_3 \to A_1 \to A_3}\;\text{S-arr}}{A_1 \to A_2 \,\&\, A_3 <: (A_1 \to A_2) \,\&\, (A_1 \to A_3)}\;\text{S-and}
$$

$$
\dfrac{\dfrac{}{(A_1 \to A_2) \,\&\, (A_3 \to A_2) <: A_1 \to A_2}\;\text{S-andL} \qquad \dfrac{A_1 \,\&\, A_3 <: A_1 \qquad A_2 <: A_2}{A_1 \to A_2 <: A_1 \,\&\, A_3 \to A_2}\;\text{S-arr}}{(A_1 \to A_2) \,\&\, (A_3 \to A_2) <: A_1 \,\&\, A_3 \to A_2}\;\text{S-trans}
$$

Rule S-distRcd, which is not found in the original BCD system, prescribes the distribution of records over intersection types. The two distributivity rules are the key to enabling nested composition. S-topArr is standard in BCD subtyping, and the new rule S-topRcd plays a similar role for record types.

Non-algorithmic. The subtyping relation in Fig. 3.3 is clearly no more than a specification due to the two subtyping axioms: rules S-refl and S-trans. A sound and complete algorithmic version is discussed in Section 3.6. Nevertheless, for the sake of establishing coherence, the declarative subtyping relation is sufficient.

$$\boxed{A <: B \rightsquigarrow co}$$ *(Declarative subtyping)*

S-REFL
$$\frac{}{A <: A \rightsquigarrow \text{id}}$$

S-TRANS
$$\frac{A_2 <: A_3 \rightsquigarrow co_1 \qquad A_1 <: A_2 \rightsquigarrow co_2}{A_1 <: A_3 \rightsquigarrow co_1 \circ co_2}$$

S-TOP
$$\frac{}{A <: \top \rightsquigarrow \text{top}}$$

S-ANDL
$$\frac{}{A_1 \& A_2 <: A_1 \rightsquigarrow \pi_1}$$

S-ANDR
$$\frac{}{A_1 \& A_2 <: A_2 \rightsquigarrow \pi_2}$$

S-AND
$$\frac{A_1 <: A_2 \rightsquigarrow co_1 \qquad A_1 <: A_3 \rightsquigarrow co_2}{A_1 <: A_2 \& A_3 \rightsquigarrow \langle co_1, co_2 \rangle}$$

S-ARR
$$\frac{B_1 <: A_1 \rightsquigarrow co_1 \qquad A_2 <: B_2 \rightsquigarrow co_2}{A_1 \to A_2 <: B_1 \to B_2 \rightsquigarrow co_1 \to co_2}$$

S-RCD
$$\frac{A <: B \rightsquigarrow co}{\{l : A\} <: \{l : B\} \rightsquigarrow co}$$

S-DISTARR
$$\frac{}{(A_1 \to A_2) \& (A_1 \to A_3) <: A_1 \to A_2 \& A_3 \rightsquigarrow \text{dist}_\to}$$

S-DISTRCD
$$\frac{}{\{l : A\} \& \{l : B\} <: \{l : A \& B\} \rightsquigarrow \text{id}}$$

S-TOPARR
$$\frac{}{\top <: \top \to \top \rightsquigarrow \text{top}_\to}$$

S-TOPRCD
$$\frac{}{\top <: \{l : \top\} \rightsquigarrow \text{id}}$$
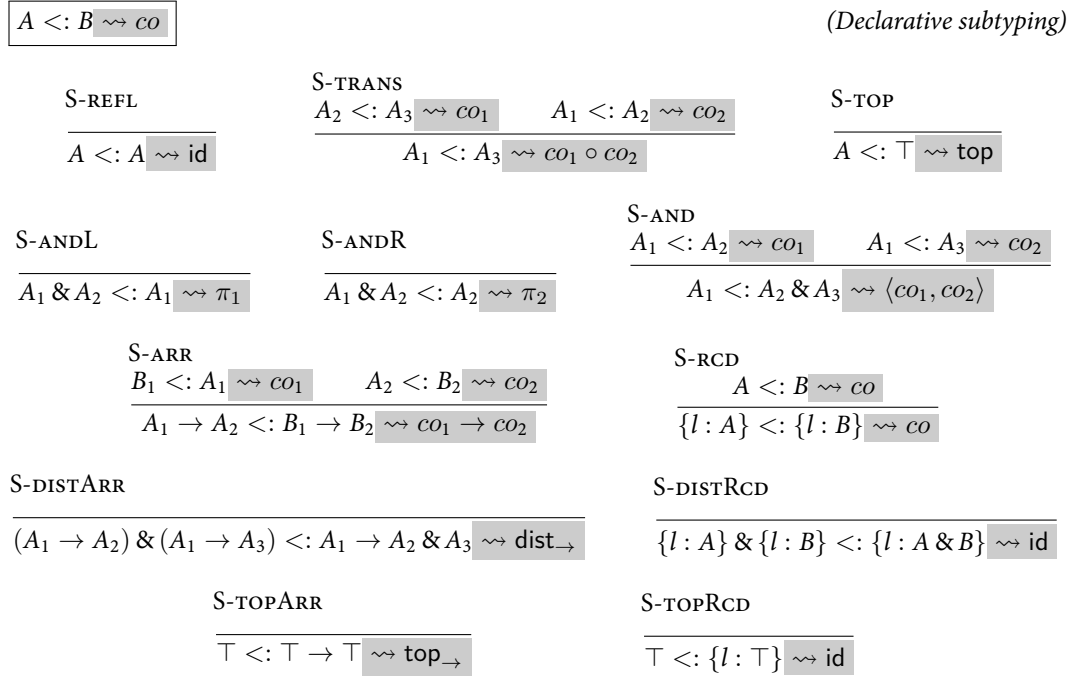
Figure 3.3: Declarative subtyping of λ_i^+

PROPERTIES OF SUBTYPING.   The subtyping relation is vacuously *reflexive* and *transitive*.

3.3.3 TYPING OF λ_i^+

The bidirectional type system for λ_i^+ is shown in Fig. 3.4. We ignore the highlighted parts for now.

TYPING RULES.   The typing rules of λ_i^+ in Fig. 3.4 are mostly ported from λ_i in Fig. 2.1. As with traditional bidirectional type systems, we employ two modes: the inference mode ($\Rightarrow$) and the checking mode ($\Leftarrow$). The inference judgment $\Gamma \vdash E \Rightarrow A$ says that we can synthesize a type A for expression E in the context Γ. The checking judgment $\Gamma \vdash E \Leftarrow A$ checks E against A in the context Γ. Most of the rules are quite standard in the literature. The merge expression $E_1 ,, E_2$ is well-typed if both sub-expressions are well-typed, and their types are *disjoint* (rule T-MERGE).

DISJOINTNESS RULES.   The set of inference rules for disjointness $A * B$ is shown in Fig. 3.5. Note that our set of disjointness rules is different from that in λ_i [Oliveira et al. 2016, Figure 10]: λ_i does not have rules D-TOPL and D-TOPR, which first appeared in F_i [Alpuim et al.

$$\boxed{\Gamma \vdash E \Rightarrow A \leadsto e} \qquad\qquad\qquad \textit{(Inference)}$$

T-TOP

$$\frac{}{\Gamma \vdash \top \Rightarrow \top \leadsto \langle\rangle}$$

T-LIT

$$\frac{}{\Gamma \vdash i \Rightarrow \mathsf{Int} \leadsto i}$$

T-VAR

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A \leadsto x}$$

T-APP

$$\frac{\Gamma \vdash E_1 \Rightarrow A_1 \to A_2 \leadsto e_1 \qquad \Gamma \vdash E_2 \Leftarrow A_1 \leadsto e_2}{\Gamma \vdash E_1\, E_2 \Rightarrow A_2 \leadsto e_1\, e_2}$$

T-ANNO

$$\frac{\Gamma \vdash E \Leftarrow A \leadsto e}{\Gamma \vdash E : A \Rightarrow A \leadsto e}$$

T-PROJ

$$\frac{\Gamma \vdash E \Rightarrow \{l : A\} \leadsto e}{\Gamma \vdash E.l \Rightarrow A \leadsto e}$$

T-MERGE

$$\frac{\Gamma \vdash E_1 \Rightarrow A_1 \leadsto e_1 \qquad \Gamma \vdash E_2 \Rightarrow A_2 \leadsto e_2 \qquad A_1 * A_2}{\Gamma \vdash E_1\,,\, E_2 \Rightarrow A_1 \,\&\, A_2 \leadsto \langle e_1, e_2 \rangle}$$

T-RCD

$$\frac{\Gamma \vdash E \Rightarrow A \leadsto e}{\Gamma \vdash \{l = E\} \Rightarrow \{l : A\} \leadsto e}$$

$$\boxed{\Gamma \vdash E \Leftarrow A \leadsto e} \qquad\qquad\qquad \textit{(Checking)}$$

T-ABS

$$\frac{\Gamma, x : A \vdash E \Leftarrow B \leadsto e}{\Gamma \vdash \lambda x.\, E \Leftarrow A \to B \leadsto \lambda x.\, e}$$

T-SUB

$$\frac{\Gamma \vdash E \Rightarrow A \leadsto e \qquad A <: B \leadsto co}{\Gamma \vdash E \Leftarrow B \leadsto co\, e}$$
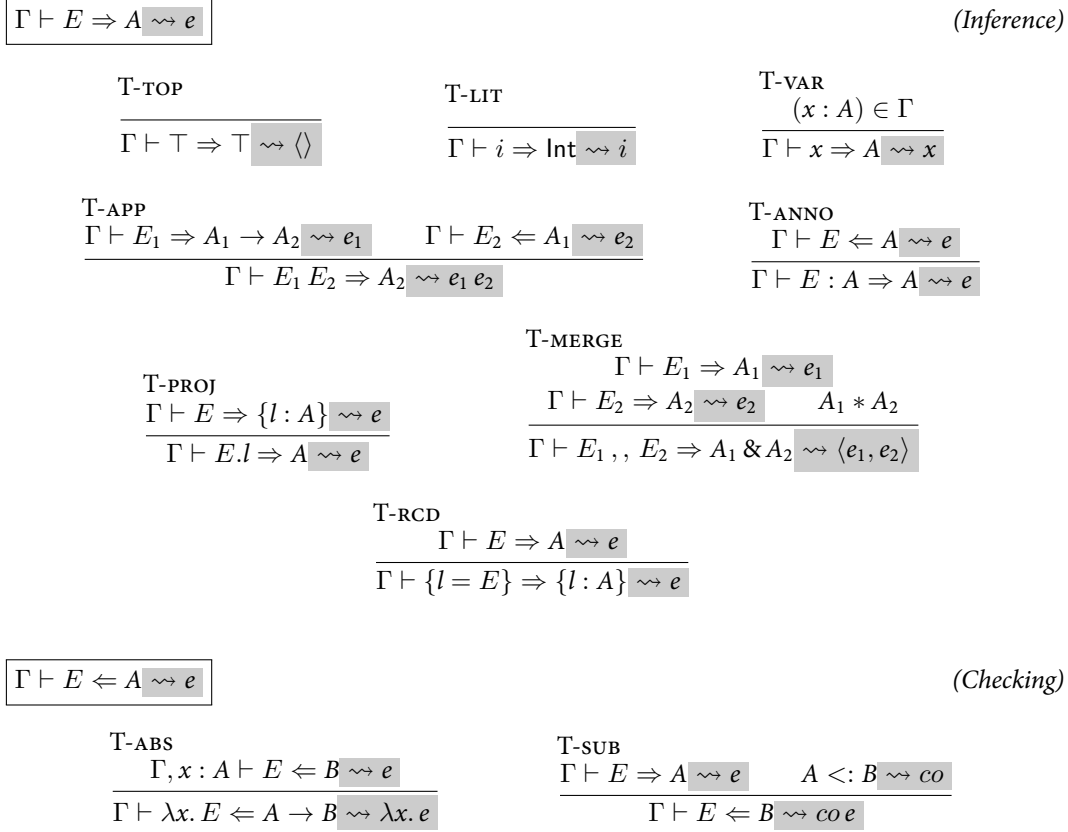
Figure 3.4: Bidirectional type system of λ_i^+

2017, Figure 3]. Disjointness axioms $A *_{ax} B$ (appearing in rule D-AX) take care of two types with different type constructors (e.g., Int and records). The disjointness relation is helpful to check whether the merge of two expressions of type A and B preserves coherence, e.g., it rule out ambiguous expressions such as $1\,,\,2$ because Int is *not* disjoint to Int.

3.4 SYNTAX AND SEMANTICS OF λ_{co}

We elaborate well-typed source expression E into target terms e. Our target language λ_{co} is the standard simply-typed call-by-value lambda calculus extended with products and coercions. The syntax of λ_{co} is shown in Fig. 3.6. The meta-function $|\cdot|$ shown in Definition 5 transforms λ_i^+ types to λ_{co} types. It is worth pointing out that we use the erasure semantics to model record labels, i.e., labels are erased during elaboration; this is different from the original publication [Bi et al. 2018] where records are also present in the target. Both work

$\boxed{A * B}$ <div style="text-align:right">*(Disjointness)*</div>

D-TOPL

$$\overline{\top * A}$$

D-TOPR

$$\overline{A * \top}$$

D-ARR
$$\frac{A_2 * B_2}{A_1 \to A_2 * B_1 \to B_2}$$

D-ANDL
$$\frac{A_1 * B \qquad A_2 * B}{A_1 \,\&\, A_2 * B}$$

D-ANDR
$$\frac{A * B_1 \qquad A * B_2}{A * B_1 \,\&\, B_2}$$

D-RCDEQ
$$\frac{A * B}{\{l : A\} * \{l : B\}}$$

D-RCDNEQ
$$\frac{l_1 \neq l_2}{\{l_1 : A\} * \{l_2 : B\}}$$

D-AX
$$\frac{A *_{ax} B}{A * B}$$

$\boxed{A *_{ax} B}$ <div style="text-align:right">*(Disjointness axioms)*</div>

DAX-SYM
$$\frac{B *_{ax} A}{A *_{ax} B}$$

DAX-INTARR
$$\overline{\mathsf{Int} *_{ax} A_1 \to A_2}$$

DAX-INTRCD
$$\overline{\mathsf{Int} *_{ax} \{l : A\}}$$

DAX-ARRRCD
$$\overline{A_1 \to A_2 *_{ax} \{l : B\}}$$

<div style="text-align:center">Figure 3.5: Disjointness</div>

Types	τ	::=	$\mathsf{Int} \mid \langle\rangle \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2$
Terms	e	::=	$x \mid i \mid \langle\rangle \mid \lambda x.\, e \mid e_1\, e_2 \mid \langle e_1, e_2 \rangle \mid co\, e$
Coercions	co	::=	$\mathsf{id} \mid co_1 \circ co_2 \mid \mathsf{top} \mid co_1 \to co_2 \mid \langle co_1, co_2 \rangle \mid \pi_1 \mid \pi_2$
			$\mid\ \mathsf{dist}_\to \mid \mathsf{top}_\to$
Values	v	::=	$\langle\rangle \mid i \mid \lambda x.\, e \mid \langle v_1, v_2 \rangle \mid (co_1 \to co_2)\, v \mid \mathsf{dist}_\to v \mid \mathsf{top}_\to v$
Typing contexts	Ψ	::=	$\bullet \mid \Psi, x : \tau$
Evaluation contexts	$\mathcal{E}$	::=	$[\cdot] \mid \mathcal{E}\, e \mid v\, \mathcal{E} \mid \langle \mathcal{E}, e \rangle \mid \langle v, \mathcal{E} \rangle \mid co\, \mathcal{E}$

<div style="text-align:center">Figure 3.6: Syntax of λ_{co}</div>

fine in λ_i^+, but discarding records makes the target calculus a bit simpler. The notation $|\cdot|$ is also overloaded for translating source contexts Γ to target contexts Ψ.

Definition 5 (Type translation $|\cdot|$ from λ_i^+ to λ_{co}).

$\|\mathsf{Int}\|$	$=$	Int	$\|\top\|$	$=$	$\langle\rangle$
$\|A \to B\|$	$=$	$\|A\| \to \|B\|$	$\|A \,\&\, B\|$	$=$	$\|A\| \times \|B\|$
$\|\{l : A\}\|$	$=$	$\|A\|$			

3.4.1 EXPLICIT COERCIONS AND COERCIVE SUBTYPING

The separate syntactic category for *explicit coercions* is a distinctive difference from the prior work [Alpuim et al. 2017; Oliveira et al. 2016] (in which they are regular terms). Our coercions are based on those of Henglein [1994], and we add more forms due to our extra subtyping rules. Metavariable *co* (hence the *co* in λ_{co}) ranges over coercions. As a cognitive

Coercion	Term	Coercion	Term
id	$\lambda x.\, x$	$co_1 \circ co_2$	$\lambda x.\, co_1\,(co_2\, x)$
top	$\lambda x.\, \langle\rangle$	$co_1 \to co_2$	$\lambda f.\, \lambda x.\, co_2\,(f\,(co_1\, x))$
π_1	$\lambda x.\, \pi_1\, x$	π_2	$\lambda x.\, \pi_2\, x$
$\langle co_1, co_2 \rangle$	$\lambda x.\, \langle co_1\, x, co_2\, x \rangle$	dist$_\to$	$\lambda x.\, \lambda y.\, \langle (\pi_1\, x)\, y, (\pi_2\, x)\, y \rangle$
top$_\to$	$\lambda x.\, \lambda y.\, \langle\rangle$		

Table 3.1: Correspondence between coercions and terms

aid, we can mentally "desugar" coercions to regular terms, which might help understand the dynamic semantics of coercions. The correspondence between coercions and terms is shown in Table 3.1. In essence, coercions express the conversion of a term from one type to another. Because of the addition of coercions, the grammar contains explicit coercion applications $co\, e$ as a term, and various unsaturated coercion applications as values. Explicit coercions are useful for the new semantic proof of coherence based on logical relations. The subtyping judgment in Fig. 3.3 has the form $A <: B \rightsquigarrow co$, which says that subtyping $A <: B$ produces a coercion co that converts terms of type $|A|$ to type $|B|$. Each subtyping rule has its own specific form of coercion.

3.4.2 Typing of λ_{co}

The typing of λ_{co} has the form $\Psi \vdash e : \tau$, and is entirely standard. Only the typing of coercion applications, shown below, deserves attention:

$$\frac{\text{T-CAPP}}{\Psi \vdash e : \tau_1 \qquad co :: \tau_1 \triangleright \tau_2}{\Psi \vdash co\, e : \tau_2}$$

Here the judgment $co :: \tau_1 \triangleright \tau_2$ expresses the typing of coercions, which are essentially functions from τ_1 to τ_2. Their typing rules correspond exactly to the subtyping rules of λ_i^+, as shown in Fig. 3.7.

3.4.3 Dynamic Semantics

The dynamic semantics of λ_{co} is shown in Fig. 3.8. We write $e \longrightarrow e'$ for reduction of expressions. The first three lines are reduction rules for coercions. They do not contribute to computation but merely rearrange coercions. Our coercion reduction rules are quite standard but not efficient in terms of space. Nevertheless, there is existing work on space-efficient coercions [Herman et al. 2010; Siek et al. 2015a], which should be applicable to our work as

$\boxed{co :: \tau_1 \triangleright \tau_2}$ *(Coercion typing)*

CT-REFL

$$\overline{\mathsf{id} :: \tau \triangleright \tau}$$

CT-TRANS
$$\frac{co_1 :: \tau_2 \triangleright \tau_3 \qquad co_2 :: \tau_1 \triangleright \tau_2}{co_1 \circ co_2 :: \tau_1 \triangleright \tau_3}$$

CT-TOP
$$\overline{\mathsf{top} :: \tau \triangleright \langle\rangle}$$

CT-TOPARR
$$\overline{\mathsf{top}_\rightarrow :: \langle\rangle \triangleright \langle\rangle \rightarrow \langle\rangle}$$

CT-ARR
$$\frac{co_1 :: \tau_1' \triangleright \tau_1 \qquad co_2 :: \tau_2 \triangleright \tau_2'}{co_1 \rightarrow co_2 :: \tau_1 \rightarrow \tau_2 \triangleright \tau_1' \rightarrow \tau_2'}$$

CT-PAIR
$$\frac{co_1 :: \tau_1 \triangleright \tau_2 \qquad co_2 :: \tau_1 \triangleright \tau_3}{\langle co_1, co_2 \rangle :: \tau_1 \triangleright \tau_2 \times \tau_3}$$

CT-PROJL
$$\overline{\pi_1 :: \tau_1 \times \tau_2 \triangleright \tau_1}$$

CT-PROJR
$$\overline{\pi_2 :: \tau_1 \times \tau_2 \triangleright \tau_2}$$

CT-DISTARR
$$\overline{\mathsf{dist}_\rightarrow :: (\tau_1 \rightarrow \tau_2) \times (\tau_1 \rightarrow \tau_3) \triangleright \tau_1 \rightarrow \tau_2 \times \tau_3}$$

Figure 3.7: Coercion typing

well. Rule R-APP is the usual β-rule that performs actual computation, and rule R-CTXT handles reduction under an evaluation context. As standard, $\longrightarrow^*$ is the reflexive, transitive closure of $\longrightarrow$. Now we can show that λ_{co} is type safe:

Theorem 3.1 (☞ Preservation). *If $\bullet \vdash e : \tau$ and $e \longrightarrow e'$, then $\bullet \vdash e' : \tau$.*

Theorem 3.2 (☞ Progress). *If $\bullet \vdash e : \tau$, then either e is a value, or there exists e' such that $e \longrightarrow e'$.*

3.4.4 ELABORATION SEMANTICS

We are now in a position to explain the elaboration judgments $\Gamma \vdash E \Rightarrow A \leadsto e$ and $\Gamma \vdash E \Leftarrow A \leadsto e$ in Fig. 3.4. The only interesting rule is rule T-SUB, which applies the coercion co produced by subtyping to the target term e to form a coercion application $co\,e$. All the other rules do straightforward translations between source and target expressions.

To conclude, we show two lemmas that relate source expressions to target terms.

Lemma 3.3 (☞ Coercions preserve types). *If $A <: B \leadsto co$, then $co :: |A| \triangleright |B|$.*

Proof. By structural induction on the derivation of subtyping. □

Lemma 3.4 (☞ Elaboration soundness). *We have that:*

- *If $\Gamma \vdash E \Rightarrow A \leadsto e$, then $|\Gamma| \vdash e : |A|$.*

- *If $\Gamma \vdash E \Leftarrow A \leadsto e$, then $|\Gamma| \vdash e : |A|$.*

$\boxed{e \longrightarrow e'}$ (*Single-step reduction*)

R-ID
$$\mathsf{id}\, v \longrightarrow v$$

R-TRANS
$$(co_1 \circ co_2)\, v \longrightarrow co_1\, (co_2\, v)$$

R-TOP
$$\mathsf{top}\, v \longrightarrow \langle\rangle$$

R-TOPARR
$$(\mathsf{top}_\rightarrow \langle\rangle)\, \langle\rangle \longrightarrow \langle\rangle$$

R-PAIR
$$\langle co_1, co_2\rangle\, v \longrightarrow \langle co_1\, v, co_2\, v\rangle$$

R-ARR
$$((co_1 \rightarrow co_2)\, v_1)\, v_2 \longrightarrow co_2\, (v_1\, (co_1\, v_2))$$

R-DISTARR
$$(\mathsf{dist}_\rightarrow \langle v_1, v_2\rangle)\, v_3 \longrightarrow \langle v_1\, v_3, v_2\, v_3\rangle$$

R-PROJL
$$\pi_1\, \langle v_1, v_2\rangle \longrightarrow v_1$$

R-PROJR
$$\pi_2\, \langle v_1, v_2\rangle \longrightarrow v_2$$

R-APP
$$(\lambda x.\, e)\, v \longrightarrow [v/x]e$$

R-CTXT
$$\frac{e \longrightarrow e'}{\mathcal{E}[e] \longrightarrow \mathcal{E}[e']}$$

Figure 3.8: Dynamic semantics of λ_{co}

Proof. By structural induction on the derivation of typing. $\qquad\square$

As a corollary, λ_i^+ is type safe due to Theorems 3.1 and 3.2 and Lemma 3.4.

3.5 COMPARISON WITH λ_i

In this section we identify major differences from λ_i (cf. Fig. 2.1), which, when taken together, yield a simpler and more elegant system. The differences may seem superficial, but they have a strong effect on coherence, our major topic in Chapter 5.

NO ORDINARY TYPES. Apart from the extra subtyping rules, there is an important difference from the λ_i subtyping relation. The subtyping relation of λ_i employs an auxiliary unary relation "*A* ordinary" (cf. rules SI-ANDL and SI-ANDR in Fig. 2.1). Ordinary types are employed to ensure determinism of subtyping (hence uniqueness of the elaboration), which plays a fundamental role for ensuring coherence and obtaining an algorithm. In λ_i^+, we show that determinism is too strong of a requirement. As we shall see in Chapter 5, it suffices to base the notion of coherence on contextual equivalence. As such, the λ_i^+ calculus discards the notion of ordinary types completely; this yields a clean and elegant formulation of the subtyping relation. Another minor difference is that due to the addition of the transitivity axiom (rule S-TRANS), rules S-ANDL and S-ANDR are simplified: an intersection type $A_1 \,\&\, A_2$ is a subtype of both A_1 and A_2, instead of the more general form $A_1 \,\&\, A_2 <: A_3$.

NO TOP-LIKE TYPES.   Another notable difference from the coercive subtyping of λ_i is that, because of the syntactic proof method, they have a special treatment for coercions of *top-like types* (see the coercion parts in rules Sɪ-ᴀɴᴅL and Sɪ-ᴀɴᴅR in Fig. 2.1). Top-like types will introduce non-determinism during subtyping, thus would potentially endanger coherence. However, as Oliveira et al. [2016] observed, any coercions for top-like types are unique, even if multiple derivations exist. For λ_i^+, as with ordinary types, we do not need this kind of ad-hoc treatment; for subtyping purposes, top-like types are handled like all other types. However, as we shall see in Chapter 4, top-like types are useful in the disjointness relation when we add the bottom type.

NO WELL-FORMEDNESS JUDGMENT.   A key difference from the type system of λ_i is the complete omission of the well-formedness judgment $\Gamma \vdash A$, which appears in rule Tɪ-ᴀʙs (our rule T-ᴀʙs) and rule Tɪ-sᴜʙ (our rule T-sᴜʙ). The sole purpose of this judgment is to ensure that all intersection types are disjoint. However, as Chapter 5 will explain, this is unnecessary for coherence, and merely complicates the type system. Thus λ_i^+ discards this well-formedness judgment altogether in favor of a simpler design that is still coherent. As a consequence, λ_i^+ supports *unrestricted intersection types*, and already subsumes λ_i even without BCD subtyping: an expression such as $1 : \mathsf{Int} \,\&\, \mathsf{Int}$ is accepted in λ_i^+ but rejected in λ_i. This simplification is based on an important observation: incoherence can only arise from merges. Therefore disjointness checking is only necessary in rule T-ᴍᴇʀɢᴇ.

3.6 ALGORITHMIC SUBTYPING

This section considers the algorithmic aspects of λ_i^+. The bidirectional type system is syntax directed, so the only source of non-determinism comes from the subtyping relation. In this section, we present an algorithm that implements the subtyping relation. While BCD subtyping is powerful, the presence of a transitivity axiom in the rules means that the subtyping relation is not algorithmic. This raises an obvious question: how to obtain an algorithm for this subtyping relation? Laurent [2012b] has shown that simply dropping the transitivity rule from BCD subtyping is not possible without losing expressiveness. Hence, this avenue for obtaining an algorithm is not available. In a 2012 article [Laurent 2012a], Laurent defined BCD subtyping without transitivity, but the system still does not deliver an algorithm. Only quite recently, Laurent [2018] presents a general approach to defining a BCD-like subtyping relation that enjoys the "sub-formula property" (read decidability). We adapt Pierce's decision procedure [Pierce 1989] for a subtyping system (closely related to BCD) to obtain a sound and complete algorithm for our BCD extension. Our algorithm extends Pierce's deci-

$$\boxed{\mathcal{Q} \vdash A \prec: B \leadsto co}$$ *(Algorithmic subtyping)*

A-AND
$$\dfrac{\mathcal{Q} \vdash A \prec: B_1 \leadsto co_1 \qquad \mathcal{Q} \vdash A \prec: B_2 \leadsto co_2}{\mathcal{Q} \vdash A \prec: B_1 \,\&\, B_2 \leadsto [\![\mathcal{Q}]\!]^{\&} \circ \langle co_1, co_2 \rangle}$$

A-ARR
$$\dfrac{\mathcal{Q}, B_1 \vdash A \prec: B_2 \leadsto co}{\mathcal{Q} \vdash A \prec: B_1 \rightarrow B_2 \leadsto co}$$

A-RCD
$$\dfrac{\mathcal{Q}, l \vdash A \prec: B \leadsto co}{\mathcal{Q} \vdash A \prec: \{l : B\} \leadsto co}$$

A-TOP
$$\dfrac{}{\mathcal{Q} \vdash A \prec: \top \leadsto [\![\mathcal{Q}]\!]^{\top} \circ \mathsf{top}}$$

A-INT
$$\dfrac{}{[] \vdash \mathsf{Int} \prec: \mathsf{Int} \leadsto \mathsf{id}}$$

A-ARRINT
$$\dfrac{[] \vdash A \prec: A_1 \leadsto co_1 \qquad \mathcal{Q} \vdash A_2 \prec: \mathsf{Int} \leadsto co_2}{A, \mathcal{Q} \vdash A_1 \rightarrow A_2 \prec: \mathsf{Int} \leadsto co_1 \rightarrow co_2}$$

A-RCDINT
$$\dfrac{\mathcal{Q} \vdash A \prec: \mathsf{Int} \leadsto co}{l, \mathcal{Q} \vdash \{l : A\} \prec: \mathsf{Int} \leadsto co}$$

A-ANDINT1
$$\dfrac{\mathcal{Q} \vdash A_1 \prec: \mathsf{Int} \leadsto co}{\mathcal{Q} \vdash A_1 \,\&\, A_2 \prec: \mathsf{Int} \leadsto co \circ \pi_1}$$

A-ANDINT2
$$\dfrac{\mathcal{Q} \vdash A_2 \prec: \mathsf{Int} \leadsto co}{\mathcal{Q} \vdash A_1 \,\&\, A_2 \prec: \mathsf{Int} \leadsto co \circ \pi_2}$$

Figure 3.9: Algorithmic subtyping of λ_i^+

sion procedure with subtyping of records and coercions. We prove in Coq that the algorithm is sound and complete with respect to the declarative specification. At the same time we find some errors and missing lemmas in Pierce's original manual proofs.

3.6.1 THE SUBTYPING ALGORITHM

While Fig. 3.3 is a fine specification of how subtyping should behave, it cannot be read directly as a subtyping algorithm for two reasons: (1) the conclusions of rules S-REFL and S-TRANS overlap with the other rules, and (2) the premises of rule S-TRANS mention a type that does not appear in the conclusion. Simply dropping the two offending rules from the system is not possible without losing expressivity Laurent [2012b]. Thus we need a different approach. Figure 3.9 shows the algorithmic subtyping judgment $\mathcal{Q} \vdash A \prec: B \leadsto co$. This judgment is the algorithmic counterpart of the declarative judgment $A <: \mathcal{Q} \Rightarrow B \leadsto co$, where $\mathcal{Q}$ stands for a queue used to track domain types and labels. Definition 6 converts $\mathcal{Q} \Rightarrow A$ to a valid type. For instance, if $\mathcal{Q} = A, B, \{l\}$, then $\mathcal{Q} \Rightarrow C$ abbreviates $A \rightarrow B \rightarrow \{l : C\}$.

Definition 6. $\mathcal{Q} \Rightarrow A$ is inductively defined as follows:

$$
\begin{aligned}
[] \Rightarrow A &= A \\
(B, \mathcal{Q}) \Rightarrow A &= B \rightarrow (\mathcal{Q} \Rightarrow A) \\
(l, \mathcal{Q}) \Rightarrow A &= \{l : \mathcal{Q} \Rightarrow A\}
\end{aligned}
$$

The basic idea of $\mathcal{Q} \vdash A \prec: B \rightsquigarrow co$ is to first perform a structural analysis of B, which descends into both sides of &'s (rule A-AND), into the right side of $\rightarrow$'s (rule A-ARR), and into the fields of records (rule A-RCD) until it reaches one of the two base cases, $\top$ or Int. If the base case is $\top$, the subtyping holds trivially (rule A-TOP). If the base case is Int, the algorithm performs a structural analysis of A, in which $\mathcal{Q}$ plays an important role. The left sides of $\rightarrow$'s are pushed onto $\mathcal{Q}$ as they are encountered in B and popped off again later, left to right, as $\rightarrow$'s are encountered in A (rule A-ARRINT). Similarly, the labels are pushed onto $\mathcal{Q}$ as they are encountered in B and popped off again later, left to right, as records are encountered in A (rule A-RCDINT). The remaining two rules A-ANDINT1 and A-ANDINT2 are similar to their declarative counterparts. Let us illustrate the algorithm in Fig. 3.10 with an example derivation (for formatting reasons we use I and S to denote Int and String respectively), which is essentially the one used by the add field in Section 3.2. The reader can try to give a corresponding derivation using the declarative subtyping and see how rule S-TRANS plays an essential role there.

Remark. Our algorithmic rules are still not deterministic (rules A-ANDINT1 and A-ANDINT2 are overlapping). In other words, to check whether $A_1 \& A_2 \prec:$ Int, we need to check if $A_1 \prec:$ Int or $A_2 \prec:$ Int. In our prototype, this is implemented using backtracking.

Now consider the coercions. Algorithmic subtyping uses the same set of coercions as declarative subtyping. However, because algorithmic subtyping has a different structure, the rules generate slightly more complicated coercions. Two meta-functions $[\![\cdot]\!]_\top$ and $[\![\cdot]\!]_\&$ used in rules A-TOP and A-AND respectively, are meant to generate correct forms of coercions. They are defined recursively on $\mathcal{Q}$ and are shown in Fig. 3.11.

3.6.2 CORRECTNESS OF THE ALGORITHM

To establish the correctness of the algorithm, we must show that the algorithm is both sound and complete with respect to the declarative specification. While soundness follows quite easily, completeness is much harder. The proof of completeness essentially follows that of Pierce [1989] in that we need to show the algorithmic subtyping is reflexive and transitive.

SOUNDNESS OF THE ALGORITHM. The following two lemmas connect the declarative subtyping with the meta-functions.

$$
\cfrac{\cfrac{D \qquad D'}{\{l\}, \mathsf{I}\,\&\,\mathsf{S}, \mathsf{I}\,\&\,\mathsf{S} \vdash \{l : \mathsf{I} \to \mathsf{I} \to \mathsf{I}\}\,\&\,\{l : \mathsf{S} \to \mathsf{S} \to \mathsf{S}\} \prec: \mathsf{I}\,\&\,\mathsf{S}}\ \text{A-\textsc{and}}}{\cfrac{\{l\}, \mathsf{I}\,\&\,\mathsf{S} \vdash \{l : \mathsf{I} \to \mathsf{I} \to \mathsf{I}\}\,\&\,\{l : \mathsf{S} \to \mathsf{S} \to \mathsf{S}\} \prec: \mathsf{I}\,\&\,\mathsf{S} \to \mathsf{I}\,\&\,\mathsf{S}}{\cfrac{\{l\} \vdash \{l : \mathsf{I} \to \mathsf{I} \to \mathsf{I}\}\,\&\,\{l : \mathsf{S} \to \mathsf{S} \to \mathsf{S}\} \prec: \mathsf{I}\,\&\,\mathsf{S} \to \mathsf{I}\,\&\,\mathsf{S} \to \mathsf{I}\,\&\,\mathsf{S}}{[]\vdash \{l : \mathsf{I} \to \mathsf{I} \to \mathsf{I}\}\,\&\,\{l : \mathsf{S} \to \mathsf{S} \to \mathsf{S}\} \prec: \{l : \mathsf{I}\,\&\,\mathsf{S} \to \mathsf{I}\,\&\,\mathsf{S} \to \mathsf{I}\,\&\,\mathsf{S}\}}\ \text{A-\textsc{rcd}}}\ \text{A-\textsc{arr}}}\ \text{A-\textsc{arr}}}
$$

$$
D = \cfrac{\cfrac{\cfrac{\cfrac{[]\vdash \mathsf{I} \prec: \mathsf{I}}{[]\vdash \mathsf{I}\,\&\,\mathsf{S} \prec: \mathsf{I}}\quad \cfrac{\cfrac{[]\vdash \mathsf{I} \prec: \mathsf{I}}{[]\vdash \mathsf{I}\,\&\,\mathsf{S} \prec: \mathsf{I}}\quad []\vdash \mathsf{I} \prec: \mathsf{I}}{\mathsf{I}\,\&\,\mathsf{S} \vdash \mathsf{I} \to \mathsf{I} \prec: \mathsf{I}}}{\mathsf{I}\,\&\,\mathsf{S}, \mathsf{I}\,\&\,\mathsf{S} \vdash \mathsf{I} \to \mathsf{I} \to \mathsf{I} \prec: \mathsf{I}}\ \text{A-\textsc{arrInt}}}{\{l\}, \mathsf{I}\,\&\,\mathsf{S}, \mathsf{I}\,\&\,\mathsf{S} \vdash \{l : \mathsf{I} \to \mathsf{I} \to \mathsf{I}\} \prec: \mathsf{I}}\ \text{A-\textsc{rcdInt}}}{\{l\}, \mathsf{I}\,\&\,\mathsf{S}, \mathsf{I}\,\&\,\mathsf{S} \vdash \{l : \mathsf{I} \to \mathsf{I} \to \mathsf{I}\}\,\&\,\{l : \mathsf{S} \to \mathsf{S} \to \mathsf{S}\} \prec: \mathsf{I}}\ \text{A-\textsc{andInt1}}
$$

$$
D' = \cfrac{\cfrac{\cfrac{\cfrac{[]\vdash \mathsf{S} \prec: \mathsf{S}}{[]\vdash \mathsf{I}\,\&\,\mathsf{S} \prec: \mathsf{S}}\quad \cfrac{\cfrac{[]\vdash \mathsf{S} \prec: \mathsf{S}}{[]\vdash \mathsf{I}\,\&\,\mathsf{S} \prec: \mathsf{S}}\quad []\vdash \mathsf{S} \prec: \mathsf{S}}{\mathsf{I}\,\&\,\mathsf{S} \vdash \mathsf{S} \to \mathsf{S} \prec: \mathsf{S}}}{\mathsf{I}\,\&\,\mathsf{S}, \mathsf{I}\,\&\,\mathsf{S} \vdash \mathsf{S} \to \mathsf{S} \to \mathsf{S} \prec: \mathsf{S}}\ \text{A-\textsc{arrInt}}}{\{l\}, \mathsf{I}\,\&\,\mathsf{S}, \mathsf{I}\,\&\,\mathsf{S} \vdash \{l : \mathsf{S} \to \mathsf{S} \to \mathsf{S}\} \prec: \mathsf{S}}\ \text{A-\textsc{rcdInt}}}{\{l\}, \mathsf{I}\,\&\,\mathsf{S}, \mathsf{I}\,\&\,\mathsf{S} \vdash \{l : \mathsf{I} \to \mathsf{I} \to \mathsf{I}\}\,\&\,\{l : \mathsf{S} \to \mathsf{S} \to \mathsf{S}\} \prec: \mathsf{S}}\ \text{A-\textsc{andInt2}}
$$

Figure 3.10: Example derivation

$$
\begin{aligned}
[\![\,[]\,]\!]^{\top} &= \mathsf{top} \\
[\![\,l, \mathcal{Q}\,]\!]^{\top} &= [\![\mathcal{Q}]\!]^{\top} \circ \mathsf{id} \\
[\![\,A, \mathcal{Q}\,]\!]^{\top} &= (\mathsf{top} \to [\![\mathcal{Q}]\!]^{\top}) \circ \mathsf{top}_{\to}
\end{aligned}
\qquad\qquad
\begin{aligned}
[\![\,[]\,]\!]^{\&} &= \mathsf{id} \\
[\![\,l, \mathcal{Q}\,]\!]^{\&} &= [\![\mathcal{Q}]\!]^{\&} \circ \mathsf{id} \\
[\![\,A, \mathcal{Q}\,]\!]^{\&} &= (\mathsf{id} \to [\![\mathcal{Q}]\!]^{\&}) \circ \mathsf{dist}_{\to}
\end{aligned}
$$

Figure 3.11: Meta-functions of coercions

Lemma 3.5 (☞). $\top <: \mathcal{Q} \Rightarrow \top \rightsquigarrow [\![\mathcal{Q}]\!]^{\top}$

Proof. By induction on the length of $\mathcal{Q}$. □

Lemma 3.6 (☞). $(\mathcal{Q} \Rightarrow A)\,\&\,(\mathcal{Q} \Rightarrow B) <: \mathcal{Q} \Rightarrow (A\,\&\,B) \rightsquigarrow [\![\mathcal{Q}]\!]^{\&}$

Proof. By induction on the length of $\mathcal{Q}$. □

The proof of soundness is straightforward.

Theorem 3.7 (☞ Soundness). *If $\mathcal{Q} \vdash A \prec: B \rightsquigarrow co$ then $A <: \mathcal{Q} \Rightarrow B \rightsquigarrow co$.*

Proof. By induction on the derivation of the algorithmic subtyping and applying Lemmas 3.5 and 3.6 where appropriate. □

COMPLETENESS OF THE ALGORITHM.   Completeness, however, is much harder. The reason is that, due to the use of $\mathcal{Q}$, reflexivity and transitivity are not entirely obvious. We need to strengthen the induction hypothesis by introducing the notion of a set, $\mathcal{U}(A)$, of "reflexive supertypes" of A, as defined below:

$$\mathcal{U}(\top) \triangleq \{\top\} \qquad \mathcal{U}(\mathsf{Int}) \triangleq \{\mathsf{Int}\} \qquad \mathcal{U}(\{l : A\}) \triangleq \{\{l : B\} \mid B \in \mathcal{U}(A)\}$$

$$\mathcal{U}(A \,\&\, B) \triangleq \mathcal{U}(A) \cup \mathcal{U}(B) \cup \{A \,\&\, B\} \qquad \mathcal{U}(A \to B) \triangleq \{A \to C \mid C \in \mathcal{U}(B)\}$$

We show two lemmas about $\mathcal{U}(A)$ that are crucial in the subsequent proofs.

Lemma 3.8 (⟦☞). $A \in \mathcal{U}(A)$

Proof. By induction on the structure of A. □

Lemma 3.9 (⟦☞). *If $A \in \mathcal{U}(B)$ and $B \in \mathcal{U}(C)$, then $A \in \mathcal{U}(C)$.*

Proof. By induction on the structure of B. □

Remark. Lemma 3.9 is not found in Pierce's proofs [Pierce 1989], which is crucial in Lemma 3.10, from which reflexivity (Lemma 3.11) follows immediately.

Lemma 3.10 (⟦☞). *If $\mathcal{Q} \Rightarrow B \in \mathcal{U}(A)$ then there exists co such that $\mathcal{Q} \vdash A \prec: B \rightsquigarrow co$.*

Proof. By induction on $\mathsf{size}(A) + \mathsf{size}(B) + \mathsf{size}(\mathcal{Q})$. □

Now it immediately follows that the algorithmic subtyping is reflexive.

Lemma 3.11 (⟦☞ Reflexivity). *For every A there exists co such that $[] \vdash A \prec: A \rightsquigarrow co$.*

Proof. Immediate from Lemma 3.8 and Lemma 3.10. □

The proof of transitivity is, to quote Pierce [1989], typically "the hardest single piece" of metatheory. We omit the details here and refer the interested reader to our Coq development.

Lemma 3.12 (⟦☞ Transitivity). *If $[] \vdash A_1 \prec: A_2 \rightsquigarrow co_1$ and $[] \vdash A_2 \prec: A_3 \rightsquigarrow co_2$, then there exists co such that $[] \vdash A_1 \prec: A_3 \rightsquigarrow co$.*

With reflexivity and transitivity in position, we show the main theorem.

Theorem 3.13 (⟦☞ Completeness). *If $A <: B \rightsquigarrow co$ then there exists co' such that $[] \vdash A \prec: B \rightsquigarrow co'$.*

Proof. By induction on the derivation of the declarative subtyping and applying Lemmas 3.11 and 3.12 where appropriate. □

Remark. Pierce's proof is wrong [Pierce 1989, pp. 20, Case F] in the case

$$
\text{S-\textsc{arr}} \\
\frac{B_1 <: A_1 \rightsquigarrow co_1 \qquad A_2 <: B_2 \rightsquigarrow co_2}{A_1 \rightarrow A_2 <: B_1 \rightarrow B_2 \rightsquigarrow co_1 \rightarrow co_2}
$$

where he concludes from the inductive hypotheses $[] \vdash B_1 \prec: A_1$ and $[] \vdash A_2 \prec: B_2$ that $[] \vdash A_1 \rightarrow A_2 \prec: B_1 \rightarrow B_2$ (his rules 6a and 3). However his rule 6a (our rule A-ARRINT) only works for *primitive types*, and is thus not applicable in this case. Instead we need a few technical lemmas to support the argument.

4   Semantics of the F_i^+ Calculus

In this chapter, we present F_i^+, the first typed calculus combining disjoint polymorphism with BCD subtyping. F_i^+ is essentially a hybrid of λ_i^+ (which does not support parametric polymorphism) and F_i (which does not account for BCD subtyping). As we will see, the combination is very expressive, enabling improved compositional designs and supporting automated composition of interpretations in programming techniques like object algebras [Oliveira and Cook 2012] and finally tagless [Carette et al. 2009]. Furthermore, as we will show in Chapter 7, F_i^+ is able to encode sophisticated concepts such as mixins/traits and dynamic inheritance. Unfortunately, the combination also introduces non-trivial complications. The main technical challenge (like for most other calculi with disjoint intersection types) is the proof of coherence—our main topic in Chapter 6.

The formalization and metatheory of F_i^+ are a significant advance over its predecessor F_i. Besides the support for distributive subtyping, F_i^+ removes several restrictions imposed by the syntactic coherence proof in F_i. In particular F_i^+ supports unrestricted intersections, which are forbidden in F_i. Unrestricted intersections enable, for example, encoding certain forms of bounded quantification [Pierce 1991]. Moreover the new proof method is more robust with respect to language extensions. For instance, F_i^+ supports the bottom type without significant complications in the proofs, while it was a challenging open problem in F_i. A final interesting aspect is that F_i^+'s type-checking is decidable. In the design space of languages with polymorphism and subtyping, similar mechanisms have been known to lead to undecidability. Pierce's seminal paper "*Bounded quantification is undecidable*" [Pierce 1994] shows that the contravariant subtyping rule for bounded quantification in $\mathsf{F}_{<:}$ leads to undecidability of subtyping. In F_i^+ the contravariant rule for disjoint quantification retains decidability. Since with unrestricted intersections F_i^+ can express several use cases of bounded quantification, F_i^+ could be an interesting and decidable alternative to $\mathsf{F}_{<:}$.

4.1 Motivation: Compositional Programming

To demonstrate the compositional properties of F_i^+ we use Gibbons and Wu's shallow embeddings of parallel prefix circuits [Gibbons and Wu 2014]. By means of several different

shallow embeddings, we first illustrate the short-comings of a state-of-the-art compositional approach, namely a *finally tagless* encoding [Carette et al. 2009] in Haskell. Next we show how parametric polymorphism and distributive intersection types provide a more elegant and compact solution in F_i^+ (with a few convenient source level constructs).

4.1.1 A FINALLY TAGLESS ENCODING IN HASKELL

The circuit DSL represents networks that map a number of inputs (known as the width) of some type A onto the same number of outputs of the same type. The outputs combine (with repetitions) one or more inputs using a binary associative operator $\oplus : A \times A \to A$. A particularly interesting class of circuits that can be expressed in the DSL are *parallel prefix circuits*. These represent computations that take $n > 0$ inputs $x_1, \ldots, x_n$ and produce n outputs $y_1, \ldots, y_n$, where $y_i = x_1 \oplus x_2 \oplus \ldots \oplus x_i$.

The DSL features 5 language primitives: two basic circuit constructors and three circuit combinators. These are captured in the Haskell type class `Circuit`:

```
class Circuit c where
  identity :: Int → c
  fan      :: Int → c
  beside   :: c → c → c
  above    :: c → c → c
  stretch  :: [Int] → c → c
```

An `identity` circuit with n inputs x_i, has n outputs $y_i = x_i$. A `fan` circuit has n inputs x_i and n outputs y_i, where $y_1 = x_1$ and $y_j = x_1 \oplus x_j$ $(j > 1)$. The binary `beside` combinator puts two circuits in parallel; the combined circuit takes the inputs of both circuits to the outputs of both circuits. The binary `above` combinator connects the outputs of the first circuit to the inputs of the second; the width of both circuits has to be same. Finally, `stretch ws c` interleaves the wires of circuit `c` with bundles of additional wires that map their input straight on their output. The `ws` parameter specifies the width of the consecutive bundles; the ith wire of `c` is preceded by a bundle of width $ws_i - 1$.

BASIC WIDTH AND DEPTH EMBEDDINGS.   Figure 4.1 shows two simple shallow embeddings, which represent a circuit respectively in terms of its width and its depth. The former denotes the number of inputs/outputs of a circuit, while the latter is the maximal number of $\oplus$ operators between any input and output. Both definitions follow the same setup: a new Haskell datatype (`Width`/`Depth`) wraps the primitive result value and provides an instance of the `Circuit` type class that interprets the 5 DSL primitives accordingly. The following code creates a so-called Brent-Kung parallel prefix circuit [Brent and Kung 1980]:

```
newtype Width = W {width :: Int}    newtype Depth = D {depth :: Int}

instance Circuit Width where        instance Circuit Depth where
  identity n   = W n                  identity n   = D 0
  fan n        = W n                  fan n        = D 1
  beside c1 c2 =                      beside c1 c2 =
    W (width c1 + width c2)             D (max (depth c1) (depth c2))
  above c1 c2  = c1                   above c1 c2  = D (depth c1 + depth c2)
  stretch ws c = W (sum ws)           stretch ws c = c
```

      (a) Width embedding   (b) Depth embedding

Figure 4.1: Two finally tagless embeddings of circuits.

```
e1 :: Width
e1 = above (beside (fan 2) (fan 2))
       (above (stretch [2, 2] (fan 2))
          (beside (beside (identity 1) (fan 2)) (identity 1)))
```

Here e1 evaluates to W {width = 4}. If we want to know the depth of the circuit, we have to change type signature to Depth.

INTERPRETING MULTIPLE WAYS.   Fortunately, with the help of polymorphism we can define a type of circuits that support multiple interpretations at once.

```
type DCircuit = forall c. Circuit c ⇒ c
```

This way we can provide a single Brent-Kung parallel prefix circuit definition that can be reused for different interpretations.

```
brentKung :: DCircuit
brentKung = above (beside (fan 2) (fan 2))
              (above (stretch [2, 2] (fan 2))
                 (beside (beside (identity 1) (fan 2)) (identity 1)))
```

A type annotation then selects the desired interpretation. For instance, brentKung :: Width yields the width and brentKung :: Depth the depth.

COMPOSITION OF EMBEDDINGS.   What is not ideal in the above code, however, is that the same brentKung circuit is processed twice, if we want to execute both interpretations. We can do better by processing the circuit only once, computing both interpretations simultaneously. The finally tagless encoding achieves this with a boilerplate instance for tuples of interpretations.

```
instance (Circuit c1, Circuit c2) ⇒ Circuit (c1, c2) where
  identity n    = (identity n, identity n)
  fan n         = (fan n, fan n)
  beside c1 c2  = (beside (fst c1) (fst c2), beside (snd c1) (snd c2))
  above c1 c2   = (above (fst c1) (fst c2), above (snd c1) (snd c2))
  stretch ws c  = (stretch ws (fst c), stretch ws (snd c))
```

Now we can get both embeddings simultaneously as follows:

```
e12 :: (Width, Depth)
e12 = brentKung
```

This evaluates to (W {width = 4}, D {depth = 2}).

COMPOSITION OF DEPENDENT INTERPRETATIONS.   The composition above is easy because the two embeddings are orthogonal. In contrast, the composition of dependent interpretations is rather cumbersome in the standard finally tagless setup. An example of the latter is the interpretation of circuits as their well-sizedness, which captures whether circuits are well-formed. This interpretation depends on the interpretation of circuits as their width.[1]

```
data WellSized = WS { wS :: Bool, ox :: Width }
```

```
instance Circuit WellSized where
 identity n    = WS True (identity n)
 fan n         = WS True (fan n)
 beside c1 c2 = WS (wS c1 && wS c2) (beside (ox c1) (ox c2))
 above c1 c2  = WS (wS c1 && wS c2 && width (ox c1) == width (ox c2))
                   (above (ox c1) (ox c2))
 stretch ws c = WS (wS c && length ws==width (ox c)) (stretch ws (ox c))
```

The WellSized datatype represents the well-sizedness of a circuit with a Boolean, and also keeps track of the circuit's width. The 5 primitives compute the well-sizedness in terms of both the width and well-sizedness of the subcomponents. What makes the code cumbersome is that it has to explicitly delegate to the Width interpretation to collect this additional information.

With the help of a substantially more complicated setup that features a dozen Haskell language extensions, and advanced programming techniques, we can make the explicit delegation implicit (see Appendix A). Nevertheless, that approach still requires a lot of boilerplate that needs to be repeated for each DSL, as well as explicit projections that need to be

---

[1] Dependent recursion schemes are also known as *zygomorphism* [Fokkinga 1989] after the ancient Greek word ζυγον for yoke. We have labeled the Width field with ox because it is pulling the yoke.

written in each interpretation. A final remark is that adding new primitives (e.g., a "right stretch" `rstretch` combinator [Hinze 2004]) can also be easily achieved in the finally tagless approach [Kiselyov 2012].

4.1.2 THE F_i^+ ENCODING

The F_i^+ setup of the circuit DSL is similar to the finally tagless approach. Instead of a `Circuit c` type class, there is a `Circuit[C]` type that gathers the 5 circuit primitives in a record. Like in Haskell, the type parameter `C` expresses that the interpretation of circuits is a parameter.

```
type Circuit[C] = {
  identity : Int → C,
  fan      : Int → C,
  beside   : C → C → C,
  above    : C → C → C,
  stretch  : List[Int] → C → C
};
```

As a side note if a new constructor (e.g., `rstretch`) is needed, then this is done by means of intersection types in F_i^+:

```
type NCircuit[C] = Circuit[C] & { rstretch : List[Int] → C → C };
```

BASIC WIDTH AND DEPTH EMBEDDINGS.   Figure 4.2 shows the two basic shallow embeddings for width and depth. In both cases, a named F_i^+ definition replaces the corresponding unnamed Haskell type class instance in providing the implementations of the 5 language primitives for a particular interpretation.

The use of the F_i^+ embeddings is different from that of their Haskell counterparts. Where Haskell implicitly selects the appropriate type class instance based on the available type information, in F_i^+ the programmer explicitly selects the implementation following the style used by object algebras. The following code does this by building a circuit with `l1` (short for `language1`).

```
l1 = language1;
e1 = l1.above (l1.beside (l1.fan 2) (l1.fan 2))
  (l1.above (l1.stretch (cons 2 (cons 2 nil)) (l1.fan 2))
    (l1.beside (l1.beside (l1.identity 1) (l1.fan 2)) (l1.identity 1)));
```

Here `e1` evaluates to `{width = 4}`. If we want to know the depth of the circuit, we have to replicate the code with `language2`.

```
type Width = { width : Int };

language1 : Circuit[Width] = {
  identity (n : Int) = { width = n },
  fan       (n : Int) = { width = n },
  beside (c1 : Width) (c2 : Width) = { width = c1.width + c2.width },
  above  (c1 : Width) (c2 : Width) = { width = c1.width },
  stretch (ws : List[Int]) (c : Width) = { width = sum ws }
};
```

```
type Depth = { depth : Int };

language2 : Circuit[Depth] = {
  identity (n : Int) = { depth = 0 },
  fan       (n : Int) = { depth = 1 },
  beside (c1 : Depth) (c2 : Depth) = { depth = max c1.depth c2.depth },
  above  (c1 : Depth) (c2 : Depth) = { depth = c1.depth + c2.depth },
  stretch (ws : List[Int]) (c : Depth) = { depth = c.depth }
};
```

Figure 4.2: Two F_i^+ embeddings of circuits.

DYNAMICALLY REUSABLE CIRCUITS.   Just like in Haskell, we can use polymorphism to define a type of circuits that can be interpreted with different languages.

```
type DCircuit = { accept : forall C. Circuit[C] → C };
```

In contrast to the Haskell solution, this implementation explicitly accepts the implementation.

```
brentKung : DCircuit = {
  accept C l = l.above (l.beside (l.fan 2) (l.fan 2))
    (l.above (l.stretch (cons 2 (cons 2 nil)) (l.fan 2))
      (l.beside (l.beside (l.identity 1) (l.fan 2)) (l.identity 1)))
};
l2 = language2;
e1 = brentKung.accept Width l1;
e2 = brentKung.accept Depth l2;
```

AUTOMATIC COMPOSITION OF LANGUAGES.   Of course, like in Haskell we can also compute both results simultaneously. However, unlike in Haskell, the composition of the two interpretation requires no boilerplate whatsoever—in particular, there is no F_i^+ counterpart of the `Circuit (c1, c2)` instance. Instead, we can just compose the two interpretations with the term-level merge operator (, ,) and specify the desired type `Circuit[Width & Depth]`.

```
l3 : Circuit[Width & Depth] = l1 ,, l2;
e3 = brentKung.accept (Width & Depth) l3;
```

Here the use of the merge operator creates a term with the intersection type `Circuit[Width]`
`& Circuit[Depth]`. Implicitly, the F_i^+ type system takes care of the details, turning this
intersection type into `Circuit[Width & Depth]`. This is possible because intersection (`&`)
distributes over function and record types.

COMPOSITION OF DEPENDENT INTERPRETATIONS. In F_i^+ the composition scales nicely to
dependent interpretations. For instance, the well-sizedness interpretation can be expressed
without explicit projections.

```
type WellSized = { wS : Bool };

language3 = {
  identity (n : Int) = { wS = true },
  fan      (n : Int) = { wS = true },
  above (c1 : WellSized & Width) (c2 : WellSized & Width) =
    { wS = c1.wS && c2.wS && c1.width == c2.width },
  beside (c1 : WellSized) (c2 : WellSized) = { wS  = c1.wS && c2.wS  },
  stretch (ws : List[Int]) (c : WellSized & Width) =
    { wS = c.wS && length ws == c.width  }
};
```

Here the `WellSized & Width` type in the `above` and `stretch` cases expresses that both the
well-sizedness and width of subcircuits must be given, and that the width implementation
is left as a dependency—when `language3` is used, then the width implementation must be
provided. Again, the distributive properties of `&` in the type system take care of merging the
two interpretations.

```
l4   = language1 ,, language3;
e4   = brentKung.accept (WellSized & Width) l4;
main = e4.wS -- Output: true
```

DISJOINT POLYMORPHISM AND DYNAMIC MERGES. While it may seem from the above ex-
amples that definitions have to be merged statically, F_i^+ in fact supports dynamic merges. For
instance, we can encapsulate the merge operator in the `combine` function while abstracting
over the two components x and y that are merged as well as over their types A and B.

```
combine A [B * A] (x : A) (y : B) = x ,, y;
```

Types	A, B, C	::=	$\mathsf{Int} \mid \top \mid \bot \mid A \rightarrow B \mid A \,\&\, B \mid \{l : A\} \mid \alpha \mid \forall (\alpha * A).\, B$
Expressions	E	::=	$x \mid i \mid \top \mid \lambda x.\, E \mid E_1\, E_2 \mid E_1\,,, E_2 \mid E : A \mid \{l = E\} \mid E.l$
		|	$\Lambda(\alpha * A).\, E \mid E\, A$
Term contexts	Γ	::=	$\bullet \mid \Gamma, x : A$
Type contexts	Δ	::=	$\bullet \mid \Delta, \alpha * A$

Figure 4.3: Syntax of F_i^+

This way the components x and y are only known at runtime and thus the merge can only happen at that time. The types A and B cannot be chosen entirely freely. For instance, if both components would contribute an implementation for the same method, which implementation is provided by the combination would be ambiguous. To avoid this problem the two types A and B have to be *disjoint*. This is expressed in the disjointness constraint * A on the quantifier of the type variable B. If a quantifier mentions no disjointness constraint, like that of A, it defaults to the trivial * ⊤ constraint which implies no restriction. With combine, we can rewrite l3 as follows (note that Width and Depth are disjoint):

```
l3 = combine Circuit[Width] Circuit[Depth] language1 language2;
```

4.2 SYNTAX AND SEMANTICS

Figure 4.3 shows the syntax of F_i^+. Metavariables A, B, C range over types. Apart from λ_i^+ types, F_i^+ also includes type variables α and disjoint quantification $\forall (\alpha * A).\, B$. One novelty in F_i^+ is the addition of the uninhabited bottom type $\bot$. Metavariable E ranges over expressions. We extend λ_i^+ expressions with two standard constructs from System F: type abstractions $\Lambda(\alpha * A).\, E$ and type applications $E\, A$. The former also includes an extra disjointness constraint A associated with the type variable α.

WELL-FORMEDNESS AND UNRESTRICTED INTERSECTIONS. The well-formedness judgments for types $\Delta \vdash A$ in Fig. 4.4 is standard and only enforces well-scoping. This is one of the key differences from F_i, which (like its predecessor λ_i) uses well-formedness to also ensure that all intersection types are disjoint. In other words, while in F_i all valid intersection types must be disjoint, in F_i^+ unrestricted intersection types such as $\mathsf{Int} \,\&\, \mathsf{Int}$ are allowed. More specifically, the well-formedness of intersection types in F_i^+ and F_i is:

WF-F_i^+

$$\frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \,\&\, B}$$

WF-F_i

$$\frac{\Delta \vdash A \qquad \Delta \vdash B \qquad \boxed{\Delta \vdash A * B}}{\Delta \vdash A \,\&\, B}$$

$$\boxed{\Delta \vdash A} \qquad\qquad\qquad\qquad\qquad\qquad \textit{(Well-formedness of types)}$$

SWFT-TOP
$$\overline{\Delta \vdash \top}$$

SWFT-INT
$$\overline{\Delta \vdash \mathsf{Int}}$$

SWFT-VAR
$$\frac{(\alpha * A) \in \Delta}{\Delta \vdash \alpha}$$

SWFT-ARROW
$$\frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \to B}$$

SWFT-ALL
$$\frac{\Delta \vdash A \qquad \Delta, \alpha * A \vdash B}{\Delta \vdash \forall(\alpha * A).\,B}$$

SWFT-AND
$$\frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \,\&\, B}$$

SWFT-RCD
$$\frac{\Delta \vdash A}{\Delta \vdash \{l : A\}}$$

$$\boxed{\Delta \vdash \Gamma} \qquad\qquad\qquad\qquad\qquad \textit{(Well-formedness of value contexts)}$$

SWFE-EMPTY
$$\overline{\Delta \vdash \bullet}$$

SWFE-VAR
$$\frac{\Delta \vdash \Gamma \qquad \Delta \vdash A}{\Delta \vdash \Gamma, x : A}$$

$$\boxed{\vdash \Delta} \qquad\qquad\qquad\qquad\qquad\qquad \textit{(Well-formedness of type contexts)}$$

SWFTE-EMPTY
$$\overline{\vdash \bullet}$$

SWFTE-VAR
$$\frac{\vdash \Delta \qquad \Delta \vdash A}{\vdash \Delta, \alpha * A}$$

Figure 4.4: Well-formedness of types and contexts

Notice that F_i has an extra disjointness condition $\Delta \vdash A * B$ in the premise. This is crucial for F_i's syntactic method for proving coherence, but also burdens the calculus with various syntactic restrictions and complicates its metatheory. For example, it requires extra effort to show that F_i only produces disjoint intersection types. As a consequence, F_i features a *weaker* substitution lemma (note the gray part in Proposition 4.1) than F_i^+ (Lemma 4.2).

Proposition 4.1 (Types are stable under substitution in F_i)**.** *If* $\Delta \vdash A$ *and* $\Delta \vdash B$ *and* $(\alpha * C) \in \Delta$ *and* $\boxed{\Delta \vdash B * C}$ *and well-formed context* $[B/\alpha]\Delta$*, then* $[B/\alpha]\Delta \vdash [B/\alpha]A$*.*

Lemma 4.2 (☞ Types are stable under substitution in F_i^+)**.** *If* $\Delta \vdash A$ *and* $\Delta \vdash B$ *and* $(\alpha * C) \in \Delta$ *and well-formed context* $[B/\alpha]\Delta$*, then* $[B/\alpha]\Delta \vdash [B/\alpha]A$*.*

Proof. By induction on the derivation of $\Delta \vdash A$. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ □

DECLARATIVE SUBTYPING OF F_i^+. We extend the subtyping of λ_i^+ with four new rules: rule S-BOT for the bottom type, and rules S-DISTALL and S-TOPALL for distributivity of disjoint quantification, as shown in Fig. 4.5. $\bot$ is a subtype of all types (rule S-BOT). Subtyping

$\boxed{A <: B \rightsquigarrow co}$ *(Declarative subtyping)*

S-REFL

$$\overline{A <: A \rightsquigarrow \mathsf{id}}$$

S-TRANS

$$\frac{A_2 <: A_3 \rightsquigarrow co_1 \qquad A_1 <: A_2 \rightsquigarrow co_2}{A_1 <: A_3 \rightsquigarrow co_1 \circ co_2}$$

S-TOP

$$\overline{A <: \top \rightsquigarrow \mathsf{top}}$$

S-RCD

$$\frac{A <: B \rightsquigarrow co}{\{l : A\} <: \{l : B\} \rightsquigarrow co}$$

S-ANDL

$$\overline{A_1 \,\&\, A_2 <: A_1 \rightsquigarrow \pi_1}$$

S-ANDR

$$\overline{A_1 \,\&\, A_2 <: A_2 \rightsquigarrow \pi_2}$$

S-ARR

$$\frac{B_1 <: A_1 \rightsquigarrow co_1 \qquad A_2 <: B_2 \rightsquigarrow co_2}{A_1 \rightarrow A_2 <: B_1 \rightarrow B_2 \rightsquigarrow co_1 \rightarrow co_2}$$

S-AND

$$\frac{A_1 <: A_2 \rightsquigarrow co_1 \qquad A_1 <: A_3 \rightsquigarrow co_2}{A_1 <: A_2 \,\&\, A_3 \rightsquigarrow \langle co_1, co_2 \rangle}$$

S-DISTARR

$$\overline{(A_1 \rightarrow A_2) \,\&\, (A_1 \rightarrow A_3) <: A_1 \rightarrow A_2 \,\&\, A_3 \rightsquigarrow \mathsf{dist}_{\rightarrow}}$$

S-TOPARR

$$\overline{\top <: \top \rightarrow \top \rightsquigarrow \mathsf{top}_{\rightarrow}}$$

S-DISTRCD

$$\overline{\{l : A\} \,\&\, \{l : B\} <: \{l : A \,\&\, B\} \rightsquigarrow \mathsf{id}}$$

S-TOPRCD

$$\overline{\top <: \{l : \top\} \rightsquigarrow \mathsf{id}}$$

S-BOT

$$\overline{\bot <: A \rightsquigarrow \mathsf{bot}}$$

S-FORALL

$$\frac{B_1 <: B_2 \rightsquigarrow co \qquad A_2 <: A_1 \rightsquigarrow co'}{\forall (\alpha * A_1).\, B_1 <: \forall (\alpha * A_2).\, B_2 \rightsquigarrow co_{\forall}}$$

S-TOPALL

$$\overline{\top <: \forall (\alpha * \top).\, \top \rightsquigarrow \mathsf{top}_{\forall}}$$

S-DISTALL

$$\overline{(\forall (\alpha * A).\, B_1) \,\&\, (\forall (\alpha * A).\, B_2) <: \forall (\alpha * A).\, B_1 \,\&\, B_2 \rightsquigarrow \mathsf{dist}_{\forall}}$$

Figure 4.5: Declarative subtyping of F_i^+

of disjoint quantification is covariant in its body, and contravariant in its disjointness constraints (rule S-FORALL). Rule S-DISTALL, similar to rule S-DISTARR, dictates that quantifiers may distribute over intersections. Rule S-TOPALL plays a similar role as rule S-TOPARR. A minor comment is that since F_i^+ features explicit polymorphism, type variables are neutral to subtyping (i.e., $\alpha <: \alpha$), which is already contained in rule S-REFL.

TYPING OF F_i^+. F_i^+ features a bidirectional type system inherited from F_i. The full set of typing rules are shown in Fig. 4.6. Again we ignore the translation parts ($\rightsquigarrow e$) and explain them in Section 4.4 The inference judgment $\Delta; \Gamma \vdash E \Rightarrow A$ says that we can synthesize the type A in the contexts Δ and Γ. The checking judgment $\Delta; \Gamma \vdash E \Leftarrow A$ asserts that

$$\boxed{\Delta;\Gamma \vdash E \Rightarrow A \rightsquigarrow e} \qquad\qquad\qquad\qquad\qquad\qquad \textit{(Inference)}$$

FT-TOP
$$\frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta;\Gamma \vdash \top \Rightarrow \top \rightsquigarrow \langle\rangle}$$

FT-INT
$$\frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta;\Gamma \vdash i \Rightarrow \mathsf{Int} \rightsquigarrow i}$$

FT-VAR
$$\frac{\vdash \Delta \qquad \Delta \vdash \Gamma \qquad (x:A) \in \Gamma}{\Delta;\Gamma \vdash x \Rightarrow A \rightsquigarrow x}$$

FT-APP
$$\frac{\Delta;\Gamma \vdash E_1 \Rightarrow A_1 \rightarrow A_2 \rightsquigarrow e_1 \qquad \Delta;\Gamma \vdash E_2 \Leftarrow A_1 \rightsquigarrow e_2}{\Delta;\Gamma \vdash E_1 E_2 \Rightarrow A_2 \rightsquigarrow e_1 e_2}$$

FT-MERGE
$$\frac{\Delta;\Gamma \vdash E_1 \Rightarrow A_1 \rightsquigarrow e_1 \qquad \Delta;\Gamma \vdash E_2 \Rightarrow A_2 \rightsquigarrow e_2 \qquad \Delta \vdash A_1 * A_2}{\Delta;\Gamma \vdash E_1\,,, E_2 \Rightarrow A_1 \& A_2 \rightsquigarrow \langle e_1, e_2 \rangle}$$

FT-ANNO
$$\frac{\Delta;\Gamma \vdash E \Leftarrow A \rightsquigarrow e}{\Delta;\Gamma \vdash E : A \Rightarrow A \rightsquigarrow e}$$

FT-RCD
$$\frac{\Delta;\Gamma \vdash E \Rightarrow A \rightsquigarrow e}{\Delta;\Gamma \vdash \{l = E\} \Rightarrow \{l : A\} \rightsquigarrow e}$$

FT-PROJ
$$\frac{\Delta;\Gamma \vdash E \Rightarrow \{l : A\} \rightsquigarrow e}{\Delta;\Gamma \vdash E.l \Rightarrow A \rightsquigarrow e}$$

FT-TABS
$$\frac{\Delta, \alpha * A;\Gamma \vdash E \Rightarrow B \rightsquigarrow e \qquad \Delta \vdash A \qquad \Delta \vdash \Gamma}{\Delta;\Gamma \vdash \Lambda(\alpha * A).\,E \Rightarrow \forall(\alpha * A).\,B \rightsquigarrow \Lambda\alpha.\,e}$$

FT-TAPP
$$\frac{\Delta;\Gamma \vdash E \Rightarrow \forall(\alpha * B).\,C \rightsquigarrow e \qquad \Delta \vdash A * B}{\Delta;\Gamma \vdash E\,A \Rightarrow [A/\alpha]C \rightsquigarrow e\,|A|}$$

$$\boxed{\Delta;\Gamma \vdash E \Leftarrow A \rightsquigarrow e} \qquad\qquad\qquad\qquad\qquad\qquad \textit{(Checking)}$$

FT-ABS
$$\frac{\Delta \vdash A \qquad \Delta;\Gamma, x:A \vdash E \Leftarrow B \rightsquigarrow e}{\Delta;\Gamma \vdash \lambda x.\,E \Leftarrow A \rightarrow B \rightsquigarrow \lambda x.\,e}$$

FT-SUB
$$\frac{\Delta;\Gamma \vdash E \Rightarrow B \rightsquigarrow e \qquad B <: A \rightsquigarrow co \qquad \Delta \vdash A}{\Delta;\Gamma \vdash E \Leftarrow A \rightsquigarrow co\,e}$$

Figure 4.6: Bidirectional type system of F_i^+

E checks against the type A in the contexts Δ and Γ. Most of the rules are quite standard in the literature. The merge expression $E_1\,,, E_2$ is well-typed if both sub-expressions are well-typed, and their types are *disjoint* (rule FT-MERGE). The disjointness relation will be explained in Section 4.3. To infer a type abstraction (rule FT-TABS), we add disjointness constraints to the type context. For a type application (rule FT-TAPP), we check that the type argument satisfies the disjointness constraints. Rules FT-MERGE and FT-TAPP are the only rules checking disjointness.

$$\boxed{\Delta \vdash A * B} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{(Disjointness)}$$

$$\frac{\lceil A \rceil}{\Delta \vdash A * B} \quad\text{FD-\textsc{topL}} \qquad\qquad \frac{\lceil B \rceil}{\Delta \vdash A * B} \quad\text{FD-\textsc{topR}} \qquad\qquad \frac{\Delta \vdash A_2 * B_2}{\Delta \vdash A_1 \to A_2 * B_1 \to B_2} \quad\text{FD-\textsc{arr}}$$

FD-\textsc{topL}
$$\frac{\lceil A \rceil}{\Delta \vdash A * B}$$

FD-\textsc{topR}
$$\frac{\lceil B \rceil}{\Delta \vdash A * B}$$

FD-\textsc{arr}
$$\frac{\Delta \vdash A_2 * B_2}{\Delta \vdash A_1 \to A_2 * B_1 \to B_2}$$

FD-\textsc{andL}
$$\frac{\Delta \vdash A_1 * B \qquad \Delta \vdash A_2 * B}{\Delta \vdash A_1 \,\&\, A_2 * B}$$

FD-\textsc{andR}
$$\frac{\Delta \vdash A * B_1 \qquad \Delta \vdash A * B_2}{\Delta \vdash A * B_1 \,\&\, B_2}$$

FD-\textsc{rcdEq}
$$\frac{\Delta \vdash A * B}{\Delta \vdash \{l : A\} * \{l : B\}}$$

FD-\textsc{rcdNeq}
$$\frac{l_1 \neq l_2}{\Delta \vdash \{l_1 : A\} * \{l_2 : B\}}$$

FD-\textsc{tvarL}
$$\frac{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash \alpha * B}$$

FD-\textsc{tvarR}
$$\frac{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash B * \alpha}$$

FD-\textsc{forall}
$$\frac{\Delta, \alpha * A_1 \,\&\, A_2 \vdash B_1 * B_2}{\Delta \vdash \forall(\alpha * A_1).\, B_1 * \forall(\alpha * A_2).\, B_2}$$

FD-\textsc{ax}
$$\frac{A *_{ax} B}{\Delta \vdash A * B}$$

$$\boxed{A *_{ax} B} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{(Disjointness axioms)}$$

\textsc{Dax-sym}
$$\frac{B *_{ax} A}{A *_{ax} B}$$

\textsc{Dax-intArr}
$$\frac{}{\mathsf{Int} *_{ax} A_1 \to A_2}$$

\textsc{Dax-intRcd}
$$\frac{}{\mathsf{Int} *_{ax} \{l : A\}}$$

\textsc{Dax-intAll}
$$\frac{}{\mathsf{Int} *_{ax} \forall(\alpha * B_1).\, B_2}$$

\textsc{Dax-arrAll}
$$\frac{}{A_1 \to A_2 *_{ax} \forall(\alpha * B_1).\, B_2}$$

\textsc{Dax-arrRcd}
$$\frac{}{A_1 \to A_2 *_{ax} \{l : B\}}$$

\textsc{Dax-allRcd}
$$\frac{}{\forall(\alpha * A_1).\, A_2 *_{ax} \{l : B\}}$$

Figure 4.7: Disjointness of F_i^+

4.3 Disjointness

We now turn to another core judgment of F_i^+—the disjointness relation, shown in Fig. 4.7. The disjointness rules are mostly inherited from F_i [Alpuim et al. 2017], but the new bottom type requires a notable change regarding disjointness with *top-like types*.

Top-like types. Top-like types are all types that are isomorphic to $\top$ (i.e., simultaneously sub- and supertypes of $\top$). Hence, they are inhabited by a single value, isomorphic to the $\top$ value. The following rules capture this notion in a syntax-directed fashion in the $\lceil A \rceil$ predicate. As a historical note, top-like types were first introduced in λ_i [Oliveira et al. 2016], and then adopted by F_i [Alpuim et al. 2017]. Their motivation is solely for enabling a syntactic

method of proving coherence, and due to the lack of BCD subtyping, they do not have a type-theoretic interpretation of top-like types.

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

$$
\begin{array}{ccccc}
\text{TL-TOP} & \begin{array}{c} \text{TL-AND} \\ \rceil A \lceil \quad \rceil B \lceil \end{array} & \begin{array}{c} \text{TL-ARR} \\ \rceil B \lceil \end{array} & \begin{array}{c} \text{TL-RCD} \\ \rceil A \lceil \end{array} & \begin{array}{c} \text{TL-ALL} \\ \rceil B \lceil \end{array} \\
\hline
\rceil \top \lceil & \rceil A \,\&\, B \lceil & \rceil A \rightarrow B \lceil & \rceil \{l : A\} \lceil & \rceil \forall(\alpha * A).\,B \lceil
\end{array}
$$

The disjointness judgment $\Delta \vdash A * B$ is helpful to check whether the merge of two expressions of type A and B preserves coherence. (As a precondition, A and B are required to be both well-formed in the context Δ.) Incoherence arises when both expressions produce distinct values for the same type, either directly when they are both of that same type, or through implicit upcasting to a common supertype. Of course we can safely disregard top-like types in this matter because they do not have two distinct values. In short, it suffices to check that the two types have only top-like supertypes in common.

Because $\bot$ and any another type A always have A as a common supertype, it follows that $\bot$ is only disjoint to A when A is top-like. More generally, if A is a top-like type, then A is disjoint to any type. This is the rationale behind the two rules FD-TOPL and FD-TOPR, which generalize and subsume $\Delta \vdash \top * A$ and $\Delta \vdash A * \top$ from F_i, and also cater to the bottom type. Two other interesting rules are FD-TVARL and FD-TVARR, which dictate that a type variable α is disjoint with some type B if its disjointness constraints A is a subtype of B. These two rules are a specialization of a more general lemma [Alpuim et al. 2017], which says that disjointness is covariant with respect to subtyping:

Lemma 4.3 (☞ Covariance of disjointness). *If $\Delta \vdash A * B$ and $B <: C$, then $\Delta \vdash A * C$.*

Proof. By double induction, first on the subtyping derivation, and then on the type A. In the case for rule S-FORALL, we need Lemma 4.4. □

Lemma 4.4 (☞ Narrowing of disjointness). *If $\Delta, \alpha * C_1 \vdash A * B$ and $C_2 <: C_1$, then $\Delta, \alpha * C_2 \vdash A * B$.*

Proof. We need to slightly generalize the lemma in the sense that the type variable is inserted in the middle, then by induction on the disjointness derivation. □

To conclude this section, we show that the disjointness relation is symmetric:

Lemma 4.5 (☞ Symmetry of disjointness). *If $\Delta \vdash A * B$, then $\Delta \vdash B * A$.*

Proof. By structural induction on the derivation of disjointness. In the case for rule FD-FORALL, apply Lemma 4.4. □

Types	τ	::=	$\mathsf{Int} \mid \langle\rangle \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \boxed{\alpha} \mid \boxed{\forall\alpha.\,\tau}$
Expressions	e	::=	$x \mid i \mid \langle\rangle \mid \lambda x.\,e \mid e_1\,e_2 \mid \langle e_1, e_2\rangle \mid co\,e \mid \boxed{\Lambda\alpha.\,e} \mid \boxed{e\,\tau}$
Coercions	co	::=	$\mathsf{id} \mid co_1 \circ co_2 \mid \mathsf{top} \mid co_1 \to co_2 \mid \langle co_1, co_2\rangle \mid \pi_1 \mid \pi_2$
			$\mid \quad \mathsf{dist}_\to \mid \mathsf{top}_\to \mid \boxed{\mathsf{bot}} \mid \boxed{co_\forall} \mid \boxed{\mathsf{top}_\forall} \mid \boxed{\mathsf{dist}_\forall}$
Values	v	::=	$i \mid \langle\rangle \mid \lambda x.\,e \mid \langle v_1, v_2\rangle \mid (co_1 \to co_2)\,v \mid \mathsf{dist}_\to v \mid \mathsf{top}_\to v$
			$\mid \quad \boxed{\Lambda\alpha.\,e} \mid \boxed{co_\forall\,v} \mid \boxed{\mathsf{top}_\forall\,v} \mid \boxed{\mathsf{dist}_\forall\,v}$
Value contexts	Ψ	::=	$\bullet \mid \Psi, x : \tau$
Type contexts	Φ	::=	$\bullet \mid \Phi, \alpha$
Evaluation contexts	$\mathcal{E}$	::=	$[\cdot] \mid \mathcal{E}\,e \mid v\,\mathcal{E} \mid \langle\mathcal{E}, e\rangle \mid \langle v, \mathcal{E}\rangle \mid co\,\mathcal{E} \mid \boxed{\mathcal{E}\,\tau}$

Figure 4.8: Syntax of F_{co}

4.4 ELABORATION AND TYPE SAFETY

Like λ_i^+, the dynamic semantics of F_i^+ is given by elaboration into a target calculus. The target calculus F_{co} is the standard call-by-value System F extended with products and coercions. The syntax of F_{co} is shown in Fig. 4.8, with the differences from λ_{co} $\boxed{\text{highlighted}}$. We extend the type translation function $|\cdot|$ to cover new constructs: $\bot$ is mapped to an uninhabited type $\forall\alpha.\,\alpha$; disjoint quantification is mapped to universal quantification, dropping the disjointness constraints.

Definition 7 (Type translation $|\cdot|$ from F_i^+ to F_{co}).

$\|\mathsf{Int}\|$	$=$	Int		$\|\top\|$	$=$	$\langle\rangle$
$\|A \to B\|$	$=$	$\|A\| \to \|B\|$		$\|A \,\&\, B\|$	$=$	$\|A\| \times \|B\|$
$\|\{l : A\}\|$	$=$	$\|A\|$		$\|\alpha\|$	$=$	α
$\|\forall(\alpha * A).\,B\|$	$=$	$\forall\alpha.\,\|B\|$		$\|\bot\|$	$=$	$\forall\alpha.\,\alpha$

COERCIONS AND COERCIVE SUBTYPING. As shown in Fig. 4.8, we extend the coercions of λ_{co} with several new coercions: bot, $co_\forall$, $\mathsf{dist}_\forall$ and $\mathsf{top}_\forall$ due to the addition of polymorphism and bottom type. As a cognitive aid, we extend Table 3.1 with new desugaring rules, shown in Table 4.1. For example, $co_\forall$ can be thought of as $\lambda f.\,\Lambda\alpha.\,co\,(f\,\alpha)$, then the expression $co_\forall\,v$ is "equal" to $\Lambda\alpha.\,co\,(v\,\alpha)$, which is why we can treat $co_\forall\,v$ as a value.

Coercion	Term	Coercion	Term
id	$\lambda x.\, x$	$co_1 \circ co_2$	$\lambda x.\, co_1\, (co_2\, x)$
top	$\lambda x.\, \langle\rangle$	$co_1 \to co_2$	$\lambda f.\, \lambda x.\, co_2\, (f\, (co_1\, x))$
π_1	$\lambda x.\, \pi_1\, x$	π_2	$\lambda x.\, \pi_2\, x$
$\langle co_1, co_2\rangle$	$\lambda x.\, \langle co_1\, x, co_2\, x\rangle$	$dist_\to$	$\lambda x.\, \lambda y.\, \langle (\pi_1\, x)\, y, (\pi_2\, x)\, y\rangle$
$top_\to$	$\lambda x.\, \lambda y.\, \langle\rangle$	$co_\forall$	$\lambda f.\, \Lambda\alpha.\, co\, (f\,\alpha)$
$top_\forall$	$\lambda x.\, \Lambda\alpha.\, \langle\rangle$	$dist_\forall$	$\lambda f.\, \Lambda\alpha.\, \langle (\pi_1\, f)\,\alpha, (\pi_2\, f)\,\alpha\rangle$
bot	$\lambda f.\, \Lambda\alpha.\, f\,\alpha$		

Table 4.1: Correspondence between coercions and terms, extended

STATIC SEMANTICS. Figure 4.9 presents the typing rules of F_{co}. Most of the rules are are similar to . Rule FT-CAPP uses the judgment $co :: \tau_1 \triangleright \tau_2$ to type coercions. We extend Fig. 3.7 with four new rules, corresponding to the four new coercions:

$$\frac{}{\mathsf{bot} :: \forall\alpha.\, \alpha \triangleright \tau} \quad \text{CT-BOT}$$

$$\frac{co :: \tau_1 \triangleright \tau_2}{co_\forall :: \forall\alpha.\, \tau_1 \triangleright \forall\alpha.\, \tau_2} \quad \text{CT-FORALL}$$

$$\frac{}{\mathsf{top}_\forall :: \langle\rangle \triangleright \forall\alpha.\, \langle\rangle} \quad \text{CT-TOPALL}$$

$$\frac{}{\mathsf{dist}_\forall :: (\forall\alpha.\, \tau_1) \times (\forall\alpha.\, \tau_2) \triangleright \forall\alpha.\, \tau_1 \times \tau_2} \quad \text{CT-DISTALL}$$

DYNAMIC SEMANTICS. We extend the evaluation context with one new form $\mathcal{E}\,\tau$ for type applications, as shown in Fig. 4.8. The set of reduction rules (Fig. 4.10) for F_{co} is a straightforward extension of λ_{co}. We have three new reduction rules R-FORALL, R-DISTALL, and R-TOPALL for the new coercions. Also we add the reduction rule R-TAPP for type applications. Now we can show that F_{co} is type-safe in the usual sense:

Theorem 4.6 (☞ **Preservation of F_{co}).** *If* $\bullet; \bullet \vdash e : \tau$ *and* $e \longrightarrow e'$, *then* $\bullet; \bullet \vdash e' : \tau$.

Theorem 4.7 (☞ **Progress of F_{co}).** *If* $\bullet; \bullet \vdash e : \tau$, *then either* e *is a value, or there exists* e' *such that* $e \longrightarrow e'$.

ELABORATION. We go back to the translation parts in Fig. 4.6. The key idea of the translation remains the same: we translate merges to pairs. For disjoint quantification and disjoint type applications (rules FT-TABS and FT-TAPP), we translate them to regular universal quantification and type applications, respectively. To conclude, we show an example translation:

$$(\Lambda(\alpha * \mathsf{Int}).\, \lambda x.x) : \forall(\alpha * \mathsf{Int}).\, \alpha \,\&\, \mathsf{Int} \to \alpha \rightsquigarrow (\pi_1 \to \mathsf{id})_\forall\, (\Lambda\alpha.\, \lambda x.\, x)$$

4 Semantics of the F$_i^+$ Calculus

$\boxed{\Phi \vdash \Psi}$ *(Well-formedness of value context)*

$$\frac{}{\Phi \vdash \bullet} \text{ WFE-EMPTY}$$

$$\frac{\Phi \vdash \tau \qquad \Phi \vdash \Psi}{\Phi \vdash \Psi, x : \tau} \text{ WFE-VAR}$$

$\boxed{\Phi \vdash \tau}$ *(Well-formedness of types)*

$$\frac{}{\Phi \vdash \mathsf{Int}} \text{ WFT-INT} \qquad \frac{\alpha \in \Phi}{\Phi \vdash \alpha} \text{ WFT-VAR} \qquad \frac{\Phi \vdash \tau_1 \quad \Phi \vdash \tau_2}{\Phi \vdash \tau_1 \rightarrow \tau_2} \text{ WFT-ARROW} \qquad \frac{\Phi \vdash \tau_1 \quad \Phi \vdash \tau_2}{\Phi \vdash \tau_1 \times \tau_2} \text{ WFT-PROD} \qquad \frac{\Phi, \alpha \vdash \tau_2}{\Phi \vdash \forall \alpha. \tau_2} \text{ WFT-ALL}$$

$\boxed{\Phi ; \Psi \vdash e : \tau}$ *(Static semantics)*

$$\frac{\Phi \vdash \Psi}{\Phi ; \Psi \vdash \langle\rangle : \langle\rangle} \text{ FT-UNIT} \qquad \frac{\Phi \vdash \Psi}{\Phi ; \Psi \vdash i : \mathsf{Int}} \text{ FT-INT} \qquad \frac{\Phi \vdash \Psi \quad (x : \tau) \in \Psi}{\Phi ; \Psi \vdash x : \tau} \text{ FT-VAR}$$

$$\frac{\Phi ; \Psi, x : \tau_1 \vdash e : \tau_2 \quad \Phi \vdash \tau_1}{\Phi ; \Psi \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \text{ FT-ABS} \qquad \frac{\Phi ; \Psi \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Phi ; \Psi \vdash e_2 : \tau_1}{\Phi ; \Psi \vdash e_1 \, e_2 : \tau_2} \text{ FT-APP}$$

$$\frac{\Phi, \alpha ; \Psi \vdash e : \tau \quad \Phi \vdash \Psi}{\Phi ; \Psi \vdash \Lambda \alpha. e : \forall \alpha. \tau} \text{ FT-TABS} \qquad \frac{\Phi ; \Psi \vdash e : \forall \alpha. \tau' \quad \Phi \vdash \tau}{\Phi ; \Psi \vdash e \, \tau : [\tau/\alpha]\tau'} \text{ FT-TAPP}$$

$$\frac{\Phi ; \Psi \vdash e_1 : \tau_1 \quad \Phi ; \Psi \vdash e_2 : \tau_2}{\Phi ; \Psi \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \text{ FT-PAIR} \qquad \frac{\Phi ; \Psi \vdash e : \tau_1 \quad co :: \tau_1 \rhd \tau_2 \quad \Phi \vdash \tau_2}{\Phi ; \Psi \vdash co \, e : \tau_2} \text{ FT-CAPP}$$

Figure 4.9: Typing rules of F$_{co}$

As with λ_i^+, we show two lemmas that relate F$_i^+$ to F$_{co}$.

Lemma 4.8 (☞ Coercions preserve types). *If $A <: B \rightsquigarrow co$, then $co :: |A| \rhd |B|$.*

Proof. By structural induction on the derivation of subtyping. ☐

Lemma 4.9 (☞ Elaboration soundness). *We have that:*

- *If $\Delta ; \Gamma \vdash E \Rightarrow A \rightsquigarrow e$, then $|\Delta| ; |\Gamma| \vdash e : |A|$.*

- *If $\Delta ; \Gamma \vdash E \Leftarrow A \rightsquigarrow e$, then $|\Delta| ; |\Gamma| \vdash e : |A|$.*

Proof. By structural induction on the derivation of typing. ☐

$\boxed{e \longrightarrow e'}$ *(Single-step reduction)*

R-ID   R-TRANS   R-TOP   R-TOPARR

$\overline{\mathsf{id}\ v \longrightarrow v}$ $\overline{(co_1 \circ co_2)\ v \longrightarrow co_1\ (co_2\ v)}$ $\overline{\mathsf{top}\ v \longrightarrow \langle\rangle}$ $\overline{(\mathsf{top}_{\rightarrow}\ \langle\rangle)\ \langle\rangle \longrightarrow \langle\rangle}$

R-PAIR   R-ARR

$\overline{\langle co_1, co_2 \rangle\ v \longrightarrow \langle co_1\ v, co_2\ v \rangle}$ $\overline{((co_1 \rightarrow co_2)\ v_1)\ v_2 \longrightarrow co_2\ (v_1\ (co_1\ v_2))}$

R-DISTARR   R-PROJL   R-PROJR

$\overline{(\mathsf{dist}_{\rightarrow}\ \langle v_1, v_2 \rangle)\ v_3 \longrightarrow \langle v_1\ v_3, v_2\ v_3 \rangle}$ $\overline{\pi_1\ \langle v_1, v_2 \rangle \longrightarrow v_1}$ $\overline{\pi_2\ \langle v_1, v_2 \rangle \longrightarrow v_2}$

R-FORALL   R-DISTALL   R-TOPALL

$\overline{(co_{\forall}\ v)\ \tau \longrightarrow co\ (v\ \tau)}$ $\overline{(\mathsf{dist}_{\forall}\ \langle v_1, v_2 \rangle)\ \tau \longrightarrow \langle v_1\ \tau, v_2\ \tau \rangle}$ $\overline{(\mathsf{top}_{\forall}\ \langle\rangle)\ \tau \longrightarrow \langle\rangle}$

R-APP   R-TAPP   R-CTXT
                                                                                                        $e \longrightarrow e'$

$\overline{(\lambda x.\ e)\ v \longrightarrow [v/x]e}$ $\overline{(\Lambda\alpha.\ e)\ \tau \longrightarrow [\tau/\alpha]e}$ $\overline{\mathcal{E}[e] \longrightarrow \mathcal{E}[e']}$

Figure 4.10: Dynamic semantics of F_{co}

4.5 ALGORITHMIC SYSTEM AND DECIDABILITY

The subtyping relation in Fig. 4.5 is highly non-algorithmic due to the presence of a transitivity rule. This section presents an alternative algorithmic formulation. The algorithmic subtyping extends that of λ_i^+, to handle disjoint quantifiers and the bottom type. We then prove that the algorithm is sound and complete with respect to declarative subtyping.

Additionally we prove that the subtyping and disjointness relations are decidable. Although the proofs of this fact are fairly straightforward, it is nonetheless remarkable since it contrasts with the subtyping relation for (full) $\mathsf{F}_{<:}$ [Cardelli and Wegner 1985], which is undecidable [Pierce 1994]. Thus while bounded quantification is infamous for its undecidability, disjoint quantification has the nicer property of being decidable.

4.5.1 ALGORITHMIC SUBTYPING RULES

Following λ_i^+, we intend the algorithmic subtyping judgment $\mathcal{Q} \vdash A \prec: B$ to be equivalent to $A <: \mathcal{Q} \Rightarrow B$, where $\mathcal{Q}$ is a queue used to track record labels, domain types and disjointness constraints. The syntax of $\mathcal{Q}$ is shown below

$$\mathcal{Q} ::= [] \mid l, \mathcal{Q} \mid B, \mathcal{Q} \mid \alpha * B, \mathcal{Q}$$

$$\boxed{\mathcal{Q} \vdash A \prec: B \rightsquigarrow co} \qquad\qquad\qquad\qquad\qquad \textit{(Algorithmic subtyping)}$$

A-TOP
$$\frac{}{\mathcal{Q} \vdash A \prec: \top \rightsquigarrow [\![\mathcal{Q}]\!]^\top \circ \mathsf{top}}$$

A-AND
$$\frac{\mathcal{Q} \vdash A \prec: B_1 \rightsquigarrow co_1 \qquad \mathcal{Q} \vdash A \prec: B_2 \rightsquigarrow co_2}{\mathcal{Q} \vdash A \prec: B_1 \,\&\, B_2 \rightsquigarrow [\![\mathcal{Q}]\!]^{\&} \circ \langle co_1, co_2 \rangle}$$

A-ARR
$$\frac{\mathcal{Q}, B_1 \vdash A \prec: B_2 \rightsquigarrow co}{\mathcal{Q} \vdash A \prec: B_1 \to B_2 \rightsquigarrow co}$$

A-RCD
$$\frac{\mathcal{Q}, l \vdash A \prec: B \rightsquigarrow co}{\mathcal{Q} \vdash A \prec: \{l : B\} \rightsquigarrow co}$$

A-FORALL
$$\frac{\mathcal{Q}, \alpha * B_1 \vdash A \prec: B_2 \rightsquigarrow co}{\mathcal{Q} \vdash A \prec: \forall(\alpha * B_1). B_2 \rightsquigarrow co}$$

A-CONST
$$\frac{}{[] \vdash c \prec: c \rightsquigarrow \mathsf{id}}$$

A-BOT
$$\frac{}{\mathcal{Q} \vdash \bot \prec: c \rightsquigarrow \mathsf{bot}}$$

A-ARRCONST
$$\frac{[] \vdash A \prec: A_1 \rightsquigarrow co_1 \qquad \mathcal{Q} \vdash A_2 \prec: c \rightsquigarrow co_2}{A, \mathcal{Q} \vdash A_1 \to A_2 \prec: c \rightsquigarrow co_1 \to co_2}$$

A-RCDCONST
$$\frac{\mathcal{Q} \vdash A \prec: c \rightsquigarrow co}{l, \mathcal{Q} \vdash \{l : A\} \prec: c \rightsquigarrow co}$$

A-ANDCONST1
$$\frac{\mathcal{Q} \vdash A_1 \prec: c \rightsquigarrow co}{\mathcal{Q} \vdash A_1 \,\&\, A_2 \prec: c \rightsquigarrow co \circ \pi_1}$$

A-ANDCONST2
$$\frac{\mathcal{Q} \vdash A_2 \prec: c \rightsquigarrow co}{\mathcal{Q} \vdash A_1 \,\&\, A_2 \prec: c \rightsquigarrow co \circ \pi_2}$$

A-ALLCONST
$$\frac{[] \vdash A \prec: A_1 \qquad \mathcal{Q} \vdash A_2 \prec: c \rightsquigarrow co}{(\alpha * A, \mathcal{Q}) \vdash \forall(\alpha * A_1). A_2 \prec: c \rightsquigarrow co_\forall}$$

Figure 4.11: Algorithmic subtyping

The full rules of the algorithmic subtyping of F_i^+ are shown Fig. 4.11.

Definition 8. $\mathcal{Q} \Rightarrow A$ is inductively defined as follows:

$[] \Rightarrow A$	$=$	A	$(B, \mathcal{Q}) \Rightarrow A$	$=$	$B \to (\mathcal{Q} \Rightarrow A)$
$(l, \mathcal{Q}) \Rightarrow A$	$=$	$\{l : \mathcal{Q} \Rightarrow A\}$	$(\alpha * B, \mathcal{Q}) \Rightarrow A$	$=$	$\forall(\alpha * B). \mathcal{Q} \Rightarrow A$

For brevity of the algorithm, we use metavariable c to mean type constants:

$$\text{Type Constants} \quad c ::= \mathsf{Int} \mid \bot \mid \alpha$$

The basic idea of $\mathcal{Q} \vdash A \prec: B$ is to perform a case analysis on B until it reaches type constants. We explain new rules regarding disjoint quantification and the bottom type. When a quantifier is encountered in B, rule A-FORALL pushes the type variables with its disjointness constraints onto $\mathcal{Q}$ and continue with the body. Correspondingly, in rule A-ALLCONST, when a quantifier is encountered in A, and the head of $\mathcal{Q}$ is a type variable, this variable is popped out and we continue with the body. Rule A-BOT is similar to its declarative counterpart. Two meta-functions $[\![\mathcal{Q}]\!]^\top$ and $[\![\mathcal{Q}]\!]^{\&}$ are meant to generate correct forms of coercions, and their definitions are shown in Fig. 4.12.

$$\llbracket [] \rrbracket^\top = \mathsf{top} \qquad\qquad\qquad \llbracket [] \rrbracket^\& = \mathsf{id}$$

$$\llbracket l, \mathcal{Q} \rrbracket^\top = \llbracket \mathcal{Q} \rrbracket^\top \circ \mathsf{id} \qquad\qquad \llbracket l, \mathcal{Q} \rrbracket^\& = \llbracket \mathcal{Q} \rrbracket^\& \circ \mathsf{id}$$

$$\llbracket A, \mathcal{Q} \rrbracket^\top = (\mathsf{top} \to \llbracket \mathcal{Q} \rrbracket^\top) \circ \mathsf{top}_\to \qquad \llbracket A, \mathcal{Q} \rrbracket^\& = (\mathsf{id} \to \llbracket \mathcal{Q} \rrbracket^\&) \circ \mathsf{dist}_\to$$

$$\llbracket \alpha * A, \mathcal{Q} \rrbracket^\top = \llbracket \mathcal{Q} \rrbracket^\top_\forall \circ \mathsf{top}_\forall \qquad\qquad \llbracket \alpha * A, \mathcal{Q} \rrbracket^\& = \llbracket \mathcal{Q} \rrbracket^\&_\forall \circ \mathsf{dist}_\forall$$

Figure 4.12: Meta-functions of coercions, extended

CORRECTNESS OF THE ALGORITHM. We prove that the algorithm is sound and complete with respect to the specification. We refer the reader to our Coq formalization for more details. We only show the two major theorems:

Theorem 4.10 (☞ Soundness). *If $\mathcal{Q} \vdash A \prec: B \rightsquigarrow co$ then $A <: \mathcal{Q} \Rightarrow B \rightsquigarrow co$.*

Theorem 4.11 (☞ Completeness). *If $A <: B \rightsquigarrow co$ then there exists co' such that $[] \vdash A \prec: B \rightsquigarrow co'$.*

4.5.2 DECIDABILITY

Moreover, we prove that our algorithmic type system is decidable. To see this, first notice that the bidirectional type system is syntax-directed, so we only need to show decidability of algorithmic subtyping and disjointness. The full (manual) proofs for decidability can be found in Appendix B.

Lemma 4.12 (Decidability of algorithmic subtyping). *Given $\mathcal{Q}$, A and B, it is decidable whether there exists co, such that $\mathcal{Q} \vdash A \prec: B \rightsquigarrow co$.*

Proof. Let the judgment $\mathcal{Q} \vdash A \prec: B \rightsquigarrow co$ be measured by $size(\mathcal{Q}) + size(A) + size(B)$. For each subtyping rule, we can show that every inductive premise is smaller than the conclusion. □

Lemma 4.13 (Decidability of disjointness checking). *Given Δ, A and B, it is decidable whether $\Delta \vdash A * B$.*

Proof. Let the judgment $\Delta \vdash A * B$ be measured by $size(A) + size(B)$. For each subtyping rule, we can show that every inductive premise is smaller than the conclusion. □

One interesting observation here is that although our disjointness quantification has a similar shape to bounded quantification $\forall(\alpha <: A). B$ in $\mathsf{F}_{<:}$ [Cardelli and Wegner 1985], sub-

typing for $\mathsf{F}_{<:}$ is undecidable [Pierce 1994]. In $\mathsf{F}_{<:}$, the subtyping relation between bounded quantification is:

$$\frac{\Delta \vdash A_2 <: A_1 \qquad \Delta, \alpha <: A_2 \vdash B_1 <: B_2}{\Delta \vdash \forall(\alpha <: A_1).\, B_1 <: \forall(\alpha <: A_2).\, B_2} \text{ \small FSUB-FORALL}$$

Compared with rule S-FORALL, both rules are contravariant on bounded/disjoint types, and covariant on the body. However, with bounded quantification it is fundamental to track the bounds in the environment, which complicates the design of the rules and makes subtyping undecidable with rule FSUB-FORALL. Decidability can be recovered by employing an invariant rule for bounded quantification (that is by forcing A_1 and A_2 to be identical). Disjoint quantification does not require such invariant rule for decidability.

PART III

COHERENCE

5 COHERENCE FOR λ_i^+

This chapter constructs a logical relation to establish coherence of λ_i^+. Finding a suitable definition of coherence for λ_i^+ is already challenging in its own right. In what follows we reproduce the steps of finding a definition for coherence that is both intuitive and applicable. Then we present the construction of the logical (equivalence) relation tailored to this definition, and the connection between logical equivalence and coherence. Chapter 6 builds on the idea in this chapter to prove coherence for F_i^+.

5.1 THE INTUITION

DUPLICATION IS HARMLESS.   While requiring that all intersections are disjoint as in λ_i is sufficient to guarantee coherence, it is not necessary. In fact, such requirement unnecessarily encumbers the subtyping definition with disjointness constraints and an ad-hoc treatment of "top-like" types. Indeed, the value $1\,,,1$ of the non-disjoint type $\text{Int \& Int}$ is entirely unambiguous, and $(1\,,,1)+3$ can obviously only result in 4. More generally, when the overlapping components of an intersection type have the same value, there is no ambiguity problem. λ_i^+ uses this idea to relax λ_i's enforcement of disjointness. In the case of a merge, it is hard to statically decide whether the two arguments have the same value, and thus λ_i^+ still requires disjointness. Yet, disjointness is no longer required for the well-formedness of types and overlapping intersections can be created implicitly through subtyping, which results in duplicating values at run time. For instance, while $1\,,,1$ is not expressible $1 : \text{Int \& Int}$ creates the equivalent value implicitly. In short, duplication is harmless and subtyping only generates duplicated values for non-disjoint types.

Two factors make establishing coherence for λ_i^+ much more difficult: the relaxation of disjointness and the adoption of the more expressive subtyping rules from the BCD system (for which λ_i lacks). These two factors mean that subtyping proofs are no longer unique and hence that there are multiple elaborations of the same source program. For instance, $\text{Int \& Int}$ is a subtype of Int in two ways: by projection on either the first or second component. Hence the fact that all elaborations yield the same result when evaluated has become a much more subtle property that requires sophisticated reasoning. For instance, we can see that coherence

holds because at run time any value of type Int & Int has identical components, and thus both projections yield the same result.

For λ_i^+ in general, we show coherence by capturing the non-ambiguity invariant in a *logical relation* [Plotkin 1973; Statman 1985; Tait 1967] and showing that it is preserved by the operational semantics. In doing so, we remove the brittleness of the previous syntactic method to prove coherence. This new proof method has several advantages. Firstly, with the new proof method, several restrictions that were enforced by λ_i to enable the syntactic proof are removed. For example, the aforementioned top-like types are not necessary; top-like types are handled like all other types. Secondly, the new proof method is more powerful because it is based on contextual equivalence rather than syntactic equality; it is more robust as the type system is extended. Finally, the removal of the ad-hoc side-conditions makes adding new extensions, such as support for BCD-style subtyping, easier. A complicating factor is that not one, but two languages are involved: the source language and the target language. In order to deal with the complexity of the elaboration semantics of λ_i^+, we employ binary logical relations that are heterogeneous, parameterized by two types; the fundamental property is also reformulated to account for bidirectional type-checking. A caveat is that our logical relation does not hold for target programs and program contexts in general, but only for those that are the image of a corresponding source program or program context. Thus we must view everything through the lens of elaboration.

5.2 In Search of Coherence

In λ_i the definition of coherence is based on α-equivalence. More specifically, the coherence property in λ_i states that any two target terms that a source expression elaborates into must be exactly the same (up to α-equivalence). Unfortunately this syntactic notion of coherence is very fragile with respect to extensions. For example, it is not obvious how to retain this notion of coherence when adding more subtyping rules such as those in Fig. 3.3.

If we permit ourselves to consider only the syntactic aspects of expressions, then very few expressions can be considered equal. The syntactic view also conflicts with the intuition that "the significance of an expression lies in its contribution to the *outcome* of a computation" [Harper 2016]. Drawing inspiration from a wide range of literature on contextual equivalence [Morris Jr 1969], we seek a context-based notion of coherence. It is helpful to consider several examples before presenting the formal definition of our new semantically founded notion of coherence.

Example 1. The same λ_i^+ expression 1 can be typed Int in many ways: for instance, by rule T-LIT; by rules T-SUB and S-REFL; or by rules T-SUB, S-TRANS, and S-REFL, resulting in transla-

λ_{co} contexts	$\mathcal{D}$	::=	$[\cdot] \mid \lambda x.\,\mathcal{D} \mid \mathcal{D}\,e \mid e\,\mathcal{D} \mid \langle \mathcal{D}, e \rangle \mid \langle e, \mathcal{D} \rangle \mid co\,\mathcal{D}$
λ_i^+ contexts	$\mathcal{C}$	::=	$[\cdot] \mid \lambda x.\,\mathcal{C} \mid \mathcal{C}\,E \mid E\,\mathcal{C} \mid E,,\mathcal{C} \mid \mathcal{C},,E \mid \mathcal{C}:A$
		$\mid$	$\{l = \mathcal{C}\} \mid \mathcal{C}.l$

Figure 5.1: Expression contexts of λ_{co} and λ_i^+

tions 1, id 1 and (id $\circ$ id) 1, respectively. It is apparent that these three λ_{co} terms are "equal" in the sense that all reduce to the same numeral 1.

5.2.1 Expression Contexts and Contextual Equivalence.

To formalize the intuition, we turn to *expression contexts*, as introduced in Section 2.5. The syntax of λ_{co} contexts $\mathcal{D}$ can be found in Fig. 5.1. The static semantics of λ_{co} is extended to expression contexts by defining the typing judgment

$$\mathcal{D} : (\Psi \vdash \tau) \mapsto (\Psi' \vdash \tau')$$

where $(\Psi \vdash \tau)$ indicates the type of the hole. This judgment is inductively defined such that if $\Psi \vdash e : \tau$, then $\Psi' \vdash \mathcal{D}\{e\} : \tau'$.

We define a *complete program* to mean any closed term of type Int. Recall the definitions of Kleene equality and contextual equivalence in Section 2.5. For ease of reference, we restate them below:

Definition 3 (Kleene Equality $\simeq$). Two complete programs, e and e', are Kleene equal, written $e \simeq e'$, if there exists i such that $e \longrightarrow^* i$ and $e' \longrightarrow^* i$.

Definition 4 (Contextual Equivalence $\simeq_{ctx}$).

$$\Psi \vdash e_1 \simeq_{ctx} e_2 : \tau \triangleq \Psi \vdash e_1 : \tau \wedge \Psi \vdash e_2 : \tau \wedge$$
$$(\forall \mathcal{D}.\ \mathcal{D} : (\Psi \vdash \tau) \mapsto (\bullet \vdash \mathsf{Int}) \implies \mathcal{D}\{e_1\} \simeq \mathcal{D}\{e_2\})$$

Regarding Example 1, it seems adequate to say that 3 and id 3 are contextually equivalent. Does this imply that coherence can be based on Definition 4? Unfortunately it cannot, as demonstrated by the following example.

Example 2. It may be counter-intuitive that two λ_{co} terms $\lambda x.\,\pi_1\,x$ and $\lambda x.\,\pi_2\,x$ should also be considered equal. To see why, first note that they are both the translations of the same λ_i^+ expression: $(\lambda x.\,x) : \mathsf{Int}\,\&\,\mathsf{Int} \rightarrow \mathsf{Int}$. What can we do with this lambda abstraction? We can apply it to 1 for example, which leads to two translations $(\lambda x.\,\pi_1\,x)\,\langle 1, 1 \rangle$ and

$(\lambda x. \pi_2 \, x) \, \langle 1, 1 \rangle$. It is obvious that both reduce to the same numeral 1. However, $\lambda x. \pi_1 \, x$ and $\lambda x. \pi_2 \, x$ are definitely *not* equal according to Definition 4, as one can find a context $[\cdot] \, \langle 1, 2 \rangle$ in which the two terms reduce to two different numerals. The problem is that $[\cdot] \, \langle 1, 2 \rangle$ should not be considered because the (non-disjoint) source expression $1 \,,\, 2$ is rejected by the type system of the source calculus λ_i^+ and thus never gets elaborated into $\langle 1, 2 \rangle$.

5.2.2 λ_i^+ CONTEXTS AND REFINED CONTEXTUAL EQUIVALENCE.

Example 2 hints at a shift from λ_{co} contexts to λ_i^+ contexts C, whose syntax is shown in Fig. 5.1. Due to the bidirectional nature of the type system, the typing judgment of C features 4 different forms:

$$\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A') \rightsquigarrow \mathcal{D} \qquad \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A') \rightsquigarrow \mathcal{D}$$

$$\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Leftarrow A') \rightsquigarrow \mathcal{D} \qquad \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Leftarrow A') \rightsquigarrow \mathcal{D}$$

We write $\mathcal{C} : (\Gamma \Leftrightarrow A) \mapsto (\Gamma' \Leftrightarrow' A') \rightsquigarrow \mathcal{D}$ to abbreviate the above 4 different forms. Take $\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A') \rightsquigarrow \mathcal{D}$ for example (whose typing rules are shown in Fig. 5.2), it reads that if $\Gamma \vdash E \Rightarrow A$, then $\Gamma' \vdash \mathcal{C}\{E\} \Rightarrow A'$. The judgment also generates a well-typed λ_{co} context $\mathcal{D}$ such that $\mathcal{D} : (|\Gamma| \vdash |A|) \mapsto (|\Gamma'| \vdash |A'|)$ holds by construction. The full typing rules appear in Appendix D. Now we are ready to refine Definition 4's contextual equivalence to take into consideration both λ_i^+ and λ_{co} contexts.

Definition 9 (λ_i^+ Contextual Equivalence).

$$\Gamma \vdash E_1 \simeq_{ctx} E_2 : A \triangleq \forall e_1, e_2. \, \Gamma \vdash E_1 \Rightarrow A \rightsquigarrow e_1 \wedge \Gamma \vdash E_2 \Rightarrow A \rightsquigarrow e_2 \wedge$$

$$(\forall \mathcal{C}, \mathcal{D}. \, \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\bullet \Rightarrow \mathsf{Int}) \rightsquigarrow \mathcal{D} \implies \mathcal{D}\{e_1\} \simeq \mathcal{D}\{e_2\})$$

In other words, two source expressions are contextually equivalent if their translations are equivalent in all possible source contexts. For brevity we only consider expressions in the inference mode. Our Coq formalization is complete with two modes. Now regarding Example 2, a possible λ_i^+ context is

$$[\cdot] \, 1 : (\bullet \Rightarrow \mathsf{Int} \,\&\, \mathsf{Int} \to \mathsf{Int}) \mapsto (\bullet \Rightarrow \mathsf{Int}) \rightsquigarrow [\cdot] \, \langle 1, 1 \rangle$$

We can verify that both $\lambda x. \pi_1 \, x$ and $\lambda x. \pi_2 \, x$ produce 1 in the context $[\cdot] \, \langle 1, 1 \rangle$. Of course we should consider all possible contexts to be certain that they are truly equal. From now on, we use the symbol $\simeq_{ctx}$ to refer to contextual equivalence in Definition 9. With Definition 9 we can formally state that λ_i^+ is coherent in the following sense:

$$\boxed{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}} \qquad\qquad \textit{(Context typing I)}$$

CTyp-empty1

$$\frac{}{[\cdot] : (\Gamma \Rightarrow A) \mapsto (\Gamma \Rightarrow A) \rightsquigarrow [\cdot]}$$

CTyp-appL1

$$\frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1 \to A_2) \rightsquigarrow \mathcal{D} \qquad \Gamma' \vdash E_2 \Leftarrow A_1 \rightsquigarrow e}{\mathcal{C}\, E_2 : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D}\, e}$$

CTyp-appR1

$$\frac{\Gamma' \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e \qquad \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Leftarrow A_1) \rightsquigarrow \mathcal{D}}{E_1\, \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow e\, \mathcal{D}}$$

CTyp-mergeL1

$$\frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1) \rightsquigarrow \mathcal{D} \qquad \Gamma' \vdash E_2 \Rightarrow A_2 \rightsquigarrow e \qquad A_1 * A_2}{\mathcal{C}\,,, E_2 : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle \mathcal{D}, e \rangle}$$

CTyp-mergeR1

$$\frac{\Gamma' \vdash E_1 \Rightarrow A_1 \rightsquigarrow e \qquad \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D} \qquad A_1 * A_2}{E_1\,,, \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle e, \mathcal{D} \rangle}$$

CTyp-anno1

$$\frac{\mathcal{C} : (\Gamma \Rightarrow B) \mapsto (\Gamma' \Leftarrow A) \rightsquigarrow \mathcal{D}}{\mathcal{C} : A : (\Gamma \Rightarrow B) \mapsto (\Gamma' \Rightarrow A) \rightsquigarrow \mathcal{D}}$$

CTyp-rcd1

$$\frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}{\{l = \mathcal{C}\} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}}$$

CTyp-proj1

$$\frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}}{\mathcal{C}.l : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}$$

Figure 5.2: λ_i^+ context typing (excerpt)

Theorem 5.1 (👉 Coherence). *We have that*

- *If $\Gamma \vdash E \Rightarrow A$ then $\Gamma \vdash E \simeq_{ctx} E : A$.*

- *If $\Gamma \vdash E \Leftarrow A$ then $\Gamma \vdash E \simeq_{ctx} E : A$.*

That is, coherence is a special case of Definition 9 where E_1 and E_2 are the same. At first glance, this appears underwhelming: of course E behaves the same as itself! The tricky part is that, if we expand it according to Definition 9, it is not E itself but all its translations e_1 and e_2 that behave the same! The rest of the chapter is devoted to proving the validity of Theorem 5.1.

5.3 The Canonicity Relation, Formally Defined

As intuitive as Definition 9 may seem, it is generally very hard to prove contextual equivalence directly, since it involves quantification over *all* possible contexts. Worse still, two kinds of contexts are involved in Theorem 5.1, which makes reasoning even more tedious. The key to simplifying the reasoning is to exploit types using logical relations [Plotkin 1973; Statman 1985; Tait 1967].

$$
\begin{aligned}
(v_1, v_2) \in \mathcal{V}[\![\mathsf{Int}; \mathsf{Int}]\!] &\triangleq \exists i.\ v_1 = v_2 = i \\
(v_1, v_2) \in \mathcal{V}[\![\{l : A\}; \{l : B\}]\!] &\triangleq (v_1, v_2) \in \mathcal{V}[\![A; B]\!] \\
(v_1, v_2) \in \mathcal{V}[\![A_1 \to B_1; A_2 \to B_2]\!] &\triangleq \forall (v_2', v_1') \in \mathcal{V}[\![A_2; A_1]\!].\ (v_1\ v_1', v_2\ v_2') \in \mathcal{E}[\![B_1; B_2]\!] \\
(\langle v_1, v_2 \rangle, v_3) \in \mathcal{V}[\![A \,\&\, B; C]\!] &\triangleq (v_1, v_3) \in \mathcal{V}[\![A; C]\!] \wedge (v_2, v_3) \in \mathcal{V}[\![B; C]\!] \\
(v_3, \langle v_1, v_2 \rangle) \in \mathcal{V}[\![C; A \,\&\, B]\!] &\triangleq (v_3, v_1) \in \mathcal{V}[\![C; A]\!] \wedge (v_3, v_2) \in \mathcal{V}[\![C; B]\!] \\
(v_1, v_2) \in \mathcal{V}[\![A; B]\!] &\triangleq \mathsf{true}\quad \mathsf{otherwise} \\
\\
(e_1, e_2) \in \mathcal{E}[\![A; B]\!] &\triangleq \exists v_1, v_2.\ e_1 \longrightarrow^* v_1 \wedge e_2 \longrightarrow^* v_2\ \wedge (v_1, v_2) \in \mathcal{V}[\![A; B]\!]
\end{aligned}
$$

Figure 5.3: ☞ The canonicity relation for λ_i^+

IN SEARCH OF A LOGICAL RELATION.   It is worth pausing to ponder what kind of relation we are looking for. The high-level intuition behind the relation is to capture the notion of "coherent" values. These values are unambiguous in all possible (source) contexts. A moment of thought leads us to the following observations:

Observation 1 (Disjoint values are unambiguous). The relation should relate values whose types are disjoint. Those values are essentially translated from merges, and since rule T-MERGE ensures disjointness, they are unambiguous. For example, one value of type Int and the other of $\{l : \mathsf{Int}\}$ can be unambiguously distinguished by any source context.

Observation 2 (Duplication is unambiguous). The relation should also relate values originating from non-disjoint intersection types, only if the values are duplicates. This may sound baffling, since the whole point of disjointness is to rule out (ambiguous) expressions such as $1 ,, 2$. However, $1 ,, 2$ never gets elaborated, and the only values corresponding to $\mathsf{Int} \,\&\, \mathsf{Int}$ are those pairs such as $\langle 1, 1 \rangle$, $\langle 2, 2 \rangle$, etc. Those values are essentially generated from rule T-SUB by subtyping and are also unambiguous.

THE CANONICITY RELATION.   In order to deal with the complexity of the elaboration semantics, we introduce in Fig. 5.3 what we call the *canonicity* relation to capture "canonical" values based on the above observations.[1] The canonicity relation is a family of binary relations over λ_{co} values that are *heterogeneous*, i.e., indexed by two λ_i^+ types. Heterogeneity allows us to relate values of different types, and in particular values whose types are disjoint. The canonicity relation seeks to combine equality checking from traditional (homogeneous) logical relations (Observation 2) with disjointness checking (Observation 1). It consists of two relations. The value relation $\mathcal{V}[\![A; B]\!]$ relates *closed* values, i.e., well-typed values with

---

[1]The logical relation is slightly different from that in the original publication [Bi et al. 2018] in that it is indexed by "source" types whereas in the publication it is indexed by "target" types. For λ_i^+, both formulations work equally fine. The choice here is mainly for consistency reasons as the logical relation for F_i^+ must be indexed by source types.

no free variables. Similarly, the expression relation $\mathcal{E}[\![A; B]\!]$ relates closed expressions. For brevity, we write $\mathcal{V}[\![A]\!]$ to mean $\mathcal{V}[\![A; A]\!]$, and $\mathcal{E}[\![A]\!]$ for $\mathcal{E}[\![A; A]\!]$.

First let us consider the relation $\mathcal{V}[\![A; B]\!]$, which specifies when two closed values v_1 and v_2 are related at the types A and B. The definition for integers and records are straightforward. Two integers are related if they are equal. For records, recall that in Section 3.4, record labels are erased during translation. Therefore two values are related at two record types of the same label if they are related at the two field types.

Functions v_1 and v_2 are related at the types $A_1 \to B_1$ and $A_2 \to B_2$ if given two arguments v_1' and v_2' related at the argument types A_1 and A_2, the functions applied to the arguments are related expressions at the result types B_1 and B_2. Note that in λ_{co}, the values v_1 and v_2 may each be a lambda abstraction, or a coercion application of a function type.

The definition of $\mathcal{V}[\![A; B]\!]$ is made more interesting when one of the indexed types is an intersection type. In that case, the relation distributes over the type constructor $\&$. It is instructive to compare the type constructor $\&$ with product types $\times$. The traditional way of relating pairs is by relating their components pairwise. That is, $\langle v_1, v_2 \rangle$ and $\langle v_1', v_2' \rangle$ are related at $A \times B$ if (1) v_1 and v_1' are related at A and (2) v_2 and v_2' are related at B. According to our definition, we also require that (3) v_1 and v_2' are related and (4) v_2 and v_1' are related. To see why this is the case, consider whether $(\langle 1, 2 \rangle, \langle 1, 2 \rangle) \in \mathcal{V}[\![\mathsf{Int} \& \mathsf{Int}]\!]$. If we regard $\mathsf{Int} \& \mathsf{Int}$ as a normal product type, then these two pairs are related. However, as remarked earlier, $\langle 1, 2 \rangle$ should not be considered as the image of some source expression at the type $\mathsf{Int} \& \mathsf{Int}$, and our definition correctly rejects it because 1 is not equal to 2, while accepting pairs such as $\langle 1, 1 \rangle, \langle 2, 2 \rangle$, etc.

The acute reader may have noticed the structural similarity between the two clauses for intersection types and the disjointness rules for intersection types:

D-ANDL
$$\frac{A_1 * B \qquad A_2 * B}{A_1 \& A_2 * B}$$

D-ANDR
$$\frac{A * B_1 \qquad A * B_2}{A * B_1 \& B_2}$$

This is not a coincidence—we can show that disjointness and the value relation are connected by the following lemma:

Lemma 5.2 (☞ Disjoint values are related)**.** *If* $A * B$ *and* $v_1 : |A|$ *and* $v_2 : |B|$, *then* $(v_1, v_2) \in \mathcal{V}[\![A; B]\!]$.

Proof. By induction on the derivation of disjointness. □

Next we consider $\mathcal{E}[\![A; B]\!]$, which is standard. Informally it expresses that two closed terms e_1 and e_2 are related if they evaluate to two values v_1 and v_2 that are related.

LOGICAL EQUIVALENCE.   The logical relation can be lifted to open terms in the usual way.
First we give the semantic interpretation of typing contexts. A *closing substitution* γ for
the typing context $\Gamma \;=\; x_1 \;:\; A_1, \ldots, x_n \;:\; A_n$ is a finite function assigning closed val-
ues $v_1 \;:\; |A_1|, \ldots, v_n \;:\; |A_n|$ to $x_1, \ldots, x_n$, respectively. We write $\gamma(e)$ for the substitution
$[v_1, \ldots, v_n / x_1, \ldots, x_n]e$. The interruption of typing contexts, written $(\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!]$ is
inductively defined as follows:

Definition 10 (☞ Interpretation of value contexts**).**

$$
\frac{}{(\emptyset, \emptyset) \in \mathcal{G}[\![\bullet]\!]}
\qquad
\frac{(\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!] \qquad (v_1, v_2) \in \mathcal{V}[\![A]\!]}{(\gamma_1[x \mapsto v_1], \gamma_2[x \mapsto v_2]) \in \mathcal{G}[\![\Gamma, x : A]\!]}
$$

The canonicity relation can be lifted to open expressions in the standard way, i.e., two open
terms are related if every pair of related closing substitutions makes them related:

Definition 11 (☞ Logical equivalence**).**

$$
\Gamma \vdash e_1 \simeq_{log} e_2 : A; B \triangleq |\Gamma| \vdash e_1 : |A| \wedge |\Gamma| \vdash e_2 : |B| \wedge
$$
$$
(\forall \gamma_1, \gamma_2. \, (\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!] \implies (\gamma_1(e_1), \gamma_2(e_2)) \in \mathcal{E}[\![A; B]\!])
$$

For succinctness, we write $\Gamma \vdash e_1 \simeq_{log} e_2 : A$ to mean $\Gamma \vdash e_1 \simeq_{log} e_2 : A; A$.

5.4 ESTABLISHING COHERENCE

With all the machinery in place, we are now ready to prove Theorem 5.1. But we need several
lemmas to set the stage.

Firstly we need the compatibility lemmas, which state that logical equivalence is preserved
by language constructs. Most of them are standard and are thus omitted. We show only two
compatibility lemmas that are specific to our logical relation:

Lemma 5.3 (☞ Coercion compatibility**).** *Suppose that* $A_1 <: A_2 \rightsquigarrow co,$

- *If* $\Gamma \vdash e_1 \simeq_{log} e_2 : A_1; A_0$ *then* $\Gamma \vdash co \, e_1 \simeq_{log} e_2 : A_2; A_0.$

- *If* $\Gamma \vdash e_1 \simeq_{log} e_2 : A_0; A_1$ *then* $\Gamma \vdash e_1 \simeq_{log} co \, e_2 : A_0; A_2.$

Proof. By induction on the subtyping derivation. □

Lemma 5.4 (☞ Merge compatibility**).** *If* $\Gamma \vdash e_1 \simeq_{log} e_1' : A$, $\Gamma \vdash e_2 \simeq_{log} e_2' : B$ *and* $A * B,$
then $\Gamma \vdash \langle e_1, e_2 \rangle \simeq_{log} \langle e_1', e_2' \rangle : A \,\&\, B.$

Proof. By the definition of logical relation and Lemma 5.2. □

The "Fundamental Property" states that any well-typed expression is related to itself by the logical relation. In our elaboration setting, we rephrase it so that any two λ_{co} terms elaborated from the *same λ_i^+* expression are related by the logical relation. To prove it, we require Theorem 5.5.

Theorem 5.5 (☞ Inference uniqueness). *If $\Gamma \vdash E \Rightarrow A_1$ and $\Gamma \vdash E \Rightarrow A_2$, then $A_1 \equiv_\alpha A_2$.*

Theorem 5.6 (☞ Fundamental property). *We have that:*

- *If $\Gamma \vdash E \Rightarrow A \rightsquigarrow e$ and $\Gamma \vdash E \Rightarrow A \rightsquigarrow e'$, then $\Gamma \vdash e \simeq_{log} e' : A$.*

- *If $\Gamma \vdash E \Leftarrow A \rightsquigarrow e$ and $\Gamma \vdash E \Leftarrow A \rightsquigarrow e'$, then $\Gamma \vdash e \simeq_{log} e' : A$.*

Proof. The proof follows by induction on the first derivation. The most interesting case is rule T-SUB

$$
\frac{\Gamma \vdash E \Rightarrow A \rightsquigarrow e \qquad A <: B \rightsquigarrow co}{\Gamma \vdash E \Leftarrow B \rightsquigarrow co\, e} \text{ T-SUB}
$$

where we need Theorem 5.5 to be able to apply the induction hypothesis. Then we apply Lemma 5.3 to say that the coercion generated preserves the relation between terms. For the other cases we use the appropriate compatibility lemmas. □

We show that logical equivalence is preserved by λ_i^+ contexts:

Lemma 5.7 (☞ Congruence). *If $\mathcal{C} : (\Gamma \Leftrightarrow A) \mapsto (\Gamma' \Leftrightarrow' A') \rightsquigarrow \mathcal{D}$, $\Gamma \vdash E_1 \Leftrightarrow A \rightsquigarrow e_1$, $\Gamma \vdash E_2 \Leftrightarrow A \rightsquigarrow e_2$ and $\Gamma \vdash e_1 \simeq_{log} e_2 : A$, then $\Gamma' \vdash \mathcal{D}\{e_1\} \simeq_{log} \mathcal{D}\{e_2\} : A'$.*

Proof. By induction on the typing derivation of the context C, and applying the compatibility lemmas where appropriate. □

Lemma 5.8 (☞ Adequacy). *If $\bullet \vdash e_1 \simeq_{log} e_2 : \mathsf{Int}$ then $e_1 \simeq e_2$.*

Proof. Adequacy follows easily from the definition of the logical relation. □

Next up is the proof that logical relation is sound with respect to contextual equivalence—that is, if two programs are logically related then they are contextually equivalent—which justifies the use of logical relation for proving contextual equivalence of programs.

Theorem 5.9 (Soundness w.r.t. contextual equivalence). *Given* $\Gamma \vdash e_1 \simeq_{log} e_2 : A$, *we have*

- *If* $\Gamma \vdash E_1 \Rightarrow A \rightsquigarrow e_1$ *and* $\Gamma \vdash E_2 \Rightarrow A \rightsquigarrow e_2$ *then* $\Gamma \vdash E_1 \simeq_{ctx} E_2 : A$.

- *If* $\Gamma \vdash E_1 \Leftarrow A \rightsquigarrow e_1$ *and* $\Gamma \vdash E_2 \Leftarrow A \rightsquigarrow e_2$ *then* $\Gamma \vdash E_1 \simeq_{ctx} E_2 : A$.

Proof. From Definition 9, we are given a context $\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\bullet \Rightarrow \mathsf{Int}) \rightsquigarrow \mathcal{D}$. By Lemma 5.7 we have $\bullet \vdash \mathcal{D}\{e_1\} \simeq_{log} \mathcal{D}\{e_2\} : \mathsf{Int}$, thus $\mathcal{D}\{e_1\} \simeq \mathcal{D}\{e_2\}$ by Lemma 5.8. $\qquad\square$

Armed with Theorem 5.6 and Theorem 5.9, coherence follows directly.

Theorem 5.1 (☞ Coherence). *We have that*

- *If* $\Gamma \vdash E \Rightarrow A$ *then* $\Gamma \vdash E \simeq_{ctx} E : A$.

- *If* $\Gamma \vdash E \Leftarrow A$ *then* $\Gamma \vdash E \simeq_{ctx} E : A$.

Proof. Immediate from Theorem 5.6 and Theorem 5.9. $\qquad\qquad\qquad\qquad\square$

5.5 SOME INTERESTING COROLLARIES

To showcase the strength of the new proof method, we can derive some interesting corollaries. For the most part, they are direct consequences of logical equivalence which carry over to contextual equivalence.

Corollary 5.10 says that merging an expression E_1 of some type with an arbitrary expression E_2 does not affect the semantics of E_1 at the same type. Corollary 5.11 and Corollary 5.12 express that merges are commutative and associative, respectively. Corollary 5.13 states that coercions from the same types are "coherent", i.e., they can be used interchangeably.

Corollary 5.10 (Neutrality). *If* $\Gamma \vdash E_1 \Rightarrow A$ *and* $\Gamma \vdash E_1 ,, E_2 \Rightarrow A$, *then* $\Gamma \vdash E_1 \simeq_{ctx} E_1 ,, E_2 : A$

Corollary 5.11 (Commutativity). *If* $\Gamma \vdash E_1 ,, E_2 \Rightarrow A$ *and* $\Gamma \vdash E_2 ,, E_1 \Rightarrow A$, *then* $\Gamma \vdash E_1 ,, E_2 \simeq_{ctx} E_2 ,, E_1 : A$.

Corollary 5.12 (Associativity). *If* $\Gamma \vdash (E_1 ,, E_2) ,, E_3 \Rightarrow A$ *and* $\Gamma \vdash E_1 ,, (E_2 ,, E_3) \Rightarrow A$, *then* $\Gamma \vdash (E_1 ,, E_2) ,, E_3 \simeq_{ctx} E_1 ,, (E_2 ,, E_3) : A$.

Corollary 5.13 (Coercions Preserve Semantics). *If* $A <: B \rightsquigarrow co_1$ *and* $A <: B \rightsquigarrow co_2$, *then* $\Gamma \vdash \lambda x. co_1 \, x \simeq_{log} \lambda x. co_2 \, x : A \rightarrow B$.

6 COHERENCE FOR F_i^+

In this section, we establish the coherence property for F_i^+. The proof strategy mostly follows that of λ_i^+, but the construction of the heterogeneous logical relation is significantly more complicated. Firstly in Section 6.1 we discuss why adding BCD subtyping to disjoint polymorphism introduces significant complications. In Section 6.2, we discuss why a natural extension of System F's logical relation to deal with disjoint polymorphism fails. The technical difficulty is *well-foundedness*, stemming from the interaction between impredicativity and disjointness. Finally in Section 6.3, we present our (predicative) logical relation that is specially crafted to prove coherence for F_i^+.

6.1 THE CHALLENGE

Before we tackle the coherence of F_i^+, let us first consider how F_i (and its predecessor λ_i) enforces coherence. Its essentially syntactic approach is to make sure that there is at most one subtyping derivation for any two types. As an immediate consequence, the produced coercions are uniquely determined and thus the calculus is clearly coherent. Key to this approach is the invariant that the type system only produces *disjoint* intersection types. As we mentioned in Section 4.2, this invariant complicates the calculus and its metatheory, and leads to a weaker substitution lemma. Moreover, the syntactic coherence approach is incompatible with BCD subtyping, which leads to multiple subtyping derivations with different coercions and requires a more general substitution lemma. For example, consider the coercions produced by $\forall (\alpha * \mathsf{Int}).\, \alpha \,\&\, \alpha <: \forall (\alpha * \mathsf{Int} \,\&\, \mathsf{Int}).\, \alpha$ (neither type is "well-formed" in the sense of F_i). Two possible ones are $\lambda f.\, \Lambda \alpha.\, \pi_1\,(f\,\alpha)$ and $\lambda f.\, \Lambda \alpha.\, \pi_2\,(f\,\alpha)$. It is not at all obvious that they should be equivalent in an appropriate sense. As we have shown in Chapter 5, to prove coherence for λ_i^+, we need a semantically-founded proof method based on logical relations. Because λ_i^+ does not feature polymorphism, the problem at hand is to incorporate support for polymorphism in this semantic approach to coherence, which turns out to be more challenging than is apparent.

6.2 Impredicativity and Disjointness at Odds

Well-foundedness issues. For F_i^+, we need to extend canonicity with additional cases to account for universally quantified types. A naive formulation of one case rule is:

$$(v_1, v_2) \in \mathcal{V}[\![\forall(\alpha * A_1). B_1 ; \forall(\alpha * A_2). B_2]\!] \triangleq$$
$$\forall C_1 * A_1, C_2 * A_2. (v_1\,|C_1|, v_2\,|C_2|) \in \mathcal{E}[\![[C_1/\alpha]B_1 ; [C_2/\alpha]B_2]\!]$$

This case is problematic because it destroys the well-foundedness of λ_i^+'s logical relation, which is based on structural induction on the type indices. Indeed, the type $[C_1/\alpha]B_1$ may well be larger than $\forall(\alpha * A_1). B_1$.

However, System F's well-known parametricity logical relation [Reynolds 1983] provides us with a means to avoid this problem. Rather than performing the type substitution immediately as in the above rule, we can defer it to a later point by adding it to an extra parameter ρ of the relation, which accumulates the deferred substitutions. This yields a modified rule where the type indices in the recursive occurrences are indeed smaller:

$$(v_1, v_2) \in \mathcal{V}[\![\forall(\alpha * A_1). B_1 ; \forall(\alpha * A_2). B_2]\!]_\rho \triangleq$$
$$\forall C_1 * A_1, C_2 * A_2. (v_1\,|C_1|, v_2\,|C_2|) \in \mathcal{E}[\![B_1 ; B_2]\!]_{\rho[\alpha \mapsto (C_1, C_2)]}$$

Of course, the deferred substitution has to be performed eventually, to be precise when the type indices are type variables.

$$(v_1, v_2) \in \mathcal{V}[\![\alpha ; \alpha]\!]_\rho \triangleq (v_1, v_2) \in \mathcal{V}[\![\rho_1(\alpha) ; \rho_2(\alpha)]\!]_\emptyset$$

Unfortunately, this way we have not only moved the type substitution to the type variable case, but also the ill-foundedness problem. Indeed, this problem is also present in System F. The standard solution is to not fix the relation R by which values at type α are related to $\mathcal{V}[\![\rho_1(\alpha) ; \rho_2(\alpha)]\!]$, but instead to make it a parameter that is tracked by ρ. This yields the following two rules for disjoint quantification and type variables:

$$(v_1, v_2) \in \mathcal{V}[\![\forall(\alpha * A_1). B_1 ; \forall(\alpha * A_2). B_2]\!]_\rho \triangleq \forall C_1 * A_1, C_2 * A_2, \mathsf{R} \subseteq C_1 \times C_2.$$
$$(v_1\,|C_1|, v_2\,|C_2|) \in \mathcal{E}[\![B_1 ; B_2]\!]_{\rho[\alpha \mapsto (C_1, C_2, \mathsf{R})]}$$
$$(v_1, v_2) \in \mathcal{V}[\![\alpha ; \alpha]\!]_\rho \triangleq (v_1, v_2) \in \rho_\mathsf{R}(\alpha)$$

Now we have finally recovered the well-foundedness of the relation. It is again structurally inductive on the size of the type indexes.

HETEROGENEOUS ISSUES.   We have not yet accounted for one major difference between the parametricity relation, from which we have borrowed ideas, and the canonicity relation, to which we have been adding. The former is homogeneous (i.e., the types of the two values is the same) and therefore has one type index, while the latter is heterogeneous (i.e., the two values may have different types) and therefore has two type indices. Thus we must also consider cases like $\mathcal{V}[\![\alpha;\mathsf{Int}]\!]$. A definition that seems to handle this case appropriately is:

$$(v_1, v_2) \in \mathcal{V}[\![\alpha;\mathsf{Int}]\!]_\rho \triangleq (v_1, v_2) \in \mathcal{V}[\![\rho_1(\alpha);\mathsf{Int}]\!]_\emptyset \tag{6.1}$$

Here is an example to motivate this case. Let $E = \Lambda(\alpha * \top).\,(\lambda x.\,x) : \alpha\,\&\,\mathsf{Int} \to \alpha\,\&\,\mathsf{Int}$. We expect that $E\,\mathsf{Int}\,1 \longrightarrow^* \langle 1, 1 \rangle$, which boils down to showing $(1, 1) \in \mathcal{V}[\![\alpha;\mathsf{Int}]\!]_{[\alpha \mapsto (\mathsf{Int},\mathsf{Int},\mathsf{R})]}$. According to Eq. (6.1), this is indeed the case. However, we run into ill-foundedness issue again, because $\rho_1(\alpha)$ could be larger than α. Alas, this time the parametricity relation has no solution for us.

6.3 THE CANONICITY RELATION FOR F_i^+

In light of the fact that substitution in the logical relation seems unavoidable in our setting, and that impredicativity is at odds with substitution, we turn to *predicativity*: we change rule FT-TAPP to its predicative version:

FT-TAPPMONO
$$\frac{\Delta;\Gamma \vdash E \Rightarrow \forall(\alpha * B).\,C \rightsquigarrow e \qquad \Delta \vdash t * B}{\Delta;\Gamma \vdash E\,t \Rightarrow [t/\alpha]C \rightsquigarrow e\,|t|}$$

where metavariable t ranges over monotypes, whose syntax is shown below

$$\text{Monotypes} \quad t ::= \mathsf{Int} \mid \top \mid t_1 \to t_2 \mid t_1\,\&\,t_2 \mid \{l : t\} \mid \alpha$$

We do not believe that predicativity is a severe restriction in practice, since many source languages (e.g., those based on the Hindley-Milner type system [Hindley 1969; Milner 1978] like Haskell and OCaml) are themselves predicative and do not require the full generality of an impredicative core language.

Luckily because monotypes do not contain $\forall$-quantifiers, substitution with monotypes preserves well-foundedness. Figure 6.1 defines the *canonicity* relation for F_i^+. The canonicity relation is a family of binary relations over F_{co} values that are *heterogeneous*, i.e., indexed by two F_i^+ types. It consists of two relations: the value relation $\mathcal{V}[\![A;B]\!]$ relates *closed* values;

$$
\begin{aligned}
(v_1, v_2) \in \mathcal{V}[\![\mathsf{Int};\mathsf{Int}]\!] &\triangleq \exists i.\, v_1 = v_2 = i \\
(v_1, v_2) \in \mathcal{V}[\![\{l:A\};\{l:B\}]\!] &\triangleq (v_1, v_2) \in \mathcal{V}[\![A;B]\!] \\
(v_1, v_2) \in \mathcal{V}[\![A_1 \to B_1; A_2 \to B_2]\!] &\triangleq \forall (v_2', v_1') \in \mathcal{V}[\![A_2;A_1]\!].\, (v_1\, v_1', v_2\, v_2') \in \mathcal{E}[\![B_1;B_2]\!] \\
(\langle v_1, v_2 \rangle, v_3) \in \mathcal{V}[\![A \,\&\, B; C]\!] &\triangleq (v_1, v_3) \in \mathcal{V}[\![A;C]\!] \wedge (v_2, v_3) \in \mathcal{V}[\![B;C]\!] \\
(v_3, \langle v_1, v_2 \rangle) \in \mathcal{V}[\![C; A \,\&\, B]\!] &\triangleq (v_3, v_1) \in \mathcal{V}[\![C;A]\!] \wedge (v_3, v_2) \in \mathcal{V}[\![C;B]\!] \\
(v_1, v_2) \in \mathcal{V}[\![\forall(\alpha * A_1).\, B_1; \forall(\alpha * A_2).\, B_2]\!] &\triangleq \forall \bullet \vdash t * A_1 \,\&\, A_2.\, (v_1\, |t|, v_2\, |t|) \in \mathcal{E}[\![[t/\alpha]B_1; [t/\alpha]B_2]\!] \\
(v_1, v_2) \in \mathcal{V}[\![A;B]\!] &\triangleq \mathsf{true} \quad \text{otherwise}
\end{aligned}
$$

$$
(e_1, e_2) \in \mathcal{E}[\![A;B]\!] \triangleq \exists v_1, v_2.\, e_1 \longrightarrow^* v_1 \wedge e_2 \longrightarrow^* v_2 \,\wedge\, (v_1, v_2) \in \mathcal{V}[\![A;B]\!]
$$

Figure 6.1: The canonicity relation for F_i^+

and the expression relation $\mathcal{E}[\![A;B]\!]$—defined in terms of the value relation—relates closed expressions.

The relation $\mathcal{V}[\![A;B]\!]$ is defined by induction on the structures of A and B. For integers, it requires the two values to be literally the same. For two records to behave the same, their fields must behave the same. For two functions to behave the same, they are required to produce outputs related at B_1 and B_2 when given related inputs at A_1 and A_2. For the next two cases regarding intersection types, the relation distributes over intersection constructor $\&$. Of particular interest is the case for disjoint quantification. Notice that it *does not* quantify over arbitrary relations, but directly substitutes α with monotype t in B_1 and B_2. This means that our canonicity relation *does not* entail parametricity [Reynolds 1983]. However, it suffices for our purposes to prove coherence. Another noticeable thing is that the value relation $\mathcal{V}[\![A;B]\!]$ (and $\mathcal{E}[\![A;B]\!]$) keeps the invariant that A and B are closed types. As a result, type variables need not to be considered in the logical relation. This simplifies things a lot. Note that when one type is $\bot$, two values are vacuously related because there simply are no values of type $\bot$. We need to show that the relation is indeed well-founded:

Lemma 6.1 (Well-foundedness). *The canonicity relation of* F_i^+ *is well-founded.*

Proof. Let $|\cdot|_\forall$ and $|\cdot|_s$ be the number of $\forall$-quantifies and the size of types, respectively. We consider the measure $\langle |\cdot|_\forall, |\cdot|_s \rangle$, where $\langle \ldots \rangle$ denotes lexicographic order. For the case of disjoint quantification, the number of $\forall$-quantifiers decreases because monotype t does not contain $\forall$-quantifiers. For the other cases, the measure of $|\cdot|_\forall$ does not increase, and the measure of $|\cdot|_s$ strictly decreases. $\qquad \square$

The logical relation is symmetric:

Lemma 6.2 (☞ Symmetry). *If* $(v_1, v_2) \in \mathcal{V}[\![A;B]\!]$ *then* $(v_2, v_1) \in \mathcal{V}[\![B;A]\!]$.

Proof. Symmetry of Fig. 5.3 is trivial. For Fig. 6.1, the proof proceeds by first induction on $|A|_\forall$, then simultaneous induction on the structures of A and B. $\qquad \square$

The interpretations of type and term contexts are given below:

Definition 12 (☞ Interpretation of type contexts).

$$\frac{}{\emptyset \in \mathcal{D}[\![\bullet]\!]} \qquad \frac{\rho \in \mathcal{D}[\![\Delta]\!] \qquad \bullet \vdash t * \rho(B)}{\rho[\alpha \mapsto t] \in \mathcal{D}[\![\Delta, \alpha * B]\!]}$$

Definition 13 (☞ Interpretation of value contexts).

$$\frac{}{(\emptyset, \emptyset) \in \mathcal{G}[\![\bullet]\!]_\rho} \qquad \frac{(\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!]_\rho \qquad (v_1, v_2) \in \mathcal{V}[\![\rho(A)]\!]}{(\gamma_1[x \mapsto v_1], \gamma_2[x \mapsto v_2]) \in \mathcal{G}[\![\Gamma, x : A]\!]_\rho}$$

The connection between disjointness and the logical relation becomes more complicated due to the addition of polymorphism. We first show values of disjoint monotypes are related.

Lemma 6.3 (☞ Disjoint values of monotypes are related). *If* $\bullet \vdash t_1 * t_2$, $\bullet; \bullet \vdash v_1 : |t_1|$ *and* $\bullet; \bullet \vdash v_2 : |t_2|$ *then* $(v_1, v_2) \in \mathcal{V}[\![t_1; t_2]\!]$.

Proof. By simultaneous induction on t_1 and t_2. □

Then we can prove a more general lemma.

Lemma 6.4 (☞ Disjoint values are related). *If* $\Delta \vdash A * B$, $\rho \in \mathcal{D}[\![\Delta]\!]$, $\bullet; \bullet \vdash v_1 : |\rho(A)|$ *and* $\bullet; \bullet \vdash v_2 : |\rho(B)|$ *then* $(v_1, v_2) \in \mathcal{V}[\![\rho(A); \rho(B)]\!]$.

Proof. By induction on the derivation of disjointness. The most interesting case is the variable rule:

$$\frac{\text{FD-\textsc{tvarL}}}{\begin{array}{cc} (\alpha * A) \in \Delta & A <: B \end{array}}{\Delta \vdash \alpha * B}$$

By the definition of ρ, we know $\rho(\alpha)$ is a monotype. If B is a polytype, then it follows easily from the definition of logical relation. If B is also a monotype, $\rho(\alpha)$ and $\rho(A)$ are disjoint by definition. Then by Lemma 4.3 and $A <: B$, we have $\rho(\alpha)$ and $\rho(B)$ are also disjoint. Finally we apply Lemma 6.3. □

6.4 Establishing Coherence

We are now ready to prove coherence for F_i^+. The proof of coherence basically follows that in Chapter 5.

F_{co} contexts	$\mathcal{D}$	$::=$	$[\cdot] \mid \lambda x.\, \mathcal{D} \mid \Lambda \alpha.\, \mathcal{D} \mid \mathcal{D}\,\tau \mid \mathcal{D}\,e \mid e\,\mathcal{D} \mid \langle \mathcal{D}, e \rangle \mid \langle e, \mathcal{D} \rangle \mid co\,\mathcal{D}$
F_i^+ contexts	$\mathcal{C}$	$::=$	$[\cdot] \mid \lambda x.\, \mathcal{C} \mid \Lambda(\alpha * A).\, \mathcal{C} \mid \mathcal{C}\, A \mid \mathcal{C}\, E \mid E\, \mathcal{C} \mid \mathcal{C}\, ,,\, E \mid E\, ,,\, \mathcal{C}$
		$\mid$	$\{l = \mathcal{C}\} \mid \mathcal{C}.l \mid \mathcal{C} : A$

Figure 6.2: Expression contexts

LOGICAL EQUIVALENCE. The canonicity relation can be lifted to open expressions in the standard way, i.e., by considering all possible interpretations of free type and term variables.

Definition 14 (Logical equivalence $\simeq_{log}$).

$$\Delta; \Gamma \vdash e_1 \simeq_{log} e_2 : A; B \triangleq |\Delta|; |\Gamma| \vdash e_1 : |A| \wedge |\Delta|; |\Gamma| \vdash e_2 : |B| \wedge$$

$$(\forall \rho, \gamma_1, \gamma_2.\ \rho \in \mathcal{D}[\![\Delta]\!] \wedge (\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!]_\rho \implies (\gamma_1(\rho_1(e_1)), \gamma_2(\rho_2(e_2))) \in \mathcal{E}[\![\rho(A); \rho(B)]\!])$$

For conciseness, we write $\Delta; \Gamma \vdash e_1 \simeq_{log} e_2 : A$ to mean $\Delta; \Gamma \vdash e_1 \simeq_{log} e_2 : A; A$.

CONTEXTUAL EQUIVALENCE. Following λ_i^+, the notion of coherence is based on *contextual equivalence*. The intuition is that two programs are equivalent if we *cannot* tell them apart in any context. More formally, we introduce *expression contexts*, whose syntax is shown in Fig. 6.2. Due to the bidirectional nature of the type system, the typing judgment of C features 4 different forms (see Appendix E), e.g., $\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A') \rightsquigarrow \mathcal{D}$ reads if $\Delta; \Gamma \vdash E \Rightarrow A$ then $\Delta'; \Gamma' \vdash \mathcal{C}\{E\} \Rightarrow A'$. The judgment also generates a well-typed F_{co} context $\mathcal{D}$. The following definition capture the notion of contextual equivalence in F_i^+:

Definition 15 (F_i^+ Contextual Equivalence).

$$\Delta; \Gamma \vdash E_1 \simeq_{ctx} E_2 : A \triangleq \forall e_1, e_2.\ \Delta; \Gamma \vdash E_1 \Rightarrow A \rightsquigarrow e_1 \wedge \Delta; \Gamma \vdash E_2 \Rightarrow A \rightsquigarrow e_2 \wedge$$

$$(\forall C, \mathcal{D}.\ \mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\bullet; \bullet \Rightarrow \mathsf{Int}) \rightsquigarrow \mathcal{D} \implies \mathcal{D}\{e_1\} \simeq \mathcal{D}\{e_2\})$$

COHERENCE. We directly show the coherence statement of F_i^+. We need several technical lemmas such as compatibility lemmas, fundamental property, etc. The interested reader can refer to our Coq formalization.

Theorem 6.5 (☞ Coherence of F_i^+). *We have that*

- *If $\Delta; \Gamma \vdash E \Rightarrow A$ then $\Delta; \Gamma \vdash E \simeq_{ctx} E : A$.*

- *If $\Delta; \Gamma \vdash E \Leftarrow A$ then $\Delta; \Gamma \vdash E \simeq_{ctx} E : A$.*

Part IV

Applications

7 First-Class Traits

In this chapter and Chapter 8, we present two applications of F_i^+. This chapter is primarily concerned with building a source-level language called SEDEL that features *typed first-class traits*, *dynamic inheritance* and *nested composition* among others. We show how to model source-level constructs for first-class traits and dynamic inheritance, supporting standard object-oriented features such as dynamic dispatching and abstract methods. It is remarkable that all of these can be explained by plain F_i^+ expressions, showing its expressive power. In Chapter 8 we conduct a case study that modularizes programming language features by the means of first-class traits.

7.1 Motivation: First-Class Classes and Dynamic Inheritance

Many dynamically typed languages (including JavaScript, Ruby, Python or Racket) support *first-class classes* [Flatt et al. 2006], or related concepts such as first-class mixins and/or traits. In those languages classes are first-class values and, like any other values, they can be passed as an argument, or returned from a function. Furthermore, first-class classes support *dynamic inheritance*: i.e., they can inherit from other classes at *run time*, enabling programmers to abstract over the inheritance hierarchy. Those features make first-class classes very powerful and expressive, and enable highly modular and reusable pieces of code, such as:

```
const mixin = Base ⇒ {
  return class extends Base { ... }
};
```

In this piece of JavaScript code, `mixin` is parameterized by a class `Base`. Note that the concrete implementation of `Base` can be even dynamically determined at run time, for example after reading a configuration file to decide which class to use as the base class. When applied to an argument, `mixin` will create a new class on-the-fly and return that as a result. Later that class can be instantiated and used to create new objects, as any other classes.

In contrast, most statically typed languages do not have first-class classes and dynamic inheritance. While all statically typed object-oriented languages allow first-class *objects* (i.e., objects can be passed as arguments and returned as results), the same is not true for classes.

Classes in languages such as Scala, Java or C++ are typically a second-class construct, and the inheritance hierarchy is *statically determined*. The closest thing to first-class classes in languages like Java or Scala are classes such as `java.lang.Class` that enable representing classes and interfaces as part of their reflective framework. `java.lang.Class` can be used to mimic some of the uses of first-class classes, but in an essentially dynamically typed way. Furthermore, simulating first-class classes using such mechanisms is highly cumbersome because classes need to be manipulated programmatically. For example instantiating a new class cannot be done using the standard **new** construct, but rather requires going through API methods of `java.lang.Class`, such as `newInstance`, for creating a new instance of a class.

Despite the popularity and expressive power of first-class classes in dynamically typed languages, there is surprisingly little work on typing of first-class classes (or related concepts such as first-class mixins or traits). First-class classes and dynamic inheritance pose well-known difficulties in terms of typing. For example, in his thesis, Bracha [1992] comments several times on the difficulties of typing dynamic inheritance and first-class mixins, and proposes the restriction to static inheritance that is also common in statically typed languages. He also observes that such restriction poses severe limitations in terms of expressiveness, but that appeared (at the time) to be a necessary compromise when typing was also desired. Only recently some progress has been made in statically typing first-class classes and dynamic inheritance. In particular there are two works in this area: Racket's gradually typed first-class classes [Takikawa et al. 2012]; and Lee et al.'s model of typed first-class classes [Lee et al. 2015]. Both works provide typed models of first-class classes, and they enable encodings of mixins [Bracha and Cook 1990] similar to those employed in dynamically typed languages.

However, as far as we known no previous work supports statically typed *first-class traits*. Traits [Ducasse et al. 2006; Schärli et al. 2003] are an alternative to mixins, and other models of (multiple) inheritance. The key difference between traits and mixins lies on the treatment of conflicts when composing multiple traits/mixins. Mixins adopt an *implicit* resolution strategy for conflicts, where the compiler automatically picks one implementation in case of conflicts. For example, Scala uses the order of mixin composition to determine which implementation to pick in case of conflicts. Traits, on the other hand, employ an *explicit* resolution strategy, where the compositions with conflicts are rejected, and the conflicts are explicitly resolved by programmers. This gives programmers fine-grained control, when conflicts arise, of selecting desired features from different components. Thus we believe traits are a better model for multiple inheritance in statically typed object-oriented languages.

7.2 Overview

This section aims at introducing first-class classes and traits, their possible uses and applications, as well as the typing challenges that arise from their use. We start by describing a hypothetical JavaScript library for text editing widgets, inspired and adapted from Racket's GUI toolkit [Takikawa et al. 2012]. The example is illustrative of typical uses of dynamic inheritance/composition, and also the typing challenges in the presence of first-class classes/traits. Without diving into technical details, we then give the corresponding typed version in SEDEL, and informally presents its salient features.

7.2.1 First-Class Classes in JavaScript

A class construct was officially added to JavaScript in the ECMAScript 2015 Language Specification [ECMA International 2015]. One purpose of adding classes to JavaScript was to support a construct that is more familiar to programmers who come from mainstream class-based languages, such as Java or C++. However classes in JavaScript are *first-class* and support functionality not easily mimicked in statically typed class-based languages.

CONVENTIONAL CLASSES. Before diving into the more advanced features of JavaScript classes, we first review the more conventional class declarations supported in JavaScript as well as many other languages. Even for conventional classes there are some interesting points to note about JavaScript that will be important when we move into a typed setting. An example of a JavaScript class declaration is:

```javascript
class Editor {
  onKey(key) {
    return "Pressing " + key;
  }
  doCut() {
    return this.onKey("C-x") + " for cutting text";
  }
  showHelp() {
    return "Version: " + this.version() + " Basic usage...";
  }
};
```

This form of class definition is standard and very similar to declarations in class-based languages (for example Java). The `Editor` class defines three methods: `onKey` for handling key events, `doCut` for cutting text and `showHelp` for displaying help message. For the purpose of demonstration, we elide the actual implementation, and replace it with plain messages.

97

We wish to bring the reader's attention to two points in the above class. Firstly, note that the `doCut` method is defined in terms of the `onKey` method via the keyword **this**. In other words the call to `onKey` is enabled by the *self* reference and is *dynamically dispatched* (i.e., the particular implementation of `onKey` will only be determined when the class or subclass is instantiated). Secondly, notice that there is no definition of the `version` method in the class body, but such method is used in the body of the `showHelp` method. In a dynamically typed language, such as JavaScript, using undefined methods is error prone—accidentally instantiating `Editor` and then calling `showHelp` will cause a run-time error! Statically-typed languages usually provide some means to protect us from this situation. For example, in Java, we would need an *abstract* `version` method, which effectively makes `Editor` an abstract class and prevents it from being instantiated. As we will see, SEDEL's treatment of abstract methods is quite different from mainstream languages. In fact, SEDEL has a unified (typing) mechanism for dealing with both dynamic dispatch and abstract methods. We will describe SEDEL's mechanism for dealing with both features and justify our design in Section 7.3.

FIRST-CLASS CLASSES AND CLASS EXPRESSIONS.   Another way to define a class in JavaScript is via a *class expression*. This is where the class model in JavaScript is very different from the traditional class model found in many mainstream object-oriented languages, such as Java, where classes are second-class (static) entities. JavaScript embraces a dynamic class model that treats classes as *first-class* expressions: a function can take classes as arguments, or return them as a result. First-class classes enable programmers to abstract over patterns in the class hierarchy and to experiment with new forms of OOP such as mixins and traits. In particular, mixins become programmer-defined constructs. We illustrate this by presenting a simple mixin that adds spell checking to an editor:

```
const spellMixin = Base ⇒ {
  return class extends Base {
    check() {
      return super.onKey("C-c") + " for spell checking";
    }
    onKey(key) {
      return "Process " + key + " on spell editor";
    }
  }
};
```

DYNAMIC INHERITANCE.   In JavaScript, a mixin is simply a function with a superclass as input and a subclass extending that superclass as an output. Concretely, `spellMixin` adds a

method `check` for spell checking. It also provides a method `onKey`. The function `spellMixin` shows the typical use of what we call *dynamic inheritance*. Note that `Base`, which is supposed to be a superclass being inherited, is *parameterized*. Therefore `spellMixin` can be applied to any base class at *run time*. This is impossible to do, in a type-safe way, in conventional statically typed class-based languages like Java or C++.[1]

It is noteworthy that not all applications of `spellMixin` to base classes are successful. Notice the use of the **super** keyword in the `check` method. If the base class does not implement the `onKey` method, then mixin application will fail with a run-time error. In a typed setting, a type system must express this requirement (i.e., the presence of the `onKey` method) on the (statically unknown) base class being inherited.

We invite the reader to pause for a while and think about what the type of `spellMixin` would look like. Clearly our type system should be flexible enough to express this kind of dynamic pattern of composition in order to accommodate mixins (or traits), but also not too lenient to allow any composition.

Mixin composition and conflicts. The powerful part of mixins is that `spellMixin`'s functionality is not tied to a particular class hierarchy and is composable with other features. For example, we can define another mixin that adds simple modal editing—as in Vim—to an arbitrary editor:

```
const modalMixin = Base ⇒ {
  return class extends Base {
    constructor() {
      super();
      this.mode = "command";
    }
    toggleMode() {
      return "toggle succeeded";
    }
    onKey(key) {
      return "Process " + key + " on modal editor";
    }
  };
};
```

---

[1]With C++ templates, it is possible to implement a so-called mixin pattern [Smaragdakis and Batory 2000], which enables extending a parameterized class. However C++ templates defer type-checking until instantiation, and such pattern still does not allow selection of the base class at run time (only at up to class instantiation time).

`modalMixin` adds a `mode` field that controls which keybindings are active, initially set to the command mode, and a method `toggleMode` that is used to switch between modes. It also provides a method `onKey`.

Now we can compose `spellMixin` with `modalMixin` to produce a combination of functionality, mimicking some form of multiple inheritance:

```
class IDEEditor extends modalMixin(spellMixin(Editor)) {
  version() {
    return 0.2;
  }
}
```

The class `IDEEditor` extends the base class `Editor` with modal editing and spell checking capabilities. It also defines the missing `version` method.

At first glance, `IDEEditor` looks quite fine, but it has a subtle issue. Recall that two mixins `modalMixin` and `spellMixin` both provide a method `onKey`, and the `Editor` class also defines an `onKey` method of its own. Now we have a name clash. A question arises as to which one gets picked inside the `IDEEditor` class. A typical mixin model resolves this issue by looking at the order of mixin applications. Mixins appearing later in the order overrides *all* the identically named methods of earlier mixins. So in our case, `onKey` in `modalMixin` gets picked. If we change the order of application to `spellMixin(modalMixin(Editor))`, then `onKey` in `spellMixin` is inherited.

THE PROBLEM OF MIXIN COMPOSITION.   From the above discussion, we can see that mixin are composed linearly: all the mixins used by a class must be applied one at a time. However, when we wish to resolve conflicts by selecting features from different mixins, we may not be able to find a suitable order. For example, when we compose the two mixins to make the class `IDEEditor`, we can choose which of them comes first, but in either order, `IDEEditor` cannot access to the `onKey` method from the `Editor` class.

TRAIT MODEL.   Because of the total ordering and the limited means for resolving conflicts imposed by the mixin model, researchers have proposed a simple compositional model called traits [Ducasse et al. 2006; Schärli et al. 2003]. Traits are lightweight entities and serve as the primitive units of code reuse. Among others, the key difference from mixins is that the order of trait composition is irrelevant, and conflicting methods must be resolved *explicitly*. This gives programmers fine-grained control, when conflicts arise, of selecting desired features from different components. Thus we believe traits are a better model for multiple inheritance in statically typed object-oriented languages, and in SEDEL we realize this vision by giving

traits a first-class status in the language, achieving more expressive power compared with traditional (second-class) traits.

SUMMARY OF TYPING CHALLENGES.   From our previous discussion, we can identify the following typing challenges for a type system to accommodate the programming patterns (first-class classes/mixins) we have just seen:

- How to account for, in a typed way, abstract methods and dynamic dispatch.

- What are the types of first-class classes or mixins.

- How to type dynamic inheritance.

- How to express constraints on method presence and absence (the use of **super** clearly demands that).

- In the presence of first-class traits, how to detect conflicts statically, even when the traits involved are not statically known.

SEDEL elegantly solves the above challenges in a unified way, as we will see next.

7.2.2 A GLANCE AT TYPED FIRST-CLASS TRAITS IN SEDEL

We now rewrite the above code in SEDEL, but this time with types. The resulting code has the same functionality as the dynamic version, but it is statically typed. All code snippets in this and later sections are runnable in our prototype. Before proceeding, we ask the reader to bear in mind that in this section we are not using traits in the most canonical way, i.e., we use traits as if they are classes (but with built-in conflict detection). This is because we are trying to stay as close as possible to the structure of the JavaScript code for ease of comparison. In Section 7.3 we will remedy this to make better use of traits.

SIMPLE TRAITS.   Below is a simple trait `editor`, which corresponds to the JavaScript class `Editor`. The `editor` trait defines the same set of methods: `on_key`, `do_cut` and `show_help`:

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

The first thing to notice is that SEDEL uses a syntax (similar to Scala's self type annotations [Odersky et al. 2004]) where we can give a type annotation to the **self** reference. In the type of **self** we use & construct to create intersection types. `Editor` and `Version` are two record types:

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

For the sake of conciseness, SEDEL uses **type** aliases to abbreviate types.

THE TYPE OF **self** ENCODES ABSTRACT METHODS.   Recall that in the JavaScript class `Editor`, the version method is undefined, but is used inside `showHelp`. How can we express this in the typed setting, if not with an abstract method? In SEDEL, the type of **self** plays the role of trait requirements. As a first approximation, the invocation of version on **self** is justified by noticing that (part of) the type of **self** (i.e., `Version`) contains the declaration of version. An interesting aspect of SEDEL's trait model is that there is no need for abstract methods. Instead, abstract methods can be simulated as requirements of a trait. Later, when the trait is composed with other traits, *all* requirements on the type of **self** must be satisfied and one of the traits in the composition must provide an implementation of the method version.

As with the JavaScript code, the on_key method is invoked on **self** in the body of do_cut. This is allowed as (part of) the type of **self** (i.e., `Editor`) contains the signature of on_key. Compared to the JavaScript class `Editor`, almost everything stays the same, except that we now have a typed version. As a side note, since SEDEL is currently a pure "functional" object-oriented language, there is no difference between fields and methods, so we can omit empty arguments and parameter parentheses.

FIRST-CLASS TRAITS AND TRAIT EXPRESSIONS.   SEDEL treats traits as first-class expressions, putting them in the same syntactic category as objects, functions, and other primitive forms. To illustrate this, we give the SEDEL version of `spellMixin`:

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

```
    on_key : String → String
};


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

This looks daunting at first, but `spell_mixin` has almost the same structure as its JavaScript counterpart `spellMixin`, albeit with some type annotations. In SEDEL, we use capital letters (A, B, ...) to denote type variables, and trait expressions **trait** [**self** : ...] **inherits** ... ⇒ {...} to create first-class traits. Trait expressions have trait types of the form **Trait**[T1, T2] where T1 and T2 denote trait requirements and functionality respectively. We will explain trait types in Section 7.3. Despite the structural similarities, there are several significant features that are unique to SEDEL (e.g., the disjointness operator *). We discuss these in the following.

Disjoint polymorphism and conflict detection. SEDEL uses a type system based on *disjoint intersection types* (cf. Chapter 3) and *disjoint polymorphism* (cf. Chapter 4). Disjoint intersections empower SEDEL to detect conflicts statically when trying to compose two traits with identically named features. For example, composing two traits a and b that both provide `foo` gives a type error (the overloaded & operator denotes trait composition):

```
    trait a ⇒ { foo = 1 };
    trait b ⇒ { foo = 2 };
    trait c inherits a & b ⇒ {}; -- type error!
```

Disjoint polymorphism, as a more advanced mechanism, allows detecting conflicts even in the presence of polymorphism—for example when a trait is parameterized and its full set of methods is not statically known. As can be seen, `spell_mixin` is actually a polymorphic function. Unlike ordinary parametric polymorphism, in SEDEL, a type variable can also have a disjointness constraint. For instance, A * Spelling & OnKey means that A can be instantiated to any type as long as it *does not* contain `check` and `on_key`. Note that these are the minimal constraints of A, as (1) A cannot contain the `on_key` method because otherwise it will conflict with `Editor`; (2) A cannot contain the `check` method because otherwise it will conflict with that in the trait body. To mimic mixins, the argument `base`, which is supposed to be some trait, serves as the "base" trait being inherited. Notice that the type variable A appears in the type of `base`, which essentially states that `base` is a trait that contains at least

those methods specified by `Editor`, and possibly more (which we do not know statically). Also note that leaving out the **override** keyword will result in a type error. The type system is forcing us to be very specific as to what is the intention of the `on_key` method because it sees the same method is also declared in `base`, and blindly inheriting `base` will definitely cause a method conflict. As a final note, the use of **super** inside `check` is allowed because the "super-trait" `base` implements `on_key`, as can be seen from its type.

TYPING DYNAMIC INHERITANCE. Disjoint polymorphism enables us to correctly type dynamic inheritance: `spell_mixin` is able to take any trait that conforms with its assigned type, equips it with the `check` method and overrides its old `on_key` method. As a side note, the use of disjoint polymorphism is essential to correctly model the mixin semantics. From the type we know `base` has some features specified by `Editor`, plus something more denoted by `A`. By inheriting `base`, it is guaranteed that the resulting trait will have everything that is already contained in `base`, plus more features. This is in some sense similar to row polymorphism [Wand 1994] in that the result trait is prohibited from forgetting methods from the argument trait. (Section 10.3 has further discussion about the relationship between row polymorphism and disjoint polymorphism.)

TYPING MIXIN COMPOSITION. Next we give the typed version of `modalMixin` as follows:

```
type ModalEdit = {mode : String, toggle_mode : String};


modal_mixin [A * ModalEdit & OnKey]
             (base : Trait[Editor & Version, Editor & A]) =
  trait [self : Editor & Version] inherits base ⇒ {
    override on_key(key : String) = "Process" ++ key ++ "on modal editor";
    mode = "command";
    toggle_mode = "toggle succeeded"
  };
```

Now the definition of `modal_mixin` should be self-explanatory. Finally we can apply both "mixins" to `editor` one at a time to create an IDE editor:

```
type IDEEditor = Editor & Version & Spelling & ModalEdit;


trait ide_editor [self : IDEEditor]
  inherits modal_mixin Spelling (spell_mixin ⊤ editor) ⇒ {
    version = "0.2"
};
```

As with the JavaScript class `IDEEditor`, we need to fill in the missing `version` method. It is easy to verify that the `on_key` method in `modal_mixin` is inherited. Compared with the untyped version, here this behavior is reasonable because we specifically tag each `on_key` method to be an overriding method. Let us take a close look at the mixin applications. Since SEDEL is currently explicitly typed, we need to provide concrete types when applying `modal_mixin` and `spell_mixin`. In the inner application (`spell_mixin ⊤ editor`), we use the top type ⊤ to instantiate `A` because the `editor` trait provides exactly those method specified by `Editor` and nothing more (hence ⊤). In the outer application, we use `Spelling` to instantiate `A` because the resulting trait of the inner application contains the `check` method. In summary, mixin applications are simply normal function applications, and conflict resolution code is implicitly embedded via the keyword **override** and the order of mixin applications. Unsurprisingly, changing the mixin application order to

>   **inherits** `spell_mixin ModalEdit (modal_mixin ⊤ editor)`

gives the same difference as in the JavaScript version (`on_key` from `spell_mixin` is inherited).

Admittedly the typed version is unnecessarily complicated as we were mimicking mixins by functions over traits. The final trait `ide_editor` suffers from the same problem as the class `IDEEditor`, since there is no obvious way to access the `on_key` method in the `editor` trait.[2] Section 7.3 makes better use of traits to simplify the editor code.

7.3 Typed First-Class Traits

In Section 7.2 we have seen some examples of first-class traits at work in SEDEL. In this section we give a detailed account of SEDEL's support for typed first-class traits, to complement what has been presented so far. In doing so, we simplify the examples in Section 7.2 to make better use of traits. Section 7.4 presents the formal type system of first-class traits.

7.3.1 Traits in SEDEL

SEDEL supports a simple, yet expressive form of traits [Schärli et al. 2003]. Traits provide a simple mechanism for fine-grained code reuse, which can be regarded as a disciplined form of multiple inheritance. A trait is similar to a mixin in that it encapsulates a collection of related methods to be added to a class. The practical difference between traits and mixins is the way conflicting features that typically arise in multiple inheritance are dealt with. Instead of automatically resolved by scoping rules, conflicts are, in SEDEL, detected by the type system, and explicitly resolved by the programmer. Compared with traditional trait models,

---

[2]In fact, as we will see in Section 7.3.5, we can still access it by the forwarding operator.

there are three interesting points about SEDEL's traits: (1) they are *statically typed*; (2) they are *first-class* values; (3) they support *dynamic inheritance*. The support for such combination of features is one of the key novelties of SEDEL. Another minor difference from traditional traits (e.g., in Scala) is that, due to the use of structural types, a trait name is *not* a type.

7.3.2 Two Roles of Traits in SEDEL

Traits as templates for creating objects. An obvious difference between traits in SEDEL and many other models of traits [Fisher and Reppy 2004; Odersky and Zenger 2005; Schärli et al. 2003] is that they directly serve as templates for objects. In many other trait models, traits are complemented by classes, which take the responsibility for object creation. In particular, most models of traits do not allow constructors for traits. However, a trait in SEDEL has a single constructor of the same name. Take our last trait `ide_editor` in Section 7.2 for example:

```
a_editor1 = new[IDEEditor] ide_editor;
```

As with conventional object-oriented languages, the keyword **new** is used to create an object. A difference to other object-oriented languages is that the keyword **new** also specifies the intended type of the object. We instantiate the `ide_editor` trait and create an object `a_editor1` of type `IDEEditor`. As we will see in Section 7.3.4, constructors with parameters can also be expressed.

It is tempting to instantiate the `editor` trait such as **new**`[Editor] editor`. However this would result in a type error, because, as we discussed, `editor` has no definition of `version`, and blindly instantiating it would cause run-time errors. This behavior is on a par with Java's abstract classes—i.e., traits with undefined methods cannot be instantiated on their own.

Traits as units of code reuse. The traditional role of traits is to serve as units of code reuse. SEDEL's traits can have this role as well. Our `spell_mixin` function in Section 7.2 is more complicated than it should be. This is because we were mimicking classes as traits, and mixins as functions over traits. Instead, traits already provide a mechanism of code reuse. To illustrate this, we simplify `spell_mixin` as follows:

```
trait spell [self : OnKey] ⇒ {
  on_key(key : String) = "Process " ++ key ++ " on spell editor";
  check = self.on_key "C-c"  ++ " for spell checking"
};
```

This is much cleaner. The trait `spell` adds a method `check`. It also defines a method `on_key`. A key difference from `spell_mixin` is that `on_key` is invoked on the **self** parameter instead

of **super**. Note that this does not necessarily mean `check` will call `on_key` defined in the same trait. As we will see, the actual behavior entirely depends on how we compose `spell` with other traits. One minor difference is that we do not need to tag `on_key` with the **override** keyword, because `spell` stands as a standalone entity. Another interesting point is that the type of **self** (i.e., `OnKey`) is not the same as that of the trait body, which also contains the `check` method. In SEDEL's traits, the type of **self** serves as trait *requirements*.

CLASSES AND/OR TRAITS.   In the literature on traits [Ducasse et al. 2006; Schärli et al. 2003], the aforementioned two roles are considered as competing. One reason of the two roles conflicting in class-based languages is because a class must adopt a fixed position in the class hierarchy and therefore it can be difficult to reuse and resolve conflicts, whereas in SEDEL, a trait is a standalone entity and is not tied to any particular hierarchy. Therefore we can view our traits either as templates for creating objects, or as units of code reuse. Another important reason why our model can do just with traits is because we have a pure language. Mutable state can often only appear in classes in imperative models of traits, which is a good reason for having both classes and traits.

7.3.3 TRAIT TYPES AND TRAIT REQUIREMENTS

OBJECT TYPES AND TRAIT TYPES.   SEDEL adopts a relatively standard foundational model of object-oriented constructs [Lee et al. 2015] where objects are encoded as records with a structural type. This is why the object `a_editor1` has the record type `IDEEditor`. In SEDEL, an object type is different from a trait type. A trait type is specified via **Trait**[T1, T2].

TRAIT REQUIREMENTS AND FUNCTIONALITY.   In general, a trait type **Trait**[T1, T2] specifies both the *requirements* T1 and the *functionality* T2 of a trait. The requirements of a trait denote the types/methods that the trait needs to support for defining the functionality it provides. For example, `spell` has type **Trait**[OnKey, OnKey & Spelling], meaning that `spell` requires some implementation of the `on_key` method, and it provides implementations for the `on_key` and `check` methods. When a trait has no requirements, the absence of a requirement is denoted by using the top type ⊤. A simplified sugar **Trait**[T] is used to denote a trait without requirements, but providing functionality T.

TRAIT REQUIREMENTS AS ABSTRACT METHODS.   Let us go back to our very first trait `editor` in Section 7.2. Note how in `editor` the type of the **self** parameter is `Editor & Version`, where `Version` contains a declaration of the `version` method that is needed for the definition of `show_help`. Note also that the trait itself does not actually contain a `version` defini-

tion. In many other object-oriented models a similar program could be achieved by having an *abstract* definition of version. In SEDEL there are no abstract definitions (methods or fields), but a similar result can be achieved via trait requirements. Requirements of a trait are met at the object creation point. For example, as we mentioned before, the editor trait alone cannot be instantiated since it lacks version. However, when it is composed with a trait that provides version, the composition can be instantiated, as shown below:

```
trait foo ⇒ {
  version = "0.2"
};
bar = new[Editor & Version] foo & editor;
```

SEDEL uses a syntax where the self parameter can be explicitly named (not necessarily named **self**) with a type annotation. When the self parameter is omitted (for example in the foo trait above), its type defaults to ⊤. This is different from most object-oriented languages, where the default type of the self parameter is the same as the class being defined. This also makes trait requirements "pay as you go" in the sense that if the self parameter is not used in the body, then there is no requirements on the trait. Otherwise, suppose the type of the self parameter in the trait foo implicitly defaults to Version:

```
trait foo [self : Version] ⇒ {
  version = "0.2"
};
```

then Version will pollute the type of the self parameter of any trait that uses foo, cascading down the inheritance hierarchy, even though **self** is not used in the body of foo.

INTERSECTION TYPES MODEL SUBTYPING.   The type IDEEditor is defined as an intersection type. Intersection types [Coppo et al. 1981; Pottinger 1980] have been woven into many modern languages these days. A notable example is Scala, which makes fundamental use of intersection types to express a class/trait that extends multiple other traits. An intersection type such as T1 & T2 contains exactly those values which can be used as values of type T1 and of type T2, and as such, T1 & T2 immediately introduces a subtyping relation between itself and its two constituent types T1 and T2. Unsurprisingly, IDEEditor is a subtype of Editor.

7.3.4 TRAITS WITH PARAMETERS AND FIRST-CLASS TRAITS

So far our uses of traits involve no parameters. Instead of inventing another trait syntax with parameters, a trait with parameters is just a function that produces a trait expression, since functions already have parameters of their own. This is one benefit of having first-class traits

in terms of language economy. To illustrate, let us simplify `modal_mixin` in a similar way as in `spell_mixin`:

```
modal (init_mode : String) = trait ⇒ {
  on_key(key : String) = "Process " ++ key ++ " on modal editor";
  mode = init_mode;
  toggle_mode = "toggle succeeded"
};
```

The first thing to notice is that `modal` is a function with one argument, and returns a trait expression, which essentially makes `modal` a trait with one parameter. Now it is easy to see that a trait declaration **trait** name [**self** : ...] ⇒ {...} is just syntactic sugar for function definition name = **trait** [**self** : ...] ⇒ {...}. The body of the `modal` trait is straightforward. We initialize the `mode` field to `init_mode`. The `modal` trait also comes with a constructor with one parameter—e.g., we can create an object via **new**[ModalEdit] (modal "insert").

7.3.5 Detecting and Resolving Conflicts in Trait Composition

A common problem in multiple inheritance is how to detect and/or resolve conflicts. For example, when inheriting from two traits that have the same field, then it is unclear which implementation to choose. There are various approaches to dealing with conflicts. The trait-based approach requires conflicts to be resolved at the level of the composition, otherwise the program is rejected by the type system. SEDEL provides a means to help resolve conflicts.

We start by assembling all the traits defined in this section to create the final editor with the same functionality as `ide_editor` in Section 7.2. Our first try is as follows:

```
ide_editor (init_mode : String) =
  -- conflict
  trait [self : IDEEditor] inherits editor & spell & modal init_mode ⇒ {
    version = "0.2"
  };
```

Unfortunately the above trait gets rejected by SEDEL because `editor`, `spell` and `modal` all define an `on_key` method. Recall that in Section 7.2, when we use a mixin-style composition, the conflict resolution code has been hardwired in the definition. However, in a trait-style composition, this is not the case: conflicts must be resolved *explicitly*. The above definition is ill-typed precisely because there is a conflicting method `on_key`, thus violating the disjointness conditions imposed by disjoint intersection types.

Resolving conflicts. To resolve the conflict, we need to explicitly state which implementation of the method `on_key` gets to stay. SEDEL provides such a means—the *exclusion*

operator (denoted by \)—which allows one to exclude a field/method from a given trait. The
following matches the behavior in Section 7.2 where on_key from the modal trait is selected:

```
ide_editor (init_mode : String) =
  trait [self : IDEEditor]
    inherits (editor \ {on_key : String → String})
         & (spell \ {on_key : String → String})
         & (modal init_mode) ⇒ {
      version = "0.2"
  };
```

Now the above code type checks. We can also select on_key from the spell trait as easily:

```
ide_editor2 (init_mode : String) =
  trait [self : IDEEditor]
    inherits spell
         & (editor \ {on_key : String → String})
         & (modal init_mode) \ {on_key : String → String} ⇒ {
      version = "0.2"
  };
```

In Section 7.2 we mentioned that in the mixin style, it is impossible to select on_key from
the editor trait, but this is not a problem now:

```
ide_editor3 (init_mode : String) =
  trait [self : IDEEditor]
    inherits editor
         & (spell \ {on_key : String → String})
         & ((modal init_mode) \ {on_key : String → String}) ⇒ {
      version = "0.2"
  };
```

Using the exclusion operator, we can drop on_key in spell and modal while keeping it in
editor.

THE FORWARDING OPERATOR. Another operator that SEDEL provides is the *forwarding*
operator, which can be useful when we want to access some method that has been explicitly
excluded in the **inherits** clause. This is a common scenario in diamond inheritance, where
super is not enough. Below we show a variant of ide_editor:

```
ide_editor4 (init_mode : String) =
  trait [self : IDEEditor]
    inherits (editor \ {on_key : String → String})
         & (spell \ {on_key : String → String})
```

```
        & (modal init_mode) ⇒ {
   version = "0.2";
   override on_key(key : String) =
     super.on_key key ++ " and " ++ (spell ^ self).on_key key
};
```

Notice that `on_key` in `spell` has been excluded. However, we can still access it by using the forwarding operator as in `spell` **^ self**, which gives full access to all the methods in `spell`. Also note that using **super** only gives us access to `on_key` in the `modal` trait. To see `ide_editor4` in action, we create a small test:

```
a_editor2 = new[IDEEditor] (ide_editor4 "command");
main = a_editor2.do_cut
-- Output:
-- "Process C-x on modal editor and Process C-x on spell editor for
    cutting text"
```

7.3.6 DISJOINT POLYMORPHISM AND DYNAMIC COMPOSITION

SEDEL supports disjoint polymorphism. The combination of disjoint polymorphism and first-class traits enables the highly modular code where traits with *statically unknown* types can be instantiated and composed in a type-safe way! The following is illustrative of this:

```
mergeTraits A [B * A] (x : Trait[A]) (y : Trait[B]) = new[A & B] x & y;
```

The `mergeTraits` function takes two traits `x` and `y` of some arbitrary types **Trait**[A] and **Trait**[B], composes them, and creates an object from the resulting composed trait. Clearly such composition cannot always work if `A` and `B` can have conflicts. However, `mergeTraits` has a constraint `B * A` that ensures that whatever types are used to instantiate `A` and `B` they must be disjoint. Thus, under the assumption that `A` and `B` are disjoint the code type-checks.

7.4 FORMALIZING TYPED FIRST-CLASS TRAITS

This section presents the syntax and semantics of SEDEL. In particular, we show how to elaborate high-level source language constructs (self-references, abstract methods, first-class traits, dynamic inheritance and so on) to F_i^+. The treatment of the self-reference and dynamic dispatching is inspired by Cook's work on the denotational semantics for inheritance [Cook 1989; Cook and Palsberg 1989]. We then prove the elaboration is type safe, i.e., well-typed SEDEL expressions are translated to well-typed F_i^+ terms. Finally we show that SEDEL is coherent. All manual proofs about SEDEL can be found in Appendix C.

Types	$\mathcal{A}, \mathcal{B}, \mathcal{C}$	$::=$	$\top \mid \mathsf{Int} \mid \mathcal{A} \rightarrow \mathcal{B} \mid \mathcal{A} \,\&\, \mathcal{B} \mid \{l : \mathcal{A}\} \mid \alpha \mid \forall(\alpha * \mathcal{A}).\, \mathcal{B} \mid \boxed{\mathbf{Trait}\,[\mathcal{A}, \mathcal{B}]}$
Expressions	$\mathcal{T}$	$::=$	$\top \mid i \mid x \mid \lambda x.\mathcal{T} \mid \mathcal{T}_1\,\mathcal{T}_2 \mid \Lambda(\alpha * \mathcal{A}).\,\mathcal{T} \mid \mathcal{T}\,\mathcal{A} \mid \mathcal{T}_1 \,,,\, \mathcal{T}_2 \mid \mathcal{T} : \mathcal{A}$
		$\mid$	$\{l = \mathcal{T}\} \mid \mathcal{T}.l \mid \mathbf{letrec}\,x : \mathcal{A} = \mathcal{T}_1\,\mathbf{in}\,\mathcal{T}_2 \mid \boxed{\mathbf{new}\,[\mathcal{A}](\,\overline{\mathcal{T}_i}^{\,i}\,) \mid \mathcal{T}_1 \wedge \mathcal{T}_2}$
		$\mid$	$\boxed{\mathbf{trait}\,[\mathsf{self} : \mathcal{B}]\,\mathbf{inherits}\,\overline{\mathcal{T}_i}^{\,i}\,\{\,\overline{l_j = \mathcal{T}_j'}^{\,j}\,\} : \mathcal{A}}$
Value Contexts	Γ	$::=$	$\bullet \mid \Gamma, x : \mathcal{A}$
Type Contexts	Δ	$::=$	$\bullet \mid \Delta, \alpha * \mathcal{A}$

Figure 7.1: SEDEL core syntax and syntactic abbreviations

7.4.1 SYNTAX

The core syntax of SEDEL is shown in Fig. 7.1, with trait related constructs highlighted . For brevity of the meta-theoretic study, we do not consider definitions, which can be added in standard ways.

TYPES. Metavariables $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ range over types. Types include a top type $\top$, integers Int, function types $\mathcal{A} \rightarrow \mathcal{B}$, intersection types $\mathcal{A} \,\&\, \mathcal{B}$, singleton record types $\{l : \mathcal{A}\}$, type variables α and disjoint (universal) quantification $\forall(\alpha * A).\, B$. The main novelty is the type of first-class traits $\mathbf{Trait}\,[\mathcal{A}, \mathcal{B}]$, which expresses the requirement $\mathcal{A}$ and the functionality $\mathcal{B}$. We will use $[\mathcal{A}/\alpha]\mathcal{B}$ to denote capture-avoiding substitution of $\mathcal{A}$ for α inside $\mathcal{B}$.

EXPRESSIONS. Metavariable $\mathcal{T}$ ranges over expressions. We start with constructs required to encode objects based on records: term variables x, lambda abstractions $\lambda x.\mathcal{T}$, function applications $\mathcal{T}_1\,\mathcal{T}_2$, singleton records $\{l = \mathcal{T}\}$, record projections $\mathcal{T}.l$, recursive let bindings $\mathbf{letrec}\,x : \mathcal{A} = \mathcal{T}_1\,\mathbf{in}\,\mathcal{T}_2$, disjoint type abstraction $\Lambda(\alpha * \mathcal{A}).\,\mathcal{T}$ and type application $\mathcal{T}\,\mathcal{A}$. The calculus also supports a merge construct $\mathcal{T}_1 \,,,\, \mathcal{T}_2$ for creating values of intersection types and annotated expressions $\mathcal{T} : \mathcal{A}$. We also include a canonical top value $\top$ and literals i.

FIRST-CLASS TRAITS AND TRAIT EXPRESSIONS. Using the vector notation $\overline{\mathcal{T}}$ to indicate a sequence of zero or more $\mathcal{T}$ (i.e., $\mathcal{T}_1, \ldots, \mathcal{T}_n$), the central construct of SEDEL is the trait expression $\mathbf{trait}\,[\mathsf{self} : \mathcal{B}]\,\mathbf{inherits}\,\overline{\mathcal{T}_i}^{\,i}\,\{\,\overline{l_j = \mathcal{T}_j'}^{\,j}\,\} : \mathcal{A}$, which specifies a list of trait expressions $\overline{\mathcal{T}_i}$ in the $\mathbf{inherits}$ clause, an explicit self parameter (with type annotation B), and a set of methods $\{\overline{l_j = E_j'}\}$. Intuitively this trait expression has type $\mathbf{Trait}\,[\mathcal{B}, \mathcal{A}]$. Unlike the conventional trait model, a trait expression denotes a first-class value: it may occur anywhere where an expression is expected. Trait instantiation expressions $\mathbf{new}\,[\mathcal{A}](\,\overline{\mathcal{T}_i}^{\,i}\,)$ instantiate a composition of trait expressions $\overline{\mathcal{T}_i}$ to create an object of type $\mathcal{A}$. Finally $\mathcal{T}_1 \wedge \mathcal{T}_2$ is the forwarding expression, where $\mathcal{T}_1$ should be some trait.

$\boxed{\Delta \vdash \mathcal{A}}$ *(Well-formedness of types)*

WF-top

$$\overline{\Delta \vdash \top}$$

WF-int

$$\overline{\Delta \vdash \mathsf{Int}}$$

WF-arr
$$\frac{\Delta \vdash \mathcal{A} \qquad \Delta \vdash \mathcal{B}}{\Delta \vdash \mathcal{A} \to \mathcal{B}}$$

WF-rcd
$$\frac{\Delta \vdash \mathcal{A}}{\Delta \vdash \{l : \mathcal{A}\}}$$

WF-var
$$\frac{(\alpha * \mathcal{A}) \in \Delta}{\Delta \vdash \alpha}$$

WF-and
$$\frac{\Delta \vdash \mathcal{A} \qquad \Delta \vdash \mathcal{B}}{\Delta \vdash \mathcal{A} \,\&\, \mathcal{B}}$$

WF-forall
$$\frac{\Delta \vdash \mathcal{A} \qquad \Delta, \alpha * \mathcal{A} \vdash \mathcal{B}}{\Delta \vdash \forall (\alpha * \mathcal{A}).\, \mathcal{B}}$$

WF-trait
$$\frac{\Delta \vdash \mathcal{A} \qquad \Delta \vdash \mathcal{B}}{\Delta \vdash \mathbf{Trait}\,[\mathcal{A}, \mathcal{B}]}$$

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

TS-refl

$$\overline{\mathcal{A} <: \mathcal{A}}$$

TS-trans
$$\frac{\mathcal{A}_2 <: \mathcal{A}_3 \qquad \mathcal{A}_1 <: \mathcal{A}_2}{\mathcal{A}_1 <: \mathcal{A}_3}$$

TS-top

$$\overline{\mathcal{A} <: \top}$$

TS-rcd
$$\frac{\mathcal{A} <: \mathcal{B}}{\{l : \mathcal{A}\} <: \{l : \mathcal{B}\}}$$

TS-arr
$$\frac{\mathcal{B}_1 <: \mathcal{A}_1 \qquad \mathcal{A}_2 <: \mathcal{B}_2}{\mathcal{A}_1 \to \mathcal{A}_2 <: \mathcal{B}_1 \to \mathcal{B}_2}$$

TS-andL

$$\overline{\mathcal{A}_1 \,\&\, \mathcal{A}_2 <: \mathcal{A}_1}$$

TS-andR

$$\overline{\mathcal{A}_1 \,\&\, \mathcal{A}_2 <: \mathcal{A}_2}$$

TS-and
$$\frac{\mathcal{A}_1 <: \mathcal{A}_2 \qquad \mathcal{A}_1 <: \mathcal{A}_3}{\mathcal{A}_1 <: \mathcal{A}_2 \,\&\, \mathcal{A}_3}$$

TS-distArr

$$\overline{(\mathcal{A}_1 \to \mathcal{A}_2) \,\&\, (\mathcal{A}_1 \to \mathcal{A}_3) <: \mathcal{A}_1 \to \mathcal{A}_2 \,\&\, \mathcal{A}_3}$$

TS-topArr

$$\overline{\top <: \top \to \top}$$

TS-distRcd

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

TS-topRcd

$$\overline{\top <: \{l : \top\}}$$

TS-trait
$$\frac{\mathcal{B}_1 <: \mathcal{A}_1 \qquad \mathcal{A}_2 <: \mathcal{B}_2}{\mathbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] <: \mathbf{Trait}\,[\mathcal{B}_1, \mathcal{B}_2]}$$

Figure 7.2: Well-formedness and subtyping of SEDEL

7.4.2 Semantics

SUBTYPING AND WELL-FORMEDNESS. Figure 7.2 shows the well-formedness and subtyping rules for SEDEL. The well-formedness rule for trait types (WF-trait) is straightforward. The subtyping rule for trait types (TS-trait) resembles the one for function types in that it is contravariant on the first type $\mathcal{A}$ and covariant on the second type $\mathcal{B}$. The rest of the rules are direct analogies of F_i^+.

DISJOINTNESS. Figure 7.3 shows the disjointness rules for traits. The disjointness checking is the underlying mechanism of conflict detection. We naturally extend the disjointness rules in F_i^+ to cover trait types. Here we discuss the rules related with traits. Rule SD-trait says that as long as the functionalities that two traits provide are disjoint, the two trait types are disjoint. Rules SD-traitArr1 and SD-traitArr2 deal with situations where one of the two

$\boxed{\Delta \vdash \mathcal{A} * \mathcal{B}}$ *(Disjointness)*

SD-topL

$$\Delta \vdash \top * \mathcal{A}$$

SD-topR

$$\Delta \vdash \mathcal{A} * \top$$

SD-arr

$$\frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \mathcal{A}_1 \to \mathcal{A}_2 * \mathcal{B}_1 \to \mathcal{B}_2}$$

SD-andL

$$\frac{\Delta \vdash \mathcal{A}_1 * \mathcal{B} \qquad \Delta \vdash \mathcal{A}_2 * \mathcal{B}}{\Delta \vdash \mathcal{A}_1 \,\&\, \mathcal{A}_2 * \mathcal{B}}$$

SD-andR

$$\frac{\Delta \vdash \mathcal{A} * \mathcal{B}_1 \qquad \Delta \vdash \mathcal{A} * \mathcal{B}_2}{\Delta \vdash \mathcal{A} * \mathcal{B}_1 \,\&\, \mathcal{B}_2}$$

SD-rcdEq

$$\frac{\Delta \vdash \mathcal{A} * \mathcal{B}}{\Delta \vdash \{l : \mathcal{A}\} * \{l : \mathcal{B}\}}$$

SD-rcdNeq

$$\frac{l_1 \neq l_2}{\Delta \vdash \{l_1 : \mathcal{A}\} * \{l_2 : \mathcal{B}\}}$$

SD-tvarL

$$\frac{(\alpha * \mathcal{A}) \in \Delta \qquad \mathcal{A} <: \mathcal{B}}{\Delta \vdash \alpha * \mathcal{B}}$$

SD-tvarR

$$\frac{(\alpha * \mathcal{A}) \in \Delta \qquad \mathcal{A} <: \mathcal{B}}{\Delta \vdash \mathcal{B} * \alpha}$$

SD-forall

$$\frac{\Delta, \alpha * \mathcal{A}_1 \,\&\, \mathcal{A}_2 \vdash \mathcal{B}_1 * \mathcal{B}_2}{\Delta \vdash \forall(\alpha * \mathcal{A}_1). \mathcal{B}_1 * \forall(\alpha * \mathcal{A}_2). \mathcal{B}_2}$$

SD-trait

$$\frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \mathbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] * \mathbf{Trait}\,[\mathcal{B}_1, \mathcal{B}_2]}$$

SD-traitArr1

$$\frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \mathbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] * \mathcal{B}_1 \to \mathcal{B}_2}$$

SD-traitArr2

$$\frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \mathcal{A}_1 \to \mathcal{A}_2 * \mathbf{Trait}\,[\mathcal{B}_1, \mathcal{B}_2]}$$

SD-ax

$$\frac{\mathcal{A} *_{ax} \mathcal{B}}{\Delta \vdash \mathcal{A} * \mathcal{B}}$$

$\boxed{\mathcal{A} *_{ax} \mathcal{B}}$ *(Disjointness axiom)*

SDax-sym

$$\frac{\mathcal{B} *_{ax} \mathcal{A}}{\mathcal{A} *_{ax} \mathcal{B}}$$

SDax-intArr

$$\mathsf{Int} *_{ax} \mathcal{A}_1 \to \mathcal{A}_2$$

SDax-intRcd

$$\mathsf{Int} *_{ax} \{l : \mathcal{A}\}$$

SDax-intAll

$$\mathsf{Int} *_{ax} \forall(\alpha * \mathcal{B}_1). \mathcal{B}_2$$

SDax-arrAll

$$\mathcal{A}_1 \to \mathcal{A}_2 *_{ax} \forall(\alpha * \mathcal{B}_1). \mathcal{B}_2$$

SDax-arrRcd

$$\mathcal{A}_1 \to \mathcal{A}_2 *_{ax} \{l : \mathcal{B}\}$$

SDax-allRcd

$$\forall(\alpha * \mathcal{A}_1). \mathcal{A}_2 *_{ax} \{l : \mathcal{B}\}$$

SDax-intTrait

$$\mathsf{Int} *_{ax} \mathbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2]$$

SDax-traitAll

$$\mathbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] *_{ax} \forall(\alpha * \mathcal{B}_1). \mathcal{B}_2$$

SDax-traitRcd

$$\mathbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] *_{ax} \{l : \mathcal{B}\}$$

Figure 7.3: Disjointness rules of SEDEL

types is a function type. At first glance, these two look strange because a trait type is *different* from a function type, and they ought to be disjoint as an axiom. The reason is that SEDEL has an elaboration semantics, and as we will see, trait types are translated to function types. In order to ensure the type safety of elaboration, we have to have special treatment for trait and function types. In principle, if SEDEL has its own dynamic semantics, then trait types are always disjoint with function types.

TYPING TRAITS. The typing rules of trait related constructs are shown in Fig. 7.4. The reader is advised to ignore the translation parts ($\leadsto E$) for now. As with F_i^+, SEDEL employs two modes: the inference mode ($\Rightarrow$) and the checking mode ($\Leftarrow$). The inference judgment $\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \mathcal{A}$ says that we can synthesize a type $\mathcal{A}$ for expression $\mathcal{T}$. The checking judgment $\Delta; \Gamma \vdash \mathcal{T} \Leftarrow \mathcal{A}$ checks $\mathcal{T}$ against $\mathcal{A}$.

To type-check a trait (rule ST-TRAIT) we first type-check if its inherited traits $\overline{\mathcal{T}_i}$ are valid traits. Note that each trait $\mathcal{T}_i$ can possibly refer to self. Methods must all be well-typed in the usual sense. Apart from these, we have several side-conditions to make sure traits are well-behaved. The disjointness judgment $\Delta \vdash \mathcal{C}_1 * .. * \mathcal{C}_n * \mathcal{C}$ ensures that we do not have conflicting methods (in inherited traits and the body). The subtyping judgments $\overline{\mathcal{B} <: \mathcal{B}_i}$ ensure that the self parameter satisfies the requirements imposed by each inherited trait. Finally the subtyping judgment $\mathcal{C}_1 \& .. \& \mathcal{C}_n \& \mathcal{C} <: \mathcal{A}$ sanity-checks that the assigned type $\mathcal{A}$ is compatible.

Trait instantiation (rule ST-NEW) requires that each instantiated trait is valid. There are also several side-conditions, which serve the same purposes as in rule ST-TRAIT. Rule ST-FORWARD says that the first operand $\mathcal{T}_1$ of the forwarding operator must be a trait. Moreover, the type of the second operand $\mathcal{T}_2$ must satisfy the requirement of $\mathcal{T}_1$.

TREATMENTS OF SUPER, EXCLUSION AND OVERRIDE. We also include a variant (rule ST-TRAITSUPER) of rule ST-TRAIT where it implicitly assumes a super variable pointing to the **inherits** clause in the context when type checking the trait body.

ST-TRAITSUPER
$$\frac{\begin{array}{c}\overline{\Delta; \Gamma, \mathsf{self} : \mathcal{B} \vdash \mathcal{T}_i \Rightarrow \mathbf{Trait}\,[\mathcal{B}_i, \mathcal{C}_i] \leadsto E_i}^{i \in 1..n} \\ \Delta; \Gamma, \mathsf{self} : \mathcal{B}, \mathsf{super} : \mathcal{C}_1 \& .. \& \mathcal{C}_n \vdash \{\, \overline{l_j = \mathcal{T}_j}^{j \in 1..m} \,\} \Rightarrow \mathcal{C} \leadsto E \\ \overline{\mathcal{B} <: \mathcal{B}_i}^{i \in 1..n} \qquad \Delta \vdash \mathcal{C}_1 * .. * \mathcal{C}_n * \mathcal{C} \qquad \mathcal{C}_1 \& .. \& \mathcal{C}_n \& \mathcal{C} <: \mathcal{A} \end{array}}{\begin{array}{c}\Delta; \Gamma \vdash \mathbf{trait}\,[\mathsf{self} : \mathcal{B}]\,\mathbf{inherits}\,\overline{\mathcal{T}_i}^{i \in 1..n}\,\{\, \overline{l_j = \mathcal{T}_j}^{j \in 1..m} \,\} : \mathcal{A} \Rightarrow \mathbf{Trait}\,[\mathcal{B}, \mathcal{A}] \leadsto \\ \lambda \mathsf{self} : |\mathcal{B}|.\,(\mathbf{let}\,\mathsf{super} = \overline{(E_i\,\mathsf{self})}^{i \in 1..n}\,\mathbf{in}\,\mathsf{super}\,,, E) \end{array}}$$

$$\boxed{\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \mathcal{A} \rightsquigarrow E}$$ *(Inference)*

ST-TOP

$$\Delta; \Gamma \vdash \top \Rightarrow \top \rightsquigarrow \top$$

ST-INT

$$\Delta; \Gamma \vdash i \Rightarrow \mathsf{Int} \rightsquigarrow i$$

ST-VAR
$$\frac{(x : \mathcal{A}) \in \Delta}{\Delta; \Gamma \vdash x \Rightarrow \mathcal{A} \rightsquigarrow x}$$

ST-APP
$$\frac{\Delta; \Gamma \vdash \mathcal{T}_1 \Rightarrow \mathcal{A}_1 \rightarrow \mathcal{A}_2 \rightsquigarrow E_1 \qquad \Delta; \Gamma \vdash \mathcal{T}_2 \Leftarrow \mathcal{A}_1 \rightsquigarrow E_2}{\Delta; \Gamma \vdash \mathcal{T}_1 \, \mathcal{T}_2 \Rightarrow \mathcal{A}_2 \rightsquigarrow E_1 \, E_2}$$

ST-MERGE
$$\frac{\Delta; \Gamma \vdash \mathcal{T}_1 \Rightarrow \mathcal{A} \rightsquigarrow E_1 \qquad \Delta; \Gamma \vdash \mathcal{T}_2 \Rightarrow \mathcal{B} \rightsquigarrow E_2 \qquad \Delta \vdash \mathcal{A} * \mathcal{B}}{\Delta; \Gamma \vdash \mathcal{T}_1 \,,, \mathcal{T}_2 \Rightarrow \mathcal{A} \& \mathcal{B} \rightsquigarrow E_1 \,,, E_2}$$

ST-ANNO
$$\frac{\Delta; \Gamma \vdash \mathcal{T} \Leftarrow \mathcal{A} \rightsquigarrow E}{\Delta; \Gamma \vdash \mathcal{T} : \mathcal{A} \Rightarrow \mathcal{A} \rightsquigarrow E : |\mathcal{A}|}$$

ST-TABS
$$\frac{\Delta \vdash \mathcal{A} \qquad \Delta, \alpha * \mathcal{A}; \Gamma \vdash \mathcal{T} \Rightarrow \mathcal{B} \rightsquigarrow E}{\Delta; \Gamma \vdash \Lambda(\alpha * \mathcal{A}). \mathcal{T} \Rightarrow \forall (\alpha * \mathcal{A}). \mathcal{B} \rightsquigarrow \Lambda(\alpha * |\mathcal{A}|). E}$$

ST-TAPP
$$\frac{\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \forall (\alpha * \mathcal{B}_1). \mathcal{B}_2 \rightsquigarrow E \qquad \Delta \vdash \mathcal{A} \qquad \Delta \vdash \mathcal{A} * \mathcal{B}_1}{\Delta; \Gamma \vdash \mathcal{T} \, \mathcal{A} \Rightarrow [\mathcal{A}/\alpha]\mathcal{B}_2 \rightsquigarrow E \, |\mathcal{A}|}$$

ST-RCD
$$\frac{\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \mathcal{A} \rightsquigarrow E}{\Delta; \Gamma \vdash \{l = \mathcal{T}\} \Rightarrow \{l : \mathcal{A}\} \rightsquigarrow \{l = E\}}$$

ST-PROJ
$$\frac{\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \{l : \mathcal{A}\} \rightsquigarrow E}{\Delta; \Gamma \vdash \mathcal{T}.l \Rightarrow \mathcal{A} \rightsquigarrow E.l}$$

ST-TRAIT
$$\frac{\overline{\Delta; \Gamma, \mathsf{self} : \mathcal{B} \vdash \mathcal{T}_i \Rightarrow \mathbf{Trait}\,[\mathcal{B}_i, \mathcal{C}_i] \rightsquigarrow E_i}^{i \in 1..n} \quad \Delta; \Gamma, \mathsf{self} : \mathcal{B} \vdash \{\overline{l_j = \mathcal{T}_j'}^{j \in 1..m}\} \Rightarrow \mathcal{C} \rightsquigarrow E \quad \overline{\mathcal{B} <: \mathcal{B}_i}^{i \in 1..n} \quad \Delta \vdash \mathcal{C}_1 * .. * \mathcal{C}_n * \mathcal{C} \quad \mathcal{C}_1 \& .. \& \mathcal{C}_n \& \mathcal{C} <: \mathcal{A}}{\Delta; \Gamma \vdash \mathbf{trait}\,[\mathsf{self} : \mathcal{B}]\,\mathbf{inherits}\,\overline{\mathcal{T}_i}^{i \in 1..n} \{\overline{l_j = \mathcal{T}_j'}^{j \in 1..m}\} : \mathcal{A} \Rightarrow \mathbf{Trait}\,[\mathcal{B}, \mathcal{A}] \rightsquigarrow \lambda \mathsf{self} : |\mathcal{B}|. (((\overline{(E_i\,\mathsf{self})}^{i \in 1..n}) \,,, E)}$$

ST-NEW
$$\frac{\overline{\Delta; \Gamma \vdash \mathcal{T}_i \Rightarrow \mathbf{Trait}\,[\mathcal{A}_i, \mathcal{B}_i] \rightsquigarrow E_i}^{i \in 1..n} \quad \overline{\mathcal{A} <: \mathcal{A}_i}^{i \in 1..n} \quad \Delta \vdash \mathcal{B}_1 * .. * \mathcal{B}_n \quad \mathcal{B}_1 \& .. \& \mathcal{B}_n <: \mathcal{A}}{\Delta; \Gamma \vdash \mathbf{new}\,[\mathcal{A}](\overline{\mathcal{T}_i}^{i \in 1..n}) \Rightarrow \mathcal{A} \rightsquigarrow \mathbf{letrec}\,\mathsf{self} : |\mathcal{A}| = \overline{(E_i\,\mathsf{self})}^{i \in 1..n}\,\mathbf{in}\,\mathsf{self}}$$

ST-FORWARD
$$\frac{\Delta; \Gamma \vdash \mathcal{T}_1 \Rightarrow \mathbf{Trait}\,[\mathcal{A}, \mathcal{B}] \rightsquigarrow E_1 \qquad \Delta; \Gamma \vdash \mathcal{T}_2 \Leftarrow \mathcal{A} \rightsquigarrow E_2}{\Delta; \Gamma \vdash \mathcal{T}_1 \wedge \mathcal{T}_2 \Rightarrow \mathcal{B} \rightsquigarrow E_1 \, E_2}$$

$$\boxed{\Delta; \Gamma \vdash \mathcal{T} \Leftarrow \mathcal{A} \rightsquigarrow E}$$ *(Checking)*

ST-ABS
$$\frac{\Delta \vdash \mathcal{A} \qquad \Delta; \Gamma, x : \mathcal{A} \vdash \mathcal{T} \Leftarrow \mathcal{B} \rightsquigarrow E}{\Delta; \Gamma \vdash \lambda x.\mathcal{T} \Leftarrow \mathcal{A} \rightarrow \mathcal{B} \rightsquigarrow \lambda x.\, E}$$

ST-SUB
$$\frac{\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \mathcal{A} \rightsquigarrow E \qquad \mathcal{A} <: \mathcal{B} \qquad \Delta \vdash \mathcal{B}}{\Delta; \Gamma \vdash \mathcal{T} \Leftarrow \mathcal{B} \rightsquigarrow E}$$

Figure 7.4: Typing rules of SEDEL

One may have also noticed that in Fig. 7.1 we did not include the exclusion operator in the core SEDEL syntax, neither does **override** appear. The reason is that in principle all uses of the exclusion operator can be replaced by type annotations. For example to exclude a `bar` field from `{foo = a, bar = b, baz = c}`, all we need is to annotate the record with type `{foo : A, baz : C}` (suppose `a` has type `A`, etc). By the subsumption rule, the resulting record is guaranteed to contain no `bar` field. In the same vein, the use of **override** can be explained using the exclusion operator. We omit all of these features in the meta-theoretic study in order to focus our attention on the essence of first-class traits. However in practice, this is rather inconvenient as we need to write down all types we wish to retain rather than the one to exclude. So in our implementation we offer all of them.

ELABORATION. The operational semantics of SEDEL is given by means of a type-directed translation into F_i^+ extended with (lazy) recursive let bindings. This extension is standard and type-safe. Let us go back to Fig. 7.4, now focusing on the translation parts, which are regular F_i^+ terms. Most of them are straightforward translations and are thus omitted. We explain the most involved rules regarding traits. In rule INF-TRAIT, a trait is translated into a lambda abstraction with self as the formal parameter. In essence a trait corresponds to what Cook and Palsberg [1989] call a *generator*. The translations of the inherited traits (i.e., $\overline{E_i}$) are each applied to self and then merged with the translation of the trait body E. Now it is clear why we require $\mathcal{B}$ (the type of self) to be a subtype of each $\mathcal{B}_i$ (the requirement of each inherited trait). Note that we abuse the vector notation here with the intention that $\overline{(E_i\,\text{self})}^{\,i\in 1..n}$ means $E_1\,\text{self}\,,,\,..\,,,\,E_n\,\text{self}$. Here is an example of translating the `ide_editor` trait from Section 7.2 into plain F_i^+ terms equipped with definitions (suppose `modal_mixin` and `spell_mixin` have been translated accordingly):

```
ide_editor (self : IDEEditor) =
    (modal_mixin Spelling (spell_mixin ⊤ editor) self) ,, {version = "0.2"};
```

The translation of the **super** keyword in rule ST-TRAITSUPER is also straightforward. That is, it becomes a let binding super with the value $\overline{(E_i\,\text{self})}^{\,i\in 1..n}$, enabling access to the inherited traits.

Rule ST-NEW show the translation of trait instantiation. First we apply every translation (i.e., E_i) of the instantiated traits to the self parameter, and then merge the applications together. The bar notation is interpreted similarly to the translation in rule ST-TRAIT. Finally we compute the *lazy* fixed-point of the resulting merge term, i.e., self-reference must be updated to refer to the whole composition. Taking the fixed-point of the traits/generators again follows the denotational inheritance model by Cook and Palsberg [1989]. This is the key to the correct implementation of dynamic dispatching. Finally, rule ST-FORWARD translates

forwarding expressions to function applications. We show the translation of the `a_editor1` object in Section 7.3 to illustrate the translation of instantiation:

```
a_editor1 = letrec self : IDEEditor = ide_editor self in self;
```

One remarkable point is that, while Cook's work is done in an untyped setting, here we apply their ideas in a setting with disjoint intersection types and disjoint polymorphism. Our work shows that disjoint intersection types blend in quite nicely with Cook's denotational model of inheritance.

FLATTENING PROPERTY.   In the literature of traits [Ducasse et al. 2006; Nierstrasz et al. 2006; Schärli et al. 2003], a distinguished feature of traits is the *flattening property*, which says that a (non-overridden) method in a trait has the same semantics as if it were implemented directly in the class that uses the trait. It would be interesting to see if our trait model has this property. One problem in formulating such a property is that flattening is a property that talks about the equivalence between a flattened class (i.e., a class where all trait methods have been inlined) and a class that reuses code from traits. Since SEDEL does not have classes, we cannot state exactly the same property. However, we believe that one way to talk about a similar property for SEDEL is to have something along the lines of the following example:

Example 3 (Flattening). Suppose we have m well-typed (i.e, conflict-free) traits:

```
trait t1 {l11 = E11, ..}, ..., trait tm {lm1 = Em1, ..}
```

each with some number of methods, then the following two expressions are contextually equivalent:

```
new (trait inherits t1 & ... & tm {})
new (trait {l11 = E11,..,lm1 = Em1,..})
```

If we elaborate these two expressions, the property boils down to whether two merge terms $(E_1\,,\,E_2)\,,\,E_3$ and $E_1\,,\,(E_2\,,\,E_3)$ have the same semantics, which is exactly what Corollary 5.12 shows. So it is not hard to see that the above two expressions are contextually equivalent. We leave it as future work to formally state and prove flattening.

7.4.3 TYPE SOUNDNESS AND COHERENCE

Since the semantics of SEDEL is defined by elaboration into F_i^+ it is easy to show that key properties of F_i^+ are also guaranteed by SEDEL. In particular, we show that the type-directed elaboration is type-safe in the sense that well-typed SEDEL expressions are elaborated into well-typed F_i^+ terms. We also show that the source language is coherent and each valid source

program has a unique (unambiguous) elaboration. We refer the reader to Appendix C for the detailed manual proofs.

We need a meta-function $|\cdot|$ that translates SEDEL types to F_i^+ types, whose definition is straightforward. Only the translation of trait types deserves attention:

$$|\mathbf{Trait}\,[\mathcal{A}, \mathcal{B}]| = |\mathcal{A}| \rightarrow |\mathcal{B}|$$

That is, trait types are translated to function types. $|\cdot|$ extends naturally to typing contexts. Now we show several lemmas that are useful in the type-safety proof.

Lemma 7.1. *If $\Delta \vdash \mathcal{A}$ then $|\Delta| \vdash |\mathcal{A}|$.*

Lemma 7.2. *If $\mathcal{A} <: \mathcal{B}$ then $|\mathcal{A}| <: |\mathcal{B}|$.*

Lemma 7.3. *If $\Delta \vdash \mathcal{A} * \mathcal{B}$ then $|\Delta| \vdash |\mathcal{A}| * |\mathcal{B}|$.*

Finally we are in a position to establish the type safety property:

Theorem 7.4 (Type-safe translation). *We have that:*

- *If $\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \mathcal{A} \rightsquigarrow E$ then $|\Delta|; |\Gamma| \vdash E \Rightarrow |\mathcal{A}|$.*

- *If $\Delta; \Gamma \vdash \mathcal{T} \Leftarrow \mathcal{A} \rightsquigarrow E$ then $|\Delta|; |\Gamma| \vdash E \Leftarrow |\mathcal{A}|$.*

Theorem 7.5 (Coherence). *Each well-typed SEDEL expression has a unique elaboration.*

Proof. By examining every elaboration rule in Fig. 7.4, it is easy to see that the elaborated F_i^+ term in the conclusion is uniquely determined by the elaborated F_i^+ terms in the premises. Then by the coherence property of F_i^+ (Theorem 6.5), we can conclude that each well-typed SEDEL expression has a unique unambiguous elaboration, thus SEDEL is coherent. □

8 Case Study: Modularizing Language Components

To further illustrate the applicability of SEDEL, we present a case study using object algebras [Oliveira and Cook 2012] and extensible Visitors [Oliveira 2009; Torgersen 2004]. Encodings of extensible designs for object algebras and extensible Visitors have been presented in mainstream languages [Oliveira 2009; Oliveira and Cook 2012; Oliveira et al. 2013; Rendel et al. 2014; Torgersen 2004]. However, prior approaches are not entirely satisfactory due to the limitations in existing mainstream object-oriented languages. In Section 8.1, we show how SEDEL makes those designs significantly simpler and convenient to use. In particular, SEDEL's encoding of extensible Visitors gives true ASTs. In Section 8.2 we show a native support for conflict-free Object Algebra combinators, thanks to first-class traits and disjoint polymorphism. Based on this technique, Section 8.3 gives a bird's-eye view of several orthogonal features of a small JavaScript-like language taken from the textbook "Anatomy of Programming Languages" [Cook 2013], and illustrates how various features can be modularly developed and composed to assemble a complete language with various operations baked in. Section 8.4 compares our SEDEL's implementation with that of the textbook using Haskell in terms of lines of code.

8.1 Object Algebras and Extensible Visitors in SEDEL

First we give a simple introduction to object algebras, a design pattern that can solve the expression problem [Cook 1990; Krishnamurthi et al. 1998; Wadler 1998] in languages like Java. Our starting point is the following code:

```
type ExpAlg[E] = {
  lit : Int → E,
  add : E → E → E
};
type IEval = { eval : Int };

trait evalAlg ⇒ {
```

```
  lit (x : Int) = {
    eval = x
  };
  add (x : IEval) (y : IEval) = {
    eval = x.eval + y.eval
  }
};
```

`ExpAlg[E]` is the generic interface of a simple arithmetic language with two cases, `lit` for literals and `add` for addition. `ExpAlg[E]` is also referred to as an *Object Algebra interface*. A concrete Object Algebra will implement such an interface by instantiating `E` with a suitable type. Here we also define one operation `IEval`, modeled by a single-field record type. A concrete Object Algebra that implements the evaluation rules is given by a trait `evalAlg`.

FIRST-CLASS OBJECT ALGEBRA VALUES.   The actual AST of this simple arithmetic language is given as an internal visitor [Oliveira et al. 2008]:

```
  type Exp = {
    accept : forall E . ExpAlg[E] → E
  };
```

Note that object algebras as implemented in languages like Java or Scala do not define the type `Exp` because this would make adding new variants very hard. Although extensible versions of this visitor pattern do exist, they usually require complex types using advanced features of generics [Oliveira and Cook 2012; Torgersen 2004]. However, as we will see, this is not a problem in SEDEL. We can build a value of `Exp` as follows:

```
  e1 : Exp = {
    accept E f = f.add (f.lit 2) (f.lit 3)
  };
```

ADDING A NEW OPERATION.   We add another operation `IPrint` to the language:

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

```
  trait printAlg ⇒ {
    lit (x : Int) = {
      print = x.toString
    };
    add (x : IPrint) (y : IPrint) = {
      print = "(" ++ x.print ++ " + " ++ y.print ++ ")"
    }
```

```
};
```

This is done by giving another trait `printAlg` that implements the additional `print` method.

ADDING A NEW CASE.   A second dimension for extension is to add another case for negation:

```
type ExpExtAlg[E] = ExpAlg[E] & { neg : E → E };

trait negEvalAlg inherits evalAlg ⇒ {
  neg (x : IEval) = {
    eval = 0 - x.eval
  }
};
trait negPrintAlg inherits printAlg ⇒ {
  neg (x : IPrint) = {
    print= "-" ++ x.print
  }
};
```

This is achieved by extending `evalAlg` and `printAlg`, implementing missing operations for negation, respectively. We define the actual AST similarly:

```
type ExtExp = {
  accept: forall E. ExpExtAlg[E] → E
};
```

and build a value of `-(2 + 3)` while reusing `e1`:

```
e2 : ExtExp = {
  accept E f = f.neg (e1.accept E f)
};
```

RELATIONS BETWEEN EXP AND EXPEXT   At this stage, it is interesting to point out an interesting subtyping relation between `Exp` and `ExtExp`: `ExpExt`, though being an *extension* of `Exp` is actually a *supertype* of `Exp`. As Oliveira [2009] observed, these relations are important for legacy and performance reasons since it means that, a value of type `Exp` can be *automatically* and *safely* coerced into a value of type `ExpExt`, allowing some interoperability between new functionality and legacy code. However, to ensure type-soundness, Scala (or other common object-oriented languages) forbids any kind of type-refinement on method parameter types. The consequence of this is that in those languages, it is impossible to express that `ExtExp` is both an extension and a supertype of `Exp`.

8.2 Dynamic Object Algebra Composition Support

When programming with object algebras, oftentimes it is necessary to compose multiple operations together in such a way that they are executed in parallel to the same input. For example, in the simple language we have been developing it can be useful to create an object that supports both printing and evaluation. Oliveira and Cook [2012] addressed this problem by proposing *Object Algebra combinators* that combine multiple algebras into one. However, as they noted, such combinators written in Java are difficult to use in practice, and they require significant amounts of boilerplate. Improved variants of Object Algebra combinators have been encoded in Scala using intersection types and an encoding of the merge operator [Oliveira et al. 2013; Rendel et al. 2014]. However, the Scala encoding of the merge operator is quite complex as it relies on low-level type-unsafe programming features such as dynamic proxies, reflection or other meta-programming techniques. In SEDEL, the combination of first-class traits, dynamic inheritance, disjoint polymorphism and nested composition allows type-safe, coherent and boilerplate-free composition of object algebras.

Abstractly speaking, what we are seeking is a combinator:

$$\mathsf{combine} \in F[A] \times F[B] \to F[A \,\&\, B]$$

That is, given two algebras with types $F[A]$ and $F[B]$ we want to derive, in a automatic way, a third algebra $F[A \,\&\, B]$ that combines the results of the two algebras. This combinator bears a similarity to a `zip`-like operation in functional programming, and it is also directly related to nested composition in object-oriented programming, as we have studied in Chapter 3. In SEDEL the definition of an Object Algebra combinator is:

```
combine A [B * A] (f : Trait[ExpExtAlg[A]])
                  (g : Trait[ExpExtAlg[B]]) : Trait[ExpExtAlg[A & B]] =
  trait inherits f & g ⇒ {};
```

That is it. None of the boilerplate in other approaches [Oliveira and Cook 2012], or type-unsafe meta-programming techniques of other approaches [Oliveira et al. 2013; Rendel et al. 2014] are needed! Three points are worth noting: (1) `combine` relies on *dynamic inheritance*. Notice how `combine` is parameterized by two traits `f` and `g`, for which their implementations are unknown statically; (2) the disjointness constraint (`B * A`) is *crucial* to ensure two algebras (`f` and `g`) are conflict-free when being composed; (3) nested composition is the underlying mechanism to automatically derive the combined algebra by appropriately invoking (delegating) behaviors in `A & B` to either `A` or `B`.

To conclude, let us see `combine` in action. We merge the evaluation and printing algebras to create a combined algebra `expEvalPrint`:

$$
\begin{array}{llll}
\text{Types} & \tau & ::= & \text{int} \mid \text{bool} \\
\text{Expressions} & e & ::= & i \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e_1 \div e_2 \quad \textit{natF} \\
& & \mid & \mathbb{B} \mid \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \qquad\qquad\quad \textit{boolF} \\
& & \mid & e_1 == e_2 \mid e_1 < e_2 \qquad\qquad\qquad\quad \textit{compF} \\
& & \mid & e_1 \,\&\&\, e_2 \mid e_1 \,||\, e_2 \qquad\qquad\qquad\quad \textit{logicF} \\
& & \mid & x \mid \textbf{var } x = e_1; e_2 \qquad\qquad\qquad\quad \textit{varF} \\
& & \mid & e_1 \, e_2 \qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{funcF} \\
\text{Programs} & pgm & ::= & decl_1 \ldots decl_n \, e \qquad\qquad\qquad\qquad\;\; \textit{funcF} \\
\text{Functions} & decl & ::= & \textbf{function } f(x : \tau)\{e\} \qquad\qquad\qquad\;\; \textit{funcF} \\
\text{Values} & v & ::= & i \mid \mathbb{B}
\end{array}
$$

Figure 8.1: Mini-JS expressions, values, and types

```
expEvalPrint = combine IEval IPrint negEvalAlg negPrintAlg;
```

We can use this algebra to create an object o that allows us to use evaluation and pretty printing at the same time:

```
o = e2.accept (IEval & IPrint)
            (new[ExpExtAlg[IEval & IPrint]] expEvalPrint);
main = o.print ++ " = " ++ o.eval.toString -- "-(2.0 + 3.0) = -5.0"
```

8.3 Case Study Overview

Now we are ready to see how the same technique scales to modularize different language features. A *feature* is an increment in program functionality [Lopez-Herrejon et al. 2005; Zave 1999]. Figure 8.1 presents the syntax of a mini-JS language [Cook 2013]; each line is annotated with the corresponding feature name. Starting from a simple arithmetic language, we gradually introduce new features and combine them with some of the existing features to form various languages. Below we briefly explain what constitutes each feature:

- *natF* and *boolF* contain, among others, literals, additions and conditional expressions.

- *compF* and *logicF* introduce comparisons between numbers and logical connectives.

- *varF* introduces local variables and variable declarations.

- *funcF* introduces top-level functions and function calls.

Besides, each feature is packed with 3 operations: evaluator, pretty printer and type checker. We also define a combined algebra with the combined behavior of all the operations.

Language	Operations			Data variants					
	eval	print	check	*natF*	*boolF*	*compF*	*logicF*	*varF*	*funcF*
`simplenat`	✓	✓		✓					
`simplebool`	✓	✓			✓				
`natbool`	✓	✓	✓	✓	✓				
`varbool`	✓	✓			✓			✓	
`varnat`	✓	✓		✓				✓	
`simplelogic`	✓	✓			✓		✓		
`varlogic`	✓	✓			✓		✓	✓	
`arith`	✓	✓	✓	✓	✓	✓			
`arithlogic`	✓	✓	✓	✓	✓	✓	✓		
`vararith`	✓	✓	✓	✓	✓	✓		✓	
`vararithlogic`	✓	✓	✓	✓	✓	✓	✓	✓	
`mini-JS`	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 8.1: Overview of the languages assembled

Having the feature set, we can synthesize different languages by selecting one or more operations, and one or more data variants, as shown in Table 8.1. For example `arith` is a simple language of arithmetic expressions, assembled from *natF*, *boolF* and *compF*. On top of that, we also define an evaluator, a pretty printer and a type checker. Note that for some languages (e.g., `simplenat`), since they have only one kind of value, we only define an evaluator and a pretty printer. We thus obtain 12 languages and 30 operations in total. The complete language `mini-JS` contains all the features and supports all the operations. The reader can refer to our supplementary material for the source code of the case study.

8.4 EVALUATION

To evaluate SEDEL's implementation of the case study, Table 8.2 compares the number of source lines of code (SLOC, lines of code without counting empty lines and comments) for SEDEL's *modular* implementation with the vanilla *non-modular* AST-based implementations in Haskell. The Haskell implementations are just straightforward AST interpreters, which duplicate code across the multiple language components.

Since SEDEL is a new language, we had to write various code that is provided in Haskell by the standard library, so they are not counted for fairness of comparison. In the left part, for each feature, we count the lines of the algebra interface (number beside the feature name), and the algebras for the operations. In the right part, for each language, we count the lines of ASTs, and those to combine previously defined operations. For example, here is the code that is needed to make the `arith` language.

Feature	eval	print	check	Lang	SEDEL	Haskell	% Reduced
natF(7)	23	7	39	`simplenat`	3	33	91%
boolF(4)	9	4	17	`simplebool`	3	16	81%
compF(4)	12	4	20	`natbool`	5	74	93%
logicF(4)	12	4	20	`varbool`	4	24	83%
varF(4)	7	4	7	`varnat`	4	41	90%
funcF(3)	10	3	9	`simplelogic`	4	28	86%
				`varlogic`	6	36	83%
				`arith`	8	94	91%
				`arithlogic`	8	114	93%
				`vararith`	8	107	93%
				`vararithlogic`	8	127	94%
				`mini-JS`	33	149	78%
Total		237			331	843	61%

Table 8.2: SLOC statistics: SEDEL implementation vs. vanilla AST implementation

```
-- Object Algebra interface
type ArithAlg[E] = NatBoolAlg[E] & CompAlg[E];
-- AST
type Arith = {
  accept : forall E. ArithAlg[E] → E
};
-- Evaluator
evalArith (e : Arith) : IEval =
  e.accept IEval
    (new[ArithAlg[IEval]] evalNatAlg & evalBoolAlg & evalCompAlg);
-- Pretty printer
ppArith (e : Arith) : IPrint =
  e.accept IPrint
    (new[ArithAlg[IPrint]] ppNatAlg & ppBoolAlg & ppCompAlg);
-- Type checker
tcArith (e : Arith) =
  e.accept ITC (new[ArithAlg[ITC]] tcNatAlg & tcBoolAlg & tcCompAlg);
```

We only need 12 lines in total: 4 lines for the AST, and 8 lines to combine the operations.

Therefore, the total SLOC of SEDEL's implementation is the sum of all the lines in the feature and language parts (237 SLOC of all features plus 94 SLOC of ASTs and operations). Although SEDEL is considerably more verbose than a functional language like Haskell, SEDEL's modular implementation for 12 languages and 30 operations in total reduces approximately 60% in terms of SLOC. The reason is that, the more frequently a feature is reused by other languages directly or indirectly, the more reduction we see in the total SLOC. For example,

natF is used across many languages. Even though `simplenat` itself *alone* has more SLOC ($40 = 7 + 23 + 7 + 3$) than that of Haskell (which has 33), we still get a huge gain when implementing other languages.

FINAL REMARKS.   We acknowledge that there are several limitations in our case study. On the one hand, SLOC is just one metric and we have not yet measured any other metrics. On the other hand, we did not compare with other modular approaches such as finally tagless [Carette et al. 2009]. Nevertheless we believe that the case study is already non-trivial in that we need to solve the expression problem. Note that Scala traits alone are not sufficient on their own to solve the expression problem. While there are solutions in both Haskell and Scala, they introduce significant complexity, as explained in Section 8.1.

Part V

Related and Future Work

9 RELATED WORK

There is a great deal of work related to this thesis. We have touched some most relevant work (notably intersection types) in Chapter 2. In this chapter, we briefly review other related work, starting with a summary of two most common approaches on coherence (Section 9.1). We then consider various existing mechanisms to foster modularity and code reuse in the rest of this chapter.

9.1 COHERENCE

In calculi that feature coercive subtyping, a semantics that interprets the subtyping judgment by introducing explicit coercions is typically defined on typing derivations rather than on typing judgments. A natural question that arises for such systems is whether the semantics is *coherent*, i.e., distinct typing derivations of the same typing judgment possess the same meaning. Since Reynolds [1991] proved the coherence of a calculus with intersection types, based on the denotational semantics for intersection types, many researchers have studied the problem of coherence in a variety of typed calculi. Below we summarize two commonly-found approaches in the literature.

9.1.1 NORMALIZATION-BASED APPROACH

The first approach is based on normalization. Breazu-Tannen et al. [1991] proved the coherence of a coercion translation from Fun [Cardelli and Wegner 1985] extended with recursive types to System F by showing that any two typing derivations of the same judgment are normalizable to a unique normal derivation where the correctness of the normalization steps is justified by an equational theory in System F. Curien and Ghelli [1992] presented a translation of System F$_\leq$ into a calculus with explicit coercions and showed that any derivations of the same judgment are translated to terms that are normalizable to a unique normal form. Following the same approach, Schwinghammer [2008] proved the coherence of coercion translation from Moggi's computational lambda calculus [Moggi 1991] with subtyping.

9.1.2 Context-based Approach

Central to the first approach is to find a normal form for a representation of the derivation and show that normal forms are unique for a given typing judgment. However, this approach cannot be directly applied to Curry-style calculi, i.e., where the lambda abstractions are not type annotated. Also this line of reasoning cannot be used when the calculus has general recursion. Biernacki and Polesiuk [2015] considered the coherence problem of coercion semantics. Their criterion for coherence of the translation is *contextual equivalence* in the target calculus. They presented a construction of logical relations for establishing so constructed coherence for coercion semantics, showing that this approach is applicable in a variety of calculi, including delimited continuations and control-effect subtyping.

Inspired by this approach, Bi et al. [2018] proposed the canonicity relation to prove coherence for a calculus with disjoint intersection types and BCD subtyping. BCD subtyping in our setting poses a non-trivial complication over Biernacki and Polesiuk's simple structural subtyping. Indeed, because any two coercions between given types are behaviorally equivalent in the target language, their coherence reasoning can all take place in the target language. This is not true in our setting, where coercions can be distinguished by arbitrary target programs, but not those that are elaborations of source programs. (Recall that $\lambda x. \pi_1 x$ and $\lambda x. \pi_2 x$ should be equated in our setting.) Hence, we have to restrict our reasoning to the latter class, which is reflected in a more complicated notion of contextual equivalence and our logical relation's non-trivial treatment of pairs. As we have shown in Chapter 6, constructing a suitable logical relation for F_i^+ is challenging. On the one hand, the original approach by Alpuim et al. [Alpuim et al. 2017] in F_i does not work any more due to the addition of BCD subtyping. On the other hand, simply combining System F's logical relation with λ_i^+'s canonicity relation does not work as expected, due to the issue of well-foundedness. To solve the problem, we employ immediate substitutions and a restriction to predicative instantiations.

9.2 BCD Subtyping and Decidability

The BCD type system was first introduced by Barendregt et al. [1983]. It is derived from a filter lambda model in order to characterize exactly the strongly normalizing terms. The BCD type system features a powerful subtyping relation, which serves as a base for our subtyping relation. Bessai et al. [2014] show how to type classes and mixins in a BCD-style record calculus with a merge-like operator [Bracha and Cook 1990] that only operates on records, and they only study a type assignment system. The decidability of BCD subtyping has been shown in several works [Kurata and Takahashi 1995; Pierce 1989; Rehof and

Urzyczyn 2011; Statman 2015]. Laurent [2012a] formalized the relation in Coq in order to eliminate transitivity cuts from it, but his formalization does not deliver an algorithm. Only recently, Laurent [2018] presents a general way of defining a BCD-like subtyping relation extended with generic contravariant/covariant type constructors that enjoys the "sub-formula property" (read decidability). The key idea is to generalize the form of subtyping from $A <: B$ to $A_1, \ldots, A_n \vdash B$, which is interpreted as meaning $A_1 \& \ldots \& A_n <: B$. Here is his subtyping system instantiated with singleton records, adapted to our setting:

$$\frac{}{\mathsf{Int} \vdash \mathsf{Int}} \qquad \frac{}{\vdash \top} \qquad \frac{\vdash B}{\vdash A \to B} \qquad \frac{\vdash A}{\vdash \{l : A\}} \qquad \frac{\Gamma, \Delta \vdash C}{\Gamma, \mathsf{Int}, \Delta \vdash C} \qquad \frac{\Gamma, \Delta \vdash C}{\Gamma, \top, \Delta \vdash C}$$

$$\frac{\Gamma, \Delta \vdash C}{\Gamma, A \to B, \Delta \vdash C} \qquad \frac{\Gamma, \Delta \vdash C}{\Gamma, \{l : B\}, \Delta \vdash C} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \& B} \qquad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, A \& B, \Delta \vdash C}$$

$$\frac{A \vdash A_1 \quad \ldots \quad A \vdash A_n \qquad B_1, \ldots, B_n \vdash B}{A_1 \to B_1, \ldots, A_n \to B_n \vdash A \to B} \qquad \frac{A_1, \ldots, A_n \vdash B}{\{l : A_1\}, \ldots, \{l : A_n\} \vdash \{l : B\}}$$

The first two rules are the base cases. The third and forth rules deal with cases where B is a "top-like" type. The next four rules are the weakening rules for integers, top types, function types and singleton records. The next two rules are the introduction and elimination rules for intersections. The last two rules combine the function distributivity rule with usual function subtyping, and record distributivity rule with usual record subtyping, respectively. Laurent proved in Coq that $A \vdash B$ if and only if $A <: B$. Our Coq formalization follows a different idea based on Pierce's decision procedure [Pierce 1989], which is shown to be easily extensible to records, parametric (disjoint) polymorphism and corresponding distributivity rules. In the course of our mechanization we identified several mistakes in Pierce's proofs, as well as some important missing lemmas. More recently, Muehlboeck and Tate [2018] presented a decidable algorithmic system (proved in Coq) with union and intersection types. Similar to F_i^+, their system also has distributive subtyping rules. They also discussed the addition of polymorphism, but left a Coq formalization for future work. In their work they regard intersections of disjoint types (e.g., $\mathsf{String} \& \mathsf{Int}$) as uninhabitable, which is different from our interpretation. As a consequence, coherence is a non-issue for them. Finally, it would be interesting to study an efficient subtyping algorithm in normal practice. As noted by Reynolds [1997], however, the worst-case inefficiency is inevitable. In fact, any typechecker for languages using intersection types is PSPACE-hard.

9.3 INTERSECTION TYPES, MERGE OPERATOR AND POLYMORPHISM

Forsythe [Reynolds 1988] has intersection types and a merge-like operator. However to ensure coherence, various restrictions were added to limit the use of merges. Forsythe only permits $p_1, , p_2$ when p_2 is either an lambda abstraction or a record, whose meaning "overrides" the corresponding type of meaning of p_1. For instance, there is a rule regarding lambda abstraction that says (adapted to our syntax):

$$\frac{\Gamma \vdash \lambda x.\, p_2 : \theta_1 \rightarrow \theta_2}{\Gamma \vdash (p_1, , \lambda x.\, p_2) : \theta_1 \rightarrow \theta_2}$$

which means that in a merge of two functions, the second one always takes precedence to the first one. In contrast, our typing rule for merges is more fine-grained in the sense that both functions are retained as long as they are disjoint. Castagna et al. [1992] proposed a coherent calculus $\lambda\&$ to study overloaded functions. $\lambda\&$ has a special merge operator that works on functions only. Dunfield [2014] proposed a calculus (which we call $\lambda_{,,}$) that shows significant expressiveness of type systems with unrestricted intersection types and a (unrestricted) merge operator. However, because of his unrestricted merge operator (allowing $1, , 2$), his calculus lacks coherence. Blaauwbroek [2017]'s $\lambda_{\wedge}^{\vee}$ enriched $\lambda_{,,}$ with BCD subtyping and computational effects, but he did not address coherence. The coherence issue for a calculus similar to $\lambda_{,,}$ was first addressed in λ_i [Oliveira et al. 2016] with the notion of disjointness, but at the cost of dropping unrestricted intersections, and a strict notion of coherence (based on α-equivalence). Later Bi et al. [2018] improved calculi with disjoint intersection types by removing several restrictions, adopted BCD subtyping and a semantic notion of coherence (based on contextual equivalence) proved using canonicity. The combination of intersection types, a merge operator and parametric polymorphism, while achieving coherence was first studied in F_i [Alpuim et al. 2017], which serves as a foundation for F_i^+. However, F_i suffered the same problems as λ_i. Additionally in F_i a bottom type is problematic due to interactions with disjoint polymorphism and the lack of unrestricted intersections. The issues can be illustrated with the well-typed F_i^+ expression $\Lambda(\alpha * \bot).\, \lambda x : \alpha.\, x, , x$. In this expression the type of $x, , x$ is $\alpha \& \alpha$. Such a merge does not violate disjointness because the only types that α can be instantiated with are top-like, and top-like types do not introduce incoherence. In F_i a type variable α can never be disjoint to another type that contains α, but (as the previous expression shows) the addition of a bottom type allows expressions where such (strict) condition does not hold. In F_i^+, we removed those restrictions, extended BCD subtyping with polymorphism, and proposed a more powerful logical relation for proving coherence.

	$\lambda_{,,}$	λ_i	$\lambda_\wedge^\vee$	λ_i^+	F_i	F_i^+
Disjointness	○	●	○	●	●	●
Unrestricted intersections	●	○	●	●	○	●
BCD subtyping	○	○	●	●	○	●
Polymorphism	○	○	○	○	●	●
Coherence	○	◐	○	●	◐	●
Bottom type	○	○	●	○	○	●

Figure 9.1: Summary of intersection calculi

Figure 9.1 summarizes the main differences between the aforementioned calculi, where ● = yes, ○ = no, and ◐ = syntactic coherence.

There are also several other calculi with intersections and polymorphism. Pierce proposed $\mathsf{F}_\wedge$ [Pierce 1991], a calculus combining intersection types and bounded quantification. Pierce translates $\mathsf{F}_\wedge$ to System F extended with products, but he left coherence as a conjecture. More recently, Castagna et al. [2014] proposed a polymorphic calculus with set-theoretic type connectives (intersections, unions, negations). But their calculus does not include a merge operator. Castagna and Lanvin [2017] also proposed a gradual type system with intersection and union types, but also without a merge operator.

9.4 ROW POLYMORPHISM AND BOUNDED POLYMORPHISM

Row polymorphism was originally proposed by Wand [1987] as a mechanism to enable type inference for a simple object-oriented language based on recursive records. These ideas were later adopted into type systems for extensible records [Gaster and Jones 1996; Harper and Pierce 1991; Leijen 2005]. Our merge operator can be seen as a generalization of record extension/concatenation, and selection is also built-in. In contrast to most record calculi, restriction is not a primitive operation in λ_i^+ and F_i^+, but can be simulated via subtyping. Disjoint quantification can simulate the *lacks* predicate often present in systems with row polymorphism. Recently Morris and McKinna [2019] presented a typed language, generalizing and abstracting existing systems of row types and row polymorphism. Alpuim et al. [2017] informally studied the relationship between row polymorphism and disjoint polymorphism, but it would be interesting to study such relationship more formally. The work of Morris and McKinna may be interesting for such study in that it gives a general framework for row type systems.

Bounded quantification is currently the dominant mechanism in major mainstream object-oriented languages supporting both subtyping and polymorphism. $\mathsf{F}_{<:}$ [Cardelli and Weg-

ner 1985] provides a simple model for bounded quantification, but type-checking in full $\mathsf{F}_{<:}$ is proved to be undecidable [Pierce 1994]. Pierce's thesis [Pierce 1991] discussed the relationship between calculi with simple polymorphism and intersection types and bounded quantification. He observed that there is a way to "encode" many forms of bounded quantification in a system with intersections and pure (unbounded) second-order polymorphism. That encoding can be easily adapted to F_i^+:

$$\forall(\alpha <: A).\, B \triangleq \forall(\alpha * \top).\, [A \,\&\, \alpha/\alpha]B$$

The idea is to replace bounded quantification by (unrestricted) universal quantification and all occurrences of α by $A \,\&\, \alpha$ in the body. Such an encoding seems to indicate that F_i^+ could be used as a decidable alternative to (full) $\mathsf{F}_{<:}$. It is worthwhile to note that this encoding does not work in F_i because $A \,\&\, \alpha$ is not well-formed (α is not disjoint to A). In other words, the encoding requires unrestricted intersections.

9.5 Typed First-Class Classes/Mixins/Traits

First-class classes have been used in Racket [Flatt et al. 2006], along with mixin support, and have shown great practical value. For example, DrRacket IDE [Findler et al. 2002] makes extensive use of layered combinations of mixins to implement text editing features. The topic of first-class classes with static typing has been explored by Takikawa et al. [2012] in Typed Racket. They designed a gradual type system that supports first-class classes. Of particular interest is their use of row polymorphism to type mixins. For example, `modal_mixin` from Section 7.2 implemented in Typed Racket has type:

```
(All (r / on-key toggle-mode)
     (Class ([on-key : (String → Void)] | r)) →
     (Class ([toggle-mode : (→ Void)] [on-key : (String → Void)] | r)))
```

As with our use of disjoint polymorphism, row polymorphism can express constraints on the presence or absence of members. As a consequence, Typed Racket ends up with two subtyping mechanisms: one for first-class classes (via row polymorphism) and the other for objects (via normal width subtyping). In contrast, SEDEL uses only one mechanism—i.e., disjoint polymorphism—to deal with both.

More recently, Lee et al. [2015] proposed a model for typed first-class classes based on tagged objects. Like our development, the semantics of their source language is defined by a translation into a target language. One notable difference to SEDEL is that they require the use of a variable rather than an expression in the **extends** clause, whereas we do not have this

restriction. In their source language, subclasses define subtypes, which limits its applicability to extensible designs. Also their target calculus is significantly more complex than ours due to the use of dependent function types and dependent sum types. As they admitted, they omit inheritance in their formalization.

Racket also supports a *dynamically typed model* of first-class traits [Flatt et al. 2006]. However, unlike Racket's first-class classes and mixins, there is no type system supporting the use of first-class traits. A key difficulty is *statically* detecting conflicts. In the mixin model this is not a problem because conflicts are implicitly resolved using the order of composition. As far as we know, SEDEL is the first design for typed first-class traits.

9.6 MIXIN-BASED INHERITANCE

Bracha and Cook's seminal paper [Bracha and Cook 1990] extends Modula-3 with mixins. Since then, many mixin-based models have been proposed [Ancona et al. 2003; Bono et al. 1999; Flatt et al. 1998]. Mixin-based inheritance requires that mixins are composed linearly, and as such, conflicts are resolved implicitly. In comparison, the trait model in SEDEL requires conflicts to be resolved explicitly. We want to emphasize that conflict detection is essential in expressing composition operators for object algebras, without running into ambiguities. Bracha's Jigsaw framework [Bracha 1992] formalized mixin composition, along with a rich trait algebra including merge, restrict, select, project, overriding and rename operators. Lagorio et al. [2012] proposed FJIG that reformulates Jigsaw constructs in a Java-like setting. Allen et al. [2003] described how to add first-class generic types—including mixins—to object-oriented languages with nominal typing. Corradi et al. [2012] described an extension of FJIG that integrates modular composition and nesting of Java-like classes. It features a set of composition operators that allow to manipulate nested classes at any depth level. In all of these systems, classes and mixins, though they enjoy static typing, are still second-class constructs, and thus their systems cannot express dynamic inheritance.

9.7 TRAIT-BASED INHERITANCE

Traits were originally proposed by Schärli et al. [2003], and later formalized by Ducasse et al. [2006] as a mechanism for fine-grained code reuse to overcome many limitations of class-based inheritance. The original proposal of traits were implemented in the dynamically typed class-based language SQUEAK/SMALLTALK. Since then various formalizations of traits in a Java-like (statically typed) setting have been proposed [Fisher and Reppy 2004; Nierstrasz et al. 2006; Scharli et al. 2003; Smith and Drossopoulou 2005]. In most of the

above proposals, trait composition complements class-based inheritance. SEDEL, in the
spirit of *pure trait-based programming languages* [Bettini and Damiani 2017; Bettini et al.
2013b], embraces traits as the sole mechanism for code reuse. The deviation from tradi-
tional class-based inheritance is not only because of its simplicity, but also because we need
a very *dynamic* form of inheritance. In comparison to the traditional trait mode, traits in
SEDEL have the following differences:

1. traditional traits cannot be instantiated but only composed with a class, whereas traits
   in SEDEL can be instantiated directly;

2. traditional traits cannot take constructor parameters whereas ours can;

3. the trait system in SEDEL lacks a proper notion of inheritance relationship, e.g., in the
   traditional trait model, if the *same* method is obtained more than once via different
   paths, there is no conflict. This is not the case in SEDEL;

4. and finally traits in SEDEL are first-class and support dynamic inheritance.

9.8 Family Polymorphism

There has been much work on family polymorphism since Ernst's original proposal [Ernst
2001]. Family polymorphism provides an elegant solution to the expression problem. Al-
though a simple Scala solution does exist without requiring family polymorphism (e.g., see
Wang and Oliveira [2016]), Scala does not support nested composition: programmers need
to manually compose all the classes from multiple extensions. Generally speaking, systems
that support family polymorphism can be divided into two categories: those that support *ob-
ject families* and those that support *class families*. The original object family approach of Beta
(e.g., virtual classes [Madsen and Moller-Pedersen 1989]) treats nested classes as attributes
of objects of the family classes, whereas in class families, classes are nested in other classes.
The former choice is considered more expressive [Ernst et al. 2006], but requires a complex
type system usually involving dependent types. The object and class family approaches have
even been combined by the work on Tribe [Clarke et al. 2007].

Object families. Virtual classes [Madsen and Moller-Pedersen 1989] as introduced in
Beta [Lehrmann Madsen et al. 1993] are based on object families. However, the virtual
class mechanism in Beta is known to be unsound [Bruce et al. 1998]. Path-dependent types
are used to ensure type safety for virtual types and virtual classes in the calculus vc [Ernst
et al. 2006]. One distinctive difference from our calculi is that vc follows the mixin-style by

allowing the rightmost class to take precedence, whereas in λ_i^+ conflicts are detected statically and resolved explicitly.

CLASS FAMILIES.   Concord [Jolly et al. 2004], Jx [Nystrom et al. 2004] and J& [Nystrom et al. 2006] follow the class family approach, where nested classes and types are attributes of the family classes directly. Jx supports *nested inheritance*, a class family mechanism that allows nesting of arbitrary depth. J& is a language that supports *nested intersection*, building on top of Jx. Similar to our calculi, intersection types play an important role in J&, which are used to compose packages/classes. However, J& does not have a merge-like operator. When conflicts arise, prefix types can be exploited to resolve the ambiguity. J&$_s$ [Qi and Myers 2009] is an extension of the Java language that adds class sharing to J&. Saito et al. [2007] identified a minimal, lightweight set of language features to enable family polymorphism,

Compared with those systems, which usually focus on getting a relatively complex Java-like language with family polymorphism, our work on λ_i^+ focuses on a minimal calculus that supports nested composition. We have shown that a calculus with the merge operator and a variant of BCD subtyping captures the essence of nested composition. Moreover, λ_i^+ enables new insights on the subtyping relations of families. Our goal in this thesis is not to support full family polymorphism which, besides nested composition, also requires dealing with other features such as self types [Bruce et al. 1995; Saito and Igarashi 2009] and mutable state. But we expect to investigate those features in the future.

9.9 LANGUAGES WITH MORE ADVANCED FORMS OF INHERITANCE

SELF [Ungar and Smith 1988] is a dynamically typed, prototype-based language with a simple and uniform object model. SELF's inheritance model is typical of what we call *mutable inheritance*, because an object's parent slot may be assigned new values at run time. Mutable inheritance is rather unstructured, and oftentimes access to any clashing methods will generate a "messageAmbiguous" error at run time. Although SEDEL's dynamic inheritance is not as powerful as mutable inheritance, its static type system can guarantee that no such errors occur at run time. Eiffel [Meyer 1987] supports a sophisticated class-based multiple inheritance with deep renaming, exclusion and repeated inheritance. Of particular interest is that in Eiffel, name collisions are considered programming errors, and ambiguities must be resolved explicitly by the programmer (by means of renaming). In this regard, SEDEL is quite like Eiffel. However, the type system in SEDEL is more lenient in that two identically named methods with different signatures can coexist. Grace [Jones et al. 2016; Noble et al. 2017] is an object-based language designed for education, where objects are created by *object con-*

structors. Since Grace has mutable fields, it has to consider many concerns when it comes to inheritance, resulting in a rather complex inheritance mechanism with various restrictions. Since SEDEL is pure, a relatively simple encoding of traits with late binding of **self** suffices for our applications. Grace's support for multiple inheritance is based on what they call *instantiable traits*. We believe that there is plenty to be learned from Grace's design of traits if we want to extend our trait model with features such as mutable state. METAFJIG [Servetto and Zucca 2014] (an extension of FJIG) supports *dynamic trait replacement* [Bettini et al. 2013a; Ducasse et al. 2006; Smith and Drossopoulou 2005], a feature for changing the behavior of an object at run time by replacing one trait for another. More recently, a Java-like language called Familia [Zhang and Myers 2017] were proposed to combine subtyping polymorphism, parametric polymorphism and family polymorphism.

9.10 MODULE SYSTEMS

In parallel to OOP, the ML module system originally proposed by MacQueen [1984] also offers powerful support for flexible program construction. There is a large body of work on ML modules. Supporting *data abstraction* is the primary focus of the module mechanism in ML. It ensures *implementer-side* data abstraction by allowing the implementer of a module to "hide" specific implementation behind an abstract interface. It also supports a form of *client-side* data abstraction where a client can develop and compile a module independently from the modules on which it depends, via the "functor" mechanism. One major limitation of the traditional ML module systems is the lack of support for mutually recursive modules. There are several proposals of extending ML with recursive modules [Crary et al. 1999; Rossberg and Dreyer 2013; Russo 2001]. Mixin modules in the Jigsaw framework [Bracha and Lindstrom 1992] provides a suite of operators for adapting and combining modules. The MixML module system [Rossberg and Dreyer 2013] incorporates mixin module composition, while retaining the full expressive powerful of ML modules. There is also work on elaborating the semantics of module systems into a smaller, well-established internal language. Rossberg et al. [2014] showed that plain System F is sufficient as an internal language for conventional ML modules. Furthermore, Rossberg [2015] proposed a redesign of ML in which modules are truly first-class values, thus unifying the core and module layers into one language.

Module systems usually put more emphasis on supporting data abstraction. Support for data abstraction adds considerable complexity, which is not needed in SEDEL. SEDEL is focused on OOP and supports, among others, method overriding, self references and dynamic dispatching, which (generally speaking) are all missing features in module systems.

10 Future Work

In this section we discuss some areas where future research might extend and/or complement the work described in this thesis.

10.1 Categorical Semantics

An interesting avenue for future work is to give a categorical semantics of disjoint intersection types. The main reason for doing so is that, as Reynolds [1988] nicely put it:

> *"by formulating succinct definitions in terms of a mathematical theory of great generality, we gain an assurance that our language will be uniform and general."*

Using category theory as the basis for the type structure of a programming language has a long history. Lambek [1985] discovered that simply-typed lambda calculus can be interpreted in any Cartesian closed category. Reynolds [1991] gives a category-theoretic presentation of a lambda calculus extended to include records, fixed points and intersection types, much similar to our λ_i^+. Of particular interest to us is his method for proving coherence. Let D denote derivations of typing, then the interpretation of a derivation $D :: \Gamma \vdash E : A$ is a morphism $[\![D :: \Gamma \vdash E : A]\!] : [\![\Gamma]\!] \to [\![A]\!]$ in a suitable "semantic" category (i.e., being Cartesian closed and possessing certain limits). Proving coherence in this presentation then amounts to establishing the commutativity of all diagrams of the following form:[1]

$$\begin{array}{ccc} & [\![D_1::\Gamma\vdash E:A]\!] & \\ [\![\Gamma]\!] & \rightrightarrows & [\![A]\!] \\ & [\![D_2::\Gamma\vdash E:A]\!] & \end{array}$$

10.1.1 Properties of Intersection Types

The key component of Reynolds' method is the interpretation of intersection types. For the sake of precision in what follows, we pause to give some basic properties of intersection types

---

[1] The proof actually needs a stronger inductive hypothesis.

that are first proved by Reynolds [1991]. First we give two definitions that are important for the discussion.

Definition 16 (Type Equivalence). Two types A and B are equivalent, written $A \approx B$, when $A <: B$ and $B <: A$.

Definition 17 (Least Upper Bounds). A *least upper bound* of A and B is a supertype of both A and B and a subtype of every common supertype of A and B—i.e., a type C such that:

- $A <: C$

- $B <: C$

- For any C', $A <: C'$ and $B <: C'$ implies $C <: C'$

According to the subtyping rules in Fig. 3.3, we can derive the following type equalities:

Proposition 10.1.

$$A_1 \mathbin{\&} (A_2 \mathbin{\&} A_3) \approx (A_1 \mathbin{\&} A_2) \mathbin{\&} A_3$$
$$\top \mathbin{\&} A \approx A$$
$$A \mathbin{\&} \top \approx A$$
$$A_1 \mathbin{\&} A_2 \approx A_2 \mathbin{\&} A_1$$
$$A \mathbin{\&} A \approx A$$
$$\{l : A_1 \mathbin{\&} A_2\} \approx \{l : A_1\} \mathbin{\&} \{l : A_2\}$$
$$A \to A_1 \mathbin{\&} A_2 \approx (A \to A_1) \mathbin{\&} (A \to A_2)$$
$$\{l : \top\} \approx \top$$
$$A \to \top \approx \top$$

Furthermore, it can be shown that every pair of λ_i^+ types has a least upper bound (unique up to $\approx$-equivalence). The following meta-function yields a least upper bound (written $A \sqcup B$) for any types A and B:

Proposition 10.2.

$$A \sqcup B = B \sqcup A$$

$$A \sqcup \top = \top$$

$$A_1 \sqcup (A_2 \,\&\, A_3) = (A_1 \sqcup A_2) \,\&\, (A_1 \sqcup A_3)$$

$$\mathsf{Int} \sqcup \{l : A\} = \top$$

$$\mathsf{Int} \sqcup (A_1 \to A_2) = \top$$

$$\{l : A\} \sqcup (A_1 \to A_2) = \top$$

$$\{l : A_1\} \sqcup \{l : A_2\} = \{l : A_1 \sqcup A_2\}$$

$$\{l_1 : A_1\} \sqcup \{l_2 : A_2\} = \top \quad when\ l_1 \neq l_2$$

$$(A_1 \to A_1') \sqcup (A_2 \to A_2') = (A_1 \,\&\, A_2) \to (A_1' \sqcup A_2')$$

10.1.2 CONNECTING WITH DISJOINTNESS

With the above propositions stated, it turns out that our disjointness rules, as given in Fig. 3.5, can be compactly formulated using $\approx$ and $\sqcup$:

Theorem 10.3. *$A * B$ if and only if $A \sqcup B \approx \top$.*

Proof. By induction on the derivation of disjointness. An interesting case is rule D-ARR

$$
\frac{\text{D-ARR} \quad A_2 * B_2}{A_1 \to A_2 * B_1 \to B_2}
$$

$A_2 \sqcup B_2 \approx \top$	By i.h
$(A_1 \to A_2) \sqcup (B_1 \to B_2) \approx (A_1 \,\&\, B_1) \to (A_2 \sqcup B_2)$	By Proposition 10.2
$(A_1 \to A_2) \sqcup (B_1 \to B_2) \approx (A_1 \,\&\, B_1) \to \top$	By above equality
$(A_1 \,\&\, B_1) \to \top \approx \top$	By Proposition 10.1
$(A_1 \to A_2) \sqcup (B_1 \to B_2) \approx \top$	By above equality

$\square$

Remark. We can view Theorem 10.3 as a specification of disjointness. Moreover, it provides an alternative approach to deriving algorithmic disjointness whenever $A \sqcup B$ is computable. However, this is not always the case for richer type structures. For instance, in the $\mathsf{F}_\wedge$ calculus [Pierce 1991], least upper bounds are not existent.

10.1.3 INTERPRETATION OF INTERSECTION TYPES

Following Reynolds, a subtyping derivation is interpreted as a morphism $[\![A <: B]\!] : [\![A]\!] \to [\![B]\!]$ with two requirements:

1. For all types A the morphism from $[\![A]\!]$ to $[\![A]\!]$ must be an identity arrow.

2. Whenever $A <: B$ and $B <: C$, the composition of $[\![A <: B]\!]$ and $[\![B <: C]\!]$ must be equal to $[\![A <: C]\!]$, i.e., $[\![A <: B]\!]; [\![B <: C]\!] = [\![A <: C]\!]$. (Here ";" denotes composition in diagrammatic order.)

These requirements actually make $[\![\cdot]\!]$ a functor from the preordered set of types (viewed as a category) to the semantic category of choice.

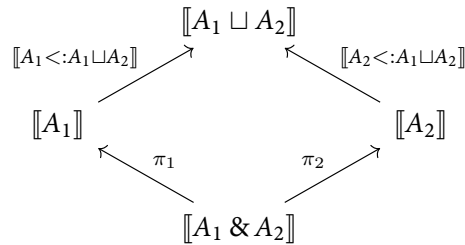*Remark.* By definition, whenever $A \approx B$ we say $[\![A]\!]$ is *isomorphic* to $[\![B]\!]$, written $[\![A]\!] \cong [\![B]\!]$.

Now we consider $[\![A_1 \& A_2]\!]$ in the following steps:

1. By rules S-ANDL and S-ANDR, there must be two morphisms, $\pi_1 : [\![A_1 \& A_2]\!] \to [\![A_1]\!]$ and $\pi_2 : [\![A_1 \& A_2]\!] \to [\![A_2]\!]$

$$
\begin{array}{ccc}
[\![A_1]\!] & & [\![A_2]\!] \\
& \nwarrow \pi_1 \quad \pi_2 \nearrow & \\
& [\![A_1 \& A_2]\!] &
\end{array}
$$
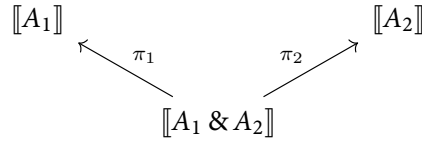
2. For any types A_1 and A_2, there exists a least upper bound $A_1 \sqcup A_2$ (Proposition 10.2), and two morphisms $[\![A_1 <: A_1 \sqcup A_2]\!] : [\![A_1]\!] \to [\![A_1 \sqcup A_2]\!]$ and $[\![A_2 <: A_1 \sqcup A_2]\!] : [\![A_2]\!] \to [\![A_1 \sqcup A_2]\!]$, and the following diagram should commute:

$$
\begin{array}{ccc}
& [\![A_1 \sqcup A_2]\!] & \\
{\scriptstyle [\![A_1 <: A_1 \sqcup A_2]\!]} \nearrow & & \nwarrow {\scriptstyle [\![A_2 <: A_1 \sqcup A_2]\!]} \\
[\![A_1]\!] & & [\![A_2]\!] \\
\nwarrow \pi_1 & & \pi_2 \nearrow \\
& [\![A_1 \& A_2]\!] &
\end{array}
$$

3. For every type A such that $A <: A_1$ and $A <: A_2$, rule S-AND implies that $A <: A_1 \& A_2$, thus a morphism from $[\![A]\!]$ to $[\![A_1 \& A_2]\!]$. Call this μ_0. The following diagram should commute:

$$
\begin{array}{ccc}
& [\![A_1 \sqcup A_2]\!] & \\
\nearrow^{[\![A_1 <: A_1 \sqcup A_2]\!]} & & \nwarrow^{[\![A_2 <: A_1 \sqcup A_2]\!]} \\
[\![A_1]\!] & & [\![A_2]\!] \\
\nwarrow^{\pi_1} & & \nearrow^{\pi_2} \\
& [\![A_1 \& A_2]\!] & \\
\nwarrow_{[\![A <: A_1]\!]} \quad \uparrow^{\mu_0} & & \nearrow_{[\![A <: A_2]\!]} \\
& [\![A]\!] &
\end{array}
$$
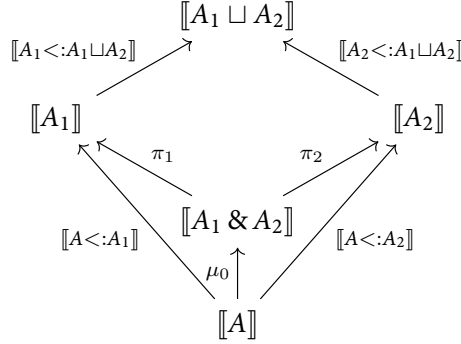
4. Furthermore, in the above diagram, we replace $[\![A]\!]$ by an arbitrary object s and $[\![A <: A_1]\!]$ and $[\![A <: A_1]\!]$ by any morphisms f_1 and f_2 that make the outer diamond commutes, and we require the "mediating morphism" μ_0 from s to $[\![A_1 \& A_2]\!]$ to be unique. Specifically, we define $[\![A_1 \& A_2]\!]$ by requiring the following diagram must commute:

$$
\begin{array}{ccc}
& [\![A_1 \sqcup A_2]\!] & \\
\nearrow^{[\![A_1 <: A_1 \sqcup A_2]\!]} & & \nwarrow^{[\![A_2 <: A_1 \sqcup A_2]\!]} \\
[\![A_1]\!] & & [\![A_2]\!] \\
\nwarrow^{\pi_1} & & \nearrow^{\pi_2} \\
& [\![A_1 \& A_2]\!] & \\
\nwarrow_{f_1} \quad \uparrow^{\mu_0} & & \nearrow_{f_2} \\
& s &
\end{array}
$$
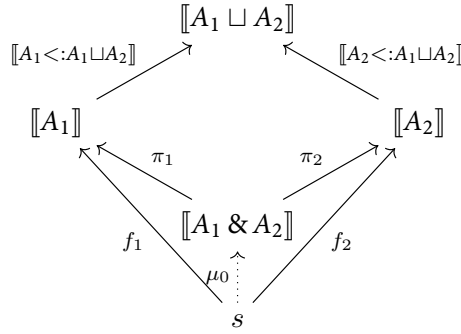
Thus we have defined $[\![A_1 \& A_2]\!]$ to be the *pullback* of $[\![A_1]\!]$, $[\![A_2]\!]$ and $[\![A_1 \sqcup A_2]\!]$.

10.1.4 INTERPRETATION OF DISJOINT INTERSECTION TYPES

Given the interpretation of intersection types, it is fairly straightforward to give the interpretation of disjoint intersection types. First recall that if $A * B$ then $A \sqcup B \approx \top$ (Theorem 10.3). Also we have $[\![\top]\!] = 1$—i.e., the terminal object. By specializing $[\![A_1 \sqcup A_2]\!]$ to be the terminal object ($[\![A_1 <: A_1 \sqcup A_2]\!]$ and $[\![A_2 <: A_1 \sqcup A_2]\!]$ are then uniquely determined), then the pullback "degenerates" to the *product* of $[\![A_1]\!]$ and $[\![A_2]\!]$. In other words, the interpretation of disjoint intersection types is given by the following theorem:

Theorem 10.4. *If $A_1 * A_2$ then $[\![A_1 \mathbin{\&} A_2]\!] \cong [\![A_1]\!] \times [\![A_2]\!]$.*

Remark. It is reassuring to see that this theorem justifies our translation of disjoint intersection types into product types, from the categorical perspective.

10.1.5 Coherence, from the Categorical Perspective

What we have developed so far is the (categorical) interpretation of disjoint intersection types. We are still half way through the ultimate goal of (re-)establishing coherence, now from the categorical perspective. The main difficulty is that we do not know yet how to interpret bidirectional typing judgments—i.e., what are $[\![\Gamma \vdash E \Rightarrow A]\!]$ and $[\![\Gamma \vdash E \Leftarrow A]\!]$, and in particular the interpretation of the merge operator. As remarked earlier, bidirectional type checking (besides disjointness) is essential to coherence. It would be exciting to see some research along the lines of the above, so that we may have a solid mathematical foundation for type systems with disjoint intersection types.

10.2 Implicit Polymorphism

Another interesting and practically useful extension is to study (predicative) implicit polymorphism, in the spirit of languages like Haskell or ML. Our F_i^+ calculus features explicit polymorphism in the sense that we need to provide types during type applications. A classic example of implicit polymorphism is the identity function $\lambda x.\, x$ of type $\forall \alpha.\, \alpha \to \alpha$. When applied to 1, for example, the type variable α will be implicitly instantiated to Int. Moreover, we are interested in *higher-rank polymorphism*, allowing polymorphic quantifiers to appear anywhere in a type. There are several approaches in the literature [Dunfield and Krishnaswami 2013; Odersky and Läufer 1996; Peyton Jones et al. 2007]. Since our declarative type system is already based on bidirectional type-checking, the work by Dunfield and Krishnaswami [2013] is particularly relevant for us. It turns out that coming up with a coherent declarative system is already very challenging, especially the disjointness relation. Below we sketch out some ideas for the initial design.

10.2.1 Declarative Subtyping

First we consider the subtyping rules. Obviously rule S-forall needs to be modified. We replace it with the following two rules:

$$
\frac{\Delta \vdash t * A_1 \qquad \Delta \vdash [t/\alpha]A_2 <: B}{\Delta \vdash \forall(\alpha * A_1).\, A_2 <: B} \text{ IS-allL}
\qquad
\frac{\Delta, \alpha * B_1 \vdash A <: B_2}{\Delta \vdash A <: \forall(\alpha * B_1).\, B_2} \text{ IS-allR}
$$

Rule IS-ALLL says that a type $\forall(\alpha * A_1).\, A_2$ is a subtype of B if some instantiation $[t/\alpha]A_2$ is a subtype of B. However, unlike Dunfield and Krishnaswami's system, in our setting, not all monotypes t that make the subtyping go through are equally fine—those that do not respect the disjointness constraints should not be considered, for the sake of coherence. Otherwise, we would allow $((\lambda x.\, x,,\, 2) : \forall(\alpha * \mathsf{Int}).\, \alpha \to \alpha \,\&\, \mathsf{Int})\, 1$ to type check, which would cause ambiguity at run time. Rule IS-ALLR says that A is a subtype of $\forall(\alpha * B_1).\, B_2$ if we can show that A is a subtype of B_2 in a context extended with $\alpha * B_1$. It is not immediately obvious that these two rules subsume rule S-FORALL, and in particular what happens to "a universal quantifier is contravariant in its disjointness constraints", which is very important in the original subtyping. It can be shown that they do subsume rule S-FORALL, as exhibited by the following derivation:

$$
\cfrac{\cfrac{\cfrac{A_2 <: A_1}{\alpha * A_2 \vdash \alpha * A_1}\; \text{FD-TVARL} \qquad \alpha * A_2 \vdash B_1 <: B_2}{\alpha * A_2 \vdash \forall(\alpha * A_1).\, B_1 <: B_2}\; \text{IS-ALLL}}{\bullet \vdash \forall(\alpha * A_1).\, B_1 <: \forall(\alpha * A_2).\, B_2}\; \text{IS-ALLR}
$$

10.2.2 Disjointness

The disjointness relation needs a major overhaul. For instance, subtyping allows $\forall(\alpha * \mathsf{Char}).\, \alpha \to \alpha <: \mathsf{Int} \to \mathsf{Int}$, and as such, $\forall(\alpha * \mathsf{Char}).\, \alpha \to \alpha$ is no longer disjoint with $\mathsf{Int} \to \mathsf{Int}$, whereas $\forall(\alpha * \mathsf{Int}).\, \alpha \to \alpha$ is disjoint with $\mathsf{Int} \to \mathsf{Int}$. A seemingly intuitive rule is as follows:

$$
\text{FD-IMPLICIT}\\
\cfrac{\Delta \vdash t_1 * A_1 \qquad \Delta \vdash [t_1/\alpha]A_2 * B_2}{\Delta \vdash \forall(\alpha * A_1).\, A_2 * B_2}
$$

In the above rule, the monotype t_1 is existentially quantified: it suffices to exhibit a disjointness derivation of $[t_1/\alpha]A_2$ and B_2 for one monotype in order to build a disjointness derivation of $\forall(\alpha * A_1).\, A_2$ and B_2. Unfortunately, this rule is incorrect as we could guess a "wrong" t_1. Take $\forall(\alpha * \mathsf{Char}).\, \alpha \to \alpha$ for example: one instantiation is $\mathsf{Bool} \to \mathsf{Bool}$, which is disjoint with $\mathsf{Int} \to \mathsf{Int}$. But as we saw, this does not mean $\forall(\alpha * \mathsf{Char}).\, \alpha \to \alpha$ is disjoint with $\mathsf{Int} \to \mathsf{Int}$. Instead we should require *all possible* instantiations are disjoint with B_2:

$$
\text{FD-IMPLICIT}\\
\cfrac{\forall t_1.\, \Delta \vdash t_1 * A_1 \implies \Delta \vdash [t_1/\alpha]A_2 * B_2}{\Delta \vdash \forall(\alpha * A_1).\, A_2 * B_2}
$$

The universal rule is very convenient as an elimination form: if we have a evidence of the disjointness between a polymorphic type and another type, we can immediately obtain the knowledge that all suitable instantiations of the former are disjoint with the latter. However, the universal rule is very hard to use as an introduction rule: it requires us to inspect every possible instantiation; it is getting even worse when we consider two polymorphic types. We do not yet fully understand all the consequences of this rule. Another idea is perhaps we should focus on the opposite side—i.e., what constitutes a non-disjointness relation. But this idea seems more radical.

10.2.3 Declarative Typing

Putting disjointness aside, now we consider the typing rules. Most of the rules stay the same. We remove rules FT-TABS and FT-TAPP, since the syntax now does not include type abstractions and type applications. We add one rule:

$$
\text{FT-GEN} \\
\dfrac{\Delta, \alpha * A; \Gamma \vdash E \Leftarrow B \rightsquigarrow e}{\Delta; \Gamma \vdash E \Leftarrow \forall(\alpha * A).\, B \rightsquigarrow \Lambda\alpha.\, e}
$$

Rule FT-GEN says that E has type $\forall(\alpha*A).\,B$ if E has type B in a context extended with $\alpha*A$. Application becomes a little more complex:

$$
\text{FT-APPI} \\
\dfrac{\Delta; \Gamma \vdash E_1 \Rightarrow A \rightsquigarrow e_1 \qquad \Delta \vdash A \rhd A_1 \to A_2 \qquad \Delta; \Gamma \vdash E_2 \Leftarrow A_1 \rightsquigarrow e_2}{\Delta; \Gamma \vdash E_1\, E_2 \Rightarrow A_2 \rightsquigarrow e_1\, e_2}
$$

The problem is that the inferred type A for E_1 could be a polymorphic quantifier. We need to eliminate universals until we reach an arrow type. To achieve this, we use a matching judgment $\Delta \vdash A \rhd A_1 \to A_2$, which says that we can synthesize an arrow type $A_1 \to A_2$ from A. Once we get an arrow type $A_1 \to A_2$, we use A_1 to check against E_2. The matching judgment [Siek et al. 2015b; Xie et al. 2018], first used in gradual type systems, is inductively defined as follows:

$$
\text{M-FORALL} \\
\dfrac{\Delta \vdash t * A_1 \qquad \Delta \vdash [t/\alpha]A_1 \rhd B_1 \to B_2}{\Delta \vdash \forall(\alpha * A_1).\, A_2 \rhd B_1 \to B_2}
\qquad\qquad
\text{M-ARR} \\
\dfrac{}{\Delta \vdash A_1 \to A_2 \rhd A_1 \to A_2}
$$

Rule M-FORALL, as with rule IS-ALLL, works by guessing instantiations of polymorphic quantifiers with the requirement that the monotype t must meet the disjointness constraints.

Rule M-ARR is trivial, returning $A_1 \rightarrow A_2$ as it is. An alternative to the matching judgment is the *application judgment* $\Delta \vdash A \bullet e \Rightarrow C$ [Dunfield and Krishnaswami 2013], which says that if we apply a term of type A to an argument e, we get something of type C.

Of course the above is only a sketch; we have not studied the declarative system in full, nor its metatheory. One potential problem is that now subtyping and disjointness are mutually recursive (e.g., rule IS-ALLL uses disjointness and rule FD-TVARL uses subtyping), which might pose difficulty in terms of formalization. For coherence, we estimate that the proof method described in this thesis should still work.

10.2.4 ALGORITHMIC SYSTEM

Having a declarative system is only a start. The major challenge is the corresponding algorithmic system. It is well-known that complete type inference is undecidable for intersection types [Barendregt et al. 1983; Coppo et al. 1981]. Some restrictions are obviously in order, leading to different points in the design space with varying degrees of expressiveness and technical difficulties. We are interested to see some research into the algorithmic system.

10.3 DISJOINT POLYMORPHISM VS. ROW POLYMORPHISM

As we have alluded to in Section 9.4, it would be interesting to study the relationship between disjoint polymorphism and row polymorphism, and in particular, whether the former subsumes the latter. As noted by Alpuim et al. [2017], disjoint polymorphism can already encode polymorphic extensible records. For the sake of comparison, we pick the record calculus $\lambda^{\|}$ of Harper and Pierce [1991]—an explicitly-typed, second-order calculus that features single-field records and a symmetric merge operator. In $\lambda^{\|}$, *compatibility constraints* are used to capture negative information about fields. For example, $r_1 \# r_2$ denotes the assertion that the record types r_1 and r_2 have disjoint sets of labels. To illustrate polymorphic extensible records in $\lambda^{\|}$, Harper and Pierce show a function that takes two "disjoint" records x_1 and x_2, where x_1 has at least a field l_1 of type Int and x_2 has at least a field l_2 of type Int, and returns the result of merging x_1 and x_2 (altering their syntax slightly):

$$\Lambda \alpha_1 \#(\{l_1 : \mathsf{Int}\}, \{l_2 : \mathsf{Int}\}). \Lambda \alpha_2 \#(\alpha_1, \{l_1 : \mathsf{Int}\}, \{l_2 : \mathsf{Int}\}).$$
$$\lambda x_1 : (\alpha_1 \| \{l_1 : \mathsf{Int}\}). \lambda x_2 : (\alpha_2 \| \{l_2 : \mathsf{Int}\}). x_1 \| x_2$$

where $r_1 \| r_2$ is the record type obtained by merging r_1 and r_2, and is only defined if $r_1 \# r_2$. The same operator is overloaded to merge two records on the term level. Central to their system is the *constrained quantification* $\forall \alpha \# R. t$, where each record type variable is associ-

ated with a list of *compatibility assumptions* R, whose elements are record types (including record type variables). The *constrained type abstraction* $\Lambda\alpha\#R. e$ is used to create values of constrained quantification.

In F_i^+, we can use disjoint quantification to express their constrained qualification, intersection types to merge record types, and the merge operator to merge records. The function mentioned above can be written in F_i^+ as follows:

$$\Lambda(\alpha_1 * \{l_1 : \mathsf{Int}\} \& \{l_2 : \mathsf{Int}\}). \Lambda(\alpha_2 * \alpha_1 \& \{l_1 : \mathsf{Int}\} \& \{l_2 : \mathsf{Int}\}).$$
$$\lambda x_1 : \alpha_1 \& \{l_1 : \mathsf{Int}\}. \lambda x_2 : \alpha_2 \& \{l_2 : \mathsf{Int}\}. x_1 \,,, x_2$$

However, the merge operator in F_i^+ is more general than its counterpart in $\lambda^{\|}$—i.e., it works on any expressions, not just records. Another important difference is that their compatibility judgment $r_1 \# r_2$ effectively implies that their records must have distinct fields, whereas F_i^+ accepts duplicate fields as long as their types are disjoint. On a related note, $\lambda^{\|}$ is powerful enough to express a polymorphic, conflict-free function that merges two records of statically unknown fields:

$$\mathsf{mergeRcd} = \Lambda\alpha_1\#\mathsf{Empty}. \Lambda\alpha_2\#\alpha_1. \lambda x_1 : \alpha_1. \lambda x_2 : \alpha_2. x_1 \| x_2$$

where Empty is the empty record type. Compare it to our "more expressive" mergeAny function:

$$\mathsf{mergeAny} = \Lambda(\alpha_1 * \top). \Lambda(\alpha_2 * \alpha_1). \lambda x_1 : \alpha_1. \lambda x_2 : \alpha_2. x_1 \,,, x_2$$

that merges *any* two expressions of statically unknown types. Thus we conjecture that F_i^+ completely subsumes $\lambda^{\|}$.

10.4 Recursive Types

One extension of particular importance for modeling object-oriented languages is *recursive types*. A great deal of lessons have been learned about calculi with recursive types and subtyping (see Pierce [2002, chap. 20]). But previous work has been focused on type systems with substantially simpler subtyping relations. For simplicity, we are interested in adding *iso-recursive types*, where a recursive type $\mu X. A$ and its one-step unfolding are transformed back and forth by a pair of functions fold and unfold. The most common definition of iso-

recursive subtyping is the *Amber rule*, popularized by Cardelli's Amber language [Cardelli 1985].

RS-AMBER
$$\frac{\Sigma, X <: Y \vdash A <: B}{\Sigma \vdash \mu X.\, A <: \mu Y.\, B}$$

RS-VAR
$$\frac{(X <: Y) \in \Sigma}{\Sigma \vdash X <: Y}$$

Rule RS-AMBER says that to show $\mu X.\, A$ is a subtype of $\mu Y.\, B$ under some set of assumptions Σ, it suffices to show $A <: B$ under the additional assumption $X <: Y$. Note that this rule, unlike most rules we have seen involving binding constructs on both sides, such as rule S-FORALL in Fig. 4.5, requires that the bound variables X and Y be renamed to be *distinct* before the rule is applied. Rule RS-VAR allows us to conclude $X <: Y$ if the assumptions assume it.

While adding the above two rules to our subtyping relation in Fig. 3.3 (and extending the other rules so that they pass Σ through from premises to conclusion) effectively yields a declarative subtyping relation with recursive types, it is not entirely straightforward as to how they affect disjointness, and in particular, under what conditions are two recursive types disjoint. An initial attempt shows that the amber rule and the disjointness rule for functions are in conflict.

THE PROBLEM.   For the ease of discussion, we do not consider top types, polymorphic types, or BCD subtyping; then a guiding principle of designing disjointness rules is the simple disjointness specification (Definition 1): two types are disjoint if and only if they share no common supertypes. Now consider two recursive types $\mu X.\, X \rightarrow \mathsf{Int}$ and $\mu Y.\, Y \rightarrow \mathsf{Int}$. It is not hard to see that they have no common supertypes (because of contravariance of function argument subtyping). According to Definition 1, they are disjoint. On the other hand, since the disjointness relation is structural, we should inspect the disjointness relation between $X \rightarrow \mathsf{Int}$ and $Y \rightarrow \mathsf{Int}$ under certain relation over X and Y we do not know yet. However, according to rule D-ARR, two functions are disjoint if their range types are disjoint; thus $X \rightarrow \mathsf{Int}$ and $Y \rightarrow \mathsf{Int}$ are not disjoint. So we have $\mu X.\, X \rightarrow \mathsf{Int}$ and $\mu Y.\, Y \rightarrow \mathsf{Int}$ are *not* disjoint: a contradiction!

POSITIVITY TO THE RESCUE.   It is not obvious how to change either rule RS-AMBER or rule D-ARR without disrupting the whole system. A possible solution is to restrict where type variables can occur. Instead of having a *general* recursive type $\mu X.\, A$ where X may occur anywhere in A, we require that X occurs *positively* in A. Specifically, X occurs positively in $A_1 \rightarrow A_2$, if (1) X *does not occur* in A_1, (2) and X occurs positively in A_2. In general, any

occurrences of X within the domain of a function type are *negative occurrences*, whereas any occurrences of X within the range of a function type are *positive occurrences*. For example, the two recursive types in the last subsection are not positive. While positivity does limit the expressiveness of types, most useful datatypes (e.g., natural numbers, lists, streams) are positive. For us, the positivity restriction for recursive types does work with the disjointness rule for function types.

With positive recursive types, here is the disjointness rule for recursive types:

$$
\begin{array}{c}
\text{D-REC} \\
A * B \\
\hline
\mu X.\, A * \mu Y.\, B
\end{array}
$$

We also need a few more disjointness axioms:

$$
\begin{array}{cccc}
\text{DAX-INTREC} & \text{DAX-RCDREC} & \text{DAX-ARRREC} & \text{DAX-INTRVAR} \\
\hline
\mathsf{Int} *_{ax} \mu X.\, B & \{l : A\} *_{ax} \mu X.\, B & A_1 \to A_2 *_{ax} \mu X.\, B & \mathsf{Int} *_{ax} X
\end{array}
$$

$$
\begin{array}{ccc}
\text{DAX-RCDRVAR} & \text{DAX-ARRRVAR} & \text{DAX-RECRVAR} \\
\hline
\{l : A\} *_{ax} X & A_1 \to A_2 *_{ax} X & \mu Y.\, A *_{ax} X
\end{array}
$$

An important observation is that any two distinct type variables are *not* disjoint. A few examples: $\mu X.\ \mathsf{Int} \to X$ and $\mu Y.\ \mathsf{Int} \to Y$ are not disjoint; $\mu X.\ \mathsf{Int} \to X$ and $\mu Y.\ \mathsf{Bool} \to Y \,\&\, \mathsf{Int}$ are not disjoint; $\mu X.\ \mathsf{Int} \to X$ and $\mu Y.\ \mathsf{Int} \to \mathsf{Int} \to Y$ are disjoint. Note that the above is only a sketch; it remains to see whether the disjointness rules are equivalent to the specification.

Another gnarly issue is coherence. To model recursive types, we need to turn to step-indexed logical relations [Ahmed 2006]. We foresee it would be a major technical challenge to adjust our coherence proof and its Coq mechanization.

10.5 OTHER EXTENSIONS

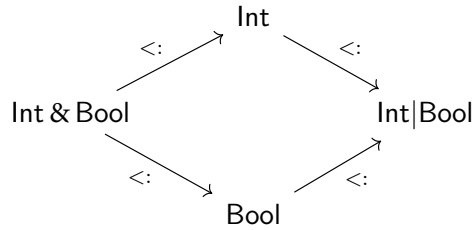There are several important extensions that should also be considered.

10.5.1 UNION TYPES

Union types—as intersections' dual—are also widely used in languages such as Ceylon and Flow. Union types introduce an interesting subtyping relation: a union type $A|B$ is a common

supertype of *A* and *B*; or more precisely, it is a *least upper bound* of *A* and *B*, as exhibited by the following subtyping rules.

<div>

UNION
$$\frac{A <: C \qquad B <: C}{A|B <: C}$$

UNIONL
$$\frac{}{A <: A|B}$$

UNIONR
$$\frac{}{B <: A|B}$$

</div>

Dunfield [2014] has shown that unions can be elaborated into sums, and the merge operator also supports union elimination with two computationally distinct branches. We think that adapting his approach to our setting should not be difficult. An immediate issue is disjointness. Adding union types to our system without any restrictions would cause ambiguity, again. For example, Int & Bool can reach to Int|Bool via two paths, as shown below, each leading to semantically different translations.

$$
\begin{array}{ccc}
 & \text{Int} & \\
 \nearrow {}^{<:} & & \searrow {}^{<:} \\
\text{Int \& Bool} & & \text{Int}|\text{Bool} \\
 \searrow {}_{<:} & & \nearrow {}_{<:} \\
 & \text{Bool} &
\end{array}
$$

More thoughts are needed to come up with a coherent system with union types.

10.5.2 NOMINAL TYPING

Many widely-used OO languages feature nominal type systems where type names play a crucial role. In previous chapters, we often define short names for long or complex compound types to improve readability, e.g., in Section 7.2, we have seen:

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

Such definitions are purely cosmetic: the name `Editor` is just an abbreviation for the record on the right-hand side, and the two are interchangeable in every context. By contrast, in OO

languages such as Java, every compound type (class declaration or interface definition) has a name, and subtyping is explicitly declared between type *names*.

To blend in with our powerful structural subtyping relation, we need to clearly decide which types are based on nominal subtyping, which are based on structural subtyping and how they interact. A rough idea, following the Moby type system [Fisher and Reppy 2000], is to separate *class types* from *object types*, as we did for trait types. Subtyping on class types is nominal, while objects are compared structurally. This is just a high-level intuition; of course there are other details (e.g., disjointness) that need to be worked out. A pleasant property of nominal systems, and also related to our extension of recursive types, is that they offer a natural account of recursive types: if we look at the amber rule RS-AMBER, an explicit subtyping relation $X <: Y$ is added to the context when two recursive types are compared.

10.5.3 MUTABLE STATE

Another direction for future work is to add mutable state, which would affect two aspects of our metatheory: type safety and coherence. For type safety, we expect that lessons learned from previous work on family polymorphism and mutability on OO languages apply to our work. For example, it is well-known that subtyping in the presence of mutable state often needs restrictions. Given such suitable restrictions we expect that type-safety in the presence of mutability still holds. For coherence, it would be a major technical challenge to adjust our coherence proof and its Coq mechanization: logical relations that account for mutable state introduce significant complexity (e.g., see Ahmed [2004]).

PART VI

EPILOGUE

11 CONCLUSION

In this thesis we have argued that the combination of disjoint intersection types, a powerful subtyping relation and parametric polymorphism greatly improve the state-of-art technique for modularity and code reuse. In the course of our investigation, we have gradually introduced three new typed calculi with increasing expressiveness:

- The λ_i^+ calculus is the basic calculus with disjoint intersection types and a powerful subtyping relation. We have shown that it captures the essence of nested composition, enables a simple solution to the expression problem. In order to prove coherence, we have proposed the canonicity relation based on logical relations.

- The F_i^+ calculus, building on λ_i^+, supports parametric polymorphism. We have shown that it improves upon the finally tagless [Carette et al. 2009] and object algebra [Oliveira and Cook 2012] approaches and support advanced compositional designs, and enables the development of highly modular and reusable programs. We have also extended the canonicity relation to establish coherence property of F_i^+.

- SEDEL—an object-oriented language design—building on F_i^+, supports, among others, typed first traits. We have illustrated the applicability of SEDEL with several example uses for first-class traits. Furthermore, we have conducted a case study that modularizes programming language interpreters. The case study demonstrates that the state-of-art encodings of extensible designs are greatly improved by SEDEL.

Of course there are several noteworthy limitations in our proposed calculi. (1) Lack of mutable state, which is a desirable feature in modern programming languages, and also very important in order for mainstream languages to adopt some of the proposed language mechanisms in this thesis. (2) Lack of recursive types, which is important for modeling object-oriented language. (3) Poor runtime performance due to redundant coercion applications (e.g., applying multiple ids). As we remarked earlier, our generated coercions are not efficient in terms of space. There is existing work on space-efficient coercions [Herman et al. 2010; Siek et al. 2015a], which we expect should be applicable to the work in this thesis.

Hopefully we have convinced the reader that disjoint intersection types have great potential to serve as a foundation for powerful, flexible and yet type-safe and easy to reason

object-oriented languages. We hope that the concepts and the methods described in this thesis may serve as a helpful guide to researchers and programmers alike in their attempts to understand and build better software. Thus this thesis serves as a stepping stone for further investigation of disjoint intersection types in conjunction with other type disciplines. A great number of open questions, new research directions lie ahead (cf. Chapter 10).

Bibliography

[Citing pages are listed after each reference.]

Amal Ahmed. 2004. *Semantics of types for mutable state*. Ph.D. Dissertation. Princeton University. [cited on page 154]

Amal Ahmed. 2006. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming (ESOP)*. [cited on page 152]

Eric E. Allen, Jonathan Bannet, and Robert Cartwright. 2003. A first-class approach to genericity. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. [cited on page 137]

João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. 2017. Disjoint Polymorphism. In *European Symposium on Programming (ESOP)*. [cited on pages 8, 9, 10, 22, 23, 35, 40, 42, 44, 66, 67, 132, 134, 135, and 149]

Nada Amin, Adriaan Moors, and Martin Odersky. 2012. Dependent object types. In *Workshop on Foundations of Object-Oriented Languages*. [cited on page 8]

Davide Ancona, Giovanni Lagorio, and Elena Zucca. 2003. Jam–designing a Java extension with mixins. In *Transactions on Programming Languages and Systems (TOPLAS)*. [cited on page 137]

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A filter lambda model and the completeness of type assignment. *The journal of symbolic logic* 48, 04 (1983), 931–940. [cited on pages 9, 16, 35, 41, 132, and 149]

Jan Bessai, Boris Düdder, Andrej Dudenhefner, Tzu-Chun Chen, and Ugo de'Liguoro. 2014. Typing Classes and Mixins with Intersection Types. In *Workshop on Intersection Types and Related Systems (ITRS)*. [cited on page 132]

Lorenzo Bettini, Sara Capecchi, and Ferruccio Damiani. 2013a. On flexible dynamic trait replacement for Java-like languages. *Science of Computer Programming* 78, 7 (2013), 907 – 932. [cited on page 140]

Bibliography

Lorenzo Bettini and Ferruccio Damiani. 2017. Xtraitj: Traits for the Java platform. *Journal of Systems and Software* 131 (2017), 419 – 441. [cited on page 138]

Lorenzo Bettini, Ferruccio Damiani, Ina Schaefer, and Fabio Strocco. 2013b. TraitRecordJ: A programming language with traits and records. *Science of Computer Programming* 78, 5 (2013), 521 – 541. [cited on page 138]

Xuan Bi and Bruno C. d. S. Oliveira. 2018. Typed First-Class Traits. In *European Conference on Object-Oriented Programming (ECOOP)*. [cited on page 13]

Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2018. The Essence of Nested Composition. In *European Conference on Object-Oriented Programming (ECOOP)*. [cited on pages 13, 35, 43, 82, 132, and 134]

Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. *Distributive Disjoint Polymorphism for Compositional Programming*. Technical Report. The University of Hong Kong. `https://bixuanzju.github.io/files/ESOP2019.pdf` [cited on page 13]

Dariusz Biernacki and Piotr Polesiuk. 2015. Logical relations for coherence of effect subtyping. In *International Conference on Typed Lambda Calculi and Applications (TLCA)*. [cited on page 132]

Lasse Blaauwbroek. 2017. *On the Interaction Between Unrestricted Union and Intersection Types and Computational Effects*. Master's thesis. Technical University Eindhoven. [cited on page 134]

Viviana Bono, Amit Patel, and Vitaly Shmatikov. 1999. A core calculus of classes and mixins. In *European Conference on Object-Oriented Programming (ECOOP)*. [cited on page 137]

Gilad Bracha. 1992. *The programming language jigsaw: mixins, modularity and multiple inheritance*. Ph.D. Dissertation. Dept. of Computer Science, University of Utah. [cited on pages 5, 6, 96, and 137]

Gilad Bracha and William R. Cook. 1990. Mixin-based inheritance. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. [cited on pages 5, 6, 24, 96, 132, and 137]

Gilad Bracha and Gary Lindstrom. 1992. Modularity meets Inheritance. In *International Conference on Computer Languages*. [cited on page 140]

Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. 1991. Inheritance as implicit coercion. *Information and Computation* 93, 1 (1991), 172–221. [cited on page 131]

Richard P Brent and Hsiang-Tsung Kung. 1980. The chip complexity of binary arithmetic. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*. 190–200. [cited on page 56]

Kim Bruce, Luca Cardelli, Giuseppe Castagna, Gary T Leavens, Benjamin Pierce, et al. 1996. On binary methods. In *Theory and Practice of Object Systems*. [cited on page 35]

Kim B Bruce, Martin Odersky, and Philip Wadler. 1998. A statically safe alternative to virtual types. In *European Conference on Object-Oriented Programming*. [cited on page 138]

Kim B. Bruce, Angela Schuett, and Robert van Gent. 1995. PolyTOIL: A Type-Safe Polymorphic Object-Oriented Language. In *European Conference on Object-Oriented Programming (ECOOP)*. [cited on page 139]

Luca Cardelli. 1985. Amber. In *Combinators and Functional Programming Languages, Thirteenth Spring School of the LITP*. [cited on page 151]

Luca Cardelli and Peter Wegner. 1985. On understanding types, data abstraction, and polymorphism. *Comput. Surveys* 17, 4 (1985), 471–523. [cited on pages 10, 71, 73, 131, and 135]

Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 05 (2009), 509. [cited on pages 10, 55, 56, 128, and 157]

Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. 1992. A calculus for overloaded functions with subtyping. In *Conference on LISP and Functional Programming*. [cited on pages 8, 17, and 134]

Giuseppe Castagna and Victor Lanvin. 2017. Gradual typing with union and intersection types. *Proceedings of the ACM on Programming Languages* 1, ICFP (aug 2017), 1–28. [cited on page 135]

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. 2014. Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. In *Principles of Programming Languages (POPL)*. [cited on page 135]

Bibliography

David Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. 2007. Tribe: More Types for Virtual Classes. In *International Conference on Aspect-Oriented Software Development (AOSD)*. [cited on pages 7 and 138]

Adriana B Compagnoni and Benjamin C Pierce. 1996. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science (MSCS)* 6, 5 (1996), 469–501. [cited on pages 8 and 15]

Steve Cook. 1987. Varieties of inheritance. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*. 35–40. [cited on page 24]

William R. Cook. 1989. *A Denotational Semantics of Inheritance*. Ph.D. Dissertation. Brown University. [cited on pages 111 and 118]

William R. Cook. 1990. Object-oriented programming versus abstract data types. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*. [cited on page 121]

William R. Cook. 2013. *Anatomy of Programming Languages*. `http://www.cs.utexas.edu/~wcook/anatomy/` [cited on pages 12, 121, and 125]

William R. Cook, Walter Hill, and Peter S Canning. 1989. Inheritance is not subtyping. In *Principles of Programming Languages (POPL)*. [cited on page 16]

William R. Cook and Jens Palsberg. 1989. A denotational semantics of inheritance and its correctness. In *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*. [cited on pages 11, 28, 111, and 117]

Mario Coppo and Mariangiola Dezani-Ciancaglini. 1978. A new type assignment for λ-terms. 19 (01 1978), 139–156. [cited on pages 8 and 15]

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 1981. Functional Characters of Solvable Terms. *Mathematical Logic Quarterly* 27, 2-6 (1981), 45–58. [cited on pages 108 and 149]

Andrea Corradi, Marco Servetto, and Elena Zucca. 2012. DeepFJig — Modular composition of nested classes. *The Journal of Object Technology* 11, 2 (2012), 1:1.   [cited on pages 7 and 137]

Karl Crary, Robert Harper, and Sidd Puri. 1999. What is a recursive module?. In *Programming Language Design and Implementation (PLDI)*. [cited on page 140]

Pierre-Louis Curien and Giorgio Ghelli. 1992. Coherence of subsumption, minimum typing and type-checking in $F_\leq$. *Mathematical Structures in Computer Science (MSCS)* 2, 01 (1992), 55. [cited on page 131]

Haskell B. Curry and Robert Feys. 1958. *Combinatory Logic*. Vol. 1. North Holland. Second edition, 1968. [cited on page 15]

Rowan Davies and Frank Pfenning. 2000. Intersection types and computational effects. In *International Conference on Functional Programming (ICFP)*. [cited on pages 8 and 20]

Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. 2006. Traits: A Mechanism for Fine-grained Reuse. *Transactions on Programming Languages and Systems (TOPLAS)* 28, 2 (2006), 331–388. [cited on pages 6, 9, 96, 100, 107, 118, 137, and 140]

Joshua Dunfield. 2014. Elaborating intersection and union types. *Journal of Functional Programming (JFP)* 24, 2-3 (2014), 133–165. [cited on pages 8, 17, 18, 19, 35, 134, and 153]

Joshua Dunfield and Neelakantan R Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism. In *International Conference on Functional Programming (ICFP)*. [cited on pages 146, 147, and 149]

Joshua Dunfield and Frank Pfenning. 2003. Type assignment for intersections and unions in call-by-value languages. In *Foundations of Software Science and Computation Structure (FoSSaCS)*. [cited on page 8]

ECMA International. 2015. *ECMAScript 2015 Language Specification* (6th ed.). Geneva. http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf [cited on page 97]

Erik Ernst. 2000. gbeta-a language with virtual attributes, Block Structure, and Propagating, Dynamic Inheritance. *DAIMI Report Series* 29, 549 (2000). [cited on page 26]

Erik Ernst. 2001. Family Polymorphism. In *European Conference on Object-Oriented Programming (ECOOP)*. [cited on pages 5, 7, 25, 36, and 138]

Erik Ernst. 2003. Higher-Order Hierarchies. In *European Conference on Object-Oriented Programming*. [cited on page 38]

Erik Ernst. 2004. The expression problem, Scandinavian style. *On Mechanisms For Specialization* (2004), 27. [cited on pages 15, 26, 27, and 35]

Bibliography

Erik Ernst, Klaus Ostermann, and William R. Cook. 2006. A Virtual Class Calculus. In *Symposium on Principles of Programming Languages (POPL)*. [cited on pages 7 and 138]

Facebook. 2014. Flow. `https://flow.org/`. [cited on page 8]

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. 2002. DrScheme: a programming environment for Scheme. *Journal of Functional Programing* 12, 2 (2002), 159–182. [cited on page 136]

Kathleen Fisher and John Reppy. 2000. Extending Moby with inheritance-based subtyping. In *European Conference on Object-Oriented Programming*. 83–107. [cited on page 154]

Kathleen Fisher and John Reppy. 2004. A typed calculus of traits. In *Workshop on Foundations of Object-Oriented Languages*. [cited on pages 106 and 137]

Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. 2006. Scheme with Classes, Mixins, and Traits. In *Programming Languages and Systems (APLAS)*. [cited on pages 95, 136, and 137]

Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1998. Classes and mixins. In *Principles of Programming Languages (POPL)*. [cited on page 137]

Maarten M Fokkinga. 1989. Tupling and mutumorphisms. *Squiggolist* 1, 4 (1989). [cited on page 58]

Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Conference on Programming Language Design and Implementation (PLDI)*. [cited on page 8]

Benedict R Gaster and Mark P Jones. 1996. *A polymorphic type system for extensible records and variants*. Technical Report. University of Nottingham. [cited on page 135]

Jeremy Gibbons and Nicolas Wu. 2014. Folding domain-specific languages: deep and shallow embeddings (functional Pearl). In *ICFP*. ACM, 339–347. [cited on page 55]

Robert Harper. 2016. *Practical Foundations for Programming Languages*. Cambridge University Press. [cited on page 78]

Robert Harper and Benjamin Pierce. 1991. A Record Calculus Based on Symmetric Concatenation. In *Principles of Programming Languages (POPL)*. [cited on pages 135 and 149]

Fritz Henglein. 1994. Dynamic typing: syntax and proof theory. *Science of Computer Programming* 22, 3 (6 1994), 197–230. [cited on pages 40 and 44]

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Symbolic Computation* 23, 2 (2010), 167. [cited on pages 45 and 157]

Roger Hindley. 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society* 146 (1969), 29–60. [cited on page 89]

Ralf Hinze. 2004. An algebra of scans. In *International Conference on Mathematics of Program Construction*. 186–210. [cited on page 59]

Paul Jolly, Sophia Drossopoulou, Christopher Anderson, and Klaus Ostermann. 2004. Simple dependent types: Concord. In *European Conference on Object-Oriented Programming Workshop on Formal Techniques for Java Programs (FTfJP)*. [cited on page 139]

Timothy Jones, Michael Homer, James Noble, and Kim B. Bruce. 2016. Object Inheritance Without Classes. In *European Conference on Object-Oriented Programming (ECOOP)*. [cited on page 139]

Oleg Kiselyov. 2012. Typed tagless final interpreters. In *Generic and Indexed Programming*. Springer, 130–174. [cited on page 59]

Shriram Krishnamurthi, Matthias Felleisen, and Daniel P Friedman. 1998. Synthesizing object-oriented and functional design to promote re-use. In *European Conference on Object-Oriented Programming*. [cited on page 121]

Toshihiko Kurata and Masako Takahashi. 1995. Decidable properties of intersection type systems. *Typed Lambda Calculi and Applications* (1995), 297–311. [cited on page 132]

Giovanni Lagorio, Marco Servetto, and Elena Zucca. 2012. Featherweight Jigsaw — Replacing inheritance by composition in Java-like languages. *Information and Computation* 214 (2012), 86 – 111. [cited on page 137]

Joachim Lambek. 1985. Cartesian closed categories and typed λ-calculi. In *LITP Spring School on Theoretical Computer Science*. 136–175. [cited on page 141]

Olivier Laurent. 2012a. Intersection types with subtyping by means of cut elimination. *Fundamenta Informaticae* 121, 1-4 (2012), 203–226. [cited on pages 48 and 133]

Olivier Laurent. 2012b. A syntactic introduction to intersection types. (2012). Unpublished note. [cited on pages 48 and 49]

Olivier Laurent. 2018. Intersection Subtyping with Constructors. In *Proceedings of the Ninth Workshop on Intersection Types and Related Systems*. [cited on pages 48 and 133]

Bibliography

Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin. 2015. A Theory of Tagged Objects. In *European Conference on Object-Oriented Programming (ECOOP)*. [cited on pages 6, 96, 107, and 136]

Ole Lehrmann Madsen, Birger Mø-Pedersen, and Kristen Nygaard. 1993. *Object-oriented Programming in the BETA Programming Language*. Addison-Wesley. [cited on page 138]

Daan Leijen. 2005. Extensible records with scoped labels. *Trends in Functional Programming* 5 (2005), 297–312. [cited on page 135]

Roberto E Lopez-Herrejon, Don Batory, and William Cook. 2005. Evaluating support for features in advanced modularization technologies. In *European Conference on Object-Oriented Programming (ECOOP)*. [cited on page 125]

David MacQueen. 1984. Modules for standard ML. In *Symposium on LISP and functional programming (LFP)*. [cited on pages 4 and 140]

O. L. Madsen and B. Moller-Pedersen. 1989. Virtual classes: a powerful mechanism in object-oriented programming. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPLAS)*. [cited on page 138]

Bertrand Meyer. 1987. Eiffel: programming for reusability and extendibility. *ACM Sigplan Notices* 22, 2 (1987), 85–94. [cited on page 139]

Microsoft. 2012. TypeScript. `https://www.typescriptlang.org/`. [cited on page 8]

Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences* 17, 3 (1978), 348–375. [cited on page 89]

Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92. [cited on page 131]

J. Garrett Morris and James McKinna. 2019. Abstracting Extensible Data Types. In *Principles of Programming Languages (POPL)*. [cited on page 135]

James Hiram Morris Jr. 1969. *Lambda-calculus models of programming languages*. Ph.D. Dissertation. Massachusetts Institute of Technology. [cited on page 78]

Fabian Muehlboeck and Ross Tate. 2018. Empowering union and intersection types with integrated subtyping. In *OOPSLA*. [cited on page 133]

Peter Naur and Brian Randell (Eds.). 1969. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO.* [cited on page 3]

Oscar Nierstrasz, Stéphane Ducasse, and Nathanael Schärli. 2006. Flattening Traits. *Journal of Object Technology* 5, 4 (2006), 129–148. [cited on pages 118 and 137]

James Noble, Andrew P. Black, Kim B. Bruce, Michael Homer, and Timothy Jones. 2017. Grace's Inheritance. *Journal of Object Technology* 16, 2 (2017), 2:1–35. [cited on page 139]

Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. 2004. Scalable extensibility via nested inheritance. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* [cited on pages 7, 25, and 139]

Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. 2006. J&: Nested Intersection for Scalable Software Composition. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* [cited on pages 4 and 139]

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. *An overview of the Scala programming language.* Technical Report. EPFL. [cited on pages 8 and 102]

Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *Symposium on Principles of Programming Languages (POPL).* [cited on page 146]

Martin Odersky and Matthias Zenger. 2005. Scalable component abstractions. In *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA).* [cited on pages 7 and 106]

Bruno C. d. S. Oliveira. 2009. Modular Visitor Components: A Practical Solution to the Expression Families Problem. In *European Conference on Object Oriented Programming (ECOOP).* [cited on pages 4, 11, 26, 121, and 123]

Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses. In *European Conference on Object-Oriented Programming (ECOOP).* [cited on pages 4, 10, 12, 26, 55, 121, 122, 124, and 157]

Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint intersection types. In *International Conference on Functional Programming (ICFP).* [cited on pages 8, 9, 10, 15, 19, 20, 22, 35, 40, 42, 44, 48, 66, and 134]

Bibliography

Bruno C. d. S. Oliveira, Tijs Van Der Storm, Alex Loh, and William R Cook. 2013. Feature-Oriented programming with object algebras. In *European Conference on Object-Oriented Programming (ECOOP)*. [cited on pages 7, 121, and 124]

Bruno C. d. S. Oliveira, Meng Wang, and Jeremy Gibbons. 2008. The visitor pattern as a reusable, generic, type-safe component. In *Object Oriented Programming: Systems, Languages and Applications (OOPSLA)*. [cited on page 122]

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical Type Inference for Arbitrary-Rank Types. *Journal of Functional Programming (JFP)* 17, 1 (2007), 1–82. [cited on page 146]

Benjamin C Pierce. 1989. *A decision procedure for the subtype relation on intersection types with bounded variables*. Technical Report. Carnegie Mellon University. [cited on pages 10, 12, 48, 49, 50, 52, 53, 132, and 133]

Benjamin C Pierce. 1991. *Programming with intersection types and bounded polymorphism*. Ph.D. Dissertation. University of Pennsylvania. [cited on pages 55, 135, 136, and 143]

Benjamin C Pierce. 1994. Bounded quantification is undecidable. *Information and Computation* 112, 1 (1994), 131–165. [cited on pages 55, 71, 74, and 136]

Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press. [cited on pages 13 and 150]

Gordon Plotkin. 1973. *Lambda-definability and logical relations*. Edinburgh University. [cited on pages 10, 31, 78, and 81]

Garrel Pottinger. 1980. A type assignment for the strongly normalizable λ-terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism* (1980), 561–577. [cited on pages 8, 15, and 108]

Xin Qi and Andrew C. Myers. 2009. Sharing Classes Between Families. In *Conference on Programming Language Design and Implementation (PLDI)*. [cited on page 139]

Redhat. 2011. Ceylon. `https://ceylon-lang.org/`. [cited on page 8]

Jakob Rehof and Paweł Urzyczyn. 2011. Finite combinatory logic with intersection types. In *International Conference on Typed Lambda Calculi and Applications*. [cited on page 132]

Tillmann Rendel, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2014. From Object Algebras to Attribute Grammars. In *Object-Oriented Programming, Systems Languages and Applications (OOPSLA)*. [cited on pages 121 and 124]

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Proceedings of the IFIP 9th World Computer Congress*. [cited on pages 88 and 90]

John C Reynolds. 1988. *Preliminary design of the programming language Forsythe*. Technical Report. Carnegie Mellon University. [cited on pages 8, 16, 17, 19, 134, and 141]

John C. Reynolds. 1991. The coherence of languages with intersection types. In *Lecture Notes in Computer Science (LNCS)*. Springer Berlin Heidelberg, 675–700. [cited on pages 8, 18, 131, 141, and 142]

John C Reynolds. 1997. Design of the programming language Forsythe. In *ALGOL-like languages*. 173–233. [cited on pages 17, 18, and 133]

Andreas Rossberg. 2015. 1ML – core and modules united (F-ing first-class modules). In *International Conference on Functional Programming (ICFP)*. [cited on page 140]

Andreas Rossberg and Derek Dreyer. 2013. Mixin' Up the ML Module System. *Transactions on Programming Languages and Systems (TOPLAS)* 35, 1 (2013), 1–84. [cited on page 140]

Andreas Rossberg, Claudio Russo, and Derek Dreyer. 2014. F-ing modules. *Journal of Functional Programming (JFP)* 24, 05 (2014), 529–607. [cited on page 140]

Claudio V. Russo. 2001. Recursive structures for standard ML. In *International Conference on Functional Programming (ICFP)*. [cited on page 140]

Chieri Saito and Atsushi Igarashi. 2009. Matching *ThisType* to subtyping. In *Symposium on Applied Computing (SAC)*. [cited on page 139]

Chieri Saito, Atsushi Igarashi, and Mirko Viroli. 2007. Lightweight family polymorphism. *Journal of Functional Programming (JFP)* 18, 03 (2007). [cited on pages 7 and 139]

Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. 2003. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming (ECOOP)*. [cited on pages 5, 6, 9, 24, 25, 96, 100, 105, 106, 107, 118, and 137]

Nathanael Scharli, St Ducasse, Roel Wuyts, Andrew Black, et al. 2003. *Traits: The formal model*. Technical Report. Oregon Health & Science University. [cited on page 137]

Jan Schwinghammer. 2008. Coherence of subsumption for monadic types. *Journal of Functional Programming (JFP)* 19, 02 (2008), 157. [cited on page 131]

Marco Servetto and Elena Zucca. 2014. A meta-circular language for active libraries. *Science of Computer Programming* 95 (2014), 219 – 253. [cited on page 140]

Jeremy Siek, Peter Thiemann, and Philip Wadler. 2015a. Blame and coercion: together again for the first time. In *Conference on Programming Language Design and Implementation (PLDI)*. [cited on pages 45 and 157]

Jeremy G. Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. 2015b. Refined Criteria for Gradual Typing. In *LIPIcs-Leibniz International Proceedings in Informatics*. [cited on page 148]

Yannis Smaragdakis and Don S. Batory. 2000. Mixin-Based Programming in C++. In *Generative and Component-Based Software Engineering (GCSE)*. [cited on page 99]

Charles Smith and Sophia Drossopoulou. 2005. Chai: Traits for Java-Like Languages. In *European Conference on Object-Oriented Programming (ECOOP)*. [cited on pages 137 and 140]

Richard Statman. 1985. Logical relations and the typed λ-calculus. *Information and Control* 65, 2-3 (1985), 85–97. [cited on pages 10, 31, 78, and 81]

Rick Statman. 2015. A Finite Model Property for Intersection Types. *Electronic Proceedings in Theoretical Computer Science* 177 (2015), 1–9. [cited on page 133]

Wouter Swierstra. 2008. Data types à la carte. *Journal of Functional Programming (JFP)* 18, 4 (2008), 423–436. [cited on pages 4 and 26]

W. W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *The Journal of symbolic logic* 32, 2 (1967), 198–212. [cited on pages 10, 31, 78, and 81]

Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Gradual typing for first-class classes. In *Object-oriented Programming: Systems, Languages and Applications (OOPSLA)*. [cited on pages 5, 6, 96, 97, and 136]

Mads Torgersen. 2004. The Expression Problem Revisited. In *European Conference on Object-Oriented Programming (ECOOP)*. [cited on pages 11, 26, 121, and 122]

David Ungar and Randall B Smith. 1988. SELF: the power of simplicity (object-oriented language). In *Thirty-Third IEEE Computer Society International Conference, Digest of Papers*. [cited on page 139]

Philip Wadler. 1998. The expression problem. *Java-genericity mailing list* (1998). [cited on pages 4, 7, 26, and 121]

Mitchell Wand. 1987. Complete type inference for simple objects. In *Symposium on Logic in Computer Science (LICS)*. [cited on page 135]

Mitchell Wand. 1994. Type inference for objects with instance variables and inheritance. *Theoretical aspects of object-oriented programming* (1994), 97–120. [cited on page 104]

Yanlin Wang and Bruno C. d. S. Oliveira. 2016. The expression problem, trivially!. In *Proceedings of the 15th International Conference on Modularity*. [cited on pages 4, 26, and 138]

Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira. 2018. Consistent Subtyping for All. In *European Symposium on Programming (ESOP)*. [cited on page 148]

Pamela Zave. 1999. Faq sheet on feature interaction. *Link: http://www. research. att. com/~ pamela/faq. html* (1999). [cited on page 125]

Mathhias Zenger and Martin Odersky. 2005. Independently Extensible Solutions to the Expression Problem. In *Foundations of Object-Oriented Languages*. [cited on pages 4, 7, and 26]

Yizhou Zhang and Andrew C. Myers. 2017. Familia: unifying interfaces, type classes, and family polymorphism. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. [cited on page 140]

Part VII

Technical Appendix

A Circuit Embeddings

```
{-

    Parallel Prefix Circuits DSL

    Finally Tagless Encoding

    Supporting zygo- and mutumorphisms

 -}


{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TypeApplications #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE UndecidableInstances #-}

{- Generic Definitions for Records -}

data Record :: [*] → * where
  Nil  :: Record '[]
  Cons :: a → Record as → Record (a ': as)

class In a as where
  project :: Record as → a

instance {-# OVERLAPPING #-} In a (a ': as) where
```

```
  project (Cons x _) = x

instance {-# OVERLAPPING #-} In a as ⇒ In a (b ': as) where
  project (Cons _ xs) = project xs

data All c :: [*] → * where
  AllNil  :: All c '[]
  AllCons :: c a ⇒ All c as → All c (a ': as)

{- Circuit DSL Infrastructure -}

class Circuit0 c where
  identity_ :: Int → c
  fan_      :: Int → c

class Circuit0 c ⇒ Circuit1 d c where
  beside_  :: Record d → Record d → c
  above_   :: Record d → Record d → c
  stretch_ :: [Int] → Record d → c

class Circuit2 d1 d2 where
  modality :: All (Circuit1 d1) d2

instance Circuit2 d1 '[] where
  modality = AllNil

instance (Circuit1 d1 a, Circuit2 d1 as) ⇒ Circuit2 d1 (a ': as) where
  modality = AllCons modality

type Circuit3 d = Circuit2 d d

identity :: forall d. Circuit3 d ⇒ Int → Record d
identity = identity' (modality @d @d)
  where
    identity' :: All (Circuit1 d1) d2 → Int → Record d2
    identity' AllNil      n = Nil
    identity' (AllCons m) n = Cons (identity_ n) (identity' m n)

fan :: forall d. Circuit3 d ⇒ Int → Record d
fan = fan' (modality @d @d)
  where
```

```
    fan' :: All (Circuit1 d1) d2 → Int → Record d2
    fan' AllNil      n = Nil
    fan' (AllCons m) n = Cons (fan_ n) (fan' m n)


beside :: forall d. Circuit3 d ⇒ Record d → Record d → Record d
beside = beside' (modality @ d @ d)
  where
    beside' :: All (Circuit1 d1) d2 → Record d1 → Record d1 → Record d2
    beside' AllNil      c1 c2  =  Nil
    beside' (AllCons m) c1 c2  =  Cons (beside_ c1 c2) (beside' m c1 c2)


above :: forall d. Circuit3 d ⇒ Record d → Record d → Record d
above = above' (modality @ d @ d)
  where
    above' :: All (Circuit1 d1) d2 → Record d1 → Record d1 → Record d2
    above' AllNil      c1 c2  =  Nil
    above' (AllCons m) c1 c2  =  Cons (above_ c1 c2) (above' m c1 c2)


stretch :: forall d. Circuit3 d ⇒ [Int] → Record d → Record d
stretch = stretch' (modality @ d @ d)
  where
    stretch' :: All (Circuit1 d1) d2 → [Int] → Record d1 → Record d2
    stretch' AllNil      ws c  =  Nil
    stretch' (AllCons m) ws c  =  Cons (stretch_ ws$ c) (stretch' m ws c)


{- Shallow Embeddings -}

-- Width

newtype Width = Width { width :: Int }

instance Circuit0 Width where
  identity_ n   = Width n
  fan_ n        = Width n

instance In Width d ⇒ Circuit1 d Width where
  beside_ c1 c2 = Width (width (project c1) + width (project c2))
  above_ c1 c2  = project c1
  stretch_ ws c = Width (sum ws)

-- Well-Sizedness
```

```
newtype WellSized = WellSized { wellSized :: Bool }

instance Circuit0 WellSized where
  identity_ n  =  WellSized True
  fan_ n       =  WellSized True

instance (In WellSized d, In Width d) ⇒ Circuit1 d WellSized where
  beside_ c1 c2 = WellSized (wellSized (project c1) && wellSized (project
    c2))
  above_ c1 c2 =
    WellSized
      (width (project c1) == width (project c2) &&
       wellSized (project c1) && wellSized (project c2))
  stretch_ ws c =
    WellSized (wellSized (project c) && length ws == width (project c))

{- Example -}

test :: Record '[Width, WellSized]
test = above (identity 5) (fan 5)

test' =
  case test of
    Cons x (Cons y Nil) → (width x, wellSized y)
```

B Decidability

Definition 18 (Size of $\mathcal{Q}$).

$$size([]) = 0$$
$$size(\mathcal{Q}, l) = size(\mathcal{Q})$$
$$size(\mathcal{Q}, A) = size(\mathcal{Q}) + size(A)$$
$$size(\mathcal{Q}, \alpha * A) = size(\mathcal{Q}) + size(A)$$

Definition 19 (Size of types).

$$size(c) = 1$$
$$size(A \to B) = size(A) + size(B) + 1$$
$$size(A \,\&\, B) = size(A) + size(B) + 1$$
$$size(\{l : A\}) = size(A) + 1$$
$$size(\forall(\alpha * A).\,B) = size(A) + size(B) + 1$$

Lemma 4.12 (Decidability of algorithmic subtyping). *Given $\mathcal{Q}$, A and B, it is decidable whether there exists co, such that $\mathcal{Q} \vdash A \prec: B \rightsquigarrow co$.*

Proof. Let the judgment $\mathcal{Q} \vdash A \prec: B \rightsquigarrow co$ be measured by $size(\mathcal{Q}) + size(A) + size(B)$. For each subtyping rule, we show that every inductive premise is smaller than the conclusion.

- Rules A-const, A-top, and A-bot have no premise.

- Case

  A-and
  $$\frac{\mathcal{Q} \vdash A \prec: B_1 \rightsquigarrow co_1 \qquad \mathcal{Q} \vdash A \prec: B_2 \rightsquigarrow co_2}{\mathcal{Q} \vdash A \prec: B_1 \,\&\, B_2 \rightsquigarrow [\![\mathcal{Q}]\!]^{\&} \circ \langle co_1, co_2 \rangle}$$

  In both premises, they have the same $\mathcal{Q}$ and A, but B_1 and B_2 are smaller than $B_1 \,\&\, B_2$.

- Case

  A-ARR

  $$\frac{\mathcal{Q}, B_1 \vdash A \prec: B_2 \rightsquigarrow co}{\mathcal{Q} \vdash A \prec: B_1 \rightarrow B_2 \rightsquigarrow co}$$

  The size of premise is smaller than the conclusion as $size(B_1 \rightarrow B_2) = size(B_1) + size(B_2) + 1$.

- Case

  A-RCD

  $$\frac{\mathcal{Q}, l \vdash A \prec: B \rightsquigarrow co}{\mathcal{Q} \vdash A \prec: \{l : B\} \rightsquigarrow co}$$

  In premise, the size is $size(\mathcal{Q}, l) + size(A) + size(B) = size(\mathcal{Q}) + size(A) + size(B)$, which is smaller than $size(\mathcal{Q}) + size(A) + size(\{l : B\}) = size(\mathcal{Q}) + size(A) + size(B) + 1$.

- Case

  A-FORALL

  $$\frac{\mathcal{Q}, \alpha * B_1 \vdash A \prec: B_2 \rightsquigarrow co}{\mathcal{Q} \vdash A \prec: \forall (\alpha * B_1). B_2 \rightsquigarrow co}$$

  The size of premise is smaller than the conclusion as $size(\mathcal{Q}) + size(A) + size(\forall (\alpha * B_1). B_2) = size(\mathcal{Q}) + size(A) + size(B_1) + size(B_2) + 1 > size(\mathcal{Q}, \alpha * B_1) + size(A) + size(B_2) = size(\mathcal{Q}) + size(B_1) + size(A) + size(B_2)$.

- Case

  A-ARRCONST

  $$\frac{[] \vdash A \prec: A_1 \rightsquigarrow co_1 \qquad \mathcal{Q} \vdash A_2 \prec: c \rightsquigarrow co_2}{A, \mathcal{Q} \vdash A_1 \rightarrow A_2 \prec: c \rightsquigarrow co_1 \rightarrow co_2}$$

  In the first premise, the size is smaller than the conclusion because of the size of $\mathcal{Q}$ and A_2. In the second premise, the size is smaller than the conclusion because $size(A_1 \rightarrow A_2) > size(A_2)$.

- Case

  A-RCDCONST

  $$\frac{\mathcal{Q} \vdash A \prec: c \rightsquigarrow co}{l, \mathcal{Q} \vdash \{l : A\} \prec: c \rightsquigarrow co}$$

  The size of premise is smaller as $size(l, \mathcal{Q}) + size(\{l : A\}) + size(c) = size(\mathcal{Q}) + size(A) + size(c) + 1 > size(\mathcal{Q}) + size(A) + size(c)$.

- Case

A-andConst1

$$\frac{\mathcal{Q} \vdash A_1 \prec: c \rightsquigarrow co}{\mathcal{Q} \vdash A_1 \& A_2 \prec: c \rightsquigarrow co \circ \pi_1}$$

The size of premise is smaller as $size(A_1 \& A_2) = size(A_1) + size(A_2) + 1 > size(A_1)$.

- Case

A-andConst2

$$\frac{\mathcal{Q} \vdash A_2 \prec: c \rightsquigarrow co}{\mathcal{Q} \vdash A_1 \& A_2 \prec: c \rightsquigarrow co \circ \pi_2}$$

The size of premise is smaller as $size(A_1 \& A_2) = size(A_1) + size(A_2) + 1 > size(A_2)$.

- Case

A-allConst

$$\frac{[] \vdash A \prec: A_1 \qquad \mathcal{Q} \vdash A_2 \prec: c \rightsquigarrow co}{(\alpha * A, \mathcal{Q}) \vdash \forall(\alpha * A_1). A_2 \prec: c \rightsquigarrow co_\forall}$$

In the first premise, the size is smaller than the conclusion because of the size of $\mathcal{Q}$ and A_2. In the second premise, the size is smaller than the conclusion because $size(\forall(\beta * A_1). A_2) > size(A_2)$.

$\square$

Lemma B.1 (Decidability of Top-like types). *Given a type A, it is decidable whether $\rceil A \lceil$.*

Proof. Induction on the judgment $\rceil A \lceil$. For each subtyping rule, we show that every inductive premise is smaller than the conclusion.

- rule TL-top has no premise.

- Case

TL-and

$$\frac{\rceil A \lceil \qquad \rceil B \lceil}{\rceil A \& B \lceil}$$

$size(A \& B) = size(A) + size(B) + 1$, which is greater than $size(A)$ and $size(B)$.

- Case

TL-arr

$$\frac{\rceil B \lceil}{\rceil A \rightarrow B \lceil}$$

$size(A \rightarrow B) = size(A) + size(B) + 1$, which is greater than $size(B)$.

- Case

  TL-RCD
  $$\frac{\rceil A\lceil}{\rceil \{l : A\}\lceil}$$

  $size(\{l : A\}) = size(A) + 1$, which is greater than $size(A)$.

- Case

  TL-ALL
  $$\frac{\rceil B\lceil}{\rceil \forall (\alpha * A).\, B\lceil}$$

  $size(\forall (\alpha * A).\, B) = size(A) + size(B) + 1$, which is greater than $size(B)$.

  $\square$

Lemma B.2 (Decidability of disjointness axioms checking). *Given A and B, it is decidable whether $A *_{ax} B$.*

Proof. $A *_{ax} B$ is decided by the shape of types, and thus it's decidable. $\square$

Lemma 4.13 (Decidability of disjointness checking). *Given Δ, A and B, it is decidable whether $\Delta \vdash A * B$.*

Proof. Let the judgment $\Delta \vdash A * B$ be measured by $size(A) + size(B)$. For each subtyping rule, we show that every inductive premise is smaller than the conclusion.

- Case

  FD-TOPL
  $$\frac{\rceil A\lceil}{\Delta \vdash A * B}$$

  By Lemma B.1, we know $\rceil A\lceil$ is decidable.

- Case

  FD-TOPR
  $$\frac{\rceil B\lceil}{\Delta \vdash A * B}$$

  By Lemma B.1, we know $\rceil B\lceil$ is decidable.

- Case

  FD-ARR
  $$\frac{\Delta \vdash A_2 * B_2}{\Delta \vdash A_1 \rightarrow A_2 * B_1 \rightarrow B_2}$$

  $size(A_1 \rightarrow A_2) + size(B_1 \rightarrow B_2) > size(A_2) + size(B_2)$.

- Case

$$\frac{\text{FD-andL}}{\Delta \vdash A_1 * B \qquad \Delta \vdash A_2 * B}{\Delta \vdash A_1 \& A_2 * B}$$

$size(A_1 \& A_2) + size(B)$ is greater than $size(A_1) + size(B)$ and $size(A_2) + size(B)$.

- Case

$$\frac{\text{FD-andR}}{\Delta \vdash A * B_1 \qquad \Delta \vdash A * B_2}{\Delta \vdash A * B_1 \& B_2}$$

$size(B_1 \& B_2) + size(A)$ is greater than $size(B_1) + size(A)$ and $size(B_2) + size(A)$.

- Case

$$\frac{\text{FD-rcdEq}}{\Delta \vdash A * B}{\Delta \vdash \{l : A\} * \{l : B\}}$$

$size(\{l : A\}) + size(\{l : B\})$ is greater than $size(A) + size(B)$.

- Case

$$\frac{\text{FD-rcdNeq}}{l_1 \neq l_2}{\Delta \vdash \{l_1 : A\} * \{l_2 : B\}}$$

It's decidable whether l_1 is equal to l_2.

- Case

$$\frac{\text{FD-tvarL}}{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash \alpha * B}$$

By Lemma 4.12, it's decidable whether $A <: B$.

- Case

$$\frac{\text{FD-tvarR}}{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash B * \alpha}$$

By Lemma 4.12, it's decidable whether $A <: B$.

- Case

$$\frac{\text{FD-forall}}{\Delta, \alpha * A_1 \& A_2 \vdash B_1 * B_2}{\Delta \vdash \forall(\alpha * A_1). B_1 * \forall(\alpha * A_2). B_2}$$

$size(\forall(\alpha * A_1).\, B_1) + size(\forall(\alpha * A_2).\, B_2)$ is greater than $size(B_1) + size(B_2)$.

- Case

$$\frac{\text{FD-\textsc{ax}}}{\quad A *_{ax} B \quad}{\Delta \vdash A * B}$$

By Lemma B.2 it is decidable.

$\square$

C   Proofs about SEDEL

Lemma 7.1. *If $\Delta \vdash \mathcal{A}$ then $|\Delta| \vdash |\mathcal{A}|$.*

Proof. By simple induction on the derivation of the judgment. $\qquad\square$

Lemma 7.2. *If $\mathcal{A} <: \mathcal{B}$ then $|\mathcal{A}| <: |\mathcal{B}|$.*

Proof. Most of them are straightforward. We only show rule TS-TRAIT.

-
$$
\begin{array}{c}
\text{TS-TRAIT} \\
\dfrac{\mathcal{B}_1 <: \mathcal{A}_1 \qquad \mathcal{A}_2 <: \mathcal{B}_2}{\mathbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] <: \mathbf{Trait}\,[\mathcal{B}_1, \mathcal{B}_2]}
\end{array}
$$

$|\mathcal{B}_1| <: |\mathcal{A}_1|$	By i.h.
$|\mathcal{A}_2| <: |\mathcal{B}_2|$	By i.h.
$|\mathcal{A}_1| \to |\mathcal{A}_2| <: |\mathcal{B}_1| \to |\mathcal{B}_2|$	By rule S-ARR

$\qquad\square$

Lemma C.1. *If $\mathcal{A} *_{ax} \mathcal{B}$ then $|\mathcal{A}| *_{ax} |\mathcal{B}|$.*

Proof. Note that $|\mathbf{Trait}\,[\mathcal{A}, \mathcal{B}]| = |\mathcal{A}| \to |\mathcal{B}|$, the rest are immediate. $\qquad\square$

Lemma 7.3. *If $\Delta \vdash \mathcal{A} * \mathcal{B}$ then $|\Delta| \vdash |\mathcal{A}| * |\mathcal{B}|$.*

Proof. By induction on the derivation of the judgment.

- Rules SD-TOPL, SD-TOPR, and SD-RCDNEQ are immediate.

-
$$
\begin{array}{c}
\text{SD-TVARL} \\
\dfrac{(\alpha * \mathcal{A}) \in \Delta \qquad \mathcal{A} <: \mathcal{B}}{\Delta \vdash \alpha * \mathcal{B}}
\end{array}
$$

$|\mathcal{A}| <: |\mathcal{B}|$ By Lemma 7.2
$(\alpha * \mathcal{A}) \in \Delta$ Given
$(\alpha * |\mathcal{A}|) \in |\Delta|$ Above
$|\Delta| \vdash \alpha * |\mathcal{B}|$ By rule FD-TVARL

-

SD-TVARR
$$\frac{(\alpha * \mathcal{A}) \in \Delta \qquad \mathcal{A} <: \mathcal{B}}{\Delta \vdash \mathcal{B} * \alpha}$$

$|\mathcal{A}| <: |\mathcal{B}|$ By Lemma 7.2
$(\alpha * \mathcal{A}) \in \Delta$ Given
$(\alpha * |\mathcal{A}|) \in |\Delta|$ Above
$|\Delta| \vdash |\mathcal{B}| * \alpha$ By rule FD-TVARR

-

SD-FORALL
$$\frac{\Delta, \alpha * \mathcal{A}_1 \mathbin{\&} \mathcal{A}_2 \vdash \mathcal{B}_1 * \mathcal{B}_2}{\Delta \vdash \forall(\alpha * \mathcal{A}_1).\, \mathcal{B}_1 * \forall(\alpha * \mathcal{A}_2).\, \mathcal{B}_2}$$

$|\Delta|, \alpha * |\mathcal{A}_1| \mathbin{\&} |\mathcal{A}_2| \vdash |\mathcal{B}_1| * |\mathcal{B}_2|$ By i.h.
$|\Delta| \vdash \forall(\alpha * |\mathcal{A}_1|).\, |\mathcal{B}_1| * \forall(\alpha * |\mathcal{A}_2|).\, |\mathcal{B}_2|$ By rule FD-FORALL

-

SD-RCDEQ
$$\frac{\Delta \vdash \mathcal{A} * \mathcal{B}}{\Delta \vdash \{l : \mathcal{A}\} * \{l : \mathcal{B}\}}$$

$|\Delta| \vdash |\mathcal{A}| * |\mathcal{B}|$ By i.h.
$|\Delta| \vdash \{l : |\mathcal{A}|\} * \{l : |\mathcal{B}|\}$ By rule FD-RCDEQ

-

SD-ARR
$$\frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \mathcal{A}_1 \to \mathcal{A}_2 * \mathcal{B}_1 \to \mathcal{B}_2}$$

$|\Delta| \vdash |\mathcal{A}_2| * |\mathcal{B}_2|$ By i.h.
$|\Delta| \vdash |\mathcal{A}_1| \to |\mathcal{A}_2| * |\mathcal{B}_1| \to |\mathcal{B}_2|$ By rule FD-ARR

-

$$\frac{\text{SD-andL}}{\Delta \vdash \mathcal{A}_1 * \mathcal{B} \qquad \Delta \vdash \mathcal{A}_2 * \mathcal{B}}{\Delta \vdash \mathcal{A}_1 \& \mathcal{A}_2 * \mathcal{B}}$$

$\lvert\Delta\rvert \vdash \lvert\mathcal{A}_1\rvert * \lvert\mathcal{B}\rvert$	By i.h.
$\lvert\Delta\rvert \vdash \lvert\mathcal{A}_2\rvert * \lvert\mathcal{B}\rvert$	By i.h.
$\lvert\Delta\rvert \vdash \lvert\mathcal{A}_1\rvert \& \lvert\mathcal{A}_2\rvert * \lvert\mathcal{B}\rvert$	By rule FD-andL

-

$$\frac{\text{SD-andR}}{\Delta \vdash \mathcal{A} * \mathcal{B}_1 \qquad \Delta \vdash \mathcal{A} * \mathcal{B}_2}{\Delta \vdash \mathcal{A} * \mathcal{B}_1 \& \mathcal{B}_2}$$

$\lvert\Delta\rvert \vdash \lvert\mathcal{A}\rvert * \lvert\mathcal{B}_1\rvert$	By i.h.
$\lvert\Delta\rvert \vdash \lvert\mathcal{A}\rvert * \lvert\mathcal{B}_2\rvert$	By i.h.
$\lvert\Delta\rvert \vdash \lvert\mathcal{A}\rvert * \lvert\mathcal{B}_1\rvert \& \lvert\mathcal{B}_2\rvert$	By rule FD-andR

-

$$\frac{\text{SD-trait}}{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \textbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] * \textbf{Trait}\,[\mathcal{B}_1, \mathcal{B}_2]}$$

$\lvert\Delta\rvert \vdash \lvert\mathcal{A}_2\rvert * \lvert\mathcal{B}_2\rvert$	By i.h.
$\lvert\Delta\rvert \vdash \lvert\mathcal{A}_1\rvert \to \lvert\mathcal{A}_2\rvert * \lvert\mathcal{B}_1\rvert \to \lvert\mathcal{B}_2\rvert$	By rule FD-arr

-

$$\frac{\text{SD-traitArr1}}{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \textbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] * \mathcal{B}_1 \to \mathcal{B}_2}$$

$\lvert\Delta\rvert \vdash \lvert\mathcal{A}_2\rvert * \lvert\mathcal{B}_2\rvert$	By i.h.
$\lvert\Delta\rvert \vdash \lvert\mathcal{A}_1\rvert \to \lvert\mathcal{A}_2\rvert * \lvert\mathcal{B}_1\rvert \to \lvert\mathcal{B}_2\rvert$	By rule FD-arr

-

$$\frac{\text{SD-traitArr2}}{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \mathcal{A}_1 \to \mathcal{A}_2 * \textbf{Trait}\,[\mathcal{B}_1, \mathcal{B}_2]}$$

$|\Delta| \vdash |\mathcal{A}_2| * |\mathcal{B}_2|$ By i.h.

$|\Delta| \vdash |\mathcal{A}_1| \rightarrow |\mathcal{A}_2| * |\mathcal{B}_1| \rightarrow |\mathcal{B}_2|$ By rule FD-ARR

-

$$\frac{\begin{array}{c}\text{SD-AX}\\ \mathcal{A} *_{ax} \mathcal{B}\end{array}}{\Delta \vdash \mathcal{A} * \mathcal{B}}$$

$|\mathcal{A}| *_{ax} |\mathcal{B}|$ By Lemma C.1

$|\Delta| \vdash |\mathcal{A}| * |\mathcal{B}|$ By rule FD-AX

$\square$

Theorem 7.4 (Type-safe translation). *We have that:*

- *If $\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \mathcal{A} \rightsquigarrow E$ then $|\Delta|; |\Gamma| \vdash E \Rightarrow |\mathcal{A}|$.*

- *If $\Delta; \Gamma \vdash \mathcal{T} \Leftarrow \mathcal{A} \rightsquigarrow E$ then $|\Delta|; |\Gamma| \vdash E \Leftarrow |\mathcal{A}|$.*

Proof. By induction on the typing judgment.

- Rules ST-TOP, ST-INT, and ST-VAR are immediate.

-

$$\frac{\begin{array}{c}\text{ST-ANNO}\\ \Delta; \Gamma \vdash \mathcal{T} \Leftarrow \mathcal{A} \rightsquigarrow E\end{array}}{\Delta; \Gamma \vdash \mathcal{T} : \mathcal{A} \Rightarrow \mathcal{A} \rightsquigarrow E : |\mathcal{A}|}$$

$|\Delta|; |\Gamma| \vdash E \Leftarrow |\mathcal{A}|$ By i.h.

$|\Delta|; |\Gamma| \vdash E : |\mathcal{A}| \Rightarrow |\mathcal{A}|$ By rule FT-ANNO

-

$$\frac{\begin{array}{c}\text{ST-APP}\\ \Delta; \Gamma \vdash \mathcal{T}_1 \Rightarrow \mathcal{A}_1 \rightarrow \mathcal{A}_2 \rightsquigarrow E_1 \qquad \Delta; \Gamma \vdash \mathcal{T}_2 \Leftarrow \mathcal{A}_1 \rightsquigarrow E_2\end{array}}{\Delta; \Gamma \vdash \mathcal{T}_1 \, \mathcal{T}_2 \Rightarrow \mathcal{A}_2 \rightsquigarrow E_1 \, E_2}$$

$|\Delta|; |\Gamma| \vdash E_1 \Rightarrow |\mathcal{A}_1| \rightarrow |\mathcal{A}_2|$ By i.h.

$|\Delta|; |\Gamma| \vdash E_2 \Leftarrow |\mathcal{A}_1|$ By i.h.

$|\Delta|; |\Gamma| \vdash E_1 \, E_2 \Rightarrow |\mathcal{A}_2|$ By rule FT-APP

-

$$\text{ST-\textsc{tapp}}$$
$$\frac{\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \forall(\alpha * \mathcal{B}_1). \mathcal{B}_2 \rightsquigarrow E \qquad \Delta \vdash \mathcal{A} \qquad \Delta \vdash \mathcal{A} * \mathcal{B}_1}{\Delta; \Gamma \vdash \mathcal{T}\, \mathcal{A} \Rightarrow [\mathcal{A}/\alpha]\mathcal{B}_2 \rightsquigarrow E\,|\mathcal{A}|}$$

$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash E \Rightarrow \forall(\alpha * \lvert\mathcal{B}_1\rvert). \lvert\mathcal{B}_2\rvert$	By i.h.
$\lvert\Delta\rvert \vdash \lvert\mathcal{A}\rvert$	By Lemma 7.1
$\lvert\Delta\rvert \vdash \lvert\mathcal{A}\rvert * \lvert\mathcal{B}_1\rvert$	By Lemma 7.3
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash E\,\lvert\mathcal{A}\rvert \Rightarrow [\lvert\mathcal{A}\rvert/\alpha]\lvert\mathcal{B}_2\rvert$	By rule FT-\textsc{tapp}

-

$$\text{ST-\textsc{merge}}$$
$$\frac{\Delta; \Gamma \vdash \mathcal{T}_1 \Rightarrow \mathcal{A} \rightsquigarrow E_1 \qquad \Delta; \Gamma \vdash \mathcal{T}_2 \Rightarrow \mathcal{B} \rightsquigarrow E_2 \qquad \Delta \vdash \mathcal{A} * \mathcal{B}}{\Delta; \Gamma \vdash \mathcal{T}_1 ,, \mathcal{T}_2 \Rightarrow \mathcal{A} \& \mathcal{B} \rightsquigarrow E_1 ,, E_2}$$

$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash E_1 \Rightarrow \lvert\mathcal{A}\rvert$	By i.h.
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash E_2 \Rightarrow \lvert\mathcal{B}\rvert$	By i.h.
$\lvert\Delta\rvert \vdash \lvert\mathcal{A}\rvert * \lvert\mathcal{B}\rvert$	By Lemma 7.3
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash E_1 ,, E_2 \Rightarrow \lvert\mathcal{A}\rvert \& \lvert\mathcal{B}\rvert$	By rule FT-\textsc{merge}

-

$$\text{ST-\textsc{rcd}}$$
$$\frac{\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \mathcal{A} \rightsquigarrow E}{\Delta; \Gamma \vdash \{l = \mathcal{T}\} \Rightarrow \{l : \mathcal{A}\} \rightsquigarrow \{l = E\}}$$

$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash E \Rightarrow \lvert\mathcal{A}\rvert$	By i.h.
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash \{l = E\} \Rightarrow \{l : \lvert\mathcal{A}\rvert\}$	By rule FT-\textsc{rcd}

-

$$\text{ST-\textsc{proj}}$$
$$\frac{\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \{l : \mathcal{A}\} \rightsquigarrow E}{\Delta; \Gamma \vdash \mathcal{T}.l \Rightarrow \mathcal{A} \rightsquigarrow E.l}$$

$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash E \Rightarrow \{l : \lvert\mathcal{A}\rvert\}$	By i.h.
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash E.l \Rightarrow \lvert\mathcal{A}\rvert$	By rule FT-\textsc{proj}

-

ST-TABS

$$\dfrac{\Delta \vdash \mathcal{A} \qquad \Delta, \alpha * \mathcal{A}; \Gamma \vdash \mathcal{T} \;\Rightarrow\; \mathcal{B} \rightsquigarrow E}{\Delta; \Gamma \vdash \Lambda(\alpha * \mathcal{A}).\, \mathcal{T} \;\Rightarrow\; \forall(\alpha * \mathcal{A}).\, \mathcal{B} \rightsquigarrow \Lambda(\alpha * |\mathcal{A}|).\, E}$$

$|\Delta| \vdash |\mathcal{A}|$	By Lemma 7.1
$|\Delta|, \alpha * |\mathcal{A}|; |\Gamma| \vdash E \;\Rightarrow\; |\mathcal{B}|$	By i.h.
$|\Delta|; |\Gamma| \vdash \Lambda(\alpha * |\mathcal{A}|).\, E \;\Rightarrow\; \forall(\alpha * |\mathcal{A}|).\, |\mathcal{B}|$	By rule FT-TABS

-

ST-LETREC

$$\dfrac{\Delta; \Gamma, x : \mathcal{A} \vdash \mathcal{T}_1 \;\Leftarrow\; \mathcal{A} \rightsquigarrow E_1 \qquad \Delta; \Gamma, x : \mathcal{A} \vdash \mathcal{T}_2 \;\Rightarrow\; \mathcal{B} \rightsquigarrow E_2}{\Delta; \Gamma \vdash \textbf{letrec}\, x : \mathcal{A} = \mathcal{T}_1 \,\textbf{in}\, \mathcal{T}_2 \;\Rightarrow\; \mathcal{B} \rightsquigarrow \textbf{letrec}\, x : |\mathcal{A}| = E_1 \,\textbf{in}\, E_2}$$

$|\Delta|; |\Gamma|, x : |\mathcal{A}| \vdash E_1 \;\Leftarrow\; |\mathcal{A}|$	By i.h.
$|\Delta|; |\Gamma|, x : |\mathcal{A}| \vdash E_2 \;\Rightarrow\; |\mathcal{B}|$	By i.h.
$|\Delta|; |\Gamma| \vdash \textbf{letrec}\, x : |\mathcal{A}| = E_1 \,\textbf{in}\, E_2 \;\Rightarrow\; |\mathcal{B}|$	

-

ST-NEW

$$\dfrac{\overline{\Delta; \Gamma \vdash \mathcal{T}_i \;\Rightarrow\; \textbf{Trait}\,[\mathcal{A}_i, \mathcal{B}_i] \rightsquigarrow E_i}^{\,i \in 1..n}}{\overline{\mathcal{A} <: \mathcal{A}_i}^{\,i \in 1..n} \qquad \Delta \vdash \mathcal{B}_1 * .. * \mathcal{B}_n \qquad \mathcal{B}_1 \,\&\, .. \,\&\, \mathcal{B}_n <: \mathcal{A}}{\Delta; \Gamma \vdash \textbf{new}\,[\mathcal{A}](\overline{\mathcal{T}_i}^{\,i \in 1..n}) \;\Rightarrow\; \mathcal{A} \rightsquigarrow \textbf{letrec}\,\text{self} : |\mathcal{A}| = \overline{(E_i\,\text{self})}^{\,i \in 1..n} \,\textbf{in}\,\text{self}}}$$

$|\Delta|; |\Gamma| \vdash E_i \;\Rightarrow\; |\mathcal{A}_i| \to |\mathcal{B}_i|$	By i.h.
$|\mathcal{A}| <: |\mathcal{A}_i|$	By Lemma 7.2
$|\Delta| \vdash |\mathcal{B}_1| * .. * |\mathcal{B}_n|$	By Lemma 7.3
$|\mathcal{B}_1| \,\&\, .. \,\&\, |\mathcal{B}_n| <: |\mathcal{A}|$	By Lemma 7.2
$|\Delta|; |\Gamma|, \text{self} : |\mathcal{A}| \vdash \text{self} \;\Rightarrow\; |\mathcal{A}|$	By rule FT-VAR
$|\Delta|; |\Gamma|, \text{self} : |\mathcal{A}| \vdash \text{self} \;\Leftarrow\; |\mathcal{A}_i|$	By rule FT-SUB
$|\Delta|; |\Gamma|, \text{self} : |\mathcal{A}| \vdash E_i\,\text{self} \;\Rightarrow\; |\mathcal{B}_i|$	By rule FT-APP
$|\Delta|; |\Gamma|, \text{self} : |\mathcal{A}| \vdash (E_1\,\text{self})\,,,\, ..\,,,\, (E_n\,\text{self}) \;\Rightarrow\; |\mathcal{B}_1| \,\&\, .. \,\&\, |\mathcal{B}_n|$	By rule FT-MERGE
$|\Delta|; |\Gamma|, \text{self} : |\mathcal{A}| \vdash (E_1\,\text{self})\,,,\, ..\,,,\, (E_n\,\text{self}) \;\Leftarrow\; |\mathcal{A}|$	By rule FT-SUB
$|\Delta|; |\Gamma| \vdash \textbf{letrec}\,\text{self} : |\mathcal{A}| = (E_1\,\text{self})\,,,\, ..\,,,\, (E_n\,\text{self}) \,\textbf{in}\,\text{self} \;\Rightarrow\; |\mathcal{A}|$	

-

ST-TRAIT

$$\overline{\Delta; \Gamma, \mathsf{self} : \mathcal{B} \vdash \mathcal{T}_i \Rightarrow \mathbf{Trait}\,[\mathcal{B}_i, \mathcal{C}_i] \rightsquigarrow E_i}^{\,i \in 1..n}$$

$$\Delta; \Gamma, \mathsf{self} : \mathcal{B} \vdash \{\,\overline{l_j = \mathcal{T}_j}'^{\,j \in 1..m}\,\} \Rightarrow \mathcal{C} \rightsquigarrow E$$

$$\frac{\overline{\mathcal{B} <: \mathcal{B}_i}^{\,i \in 1..n} \qquad \Delta \vdash \mathcal{C}_1 * .. * \mathcal{C}_n * \mathcal{C} \qquad \mathcal{C}_1 \,\&\, .. \,\&\, \mathcal{C}_n \,\&\, \mathcal{C} <: \mathcal{A}}{\Delta; \Gamma \vdash \mathbf{trait}\,[\mathsf{self} : \mathcal{B}]\,\mathbf{inherits}\,\overline{\mathcal{T}_i}^{\,i \in 1..n}\,\{\,\overline{l_j = \mathcal{T}_j}'^{\,j \in 1..m}\,\} : \mathcal{A} \Rightarrow \mathbf{Trait}\,[\mathcal{B}, \mathcal{A}] \rightsquigarrow}$$
$$\lambda \mathsf{self} : |\mathcal{B}|.\,(((\overline{(E_i\,\mathsf{self})}^{\,i \in 1..n})\,,,\,E)$$

$\|\Delta\|; \|\Gamma\|, \mathsf{self} : \|\mathcal{B}\| \vdash E_i \Rightarrow \|\mathcal{B}_i\| \to \|\mathcal{C}_i\|$	By i.h.
$\|\Delta\|; \|\Gamma\|, \mathsf{self} : \|\mathcal{B}\| \vdash E \Rightarrow \|\mathcal{C}\|$	By i.h.
$\|\mathcal{B}\| <: \|\mathcal{B}_i\|$	By Lemma 7.2
$\|\Delta\| \vdash \|\mathcal{C}_1\| * .. * \|\mathcal{C}_n\| * \|\mathcal{C}\|$	By Lemma 7.1
$\|\mathcal{C}_1\| \,\&\, .. \,\&\, \|\mathcal{C}_n\| \,\&\, \|\mathcal{C}\| <: \|\mathcal{A}\|$	By Lemma 7.2
$\|\Delta\|; \|\Gamma\|, \mathsf{self} : \|\mathcal{B}\| \vdash \mathsf{self} \Rightarrow \|\mathcal{B}\|$	By rule FT-VAR
$\|\Delta\|; \|\Gamma\|, \mathsf{self} : \|\mathcal{B}\| \vdash \mathsf{self} \Leftarrow \|\mathcal{B}_i\|$	By rule FT-SUB
$\|\Delta\|; \|\Gamma\|, \mathsf{self} : \|\mathcal{B}\| \vdash E_i\,\mathsf{self} \Rightarrow \|\mathcal{C}_i\|$	By rule TI-APP
$\|\Delta\|; \|\Gamma\|, \mathsf{self} : \|\mathcal{B}\| \vdash (E_1\,\mathsf{self})\,,,\,..\,,,\,(E_n\,\mathsf{self})\,,,\,E \Rightarrow \|\mathcal{C}_1\| \,\&\, .. \,\&\, \|\mathcal{C}_n\| \,\&\, \|\mathcal{C}\|$	By rule FT-MERGE
$\|\Delta\|; \|\Gamma\|, \mathsf{self} : \|\mathcal{B}\| \vdash (E_1\,\mathsf{self})\,,,\,..\,,,\,(E_n\,\mathsf{self})\,,,\,E \Rightarrow \|\mathcal{A}\|$	By rule FT-SUB
$\|\Delta\|; \|\Gamma\| \vdash \lambda\mathsf{self} : \|\mathcal{B}\|.\,(E_1\,\mathsf{self})\,,,\,..\,,,\,(E_n\,\mathsf{self})\,,,\,E \Rightarrow \|\mathcal{B}\| \to \|\mathcal{A}\|$	

-

ST-FORWARD

$$\frac{\Delta; \Gamma \vdash \mathcal{T}_1 \Rightarrow \mathbf{Trait}\,[\mathcal{A}, \mathcal{B}] \rightsquigarrow E_1 \qquad \Delta; \Gamma \vdash \mathcal{T}_2 \Leftarrow \mathcal{A} \rightsquigarrow E_2}{\Delta; \Gamma \vdash \mathcal{T}_1 \wedge \mathcal{T}_2 \Rightarrow \mathcal{B} \rightsquigarrow E_1\,E_2}$$

$\|\Delta\|; \|\Gamma\| \vdash E_1 \Rightarrow \|\mathcal{A}\| \to \|\mathcal{B}\|$	By i.h.
$\|\Delta\|; \|\Gamma\| \vdash E_2 \Leftarrow \|\mathcal{A}\|$	By i.h.
$\|\Delta\|; \|\Gamma\| \vdash E_1\,E_2 \Rightarrow \|\mathcal{B}\|$	By rule FT-APP

-

ST-ABS

$$\frac{\Delta \vdash \mathcal{A} \qquad \Delta; \Gamma, x : \mathcal{A} \vdash \mathcal{T} \Leftarrow \mathcal{B} \rightsquigarrow E}{\Delta; \Gamma \vdash \lambda x.\mathcal{T} \Leftarrow \mathcal{A} \to \mathcal{B} \rightsquigarrow \lambda x.\,E}$$

$\|\Delta\| \vdash \|\mathcal{A}\|$	By Lemma 7.1
$\|\Delta\|; \|\Gamma\|, x : \|\mathcal{A}\| \vdash E \Leftarrow \|\mathcal{B}\|$	By i.h.

$|\Delta|; |\Gamma| \vdash \lambda x. E \; \Leftarrow \; |\mathcal{A}| \to |\mathcal{B}|$ By rule FT-ABS

-         $\dfrac{\text{ST-SUB} \\ \Delta; \Gamma \vdash \mathcal{T} \; \Rightarrow \; \mathcal{A} \rightsquigarrow E \qquad \mathcal{A} <: \mathcal{B} \qquad \Delta \vdash \mathcal{B}}{\Delta; \Gamma \vdash \mathcal{T} \; \Leftarrow \; \mathcal{B} \rightsquigarrow E}$

$|\Delta|; |\Gamma| \vdash E \; \Rightarrow \; |\mathcal{A}|$ By i.h.
$|\mathcal{A}| <: |\mathcal{B}|$ By Lemma 7.2
$|\Delta| \vdash |\mathcal{B}|$ By Lemma 7.1
$|\Delta|; |\Gamma| \vdash E \; \Leftarrow \; |\mathcal{B}|$ By rule FT-SUB

$\square$

D $\quad \lambda_i^+$ Typing Rules, in Full

$\boxed{A <: B \rightsquigarrow co}$ *(Declarative subtyping)*

S-REFL
$$\frac{}{A <: A \rightsquigarrow \mathsf{id}}$$

S-TRANS
$$\frac{A_2 <: A_3 \rightsquigarrow co_1 \qquad A_1 <: A_2 \rightsquigarrow co_2}{A_1 <: A_3 \rightsquigarrow co_1 \circ co_2}$$

S-TOP
$$\frac{}{A <: \top \rightsquigarrow \mathsf{top}}$$

S-RCD
$$\frac{A <: B \rightsquigarrow co}{\{l : A\} <: \{l : B\} \rightsquigarrow co}$$

S-ARR
$$\frac{B_1 <: A_1 \rightsquigarrow co_1 \qquad A_2 <: B_2 \rightsquigarrow co_2}{A_1 \to A_2 <: B_1 \to B_2 \rightsquigarrow co_1 \to co_2}$$

S-ANDL
$$\frac{}{A_1 \,\&\, A_2 <: A_1 \rightsquigarrow \pi_1}$$

S-ANDR
$$\frac{}{A_1 \,\&\, A_2 <: A_2 \rightsquigarrow \pi_2}$$

S-AND
$$\frac{A_1 <: A_2 \rightsquigarrow co_1 \qquad A_1 <: A_3 \rightsquigarrow co_2}{A_1 <: A_2 \,\&\, A_3 \rightsquigarrow \langle co_1, co_2 \rangle}$$

S-TOPARR
$$\frac{}{\top <: \top \to \top \rightsquigarrow \mathsf{top}_\to}$$

S-TOPRCD
$$\frac{}{\top <: \{l : \top\} \rightsquigarrow \mathsf{id}}$$

S-DISTRCD
$$\frac{}{\{l : A\} \,\&\, \{l : B\} <: \{l : A \,\&\, B\} \rightsquigarrow \mathsf{id}}$$

S-DISTARR
$$\frac{}{(A_1 \to A_2) \,\&\, (A_1 \to A_3) <: A_1 \to A_2 \,\&\, A_3 \rightsquigarrow \mathsf{dist}_\to}$$

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

D-TOPL
$$\frac{}{\top * A}$$

D-TOPR
$$\frac{}{A * \top}$$

D-ARR
$$\frac{A_2 * B_2}{A_1 \to A_2 * B_1 \to B_2}$$

D-ANDL
$$\frac{A_1 * B \qquad A_2 * B}{A_1 \,\&\, A_2 * B}$$

D-ANDR
$$\frac{A * B_1 \qquad A * B_2}{A * B_1 \,\&\, B_2}$$

D-RCDEQ
$$\frac{A * B}{\{l : A\} * \{l : B\}}$$

D-RCDNEQ
$$\frac{l_1 \neq l_2}{\{l_1 : A\} * \{l_2 : B\}}$$

D-AX
$$\frac{A *_{ax} B}{A * B}$$

D λ_i^+ Typing Rules, in Full

$\boxed{A *_{ax} B}$ *(Disjointness axioms)*

Dax-sym
$$\frac{B *_{ax} A}{A *_{ax} B}$$

Dax-intArr
$$\frac{}{\mathsf{Int} *_{ax} A_1 \to A_2}$$

Dax-intRcd
$$\frac{}{\mathsf{Int} *_{ax} \{l : A\}}$$

Dax-arrRcd
$$\frac{}{A_1 \to A_2 *_{ax} \{l : B\}}$$

$\boxed{\Gamma \vdash E \Rightarrow A \rightsquigarrow e}$ *(Inference)*

T-top
$$\frac{}{\Gamma \vdash \top \Rightarrow \top \rightsquigarrow \langle \rangle}$$

T-lit
$$\frac{}{\Gamma \vdash i \Rightarrow \mathsf{Int} \rightsquigarrow i}$$

T-var
$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A \rightsquigarrow x}$$

T-app
$$\frac{\Gamma \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e_1 \qquad \Gamma \vdash E_2 \Leftarrow A_1 \rightsquigarrow e_2}{\Gamma \vdash E_1\, E_2 \Rightarrow A_2 \rightsquigarrow e_1\, e_2}$$

T-anno
$$\frac{\Gamma \vdash E \Leftarrow A \rightsquigarrow e}{\Gamma \vdash E : A \Rightarrow A \rightsquigarrow e}$$

T-proj
$$\frac{\Gamma \vdash E \Rightarrow \{l : A\} \rightsquigarrow e}{\Gamma \vdash E.l \Rightarrow A \rightsquigarrow e}$$

T-merge
$$\frac{\Gamma \vdash E_1 \Rightarrow A_1 \rightsquigarrow e_1 \qquad \Gamma \vdash E_2 \Rightarrow A_2 \rightsquigarrow e_2 \qquad A_1 * A_2}{\Gamma \vdash E_1\,,, E_2 \Rightarrow A_1 \,\&\, A_2 \rightsquigarrow \langle e_1, e_2 \rangle}$$

T-rcd
$$\frac{\Gamma \vdash E \Rightarrow A \rightsquigarrow e}{\Gamma \vdash \{l = E\} \Rightarrow \{l : A\} \rightsquigarrow e}$$

$\boxed{\Gamma \vdash E \Leftarrow A \rightsquigarrow e}$ *(Checking)*

T-abs
$$\frac{\Gamma, x : A \vdash E \Leftarrow B \rightsquigarrow e}{\Gamma \vdash \lambda x.\, E \Leftarrow A \to B \rightsquigarrow \lambda x.\, e}$$

T-sub
$$\frac{\Gamma \vdash E \Rightarrow A \rightsquigarrow e \qquad A <: B \rightsquigarrow co}{\Gamma \vdash E \Leftarrow B \rightsquigarrow co\, e}$$

$\boxed{\mathcal{Q} \vdash A \prec: B \rightsquigarrow co}$ *(Algorithmic subtyping)*

A-int
$$\frac{}{[]\, \vdash \mathsf{Int} \prec: \mathsf{Int} \rightsquigarrow \mathsf{id}}$$

A-and
$$\frac{\mathcal{Q} \vdash A \prec: B_1 \rightsquigarrow co_1 \qquad \mathcal{Q} \vdash A \prec: B_2 \rightsquigarrow co_2}{\mathcal{Q} \vdash A \prec: B_1 \,\&\, B_2 \rightsquigarrow [\![\mathcal{Q}]\!]^{\&} \circ \langle co_1, co_2 \rangle}$$

A-arr
$$\frac{\mathcal{Q}, B_1 \vdash A \prec: B_2 \rightsquigarrow co}{\mathcal{Q} \vdash A \prec: B_1 \to B_2 \rightsquigarrow co}$$

A-rcd
$$\frac{\mathcal{Q}, l \vdash A \prec: B \rightsquigarrow co}{\mathcal{Q} \vdash A \prec: \{l : B\} \rightsquigarrow co}$$

A-top
$$\frac{}{\mathcal{Q} \vdash A \prec: \top \rightsquigarrow [\![\mathcal{Q}]\!]^{\top} \circ \mathsf{top}}$$

A-arrInt
$$\frac{[] \vdash A \prec: A_1 \rightsquigarrow co_1 \qquad \mathcal{Q} \vdash A_2 \prec: \mathsf{Int} \rightsquigarrow co_2}{A, \mathcal{Q} \vdash A_1 \rightarrow A_2 \prec: \mathsf{Int} \rightsquigarrow co_1 \rightarrow co_2}$$

A-rcdInt
$$\frac{\mathcal{Q} \vdash A \prec: \mathsf{Int} \rightsquigarrow co}{l, \mathcal{Q} \vdash \{l : A\} \prec: \mathsf{Int} \rightsquigarrow co}$$

A-andInt1
$$\frac{\mathcal{Q} \vdash A_1 \prec: \mathsf{Int} \rightsquigarrow co}{\mathcal{Q} \vdash A_1 \& A_2 \prec: \mathsf{Int} \rightsquigarrow co \circ \pi_1}$$

A-andInt2
$$\frac{\mathcal{Q} \vdash A_2 \prec: \mathsf{Int} \rightsquigarrow co}{\mathcal{Q} \vdash A_1 \& A_2 \prec: \mathsf{Int} \rightsquigarrow co \circ \pi_2}$$

$$\boxed{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}} \qquad\qquad \textit{(Context typing I)}$$

CTyp-empty1
$$\frac{}{[\cdot] : (\Gamma \Rightarrow A) \mapsto (\Gamma \Rightarrow A) \rightsquigarrow [\cdot]}$$

CTyp-appL1
$$\frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1 \rightarrow A_2) \rightsquigarrow \mathcal{D} \qquad \Gamma' \vdash E_2 \Leftarrow A_1 \rightsquigarrow e}{\mathcal{C}\, E_2 : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D}\, e}$$

CTyp-appR1
$$\frac{\Gamma' \vdash E_1 \Rightarrow A_1 \rightarrow A_2 \rightsquigarrow e \qquad \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Leftarrow A_1) \rightsquigarrow \mathcal{D}}{E_1\, \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow e\, \mathcal{D}}$$

CTyp-mergeL1
$$\frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1) \rightsquigarrow \mathcal{D} \qquad \Gamma' \vdash E_2 \Rightarrow A_2 \rightsquigarrow e \qquad A_1 * A_2}{\mathcal{C}\, ,,\, E_2 : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1 \& A_2) \rightsquigarrow \langle \mathcal{D}, e \rangle}$$

CTyp-mergeR1
$$\frac{\Gamma' \vdash E_1 \Rightarrow A_1 \rightsquigarrow e \qquad \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D} \qquad A_1 * A_2}{E_1\, ,,\, \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1 \& A_2) \rightsquigarrow \langle e, \mathcal{D} \rangle}$$

CTyp-rcd1
$$\frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}{\{l = \mathcal{C}\} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}}$$

CTyp-proj1
$$\frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}}{\mathcal{C}.l : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}$$

CTyp-anno1
$$\frac{\mathcal{C} : (\Gamma \Rightarrow B) \mapsto (\Gamma' \Leftarrow A) \rightsquigarrow \mathcal{D}}{\mathcal{C} : A : (\Gamma \Rightarrow B) \mapsto (\Gamma' \Rightarrow A) \rightsquigarrow \mathcal{D}}$$

$$\boxed{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Leftarrow B) \rightsquigarrow \mathcal{D}}$$ *(Context typing II)*

CTYP-EMPTY2

$$\frac{}{[\cdot] : (\Gamma \Leftarrow A) \mapsto (\Gamma \Leftarrow A) \rightsquigarrow [\cdot]}$$

CTYP-ABS2

$$\frac{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma', x : A_1 \Leftarrow A_2) \rightsquigarrow \mathcal{D} \qquad x \notin \Gamma'}{\lambda x. \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Leftarrow A_1 \to A_2) \rightsquigarrow \lambda x. \mathcal{D}}$$

$$\boxed{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}$$ *(Context typing III)*

CTYP-APPL2

$$\frac{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_1 \to A_2) \rightsquigarrow \mathcal{D} \qquad \Gamma' \vdash E_2 \Leftarrow A_1 \rightsquigarrow e}{\mathcal{C}\, E_2 : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D}\, e}$$

CTYP-APPR2

$$\frac{\Gamma' \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e \qquad \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Leftarrow A_1) \rightsquigarrow \mathcal{D}}{E_1\, \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow e\, \mathcal{D}}$$

CTYP-MERGEL2

$$\frac{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_1) \rightsquigarrow \mathcal{D} \qquad \Gamma' \vdash E_2 \Rightarrow A_2 \rightsquigarrow e \qquad A_1 * A_2}{\mathcal{C} \,,, E_2 : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle \mathcal{D}, e \rangle}$$

CTYP-MERGER2

$$\frac{\Gamma' \vdash E_1 \Rightarrow A_1 \rightsquigarrow e \qquad \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D} \qquad A_1 * A_2}{E_1 \,,, \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle e, \mathcal{D} \rangle}$$

CTYP-RCD2

$$\frac{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}{\{l = \mathcal{C}\} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}}$$

CTYP-PROJ2

$$\frac{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}}{\mathcal{C}.l : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}$$

CTYP-ANNO2

$$\frac{\mathcal{C} : (\Gamma \Leftarrow B) \mapsto (\Gamma' \Leftarrow A) \rightsquigarrow \mathcal{D}}{\mathcal{C} : A : (\Gamma \Leftarrow B) \mapsto (\Gamma' \Rightarrow A) \rightsquigarrow \mathcal{D}}$$

$$\boxed{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Leftarrow B) \rightsquigarrow \mathcal{D}}$$ *(Context typing IV)*

CTYP-ABS1

$$\frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma', x : A_1 \Leftarrow A_2) \rightsquigarrow \mathcal{D} \qquad x \notin \Gamma'}{\lambda x. \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Leftarrow A_1 \to A_2) \rightsquigarrow \lambda x. \mathcal{D}}$$

D.1 λ_{co}

$\boxed{co :: \tau_1 \rhd \tau_2}$ *(Coercion typing)*

CT-REFL

$$\overline{\mathsf{id} :: \tau \rhd \tau}$$

CT-TRANS

$$\frac{co_1 :: \tau_2 \rhd \tau_3 \qquad co_2 :: \tau_1 \rhd \tau_2}{co_1 \circ co_2 :: \tau_1 \rhd \tau_3}$$

CT-TOP

$$\overline{\mathsf{top} :: \tau \rhd \langle\rangle}$$

CT-TOPARR

$$\overline{\mathsf{top}_\to :: \langle\rangle \rhd \langle\rangle \to \langle\rangle}$$

CT-ARR

$$\frac{co_1 :: \tau_1' \rhd \tau_1 \qquad co_2 :: \tau_2 \rhd \tau_2'}{co_1 \to co_2 :: \tau_1 \to \tau_2 \rhd \tau_1' \to \tau_2'}$$

CT-PAIR

$$\frac{co_1 :: \tau_1 \rhd \tau_2 \qquad co_2 :: \tau_1 \rhd \tau_3}{\langle co_1, co_2 \rangle :: \tau_1 \rhd \tau_2 \times \tau_3}$$

CT-PROJL

$$\overline{\pi_1 :: \tau_1 \times \tau_2 \rhd \tau_1}$$

CT-PROJR

$$\overline{\pi_2 :: \tau_1 \times \tau_2 \rhd \tau_2}$$

CT-DISTARR

$$\overline{\mathsf{dist}_\to :: (\tau_1 \to \tau_2) \times (\tau_1 \to \tau_3) \rhd \tau_1 \to \tau_2 \times \tau_3}$$

$\boxed{e \longrightarrow e'}$ *(Single-step reduction)*

R-ID

$$\overline{\mathsf{id}\, v \longrightarrow v}$$

R-TRANS

$$\overline{(co_1 \circ co_2)\, v \longrightarrow co_1\,(co_2\, v)}$$

R-TOP

$$\overline{\mathsf{top}\, v \longrightarrow \langle\rangle}$$

R-TOPARR

$$\overline{(\mathsf{top}_\to\, \langle\rangle)\, \langle\rangle \longrightarrow \langle\rangle}$$

R-PAIR

$$\overline{\langle co_1, co_2 \rangle\, v \longrightarrow \langle co_1\, v, co_2\, v \rangle}$$

R-ARR

$$\overline{((co_1 \to co_2)\, v_1)\, v_2 \longrightarrow co_2\,(v_1\,(co_1\, v_2))}$$

R-DISTARR

$$\overline{(\mathsf{dist}_\to\, \langle v_1, v_2 \rangle)\, v_3 \longrightarrow \langle v_1\, v_3, v_2\, v_3 \rangle}$$

R-PROJL

$$\overline{\pi_1\, \langle v_1, v_2 \rangle \longrightarrow v_1}$$

R-PROJR

$$\overline{\pi_2\, \langle v_1, v_2 \rangle \longrightarrow v_2}$$

R-APP

$$\overline{(\lambda x.\, e)\, v \longrightarrow [v/x]e}$$

R-CTXT

$$\frac{e \longrightarrow e'}{\mathcal{E}[e] \longrightarrow \mathcal{E}[e']}$$

E F_i^+ Typing Rules, in Full

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

SWFT-TOP
$$\frac{}{\Delta \vdash \top}$$

SWFT-INT
$$\frac{}{\Delta \vdash \mathsf{Int}}$$

SWFT-VAR
$$\frac{(\alpha * A) \in \Delta}{\Delta \vdash \alpha}$$

SWFT-ARROW
$$\frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \to B}$$

SWFT-ALL
$$\frac{\Delta \vdash A \qquad \Delta, \alpha * A \vdash B}{\Delta \vdash \forall(\alpha * A).\, B}$$

SWFT-AND
$$\frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \,\&\, B}$$

SWFT-RCD
$$\frac{\Delta \vdash A}{\Delta \vdash \{l : A\}}$$

$\boxed{\Delta \vdash \Gamma}$ *(Well-formedness of value context)*

SWFE-EMPTY
$$\frac{}{\Delta \vdash \bullet}$$

SWFE-VAR
$$\frac{\Delta \vdash \Gamma \qquad \Delta \vdash A}{\Delta \vdash \Gamma, x : A}$$

$\boxed{\vdash \Delta}$ *(Well-formedness of type context)*

SWFTE-EMPTY
$$\frac{}{\vdash \bullet}$$

SWFTE-VAR
$$\frac{\vdash \Delta \qquad \Delta \vdash A}{\vdash \Delta, \alpha * A}$$

$\boxed{A <: B \leadsto co}$ *(Declarative subtyping)*

S-REFL
$$\frac{}{A <: A \leadsto \mathsf{id}}$$

S-TRANS
$$\frac{A_2 <: A_3 \leadsto co_1 \qquad A_1 <: A_2 \leadsto co_2}{A_1 <: A_3 \leadsto co_1 \circ co_2}$$

S-TOP
$$\frac{}{A <: \top \leadsto \mathsf{top}}$$

S-RCD
$$\frac{A <: B \leadsto co}{\{l : A\} <: \{l : B\} \leadsto co}$$

S-ARR
$$\frac{B_1 <: A_1 \leadsto co_1 \qquad A_2 <: B_2 \leadsto co_2}{A_1 \to A_2 <: B_1 \to B_2 \leadsto co_1 \to co_2}$$

S-forall
$$\frac{B_1 <: B_2 \rightsquigarrow co \qquad A_2 <: A_1 \rightsquigarrow co'}{\forall (\alpha * A_1).\, B_1 <: \forall (\alpha * A_2).\, B_2 \rightsquigarrow co_\forall}$$

S-andL
$$\frac{}{A_1 \,\&\, A_2 <: A_1 \rightsquigarrow \pi_1}$$

S-andR
$$\frac{}{A_1 \,\&\, A_2 <: A_2 \rightsquigarrow \pi_2}$$

S-and
$$\frac{A_1 <: A_2 \rightsquigarrow co_1 \qquad A_1 <: A_3 \rightsquigarrow co_2}{A_1 <: A_2 \,\&\, A_3 \rightsquigarrow \langle co_1, co_2 \rangle}$$

S-topArr
$$\frac{}{\top <: \top \to \top \rightsquigarrow \mathsf{top}_\to}$$

S-topRcd
$$\frac{}{\top <: \{l : \top\} \rightsquigarrow \mathsf{id}}$$

S-topAll
$$\frac{}{\top <: \forall (\alpha * \top).\, \top \rightsquigarrow \mathsf{top}_\forall}$$

S-distArr
$$\frac{}{(A_1 \to A_2) \,\&\, (A_1 \to A_3) <: A_1 \to A_2 \,\&\, A_3 \rightsquigarrow \mathsf{dist}_\to}$$

S-distRcd
$$\frac{}{\{l : A\} \,\&\, \{l : B\} <: \{l : A \,\&\, B\} \rightsquigarrow \mathsf{id}}$$

S-distAll
$$\frac{}{(\forall (\alpha * A).\, B_1) \,\&\, (\forall (\alpha * A).\, B_2) <: \forall (\alpha * A).\, B_1 \,\&\, B_2 \rightsquigarrow \mathsf{dist}_\forall}$$

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

FD-topL
$$\frac{\rceil A \lceil}{\Delta \vdash A * B}$$

FD-topR
$$\frac{\rceil B \lceil}{\Delta \vdash A * B}$$

FD-arr
$$\frac{\Delta \vdash A_2 * B_2}{\Delta \vdash A_1 \to A_2 * B_1 \to B_2}$$

FD-andL
$$\frac{\Delta \vdash A_1 * B \qquad \Delta \vdash A_2 * B}{\Delta \vdash A_1 \,\&\, A_2 * B}$$

FD-andR
$$\frac{\Delta \vdash A * B_1 \qquad \Delta \vdash A * B_2}{\Delta \vdash A * B_1 \,\&\, B_2}$$

FD-rcdEq
$$\frac{\Delta \vdash A * B}{\Delta \vdash \{l : A\} * \{l : B\}}$$

FD-rcdNeq
$$\frac{l_1 \neq l_2}{\Delta \vdash \{l_1 : A\} * \{l_2 : B\}}$$

FD-tvarL
$$\frac{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash \alpha * B}$$

FD-tvarR
$$\frac{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash B * \alpha}$$

FD-forall
$$\frac{\Delta, \alpha * A_1 \,\&\, A_2 \vdash B_1 * B_2}{\Delta \vdash \forall (\alpha * A_1).\, B_1 * \forall (\alpha * A_2).\, B_2}$$

FD-ax
$$\frac{A *_{ax} B}{\Delta \vdash A * B}$$

$$\boxed{A *_{ax} B} \hspace{5cm} \textit{(Disjointness axioms)}$$

$$\frac{\text{Dax-sym}}{B *_{ax} A}$$
$$\overline{A *_{ax} B}$$

$$\frac{\text{Dax-intArr}}{\text{Int} *_{ax} A_1 \to A_2}$$

$$\frac{\text{Dax-intRcd}}{\text{Int} *_{ax} \{l : A\}}$$

$$\frac{\text{Dax-intAll}}{\text{Int} *_{ax} \forall(\alpha * B_1).B_2}$$

$$\frac{\text{Dax-arrAll}}{A_1 \to A_2 *_{ax} \forall(\alpha * B_1).B_2}$$

$$\frac{\text{Dax-arrRcd}}{A_1 \to A_2 *_{ax} \{l : B\}}$$

$$\frac{\text{Dax-allRcd}}{\forall(\alpha * A_1).A_2 *_{ax} \{l : B\}}$$

$$\boxed{\Delta; \Gamma \vdash E \Rightarrow A \rightsquigarrow e} \hspace{4cm} \textit{(Inference)}$$

$$\frac{\text{FT-top}}{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash \top \Rightarrow \top \rightsquigarrow \langle\rangle}$$

$$\frac{\text{FT-int}}{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash i \Rightarrow \text{Int} \rightsquigarrow i}$$

$$\frac{\text{FT-var}}{\vdash \Delta \qquad \Delta \vdash \Gamma \qquad (x : A) \in \Gamma}{\Delta; \Gamma \vdash x \Rightarrow A \rightsquigarrow x}$$

$$\frac{\text{FT-app}}{\Delta; \Gamma \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e_1 \qquad \Delta; \Gamma \vdash E_2 \Leftarrow A_1 \rightsquigarrow e_2}{\Delta; \Gamma \vdash E_1\, E_2 \Rightarrow A_2 \rightsquigarrow e_1\, e_2}$$

$$\frac{\text{FT-merge}}{\Delta; \Gamma \vdash E_1 \Rightarrow A_1 \rightsquigarrow e_1 \qquad \Delta; \Gamma \vdash E_2 \Rightarrow A_2 \rightsquigarrow e_2 \qquad \Delta \vdash A_1 * A_2}{\Delta; \Gamma \vdash E_1 ,, E_2 \Rightarrow A_1 \,\&\, A_2 \rightsquigarrow \langle e_1, e_2 \rangle}$$

$$\frac{\text{FT-anno}}{\Delta; \Gamma \vdash E \Leftarrow A \rightsquigarrow e}{\Delta; \Gamma \vdash E : A \Rightarrow A \rightsquigarrow e}$$

$$\frac{\text{FT-tabs}}{\Delta, \alpha * A; \Gamma \vdash E \Rightarrow B \rightsquigarrow e \qquad \Delta \vdash A \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash \Lambda(\alpha * A).E \Rightarrow \forall(\alpha * A).B \rightsquigarrow \Lambda\alpha.e}$$

$$\frac{\text{FT-tapp}}{\Delta; \Gamma \vdash E \Rightarrow \forall(\alpha * B).C \rightsquigarrow e \qquad \Delta \vdash A * B}{\Delta; \Gamma \vdash E\, A \Rightarrow [A/\alpha]C \rightsquigarrow e\, |A|}$$

$$\frac{\text{FT-rcd}}{\Delta; \Gamma \vdash E \Rightarrow A \rightsquigarrow e}{\Delta; \Gamma \vdash \{l = E\} \Rightarrow \{l : A\} \rightsquigarrow e}$$

$$\frac{\text{FT-proj}}{\Delta; \Gamma \vdash E \Rightarrow \{l : A\} \rightsquigarrow e}{\Delta; \Gamma \vdash E.l \Rightarrow A \rightsquigarrow e}$$

$$\boxed{\Delta;\Gamma \vdash E \Leftarrow A \rightsquigarrow e} \hspace{4cm} \textit{(Checking)}$$

FT-ABS

$$\frac{\Delta \vdash A \qquad \Delta;\Gamma, x : A \vdash E \Leftarrow B \rightsquigarrow e}{\Delta;\Gamma \vdash \lambda x.\, E \Leftarrow A \rightarrow B \rightsquigarrow \lambda x.\, e}$$

FT-SUB

$$\frac{\Delta;\Gamma \vdash E \Rightarrow B \rightsquigarrow e \qquad B <: A \rightsquigarrow co \qquad \Delta \vdash A}{\Delta;\Gamma \vdash E \Leftarrow A \rightsquigarrow co\, e}$$

$$\boxed{\mathcal{Q} \vdash A \prec: B \rightsquigarrow co} \hspace{4cm} \textit{(Algorithmic subtyping)}$$

A-CONST

$$\frac{}{[\,] \vdash c \prec: c \rightsquigarrow \mathsf{id}}$$

A-AND

$$\frac{\mathcal{Q} \vdash A \prec: B_1 \rightsquigarrow co_1 \qquad \mathcal{Q} \vdash A \prec: B_2 \rightsquigarrow co_2}{\mathcal{Q} \vdash A \prec: B_1 \,\&\, B_2 \rightsquigarrow [\![\mathcal{Q}]\!]^{\&} \circ \langle co_1, co_2 \rangle}$$

A-ARR

$$\frac{\mathcal{Q}, B_1 \vdash A \prec: B_2 \rightsquigarrow co}{\mathcal{Q} \vdash A \prec: B_1 \rightarrow B_2 \rightsquigarrow co}$$

A-RCD

$$\frac{\mathcal{Q}, l \vdash A \prec: B \rightsquigarrow co}{\mathcal{Q} \vdash A \prec: \{l : B\} \rightsquigarrow co}$$

A-TOP

$$\frac{}{\mathcal{Q} \vdash A \prec: \top \rightsquigarrow [\![\mathcal{Q}]\!]^{\top} \circ \mathsf{top}}$$

A-FORALL

$$\frac{\mathcal{Q}, \alpha * B_1 \vdash A \prec: B_2 \rightsquigarrow co}{\mathcal{Q} \vdash A \prec: \forall(\alpha * B_1).\, B_2 \rightsquigarrow co}$$

A-ARRCONST

$$\frac{[\,] \vdash A \prec: A_1 \rightsquigarrow co_1 \qquad \mathcal{Q} \vdash A_2 \prec: c \rightsquigarrow co_2}{A, \mathcal{Q} \vdash A_1 \rightarrow A_2 \prec: c \rightsquigarrow co_1 \rightarrow co_2}$$

A-RCDCONST

$$\frac{\mathcal{Q} \vdash A \prec: c \rightsquigarrow co}{l, \mathcal{Q} \vdash \{l : A\} \prec: c \rightsquigarrow co}$$

A-ANDCONST1

$$\frac{\mathcal{Q} \vdash A_1 \prec: c \rightsquigarrow co}{\mathcal{Q} \vdash A_1 \,\&\, A_2 \prec: c \rightsquigarrow co \circ \pi_1}$$

A-ANDCONST2

$$\frac{\mathcal{Q} \vdash A_2 \prec: c \rightsquigarrow co}{\mathcal{Q} \vdash A_1 \,\&\, A_2 \prec: c \rightsquigarrow co \circ \pi_2}$$

A-ALLCONST

$$\frac{[\,] \vdash A \prec: A_1 \qquad \mathcal{Q} \vdash A_2 \prec: c \rightsquigarrow co}{(\alpha * A, \mathcal{Q}) \vdash \forall(\alpha * A_1).\, A_2 \prec: c \rightsquigarrow co_\forall}$$

$$\boxed{\mathcal{C} : (\Delta;\Gamma \Rightarrow A) \mapsto (\Delta';\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}} \hspace{3cm} \textit{(Context typing I)}$$

FCTYP-EMPTY1

$$\frac{}{[\cdot] : (\Delta;\Gamma \Rightarrow A) \mapsto (\Delta;\Gamma \Rightarrow A) \rightsquigarrow [\cdot]}$$

FCTYP-APPL1

$$\frac{\mathcal{C} : (\Delta;\Gamma \Rightarrow A) \mapsto (\Delta';\Gamma' \Rightarrow A_1 \rightarrow A_2) \rightsquigarrow \mathcal{D} \qquad \Delta';\Gamma' \vdash E_2 \Leftarrow A_1 \rightsquigarrow e}{\mathcal{C}\, E_2 : (\Delta;\Gamma \Rightarrow A) \mapsto (\Delta';\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D}\, e}$$

FCTyp-appR1
$$\frac{\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Leftarrow A_1) \rightsquigarrow \mathcal{D} \qquad \Delta'; \Gamma' \vdash E_1 \Rightarrow A_1 \rightarrow A_2 \rightsquigarrow e}{E_1 \, \mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_2) \rightsquigarrow e \, \mathcal{D}}$$

FCTyp-mergeL1
$$\frac{\begin{array}{c} \mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_1) \rightsquigarrow \mathcal{D} \\ \Delta'; \Gamma' \vdash E_2 \Rightarrow A_2 \rightsquigarrow e \qquad \Delta' \vdash A_1 * A_2 \end{array}}{\mathcal{C} \, ,, E_2 : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle \mathcal{D}, e \rangle}$$

FCTyp-mergeR1
$$\frac{\begin{array}{c} \mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D} \\ \Delta'; \Gamma' \vdash E_1 \Rightarrow A_1 \rightsquigarrow e \qquad \Delta' \vdash A_1 * A_2 \end{array}}{E_1 \, ,, \mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle e, \mathcal{D} \rangle}$$

FCTyp-rcd1
$$\frac{\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}{\{l = \mathcal{C}\} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}}$$

FCTyp-proj1
$$\frac{\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}}{\mathcal{C}.l : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}$$

FCTyp-anno1
$$\frac{\mathcal{C} : (\Delta; \Gamma \Rightarrow B) \mapsto (\Delta'; \Gamma' \Leftarrow A) \rightsquigarrow \mathcal{D}}{\mathcal{C} : A : (\Delta; \Gamma \Rightarrow B) \mapsto (\Delta'; \Gamma' \Rightarrow A) \rightsquigarrow \mathcal{D}}$$

FCTyp-tabs1
$$\frac{\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta', \alpha * B; \Gamma' \Rightarrow B') \rightsquigarrow \mathcal{D} \qquad \Delta' \vdash B \qquad \Delta' \vdash \Gamma'}{\Lambda(\alpha * B). \mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow \forall(\alpha * B). B') \rightsquigarrow \Lambda\alpha. \mathcal{D}}$$

FCTyp-tapp1
$$\frac{\begin{array}{c} \mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow \forall(\alpha * A_1). A_2) \rightsquigarrow \mathcal{D} \\ \Delta' \vdash B \qquad \Delta' \vdash B * A_1 \end{array}}{\mathcal{C} \, B : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow [B/\alpha]A_2) \rightsquigarrow \mathcal{D} \, |B|}$$

$$\boxed{\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Leftarrow B) \rightsquigarrow \mathcal{D}}$$ *(Context typing II)*

FCTYP-EMPTY2

$$[\cdot] : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta; \Gamma \Leftarrow A) \rightsquigarrow [\cdot]$$

FCTYP-ABS2

$$\frac{\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma', x : A_1 \Leftarrow A_2) \rightsquigarrow \mathcal{D} \qquad \Delta' \vdash A_1}{\lambda x.\,\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Leftarrow A_1 \to A_2) \rightsquigarrow \lambda x.\,\mathcal{D}}$$

$$\boxed{\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}$$ *(Context typing III)*

FCTYP-APPL2

$$\frac{\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_1 \to A_2) \rightsquigarrow \mathcal{D} \qquad \Delta'; \Gamma' \vdash E_2 \Leftarrow A_1 \rightsquigarrow e}{\mathcal{C}\, E_2 : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D}\, e}$$

FCTYP-APPR2

$$\frac{\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Leftarrow A_1) \rightsquigarrow \mathcal{D} \qquad \Delta'; \Gamma' \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e}{E_1\, \mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_2) \rightsquigarrow e\, \mathcal{D}}$$

FCTYP-MERGEL2

$$\frac{\begin{array}{c} \mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_1) \rightsquigarrow \mathcal{D} \\ \Delta'; \Gamma' \vdash E_2 \Rightarrow A_2 \rightsquigarrow e \qquad \Delta' \vdash A_1 * A_2 \end{array}}{\mathcal{C}\, ,, E_2 : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle \mathcal{D}, e \rangle}$$

FCTYP-MERGER2

$$\frac{\begin{array}{c} \mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D} \\ \Delta'; \Gamma' \vdash E_1 \Rightarrow A_1 \rightsquigarrow e \qquad \Delta' \vdash A_1 * A_2 \end{array}}{E_1 \,,, \mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle e, \mathcal{D} \rangle}$$

FCTYP-RCD2

$$\frac{\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}{\{l = \mathcal{C}\} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}}$$

FCTYP-PROJ2

$$\frac{\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}}{\mathcal{C}.l : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}$$

FCTYP-ANNO2

$$\frac{\mathcal{C} : (\Delta; \Gamma \Leftarrow B) \mapsto (\Delta'; \Gamma' \Leftarrow A) \rightsquigarrow \mathcal{D}}{\mathcal{C} : A : (\Delta; \Gamma \Leftarrow B) \mapsto (\Delta'; \Gamma' \Rightarrow A) \rightsquigarrow \mathcal{D}}$$

FCTYP-TABS2

$$\frac{\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta', \alpha * B; \Gamma' \Rightarrow B') \rightsquigarrow \mathcal{D} \qquad \Delta' \vdash B \qquad \Delta' \vdash \Gamma'}{\Lambda(\alpha * B).\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow \forall(\alpha * B).B') \rightsquigarrow \Lambda\alpha.\mathcal{D}}$$

FCTYP-TAPP2

$$\frac{\begin{array}{c} \mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow \forall(\alpha * A_1).A_2) \rightsquigarrow \mathcal{D} \\ \Delta' \vdash B \qquad \Delta' \vdash B * A_1 \end{array}}{\mathcal{C}\,B : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow [B/\alpha]A_2) \rightsquigarrow \mathcal{D}\,|B|}$$

$$\boxed{\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Leftarrow B) \rightsquigarrow \mathcal{D}} \qquad\qquad\qquad \textit{(Context typing IV)}$$

FCTYP-ABS1

$$\frac{\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma', x : A_1 \Leftarrow A_2) \rightsquigarrow \mathcal{D} \qquad \Delta' \vdash A_1}{\lambda x.\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Leftarrow A_1 \to A_2) \rightsquigarrow \lambda x.\mathcal{D}}$$

E.1 F_{co}

$$\boxed{\Phi \vdash \Psi} \qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{(Well-formedness of value context)}$$

WFE-EMPTY

$$\frac{}{\Phi \vdash \bullet}$$

WFE-VAR

$$\frac{\Phi \vdash \tau \qquad \Phi \vdash \Psi}{\Phi \vdash \Psi, x : \tau}$$

$$\boxed{\Phi \vdash \tau} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{(Well-formedness of types)}$$

WFT-INT

$$\frac{}{\Phi \vdash \mathsf{Int}}$$

WFT-VAR

$$\frac{\alpha \in \Phi}{\Phi \vdash \alpha}$$

WFT-ARROW

$$\frac{\Phi \vdash \tau_1 \qquad \Phi \vdash \tau_2}{\Phi \vdash \tau_1 \to \tau_2}$$

WFT-PROD

$$\frac{\Phi \vdash \tau_1 \qquad \Phi \vdash \tau_2}{\Phi \vdash \tau_1 \times \tau_2}$$

WFT-ALL

$$\frac{\Phi, \alpha \vdash \tau_2}{\Phi \vdash \forall\alpha.\tau_2}$$

$$\boxed{co :: \tau_1 \rhd \tau_2} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{(Coercion typing)}$$

CT-REFL

$$\frac{}{\mathsf{id} :: \tau \rhd \tau}$$

CT-TRANS

$$\frac{co_1 :: \tau_2 \rhd \tau_3 \qquad co_2 :: \tau_1 \rhd \tau_2}{co_1 \circ co_2 :: \tau_1 \rhd \tau_3}$$

CT-TOP

$$\frac{}{\mathsf{top} :: \tau \rhd \langle\rangle}$$

CT-TOPARR

$$\frac{}{\mathsf{top}_{\to} :: \langle\rangle \rhd \langle\rangle \to \langle\rangle}$$

CT-ARR

$$\frac{co_1 :: \tau_1' \rhd \tau_1 \qquad co_2 :: \tau_2 \rhd \tau_2'}{co_1 \to co_2 :: \tau_1 \to \tau_2 \rhd \tau_1' \to \tau_2'}$$

CT-PAIR
$$\frac{co_1 :: \tau_1 \triangleright \tau_2 \qquad co_2 :: \tau_1 \triangleright \tau_3}{\langle co_1, co_2 \rangle :: \tau_1 \triangleright \tau_2 \times \tau_3}$$

CT-PROJL
$$\frac{}{\pi_1 :: \tau_1 \times \tau_2 \triangleright \tau_1}$$

CT-PROJR
$$\frac{}{\pi_2 :: \tau_1 \times \tau_2 \triangleright \tau_2}$$

CT-FORALL
$$\frac{co :: \tau_1 \triangleright \tau_2}{co_\forall :: \forall \alpha. \tau_1 \triangleright \forall \alpha. \tau_2}$$

CT-DISTARR
$$\frac{}{\mathsf{dist}_\rightarrow :: (\tau_1 \rightarrow \tau_2) \times (\tau_1 \rightarrow \tau_3) \triangleright \tau_1 \rightarrow \tau_2 \times \tau_3}$$

CT-TOPALL
$$\frac{}{\mathsf{top}_\forall :: \langle \rangle \triangleright \forall \alpha. \langle \rangle}$$

CT-DISTALL
$$\frac{}{\mathsf{dist}_\forall :: (\forall \alpha. \tau_1) \times (\forall \alpha. \tau_2) \triangleright \forall \alpha. \tau_1 \times \tau_2}$$

$\boxed{\Phi; \Psi \vdash e : \tau}$ *(Static semantics)*

FT-UNIT
$$\frac{\Phi \vdash \Psi}{\Phi; \Psi \vdash \langle \rangle : \langle \rangle}$$

FT-INT
$$\frac{\Phi \vdash \Psi}{\Phi; \Psi \vdash i : \mathsf{Int}}$$

FT-VAR
$$\frac{\Phi \vdash \Psi \qquad (x : \tau) \in \Psi}{\Phi; \Psi \vdash x : \tau}$$

FT-ABS
$$\frac{\Phi; \Psi, x : \tau_1 \vdash e : \tau_2 \qquad \Phi \vdash \tau_1}{\Phi; \Psi \vdash \lambda x. e : \tau_1 \rightarrow \tau_2}$$

FT-APP
$$\frac{\Phi; \Psi \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Phi; \Psi \vdash e_2 : \tau_1}{\Phi; \Psi \vdash e_1 \, e_2 : \tau_2}$$

FT-TABS
$$\frac{\Phi, \alpha; \Psi \vdash e : \tau \qquad \Phi \vdash \Psi}{\Phi; \Psi \vdash \Lambda \alpha. e : \forall \alpha. \tau}$$

FT-TAPP
$$\frac{\Phi; \Psi \vdash e : \forall \alpha. \tau' \qquad \Phi \vdash \tau}{\Phi; \Psi \vdash e \, \tau : [\tau/\alpha]\tau'}$$

FT-PAIR
$$\frac{\Phi; \Psi \vdash e_1 : \tau_1 \qquad \Phi; \Psi \vdash e_2 : \tau_2}{\Phi; \Psi \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

FT-CAPP
$$\frac{\Phi; \Psi \vdash e : \tau_1 \qquad co :: \tau_1 \triangleright \tau_2 \qquad \Phi \vdash \tau_2}{\Phi; \Psi \vdash co \, e : \tau_2}$$

$\boxed{e \longrightarrow e'}$ *(Single-step reduction)*

R-ID
$$\frac{}{\mathsf{id} \, v \longrightarrow v}$$

R-TRANS
$$\frac{}{(co_1 \circ co_2) \, v \longrightarrow co_1 \, (co_2 \, v)}$$

R-TOP
$$\frac{}{\mathsf{top} \, v \longrightarrow \langle \rangle}$$

R-TOPARR
$$\frac{}{(\mathsf{top}_\rightarrow \langle \rangle) \, \langle \rangle \longrightarrow \langle \rangle}$$

R-PAIR
$$\frac{}{\langle co_1, co_2 \rangle \, v \longrightarrow \langle co_1 \, v, co_2 \, v \rangle}$$

R-ARR
$$\frac{}{((co_1 \rightarrow co_2) \, v_1) \, v_2 \longrightarrow co_2 \, (v_1 \, (co_1 \, v_2))}$$

R-DISTARR
$$\frac{}{(\mathsf{dist}_\rightarrow \langle v_1, v_2 \rangle) \, v_3 \longrightarrow \langle v_1 \, v_3, v_2 \, v_3 \rangle}$$

R-PROJL
$$\frac{}{\pi_1 \langle v_1, v_2 \rangle \longrightarrow v_1}$$

R-PROJR
$$\frac{}{\pi_2 \langle v_1, v_2 \rangle \longrightarrow v_2}$$

R-FORALL

$$(co_\forall \, v) \, \tau \longrightarrow co \, (v \, \tau)$$

R-TOPALL

$$(\mathsf{top}_\forall \, \langle \rangle) \, \tau \longrightarrow \langle \rangle$$

R-DISTALL

$$(\mathsf{dist}_\forall \, \langle v_1, v_2 \rangle) \, \tau \longrightarrow \langle v_1 \, \tau, v_2 \, \tau \rangle$$

R-TAPP

$$(\Lambda \alpha. \, e) \, \tau \longrightarrow [\tau/\alpha]e$$

R-APP

$$(\lambda x. \, e) \, v \longrightarrow [v/x]e$$

R-CTXT

$$\frac{e \longrightarrow e'}{\mathcal{E}[e] \longrightarrow \mathcal{E}[e']}$$

F SEDEL TYPING RULES, IN FULL

$\boxed{\Delta \vdash \mathcal{A}}$ *(Well-formedness of types)*

$$
\text{WF-top}\;\frac{}{\Delta \vdash \top}
\qquad
\text{WF-int}\;\frac{}{\Delta \vdash \mathsf{Int}}
\qquad
\text{WF-arr}\;\frac{\Delta \vdash \mathcal{A} \qquad \Delta \vdash \mathcal{B}}{\Delta \vdash \mathcal{A} \to \mathcal{B}}
\qquad
\text{WF-rcd}\;\frac{\Delta \vdash \mathcal{A}}{\Delta \vdash \{l : \mathcal{A}\}}
\qquad
\text{WF-var}\;\frac{(\alpha * \mathcal{A}) \in \Delta}{\Delta \vdash \alpha}
$$

$$
\text{WF-and}\;\frac{\Delta \vdash \mathcal{A} \qquad \Delta \vdash \mathcal{B}}{\Delta \vdash \mathcal{A} \,\&\, \mathcal{B}}
\qquad
\text{WF-forall}\;\frac{\Delta \vdash \mathcal{A} \qquad \Delta, \alpha * \mathcal{A} \vdash \mathcal{B}}{\Delta \vdash \forall(\alpha * \mathcal{A}).\,\mathcal{B}}
\qquad
\text{WF-trait}\;\frac{\Delta \vdash \mathcal{A} \qquad \Delta \vdash \mathcal{B}}{\Delta \vdash \mathbf{Trait}\,[\mathcal{A}, \mathcal{B}]}
$$

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

$$
\text{TS-refl}\;\frac{}{\mathcal{A} <: \mathcal{A}}
\qquad
\text{TS-trans}\;\frac{\mathcal{A}_2 <: \mathcal{A}_3 \qquad \mathcal{A}_1 <: \mathcal{A}_2}{\mathcal{A}_1 <: \mathcal{A}_3}
\qquad
\text{TS-top}\;\frac{}{\mathcal{A} <: \top}
\qquad
\text{TS-rcd}\;\frac{\mathcal{A} <: \mathcal{B}}{\{l : \mathcal{A}\} <: \{l : \mathcal{B}\}}
$$

$$
\text{TS-arr}\;\frac{\mathcal{B}_1 <: \mathcal{A}_1 \qquad \mathcal{A}_2 <: \mathcal{B}_2}{\mathcal{A}_1 \to \mathcal{A}_2 <: \mathcal{B}_1 \to \mathcal{B}_2}
\qquad
\text{TS-andL}\;\frac{}{\mathcal{A}_1 \,\&\, \mathcal{A}_2 <: \mathcal{A}_1}
\qquad
\text{TS-andR}\;\frac{}{\mathcal{A}_1 \,\&\, \mathcal{A}_2 <: \mathcal{A}_2}
$$

$$
\text{TS-and}\;\frac{\mathcal{A}_1 <: \mathcal{A}_2 \qquad \mathcal{A}_1 <: \mathcal{A}_3}{\mathcal{A}_1 <: \mathcal{A}_2 \,\&\, \mathcal{A}_3}
\qquad
\text{TS-distArr}\;\frac{}{(\mathcal{A}_1 \to \mathcal{A}_2) \,\&\, (\mathcal{A}_1 \to \mathcal{A}_3) <: \mathcal{A}_1 \to \mathcal{A}_2 \,\&\, \mathcal{A}_3}
$$

$$
\text{TS-topArr}\;\frac{}{\top <: \top \to \top}
\qquad
\text{TS-distRcd}\;\frac{}{\{l : \mathcal{A}\} \,\&\, \{l : \mathcal{B}\} <: \{l : \mathcal{A} \,\&\, \mathcal{B}\}}
\qquad
\text{TS-topRcd}\;\frac{}{\top <: \{l : \top\}}
$$

$$
\text{TS-trait}\;\frac{\mathcal{B}_1 <: \mathcal{A}_1 \qquad \mathcal{A}_2 <: \mathcal{B}_2}{\mathbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] <: \mathbf{Trait}\,[\mathcal{B}_1, \mathcal{B}_2]}
$$

$\boxed{\Delta \vdash \mathcal{A} * \mathcal{B}}$ *(Disjointness)*

SD-topL

$$\overline{\Delta \vdash \top * \mathcal{A}}$$

SD-topR

$$\overline{\Delta \vdash \mathcal{A} * \top}$$

SD-arr

$$\frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \mathcal{A}_1 \to \mathcal{A}_2 * \mathcal{B}_1 \to \mathcal{B}_2}$$

SD-andL

$$\frac{\Delta \vdash \mathcal{A}_1 * \mathcal{B} \qquad \Delta \vdash \mathcal{A}_2 * \mathcal{B}}{\Delta \vdash \mathcal{A}_1 \,\&\, \mathcal{A}_2 * \mathcal{B}}$$

SD-andR

$$\frac{\Delta \vdash \mathcal{A} * \mathcal{B}_1 \qquad \Delta \vdash \mathcal{A} * \mathcal{B}_2}{\Delta \vdash \mathcal{A} * \mathcal{B}_1 \,\&\, \mathcal{B}_2}$$

SD-rcdEq

$$\frac{\Delta \vdash \mathcal{A} * \mathcal{B}}{\Delta \vdash \{l : \mathcal{A}\} * \{l : \mathcal{B}\}}$$

SD-rcdNeq

$$\frac{l_1 \neq l_2}{\Delta \vdash \{l_1 : \mathcal{A}\} * \{l_2 : \mathcal{B}\}}$$

SD-tvarL

$$\frac{(\alpha * \mathcal{A}) \in \Delta \qquad \mathcal{A} <: \mathcal{B}}{\Delta \vdash \alpha * \mathcal{B}}$$

SD-tvarR

$$\frac{(\alpha * \mathcal{A}) \in \Delta \qquad \mathcal{A} <: \mathcal{B}}{\Delta \vdash \mathcal{B} * \alpha}$$

SD-forall

$$\frac{\Delta, \alpha * \mathcal{A}_1 \,\&\, \mathcal{A}_2 \vdash \mathcal{B}_1 * \mathcal{B}_2}{\Delta \vdash \forall (\alpha * \mathcal{A}_1).\, \mathcal{B}_1 * \forall (\alpha * \mathcal{A}_2).\, \mathcal{B}_2}$$

SD-trait

$$\frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \mathbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] * \mathbf{Trait}\,[\mathcal{B}_1, \mathcal{B}_2]}$$

SD-traitArr1

$$\frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \mathbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] * \mathcal{B}_1 \to \mathcal{B}_2}$$

SD-traitArr2

$$\frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \mathcal{A}_1 \to \mathcal{A}_2 * \mathbf{Trait}\,[\mathcal{B}_1, \mathcal{B}_2]}$$

SD-ax

$$\frac{\mathcal{A} *_{ax} \mathcal{B}}{\Delta \vdash \mathcal{A} * \mathcal{B}}$$

$\boxed{\mathcal{A} *_{ax} \mathcal{B}}$ *(Disjointness axiom)*

SDax-sym

$$\frac{\mathcal{B} *_{ax} \mathcal{A}}{\mathcal{A} *_{ax} \mathcal{B}}$$

SDax-intArr

$$\overline{\mathsf{Int} *_{ax} \mathcal{A}_1 \to \mathcal{A}_2}$$

SDax-intRcd

$$\overline{\mathsf{Int} *_{ax} \{l : \mathcal{A}\}}$$

SDax-intAll

$$\overline{\mathsf{Int} *_{ax} \forall (\alpha * \mathcal{B}_1).\, \mathcal{B}_2}$$

SDax-arrAll

$$\overline{\mathcal{A}_1 \to \mathcal{A}_2 *_{ax} \forall (\alpha * \mathcal{B}_1).\, \mathcal{B}_2}$$

SDax-arrRcd

$$\overline{\mathcal{A}_1 \to \mathcal{A}_2 *_{ax} \{l : \mathcal{B}\}}$$

SDax-allRcd

$$\overline{\forall (\alpha * \mathcal{A}_1).\, \mathcal{A}_2 *_{ax} \{l : \mathcal{B}\}}$$

SDax-intTrait

$$\overline{\mathsf{Int} *_{ax} \mathbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2]}$$

SDax-traitAll

$$\overline{\mathbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] *_{ax} \forall (\alpha * \mathcal{B}_1).\, \mathcal{B}_2}$$

SDax-traitRcd

$$\overline{\mathbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] *_{ax} \{l : \mathcal{B}\}}$$

$$\boxed{\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \mathcal{A} \rightsquigarrow E} \qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{(Inference)}$$

ST-top
$$\frac{}{\Delta; \Gamma \vdash \top \Rightarrow \top \rightsquigarrow \top}$$

ST-int
$$\frac{}{\Delta; \Gamma \vdash i \Rightarrow \mathsf{Int} \rightsquigarrow i}$$

ST-var
$$\frac{(x : \mathcal{A}) \in \Delta}{\Delta; \Gamma \vdash x \Rightarrow \mathcal{A} \rightsquigarrow x}$$

ST-app
$$\frac{\Delta; \Gamma \vdash \mathcal{T}_1 \Rightarrow \mathcal{A}_1 \rightarrow \mathcal{A}_2 \rightsquigarrow E_1 \qquad \Delta; \Gamma \vdash \mathcal{T}_2 \Leftarrow \mathcal{A}_1 \rightsquigarrow E_2}{\Delta; \Gamma \vdash \mathcal{T}_1 \, \mathcal{T}_2 \Rightarrow \mathcal{A}_2 \rightsquigarrow E_1 \, E_2}$$

ST-merge
$$\frac{\Delta; \Gamma \vdash \mathcal{T}_1 \Rightarrow \mathcal{A} \rightsquigarrow E_1 \qquad \Delta; \Gamma \vdash \mathcal{T}_2 \Rightarrow \mathcal{B} \rightsquigarrow E_2 \qquad \Delta \vdash \mathcal{A} * \mathcal{B}}{\Delta; \Gamma \vdash \mathcal{T}_1 \,,, \mathcal{T}_2 \Rightarrow \mathcal{A} \, \& \, \mathcal{B} \rightsquigarrow E_1 \,,, E_2}$$

ST-anno
$$\frac{\Delta; \Gamma \vdash \mathcal{T} \Leftarrow \mathcal{A} \rightsquigarrow E}{\Delta; \Gamma \vdash \mathcal{T} : \mathcal{A} \Rightarrow \mathcal{A} \rightsquigarrow E : |\mathcal{A}|}$$

ST-tabs
$$\frac{\Delta \vdash \mathcal{A} \qquad \Delta, \alpha * \mathcal{A}; \Gamma \vdash \mathcal{T} \Rightarrow \mathcal{B} \rightsquigarrow E}{\Delta; \Gamma \vdash \Lambda(\alpha * \mathcal{A}). \, \mathcal{T} \Rightarrow \forall(\alpha * \mathcal{A}). \, \mathcal{B} \rightsquigarrow \Lambda(\alpha * |\mathcal{A}|). \, E}$$

ST-tapp
$$\frac{\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \forall(\alpha * \mathcal{B}_1). \, \mathcal{B}_2 \rightsquigarrow E \qquad \Delta \vdash \mathcal{A} \qquad \Delta \vdash \mathcal{A} * \mathcal{B}_1}{\Delta; \Gamma \vdash \mathcal{T} \, \mathcal{A} \Rightarrow [\mathcal{A}/\alpha]\mathcal{B}_2 \rightsquigarrow E \, |\mathcal{A}|}$$

ST-rcd
$$\frac{\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \mathcal{A} \rightsquigarrow E}{\Delta; \Gamma \vdash \{l = \mathcal{T}\} \Rightarrow \{l : \mathcal{A}\} \rightsquigarrow \{l = E\}}$$

ST-proj
$$\frac{\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \{l : \mathcal{A}\} \rightsquigarrow E}{\Delta; \Gamma \vdash \mathcal{T}.l \Rightarrow \mathcal{A} \rightsquigarrow E.l}$$

ST-trait
$$\frac{\begin{array}{c} \overline{\Delta; \Gamma, \mathsf{self} : \mathcal{B} \vdash \mathcal{T}_i \Rightarrow \mathbf{Trait}\,[\mathcal{B}_i, \mathcal{C}_i] \rightsquigarrow E_i}^{\,i \in 1..n} \\ \Delta; \Gamma, \mathsf{self} : \mathcal{B} \vdash \{\overline{l_j = \mathcal{T}_j'}^{\,j \in 1..m}\} \Rightarrow \mathcal{C} \rightsquigarrow E \\ \overline{\mathcal{B} <: \mathcal{B}_i}^{\,i \in 1..n} \qquad \Delta \vdash \mathcal{C}_1 * .. * \mathcal{C}_n * \mathcal{C} \qquad \mathcal{C}_1 \, \& \, .. \, \& \, \mathcal{C}_n \, \& \, \mathcal{C} <: \mathcal{A} \end{array}}{\begin{array}{c} \Delta; \Gamma \vdash \mathbf{trait}\,[\mathsf{self} : \mathcal{B}]\,\mathbf{inherits}\,\overline{\mathcal{T}_i}^{\,i \in 1..n} \, \{\overline{l_j = \mathcal{T}_j'}^{\,j \in 1..m}\} : \mathcal{A} \Rightarrow \mathbf{Trait}\,[\mathcal{B}, \mathcal{A}] \rightsquigarrow \\ \lambda \mathsf{self} : |\mathcal{B}|. \, (((\overline{(E_i\,\mathsf{self})}^{\,i \in 1..n}) \,,, E) \end{array}}$$

ST-NEW

$$\frac{\overline{\Delta; \Gamma \vdash \mathcal{T}_i \Rightarrow \mathbf{Trait}\,[\mathcal{A}_i, \mathcal{B}_i] \rightsquigarrow E_i}^{\,i \in 1..n}}{\overline{\mathcal{A} <: \mathcal{A}_i}^{\,i \in 1..n} \qquad \Delta \vdash \mathcal{B}_1 * .. * \mathcal{B}_n \qquad \mathcal{B}_1 \& .. \& \mathcal{B}_n <: \mathcal{A}}$$
$$\Delta; \Gamma \vdash \mathbf{new}\,[\mathcal{A}](\,\overline{\mathcal{T}_i}^{\,i \in 1..n}\,) \Rightarrow \mathcal{A} \rightsquigarrow \mathbf{letrec}\,\mathsf{self} : |\mathcal{A}| = \overline{(E_i\,\mathsf{self})}^{\,i \in 1..n}\,\mathbf{in}\,\mathsf{self}$$

ST-FORWARD

$$\frac{\Delta; \Gamma \vdash \mathcal{T}_1 \Rightarrow \mathbf{Trait}\,[\mathcal{A}, \mathcal{B}] \rightsquigarrow E_1 \qquad \Delta; \Gamma \vdash \mathcal{T}_2 \Leftarrow \mathcal{A} \rightsquigarrow E_2}{\Delta; \Gamma \vdash \mathcal{T}_1 \wedge \mathcal{T}_2 \Rightarrow \mathcal{B} \rightsquigarrow E_1\,E_2}$$

$$\boxed{\Delta; \Gamma \vdash \mathcal{T} \Leftarrow \mathcal{A} \rightsquigarrow E} \hspace{4cm} \textit{(Checking)}$$

ST-ABS

$$\frac{\Delta \vdash \mathcal{A} \qquad \Delta; \Gamma, x : \mathcal{A} \vdash \mathcal{T} \Leftarrow \mathcal{B} \rightsquigarrow E}{\Delta; \Gamma \vdash \lambda x.\mathcal{T} \Leftarrow \mathcal{A} \to \mathcal{B} \rightsquigarrow \lambda x.\,E}$$

ST-SUB

$$\frac{\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \mathcal{A} \rightsquigarrow E \qquad \mathcal{A} <: \mathcal{B} \qquad \Delta \vdash \mathcal{B}}{\Delta; \Gamma \vdash \mathcal{T} \Leftarrow \mathcal{B} \rightsquigarrow E}$$