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 27, 2018

Abstract

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, as the software we use daily is getting more and more complex and hard 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 help programmers write 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. Several reasons have been raised for the lack of satisfactory solutions, but one that is constantly pointed out is the inadequacy of existing programming language features 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 object-oriented 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 relation with records, universal quantification and coercions. To address the second problem, this thesis proposes a powerful proof method to establish coherence. Thus 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 allows 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 the Coq proof assistant 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, family polymorphism—by a single lightweight mechanism, thus providing new insights into software modularity and extensibility.

Declaration

I declare that this thesis represents my own work, except where due acknowledgement 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 27, 2018

Acknowledgments

First and foremost, I would like to thank my PhD supervisor Dr. Bruno C. d. S. Oliveira for his continued support and mentorship throughout my studies. I would also like to thank my co-supervisor 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. Secondly, I would like to express my gratitude to Prof. Tom Schrijvers from KU Leuven. I enjoyed very much our collaboration on a topic of mutual interest, which lead to one of the key publications for this PhD dissertation. I learned a lot from you! During the years I spent at HKU, I had the opportunity to collaborate with a number of excellent fellow researchers, to name a few, Tomas Tauber, Zhiyuan Shi, Weixin Zhang, Huang Li, Yanpeng Yang and Ningning Xie. 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

1 Introduction 1

1.1 Motivation . . . 1

1.1.1 First-Class Classes . . . 3

1.1.2 (First-Class) Mixins and Traits . . . 3

1.1.3 Family Polymorphism and Nested Composition . . . 5

1.2 Our Proposed Solution . . . 5

1.2.1 Disjoint Intersection Types . . . 6

1.3 Contributions . . . 7

1.4 Structure of the Thesis . . . 10

2 Background 13

2.1 Intersection Types . . . 13

2.1.1 The Merge Operator . . . 15

2.1.2 (In)Coherence . . . 17

2.1.3 Disjoint Intersection Types . . . 18

2.1.4 Disjoint Polymorphism . . . 20

2.2 Mixins and Traits . . . 21

2.3 Family Polymorphism and Nested Composition . . . 23

2.4 Functional Object Encodings . . . 26

2.5 Program Equivalence and Logical Relations . . . 28

I Type Systems 31

3 Semantics of the λ_i^+ Calculus 33

3.1 Introduction . . . 33

3.2 λ_i^+ by Examples . . . 34

3.2.1 The expression problem, λ_i^+ Style . . . 34

3.3 Syntax and Semantics of λ_i^+ . . . 37

3.3.1 Syntax . . . 38

3.3.2 Declarative Subtyping . . . 39
3.3.3 Typing of λ_i^+ . . . 40
3.4 Syntax and Semantics of λ_{co} . . . 41
3.4.1 Explicit Coercions and Coercive Subtyping . . . 43
3.4.2 Typing of λ_{co} . . . 43
3.4.3 Dynamic Semantics . . . 44
3.4.4 Elaboration Semantics . . . 44
3.5 Comparison with λ_i . . . 45
3.6 Algorithmic Subtyping . . . 46
3.6.1 The Subtyping Algorithm . . . 47
3.6.2 Correctness of the Algorithm . . . 49

4 Semantics of the F_i^+ Calculus 53
4.1 Motivation . . . 53
4.2 Syntax and Semantics . . . 55
4.3 Disjointness . . . 59
4.4 Elaboration and Type Safety . . . 61

II Coherence 67

5 Coherence for λ_i^+ 69
5.1 The Intuition . . . 69
5.2 In Search of Coherence . . . 70
5.2.1 Expression Contexts and Contextual Equivalence. . . . 71
5.2.2 λ_i^+ Contexts and Refined Contextual Equivalence. . . . 72
5.3 The Canonicity Relation, Formally Defined . . . 74
5.4 Establishing Coherence . . . 77
5.5 Some Interesting Corollaries . . . 78

6 Coherence for F_i^+ 81
6.1 The Challenges . . . 81
6.2 Review of the Parametric Logical Relation . . . 82
6.3 Impredicativity and Disjointness at Odds . . . 83
6.4 Predicative Logical Relation . . . 85
6.5 Establishing Coherence . . . 86

III Applications 91

7 First-Class Traits 93
7.1 Motivation: First-Class Classes and Dynamic Inheritance 93
7.2 Overview 95
7.2.1 First-Class Classes in JavaScript 95
7.2.2 A Glance at Typed First-Class Traits in SEDEL 99
7.3 Typed First-Class Traits 103
7.3.1 Traits in SEDEL 103
7.3.2 Two Roles of Traits in SEDEL 104
7.3.3 Trait Types and Trait Requirements 105
7.3.4 Traits with Parameters and First-Class Traits 106
7.3.5 Detecting and Resolving Conflicts in Trait Composition 107
7.3.6 Disjoint Polymorphism and Dynamic Composition 109
7.4 Formalizing Typed First-Class Traits 109
7.4.1 Syntax 109
7.4.2 Semantics 111
7.4.3 Type Soundness and Coherence 116

8 Case Study: Modularizing Language Components 119
8.1 Object Algebras and Extensible Visitors in SEDEL 119
8.2 Dynamic Object Algebra Composition Support 122
8.3 Case Study Overview 123
8.4 Evaluation 124

9 Related Work 127
9.1 Coherence 127
9.1.1 Normalization-based Approach 127
9.1.2 Context-based Approach 128
9.2 BCD Subtyping and Decidability 128
9.3 Intersection Types and the Merge Operator 129
9.4 Intersection Types and Polymorphism 130
9.5 Intersection Types and Multiple Inheritance 131
9.6 Row Polymorphism and Extensible Records 132
9.7 Typed First-Class Classes/Mixins/Traits 133
9.8 Mixin-Based Inheritance 134
9.9 Trait-Based Inheritance 134

9.10 Family Polymorphism 135
9.11 Languages with More Advanced Forms of Inheritance 136
9.12 Module Systems 137

10 Future Work 139
10.1 On Categorical Semantics 139
10.1.1 Properties of Intersection Types. 139
10.1.2 Connecting with Disjointness. 141
10.1.3 Interpretation of Intersection Types. 142
10.1.4 Interpretation of Disjoint Intersection Types. 143
10.1.5 Coherence, from the categorical perspective? 144
10.2 On Implicit Polymorphism 144
10.2.1 Declarative Subtyping. 144
10.2.2 Disjointness. 145
10.2.3 Declarative Typing. 146
10.2.4 Algorithmic System. 147
10.3 Disjoint Polymorphism vs. Row Polymorphism 147
10.4 Recursive Types 149
10.5 Other Extensions 151
10.5.1 Union Types. 151
10.5.2 Nominal Typing. 151
10.5.3 Mutable State. 152

11 Conclusion 153

A Proofs about SEDEL 155

B λ_i^+ Typing Rules, in Full 163
B.1 λ_{co} 167

C F_i^+ Typing Rules, in Full 169
C.1 F_{co} 175

D SEDEL Typing Rules, in Full 177

Bibliography 181

List of Figures

2.1 Type system of λ_i . . . 19
2.2 Multiple inheritance and mixins . . . 22
2.3 Traits and conflicts . . . 23
2.4 The expression problem, Scandinavian Style . . . 25

3.1 Summary of the relationships between language components . . . 38
3.2 Syntax of λ_i^+ . . . 38
3.3 Declarative subtyping of F_i . . . 39
3.4 Bidirectional type system of λ_i^+ . . . 41
3.5 Disjointness . . . 42
3.6 λ_{co} syntax . . . 42
3.7 Coercion typing . . . 44
3.8 Dynamic semantics of λ_{co} . . . 45
3.9 Algorithmic subtyping of λ_i^+ . . . 47
3.10 Example derivation . . . 49
3.11 Meta-functions of coercions . . . 49

4.1 Syntax of F_i^+ . . . 55
4.2 Well-formedness of types and contexts . . . 56
4.3 Declarative subtyping of F_i^+ . . . 57
4.4 Bidirectional type system of F_i^+ . . . 58
4.5 Disjointness of F_i^+ . . . 59
4.6 Syntax of F_{co} . . . 61
4.7 Typing rules of F_{co} . . . 63
4.8 Dynamic semantics of F_{co} . . . 64
4.9 Algorithmic subtyping of F_i^+ . . . 65

5.1 Expression contexts of λ_{co} and λ_i^+ . . . 71
5.2 λ_i^+ context typing (excerpt) . . . 73
5.3 ☞ The canonicity relation for λ_i^+ . . . 74

6.1 A logical relation for System F . . . 82
6.2 ☞ The canonicity relation for F_i^+ . . . 86
6.3 Expression contexts of F_{co} and F_i^+ . . . 87
6.4 F_i^+ context typing (excerpt) . . . 88

7.1 SEDEL core syntax and syntactic abbreviations . . . 110
7.2 Well-formedness and subtyping of SEDEL . . . 111
7.3 Disjointness rules of SEDEL . . . 112
7.4 Typing rules of SEDEL . . . 114

8.1 Mini-JS expressions, values, and types . . . 123

List of Tables

3.1 Correspondence between coercions and terms 43

4.1 Correspondence between coercions and terms, extended 62

8.1 Overview of the languages assembled 124

8.2 SLOC statistics: SEDEL implementation vs. vanilla AST implementation . 125

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 hard 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 help programmers write 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 current mainstream languages 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 fifth requirement. The last two requirements were proposed by Nystrom et al. [2006]. Scalability (6) is often but not necessarily satisfied by separate compilation; it is important for extending large software. Non-destructive extension (7) 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 prior 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 and 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 is three powerful linguistic mechanisms to foster 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 in this thesis is to present 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 thing 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 [Cardelli 1984] 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 ordering 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 these mechanisms usually focus on getting a relatively complex Java-like language with family polymorphism. One of the motivations 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 extending and composing existing code more easily and safely 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 many examples and rigorous analysis in this thesis, we hope to convince readers that disjoint intersection types is 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 the power of parametric polymorphism, we can further express a very dynamic form of mixin-style compositions, enabling 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

A central 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 intersection increases only the expressiveness of types (more precise properties can be checked) and not of terms. For this reason, Dunfield [2014] argues that refinement intersection is unsuited for encoding various useful language features that require the *merge operator* (or an equivalent term-level operator).

Unrestrained intersection types with a merge operator as an *explicit* introduction form for intersections 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. A bidirectional type system and the disjointness restrictions ensure that the semantics of the resulting calculi remains 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 the same field-/method names 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 prior 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 exploring their meta-theoretical properties using logical relations.

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 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 adoption of BCD subtyping, which includes distributivity rules between function/record types and intersection types. These rules are the delta that enable extending the simple forms of multiple inheritance/composition supported by prior 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 allow for a simple syntactic proof of coherence. However such restrictions makes 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 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 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 polymorphic called disjoint polymorphism [Alpuim et al. 2017]. The addition of parametric polymorphic greatly increases the expressiveness power of λ_i^+: F_i^+ is able to express *deep* conflict-free mixin composition in the presence of parametric polymorphism, which is extremely useful in the encodings of extensible designs. The key contribution is the extension of BCD subtyping with disjointness polymorphism. The extension is non-trivial in that we need to carefully retain coherence. The technical difficulty is *well-foundedness*, stemming from the interaction between impredicativity and disjointness. To address this, we extend the canonicity relation with the restriction of predicativity.

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 and BCD-like subtyping.

- A comprehensive Coq mechanization of all metatheory, including type safety, coherence, algorithmic soundness and completeness, etc.[1] This has notably revealed several

[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.

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.[2] 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.

1.4 Structure of the Thesis

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 I: 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 II: Chapters 5 and 6 explore the issue of coherence. In Chapter 5 we first propose a semantically-founded definition of coherence. We then use 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^+.

Part III: 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.

[2] The Coq formalization and implementation are available at https://github.com/bixuanzju/phd-thesis-artifact.

Chapter 9 reviews related work, Chapter 10 presents some future work and finally Chapter 11 concludes.

This thesis is largely based on two publications by the author, which are shown in the list below with corresponding chapters indicated. The work on F_i^+ is based on an on-going draft by the author (Chapters 4 and 6). 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 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 languages theory and object-oriented programming. We recommend Pierce's excellent textbook on programming languages [Pierce 2002] for a general introduction.

2 Background

The 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 2.3 we introduce family polymorphism by means of presenting Ernst's elegant solution [Ernst 2004] to the expression problem. In Section 2.2 we review the concepts of mixins and traits, their drawbacks and strengths. 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 to introduce 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 OOP, intersection types are long

[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.

known to model *multiple inheritance* [Compagnoni and Pierce 1996]. 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:

$$\text{S-interL}\ \frac{}{A \,\&\, B <: A} \qquad \text{S-interR}\ \frac{}{A \,\&\, B <: B} \qquad \text{S-inter}\ \frac{C <: A \qquad C <: B}{C <: A \,\&\, B}$$

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. Multiple-field record types can be thought of as an intersection of single-field record types. Thus, instead of
$$\{l_1 : A_1, \, \dots, l_n : A_n\}$$
we can write
$$\{l_1 : A_1\} \,\&\, \dots \,\&\, \{l_n : A_n\}$$
Note that the width and depth subtyping rules 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:

$$\frac{}{(A_1 \rightarrow A_2) \,\&\, (A_1 \rightarrow A_3) <: A_1 \rightarrow A_2 \,\&\, A_3}\;\textsc{S-distArr}$$

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:

$$\frac{}{\{l : A\} \,\&\, \{l : B\} <: \{l : A \,\&\, B\}}\;\textsc{S-distRcd}$$

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

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

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

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 rules; so we need not to add the elimination rule explicitly to the calculus.

2.1.1 The Merge Operator

Intersection types were first incorporated into a practical programming language named 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 multiple-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:

$$\textsc{mergeL}\ \frac{E_1 : A}{E_1,, E_2 : A} \qquad \textsc{mergeR}\ \frac{E_2 : A}{E_1,, E_2 : A}$$

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 $E_1,, E_2$:

$$\dfrac{\dfrac{E_1 : \mathsf{Int} \to \mathsf{Int}}{E_1,, E_2 : \mathsf{Int} \to \mathsf{Int}}\ \textsc{mergeL} \qquad \dfrac{E_2 : \mathsf{Int}}{E_1,, E_2 : \mathsf{Int}}\ \textsc{mergeR}}{E_1,, E_2 : (\mathsf{Int} \to \mathsf{Int})\ \&\ \mathsf{Int}}\ \textsc{interI}$$

Dunfield then showed how to give a semantics to a calculus with unrestricted intersection types by a type-directed elaboration to a simply-typed lambda calculus extended with 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 example, 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 type systems 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 always 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: $\mathsf{Int}\,\&\,\mathsf{Bool}$ is clearly a good intersection. But it is less clear as to what constituents a "good" (read unambiguous) intersection type in general. A few moments of thoughts lead 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]):

$$(1\,,\,,\,\text{“}c\text{”})\,,\,,\,(2\,,\,,\,\mathsf{true})$$

The first component $(1\,,\,,\,\text{“}c\text{”})$ has type $\mathsf{Int}\,\&\,\mathsf{String}$ and the second component $(2\,,\,,\,\mathsf{true})$ has type $\mathsf{Int}\,\&\,\mathsf{Bool}$. 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 take 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 in an elegant way by Oliveira et al. [2016] with the notion of *disjoint intersection types*, 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:

$$\begin{array}{ll} \text{Types} & A ::= \mathsf{Int} \mid A_1 \to A_2 \mid A_1 \,\&\, A_2 \\ \text{Terms} & E ::= i \mid x \mid \lambda x.\, E \mid E_1\, E_2 \mid E_1 ,, E_2 \mid E : A \end{array}$$

Its full (bidirectional) type system is shown in Fig. 2.1. 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). $A * B \triangleq \nexists C.\ A <: C \wedge B <: C$

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 types: 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:

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

which is impossible to express in λ_i.

To ensure that subtyping produces unique coercions, they also employ the notion of *ordinary types* [Davies and Pfenning 2000]—those that are not intersection types—and use the

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

$$\textsc{Si-int}\ \frac{}{\mathsf{Int} <: \mathsf{Int} \rightsquigarrow \lambda x.\, x} \qquad \textsc{Si-top}\ \frac{}{A <: \top \rightsquigarrow \lambda x.\, \langle\rangle}$$

$$\textsc{Si-arr}\ \frac{B_1 <: A_1 \rightsquigarrow e_1 \qquad A_2 <: B_2 \rightsquigarrow e_2}{A_1 \to A_2 <: B_1 \to B_2 \rightsquigarrow \lambda f.\, \lambda x.\, e_2\, (f\, (e_1\, x))} \qquad \textsc{Si-and}\ \frac{A_1 <: A_2 \rightsquigarrow e_1 \qquad A_1 <: A_3 \rightsquigarrow e_2}{A_1 <: A_2 \,\&\, A_3 \rightsquigarrow \lambda x.\, \langle e_1\, x, e_2\, x\rangle}$$

$$\textsc{Si-andL}\ \frac{A_1 <: A_3 \rightsquigarrow e \qquad A_3\ \mathsf{ordinary}}{A_1 \,\&\, A_2 <: A_3 \rightsquigarrow \lambda x.\, [\![A_3]\!]_{e\,(\pi_1\, x)}} \qquad \textsc{Si-andR}\ \frac{A_2 <: A_3 \rightsquigarrow e \qquad A_3\ \mathsf{ordinary}}{A_1 \,\&\, A_2 <: A_3 \rightsquigarrow \lambda x.\, [\![A_3]\!]_{e\,(\pi_2\, x)}}$$

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

$$\textsc{wf-int}\ \frac{}{\Gamma \vdash \mathsf{Int}} \qquad \textsc{wf-top}\ \frac{}{\Gamma \vdash \top} \qquad \textsc{wf-and}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash B \qquad A * B}{\Gamma \vdash A \,\&\, B} \qquad \textsc{wf-arr}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B}$$

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

$$\textsc{Ti-top}\ \frac{}{\Gamma \vdash \top \Rightarrow \top \rightsquigarrow \langle\rangle} \qquad \textsc{Ti-lit}\ \frac{}{\Gamma \vdash i \Rightarrow \mathsf{Int} \rightsquigarrow i} \qquad \textsc{Ti-var}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A \rightsquigarrow x}$$

$$\textsc{Ti-app}\ \frac{\Gamma \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e_1 \qquad \Gamma \vdash E_2 \Leftarrow A_1 \rightsquigarrow e_2}{\Gamma \vdash E_1\, E_2 \Rightarrow A_2 \rightsquigarrow e_1\, e_2} \qquad \textsc{Ti-anno}\ \frac{\Gamma \vdash E \Leftarrow A \rightsquigarrow e}{\Gamma \vdash E : A \Rightarrow A \rightsquigarrow e}$$

$$\textsc{Ti-merge}\ \frac{\Gamma \vdash E_1 \Rightarrow A_1 \rightsquigarrow e_1 \qquad \Gamma \vdash E_2 \Rightarrow A_2 \rightsquigarrow e_2 \qquad A_1 * A_2}{\Gamma \vdash E_1 ,, E_2 \Rightarrow A_1 \,\&\, A_2 \rightsquigarrow \langle e_1, e_2\rangle}$$

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

$$\textsc{Ti-abs}\ \frac{\Gamma \vdash A \qquad \Gamma, x : A \vdash E \Leftarrow B \rightsquigarrow e}{\Gamma \vdash \lambda x.\, E \Leftarrow A \to B \rightsquigarrow \lambda x.\, e} \qquad \textsc{Ti-sub}\ \frac{\Gamma \vdash B \qquad \Gamma \vdash E \Rightarrow A \rightsquigarrow e \qquad A <: B \rightsquigarrow e'}{\Gamma \vdash E \Leftarrow B \rightsquigarrow e'\, e}$$

Figure 2.1: Type system of λ_i

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 $\top$.

$\top$ brings extra complications, because Definition 1 does not hold anymore ($\top$ 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 $\top$ (such as $\top \,\&\, \top$, $\top \,\&\, \top \,\&\, \top, \ldots$), and captured by a predicate $\rceil \cdot \lceil$. An important observation is that any coercions for top-like types are unique, *even if multiple derivations exist*. The meta-function $[\![A]\!]$ used in rules SI-ANDL and SI-ANDR define coercions for top-like types. With top-types, Definition 1 is refined to account for $\top$, as shown in Definition 2.

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

However, a careful analysis of Definition 2 shows that intersection types such as $\top \,\&\, \top$ and $\top \,\&\, \mathsf{Int}$ are not well-formed because their constitute types are not disjoint. This is one of the limitations in λ_i, since "a merge of two $\top$-types will always return the same value regardless of which component of the merge is chosen" [Alpuim et al. 2017]. In other words, $\top$ 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 Disjoint Polymorphism

Disjoint polymorphism, first proposed by Alpuim et al. [2017] in the F_i calculus, is a more advanced mechanism to combine disjointness 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,

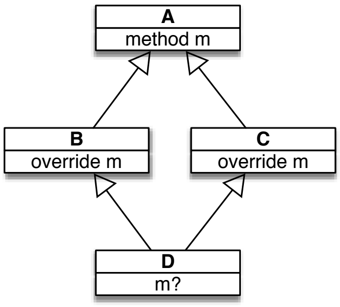
```
mergeThree [X * Int & Bool] (x : X) : X & Int & Bool = x ,, 2 ,, True;
```

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

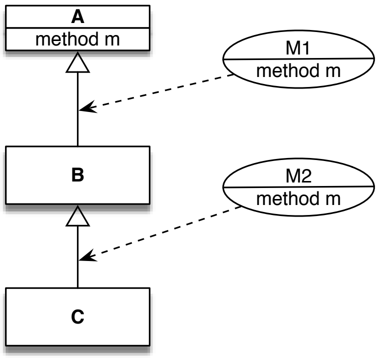
In essence disjoint intersection types and disjoint polymorphism retain most of the expressive power of the merge operator. For example, as noted by Alpuim et al. [2017], they can be used to model powerful forms of extensible records. However, forcing every intersection types to be disjoint is unnecessarily restrictive. For instance, $1 : \mathsf{Int} \,\&\, \mathsf{Int}$ 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,



(a) The diamond problem



(b) Mixin composition

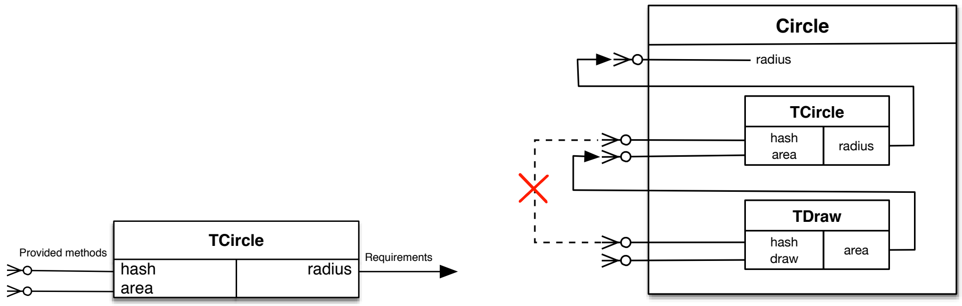
Figure 2.2: Multiple inheritance and mixins

which may not be desirable. On the other hand, putting those methods in a *lower* position results in code duplication. To overcome this limitation, *multiple inheritance* was proposed as a generalization of single inheritance. However, as 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`.



(a) A simple trait (b) Trait composition with conflicts

Figure 2.3: Traits and conflicts

In respondence 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 programs. 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. Also 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 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 OO 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 cases, `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 Lang {
  virtual class Exp {
    String toString() {}
  }
  virtual class Lit extends Exp {
    int value;
    Lit(int value) {
      this.value = value;
    }
    String toString() {
      return value;
    }
  }
  virtual class Add extends Exp {
    Exp left,right;
    Add(Exp left, Exp right) {
      this.left = left;
      this.right = right;
    }
    String toString() {
      return left + "+" + right;
    }
  }
}
```

(a) Base family: the language Lang

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

(b) Extending in two dimensions

Figure 2.4: The expression problem, Scandinavian Style

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 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 to model 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 field accessing on records: `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 $\oplus$. (`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 program 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 context 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)**.** 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)**.**

$$\begin{aligned}\Psi \vdash e_1 \simeq_{ctx} e_2 : \tau \triangleq\ & \Psi \vdash e_1 : \tau \wedge \Psi \vdash e_2 : \tau \wedge \\ & (\forall \mathcal{D}.\, \mathcal{D} : (\Psi \vdash \tau) \mapsto (\cdot \vdash \mathsf{Int}) \Longrightarrow \mathcal{D}\{e_1\} \simeq \mathcal{D}\{e_2\})\end{aligned}$$

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 τ as follows:

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

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 range 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 1. *For closed expression $e : \tau$ and $e' : \tau$, $(e, e') \in \mathcal{E}[\![\tau]\!]$ if and only if $\cdot \vdash e \simeq_{ctx} e' : \tau$.*

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 I

Type Systems

3 Semantics of the λ_i^+ Calculus

This chapter presents λ_i^+ [1], a calculus based on λ_i [Oliveira et al. 2016] with disjoint intersection types that features both BCD-style subtyping and the 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 enable 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] (BCD). 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 any obvious practical applications. This chapter shows how to deal with the complications

[1] It was also called NeColus in the original publication [Bi et al. 2018].

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 allow 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 `e` of expression $-2 + 3$ that 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 big 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, and we have created an implementation for `LangEval` with `implLang ,, implEval` where `implLang : Lang`. That 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 OO 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.

3.3 Syntax and Semantics of λ_i^+

In this section we formally present the syntax and semantics of λ_i^+. Compared to prior 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-style 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

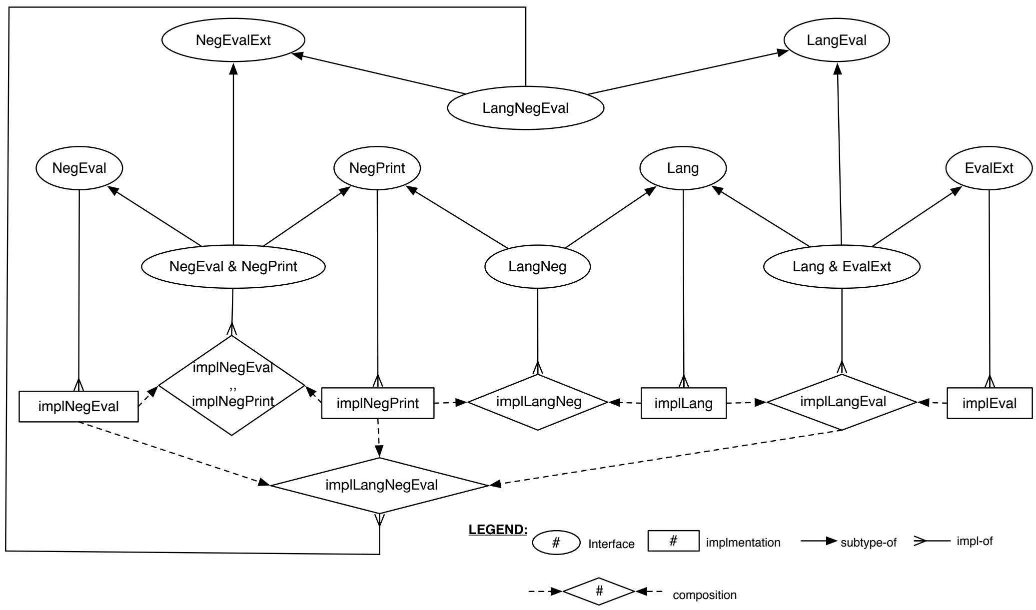


Figure 3.1: Summary of the relationships between language components

Types	A, B, C	$::=$	$\rho \mid \top \mid A \to B \mid A \,\&\, B \mid \{l : A\}$
Expressions	E	$::=$	$x \mid i \mid \top \mid \lambda x.\, E \mid E_1\, E_2 \mid E_1 \,,, E_2 \mid E : A$
		$\mid$	$\{l = E\} \mid E.l$
Typing contexts	Γ	$::=$	$\cdot \mid \Gamma, x : A$

Figure 3.2: Syntax of λ_i^+

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 primitive types (e.g., Int, String, denoted by metavariable ρ), a top type $\top$, function types $A \to B$, intersection types $A \,\&\, B$, and singleton 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 $E : A$, singleton records $\{l = E\}$, and record selections $E.l$.

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

$$\textsc{S-refl}\ \frac{}{A <: A \rightsquigarrow \mathsf{id}} \qquad \textsc{S-trans}\ \frac{A_2 <: A_3 \rightsquigarrow c_1 \qquad A_1 <: A_2 \rightsquigarrow c_2}{A_1 <: A_3 \rightsquigarrow c_1 \circ c_2} \qquad \textsc{S-top}\ \frac{}{A <: \top \rightsquigarrow \mathsf{top}}$$

$$\textsc{S-rcd}\ \frac{A <: B \rightsquigarrow c}{\{l : A\} <: \{l : B\} \rightsquigarrow c} \qquad \textsc{S-arr}\ \frac{B_1 <: A_1 \rightsquigarrow c_1 \qquad A_2 <: B_2 \rightsquigarrow c_2}{A_1 \to A_2 <: B_1 \to B_2 \rightsquigarrow c_1 \to c_2}$$

$$\textsc{S-andL}\ \frac{}{A_1 \,\&\, A_2 <: A_1 \rightsquigarrow \pi_1} \qquad \textsc{S-andR}\ \frac{}{A_1 \,\&\, A_2 <: A_2 \rightsquigarrow \pi_2}$$

$$\textsc{S-and}\ \frac{A_1 <: A_2 \rightsquigarrow c_1 \qquad A_1 <: A_3 \rightsquigarrow c_2}{A_1 <: A_2 \,\&\, A_3 \rightsquigarrow \langle c_1, c_2 \rangle} \qquad \textsc{S-topArr}\ \frac{}{\top <: \top \to \top \rightsquigarrow \mathsf{top}_{\to}}$$

$$\textsc{S-topRcd}\ \frac{}{\top <: \{l : \top\} \rightsquigarrow \mathsf{id}} \qquad \textsc{S-distArr}\ \frac{}{(A_1 \to A_2) \,\&\, (A_1 \to A_3) <: A_1 \to A_2 \,\&\, A_3 \rightsquigarrow \mathsf{dist}_{\to}}$$

$$\textsc{S-distRcd}\ \frac{}{\{l : A\} \,\&\, \{l : B\} <: \{l : A \,\&\, B\} \rightsquigarrow \mathsf{id}}$$

Figure 3.3: Declarative subtyping of F_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 rules are essentially those of the BCD type system [Barendregt et al. 1983], extended with subtyping for singleton records. Rules S-top and S-rcd for top types and record types are straightforward. Rule S-arr for function subtyping is standard. Rules S-andL, S-andR, and S-and for intersection types axiomatize that $A \,\&\, B$ is the greatest lower bound of A and B. 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:

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

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

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 enable 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.

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 Section 3.3.3. Again we ignore the highlighted parts for now.

Typing Rules and Disjointness. The typing rules of λ_i^+ 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. The set of inference rules of 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. 2017, Figure 3]. The disjointness judgment is important in order to rule out ambiguous expressions such as $1 \,,,\, 2$. Most of the typing and disjointness rules are standard and are explained in detail in previous work [Alpuim et al. 2017; Oliveira et al. 2016].

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

$$\frac{}{\Gamma \vdash \top \Rightarrow \top \rightsquigarrow \langle\rangle}\ \textsc{T-top} \qquad \frac{}{\Gamma \vdash i \Rightarrow \rho \rightsquigarrow i}\ \textsc{T-lit} \qquad \frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A \rightsquigarrow x}\ \textsc{T-var}$$

$$\frac{\Gamma \vdash E_1 \Rightarrow A_1 \rightarrow A_2 \rightsquigarrow e_1 \qquad \Gamma \vdash E_2 \Leftarrow A_1 \rightsquigarrow e_2}{\Gamma \vdash E_1\, E_2 \Rightarrow A_2 \rightsquigarrow e_1\, e_2}\ \textsc{T-app} \qquad \frac{\Gamma \vdash E \Leftarrow A \rightsquigarrow e}{\Gamma \vdash E : A \Rightarrow A \rightsquigarrow e}\ \textsc{T-anno}$$

$$\frac{\Gamma \vdash E \Rightarrow \{l : A\} \rightsquigarrow e}{\Gamma \vdash E.l \Rightarrow A \rightsquigarrow e}\ \textsc{T-proj} \qquad \frac{\Gamma \vdash E_1 \Rightarrow A_1 \rightsquigarrow e_1 \qquad \Gamma \vdash E_2 \Rightarrow A_2 \rightsquigarrow e_2 \qquad A_1 * A_2}{\Gamma \vdash E_1 ,, E_2 \Rightarrow A_1 \,\&\, A_2 \rightsquigarrow \langle e_1, e_2 \rangle}\ \textsc{T-merge}$$

$$\frac{\Gamma \vdash E \Rightarrow A \rightsquigarrow e}{\Gamma \vdash \{l = E\} \Rightarrow \{l : A\} \rightsquigarrow e}\ \textsc{T-rcd}$$

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

$$\frac{\Gamma, x : A \vdash E \Leftarrow B \rightsquigarrow e}{\Gamma \vdash \lambda x.\, E \Leftarrow A \rightarrow B \rightsquigarrow \lambda x.\, e}\ \textsc{T-abs} \qquad \frac{\Gamma \vdash E \Rightarrow A \rightsquigarrow e \qquad A <: B \rightsquigarrow c}{\Gamma \vdash E \Leftarrow B \rightsquigarrow c\, e}\ \textsc{T-sub}$$

Figure 3.4: Bidirectional type system of λ_i^+

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. Note that this is different from the original publication [Bi et al. 2018] where we keep labels in the target. Both work fine in λ_i^+, but discarding records would make the target calculus a bit simpler. The notation $|\cdot|$ is also overloaded for translating source contexts Γ to target contexts Ψ.

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

$$\frac{}{\top * A}\ \textsc{D-topL} \qquad \frac{}{A * \top}\ \textsc{D-topR} \qquad \frac{A_2 * B_2}{A_1 \to A_2 * B_1 \to B_2}\ \textsc{D-arr} \qquad \frac{A_1 * B \qquad A_2 * B}{A_1 \,\&\, A_2 * B}\ \textsc{D-andL}$$

$$\frac{A * B_1 \qquad A * B_2}{A * B_1 \,\&\, B_2}\ \textsc{D-andR} \qquad \frac{A * B}{\{l : A\} * \{l : B\}}\ \textsc{D-rcdEq} \qquad \frac{l_1 \neq l_2}{\{l_1 : A\} * \{l_2 : B\}}\ \textsc{D-rcdNeq} \qquad \frac{A *_{ax} B}{A * B}\ \textsc{D-ax}$$

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

$$\frac{B *_{ax} A}{A *_{ax} B}\ \textsc{Dax-sym} \qquad \frac{}{\rho *_{ax} A_1 \to A_2}\ \textsc{Dax-intArr} \qquad \frac{}{\rho *_{ax} \{l : A\}}\ \textsc{Dax-intRcd} \qquad \frac{}{A_1 \to A_2 *_{ax} \{l : B\}}\ \textsc{Dax-arrRcd}$$

Figure 3.5: Disjointness

Types	τ	::=	$\rho \mid \langle\rangle \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2$
Terms	e	::=	$x \mid i \mid \langle\rangle \mid \lambda x.\, e \mid e_1\, e_2 \mid \langle e_1, e_2\rangle \mid c\, e$
Coercions	c	::=	$\mathsf{id} \mid c_1 \circ c_2 \mid \mathsf{top} \mid c_1 \to c_2 \mid \langle c_1, c_2\rangle \mid \pi_1 \mid \pi_2$
		$\mid$	$\mathsf{dist}_{\to} \mid \mathsf{top}_{\to}$
Values	v	::=	$\langle\rangle \mid i \mid \lambda x.\, e \mid \langle v_1, v_2\rangle \mid (c_1 \to c_2)\, v \mid \mathsf{dist}_{\to}\, v \mid \mathsf{top}_{\to}\, v$
Typing contexts	Ψ	::=	$\cdot \mid \Psi, x : \tau$
Evaluation Contexts	$\mathcal{E}$	::=	$[\cdot] \mid \mathcal{E}\, e \mid v\, \mathcal{E} \mid \langle \mathcal{E}, e\rangle \mid \langle v, \mathcal{E}\rangle \mid c\, \mathcal{E}$

Figure 3.6: λ_{co} syntax

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

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

Coercion	Term	Coercion	Term
id	$\lambda x.\, x$	$c_1 \circ c_2$	$\lambda x.\, c_1\,(c_2\, x)$
top	$\lambda x.\, \langle\rangle$	$c_1 \to c_2$	$\lambda f.\, \lambda x.\, c_2\,(f\,(c_1\, x))$
π_1	$\lambda x.\, \pi_1\, x$	π_2	$\lambda x.\, \pi_2\, x$
$\langle c_1, c_2\rangle$	$\lambda x.\, \langle c_1\, x, c_2\, x\rangle$	$\mathsf{dist}_{\to}$	$\lambda x.\, \lambda y.\, \langle(\pi_1\, x)\, y, (\pi_2\, x)\, y\rangle$
$\mathsf{top}_{\to}$	$\lambda x.\, \lambda y.\, \langle\rangle$		

Table 3.1: Correspondence between coercions and terms

3.4.1 Explicit Coercions and Coercive Subtyping

The separate syntactic category for *explicit coercions* is a distinctive difference from the prior works (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 c ranges over coercions.[2] As a cognitive 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 $c\, e$ as a term, and various unsaturated coercion applications as values. The use of explicit coercions is useful for the new semantic proof of coherence based on logical relations. The subtyping judgment in Fig. 3.3 has the form $A <: B \rightsquigarrow c$, which says that the subtyping derivation of $A <: B$ produces a coercion c 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 entirely standard. Only the typing of coercion applications, shown below, deserves attention:

$$\textsc{T-CApp}\quad \frac{\Psi \vdash e : \tau_1 \qquad c \vdash \tau_1 \rhd \tau_2}{\Psi \vdash c\, e : \tau_2}$$

Here the judgment $c \vdash \tau_1 \rhd \tau_2$ 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.

$\boxed{c \vdash \tau_1 \rhd \tau_2}$ *(Coercion typing)*

$$\text{CT-REFL}\ \frac{}{\mathsf{id} \vdash \tau \rhd \tau} \qquad \text{CT-TRANS}\ \frac{c_1 \vdash \tau_2 \rhd \tau_3 \qquad c_2 \vdash \tau_1 \rhd \tau_2}{c_1 \circ c_2 \vdash \tau_1 \rhd \tau_3} \qquad \text{CT-TOP}\ \frac{}{\mathsf{top} \vdash \tau \rhd \langle\rangle}$$

$$\text{CT-TOPARR}\ \frac{}{\mathsf{top}_{\rightarrow} \vdash \langle\rangle \rhd \langle\rangle \rightarrow \langle\rangle} \qquad \text{CT-ARR}\ \frac{c_1 \vdash \tau_1' \rhd \tau_1 \qquad c_2 \vdash \tau_2 \rhd \tau_2'}{c_1 \rightarrow c_2 \vdash \tau_1 \rightarrow \tau_2 \rhd \tau_1' \rightarrow \tau_2'}$$

$$\text{CT-PAIR}\ \frac{c_1 \vdash \tau_1 \rhd \tau_2 \qquad c_2 \vdash \tau_1 \rhd \tau_3}{\langle c_1, c_2 \rangle \vdash \tau_1 \rhd \tau_2 \times \tau_3} \qquad \text{CT-PROJL}\ \frac{}{\pi_1 \vdash \tau_1 \times \tau_2 \rhd \tau_1} \qquad \text{CT-PROJR}\ \frac{}{\pi_2 \vdash \tau_1 \times \tau_2 \rhd \tau_2}$$

$$\text{CT-DISTARR}\ \frac{}{\mathsf{dist}_{\rightarrow} \vdash (\tau_1 \rightarrow \tau_2) \times (\tau_1 \rightarrow \tau_3) \rhd \tau_1 \rightarrow \tau_2 \times \tau_3}$$

Figure 3.7: Coercion typing

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 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 1 (☞ Preservation). *If $\cdot \vdash e : \tau$ and $e \longrightarrow e'$, then $\cdot \vdash e' : \tau$.*

Theorem 2 (☞ Progress). *If $\cdot \vdash e : \tau$, 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 $\Gamma \vdash E \Rightarrow A \rightsquigarrow e$ and $\Gamma \vdash E \Leftarrow A \rightsquigarrow e$ in Section 3.3.3. The only interesting rule is rule T-SUB, which applies the

[2] Coercions π_1 and π_2 subsume the first and second projection of pairs, respectively.

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

$$\frac{}{\mathsf{id}\ v \longrightarrow v}\ \textsc{R-Id} \qquad \frac{}{(c_1 \circ c_2)\ v \longrightarrow c_1\ (c_2\ v)}\ \textsc{R-Trans} \qquad \frac{}{\mathsf{top}\ v \longrightarrow \langle\rangle}\ \textsc{R-Top} \qquad \frac{}{(\mathsf{top}_{\rightarrow}\ \langle\rangle)\ \langle\rangle \longrightarrow \langle\rangle}\ \textsc{R-TopArr}$$

$$\frac{}{\langle c_1, c_2\rangle\ v \longrightarrow \langle c_1\ v, c_2\ v\rangle}\ \textsc{R-Pair} \qquad \frac{}{((c_1 \to c_2)\ v_1)\ v_2 \longrightarrow c_2\ (v_1\ (c_1\ v_2))}\ \textsc{R-Arr}$$

$$\frac{}{(\mathsf{dist}_{\rightarrow}\ \langle v_1, v_2\rangle)\ v_3 \longrightarrow \langle v_1\ v_3, v_2\ v_3\rangle}\ \textsc{R-DistArr} \qquad \frac{}{\pi_1\ \langle v_1, v_2\rangle \longrightarrow v_1}\ \textsc{R-ProjL} \qquad \frac{}{\pi_2\ \langle v_1, v_2\rangle \longrightarrow v_2}\ \textsc{R-ProjR}$$

$$\frac{}{(\lambda x.\ e)\ v \longrightarrow [v/x]e}\ \textsc{R-App} \qquad \frac{e \longrightarrow e'}{\mathcal{E}[e] \longrightarrow \mathcal{E}[e']}\ \textsc{R-Ctxt}$$

Figure 3.8: Dynamic semantics of λ_{co}

coercion c produced by subtyping to the target term e to form a coercion application $c\,e$. 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 1 (☞ Coercions preserve types). *If $A <: B \rightsquigarrow c$, then $c \vdash |A| \rhd |B|$.*

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

Lemma 2 (☞ Elaboration soundness). *We have that:*

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

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

As a corollary, λ_i^+ is type safe due to Theorems 1 and 2 and Lemma 2.

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 (and 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. Therefore, 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 their syntactic proof method, they have special treatment for coercions of *top-like types* (see the coercion parts in rules Si-andL and Si-andR in Fig. 2.1). Top 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; top-like types are handled like all other types.

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 Ti-abs (our rule T-abs) and rule Ti-sub (our rule T-sub). The sole purpose of this judgment is to enforce that all intersection types are disjoint. However, as Chapter 5 will explain, the syntactic restriction is unnecessary for coherence, and merely complicates the type system. Thus λ_i^+ discards this well-formedness judgment altogether in favour of a simpler design that is still coherent. As a consequence, λ_i^+ already subsumes λ_i even without adding 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 originate in merges. Therefore disjointness checking is only necessary in rule T-merge.

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 sub-

$\boxed{\mathcal{L} \vdash A \prec: B \rightsquigarrow c}$ *(Algorithmic subtyping)*

$$\textsc{A-prim}\quad \frac{}{[\,] \vdash \rho \prec: \rho \rightsquigarrow \mathsf{id}} \qquad \textsc{A-and}\quad \frac{\mathcal{L} \vdash A \prec: B_1 \rightsquigarrow c_1 \qquad \mathcal{L} \vdash A \prec: B_2 \rightsquigarrow c_2}{\mathcal{L} \vdash A \prec: B_1 \,\&\, B_2 \rightsquigarrow [\![\mathcal{L}]\!]_{\&} \circ \langle c_1, c_2 \rangle}$$

$$\textsc{A-arr}\quad \frac{\mathcal{L}, B_1 \vdash A \prec: B_2 \rightsquigarrow c}{\mathcal{L} \vdash A \prec: B_1 \to B_2 \rightsquigarrow c} \qquad \textsc{A-rcd}\quad \frac{\mathcal{L}, \{l\} \vdash A \prec: B \rightsquigarrow c}{\mathcal{L} \vdash A \prec: \{l : B\} \rightsquigarrow c} \qquad \textsc{A-top}\quad \frac{}{\mathcal{L} \vdash A \prec: \top \rightsquigarrow [\![\mathcal{L}]\!]_{\top} \circ \mathsf{top}}$$

$$\textsc{A-arrR}\quad \frac{[\,] \vdash A \prec: A_1 \rightsquigarrow c_1 \qquad \mathcal{L} \vdash A_2 \prec: B \rightsquigarrow c_2 \qquad B\ \mathsf{rigid}}{A, \mathcal{L} \vdash A_1 \to A_2 \prec: B \rightsquigarrow c_1 \to c_2} \qquad \textsc{A-rcdR}\quad \frac{\mathcal{L} \vdash A \prec: B \rightsquigarrow c \qquad B\ \mathsf{rigid}}{\{l\}, \mathcal{L} \vdash \{l : A\} \prec: B \rightsquigarrow c}$$

$$\textsc{A-andR1}\quad \frac{\mathcal{L} \vdash A_1 \prec: B \rightsquigarrow c \qquad B\ \mathsf{rigid}}{\mathcal{L} \vdash A_1 \,\&\, A_2 \prec: B \rightsquigarrow c \circ \pi_1} \qquad \textsc{A-andR2}\quad \frac{\mathcal{L} \vdash A_2 \prec: B \rightsquigarrow c \qquad B\ \mathsf{rigid}}{\mathcal{L} \vdash A_1 \,\&\, A_2 \prec: B \rightsquigarrow c \circ \pi_2}$$

Figure 3.9: Algorithmic subtyping of λ_i^+

typing is well-known, the presence of a transitivity axiom in the rules means that the system 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 the BCD system is not possible without losing expressivity. 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 decision procedure with subtyping of singleton records and coercions. We prove in Coq that the algorithm is sound and complete with respect to the declarative version. At the same time we find some errors and missing lemmas in Pierce's original manual proofs.

3.6.1 The Subtyping Algorithm

Figure 3.9 shows the algorithmic subtyping judgment $\mathcal{L} \vdash A \prec: B \rightsquigarrow c$. This judgment is the algorithmic counterpart of the declarative judgment $A <: \mathcal{L} \Rightarrow B \rightsquigarrow c$, where the symbol $\mathcal{L}$

stands for a sequence of types and labels. Definition 6 converts $\mathcal{L} \Rightarrow A$ to a valid type. For instance, if $\mathcal{L} = A, B, \{l\}$, then $\mathcal{L} \Rightarrow C$ abbreviates $A \to B \to \{l : C\}$.

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

$$[] \Rightarrow A = A \qquad (\mathcal{L}, B) \Rightarrow A = \mathcal{L} \Rightarrow (B \to A) \qquad (\mathcal{L}, \{l\}) \Rightarrow A = \mathcal{L} \Rightarrow \{l : A\}$$

The basic idea of $\mathcal{L} \vdash A \prec: B \rightsquigarrow c$ is to first perform a structural analysis of B, which descends into both sides of $\&$'s (rule A-AND), into the right side of $\to$'s (rule A-ARR), and into the fields of records (rule A-RCD) until it reaches one of the two base cases, $\top$ or B rigid. If the base case is $\top$, then the subtyping holds trivially (rule A-TOP). For the other base case, we introduce the notion of *rigid types*—those that do not have distributivity rules—captured by the predicate "A rigid" over types, and defined as follows:

Definition 7 (Rigid types)**.**

$$\rho \ \mathsf{rigid}$$

For now, the only rigid type is the primitive type ρ, but will be extended when we have richer types. If the base case is a rigid type, the algorithm performs a structural analysis of A, in which $\mathcal{L}$ plays an important role. The left sides of $\to$'s are pushed onto $\mathcal{L}$ as they are encountered in B and popped off again later, left to right, as $\to$'s are encountered in A (rule A-ARRR). Similarly, the labels are pushed onto $\mathcal{L}$ as they are encountered in B and popped off again later, left to right, as records are encountered in A (rule A-RCDR). The remaining rules 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 are both rigid types), which is essentially the one used by the `add` field in Section 3.2. The readers 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-ANDR1 and A-ANDR2 are overlapping). In other words, to check whether $A_1 \,\&\, A_2 \prec: B$, we need to check if $A_1 \prec: B$ or $A_2 \prec: B$. 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 $\mathcal{L}$ and are shown in Fig. 3.11.

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

$$D = \dfrac{\dfrac{\dfrac{\dfrac{[] \vdash \mathsf{I} \prec: \mathsf{I}}{[] \vdash \mathsf{I}\,\&\,\mathsf{S} \prec: \mathsf{I}} \qquad \dfrac{\dfrac{[] \vdash \mathsf{I} \prec: \mathsf{I}}{[] \vdash \mathsf{I}\,\&\,\mathsf{S} \prec: \mathsf{I}} \qquad [] \vdash \mathsf{I} \prec: \mathsf{I}}{\mathsf{I}\,\&\,\mathsf{S} \vdash \mathsf{I} \to \mathsf{I} \prec: \mathsf{I}}}{\mathsf{I}\,\&\,\mathsf{S}, \mathsf{I}\,\&\,\mathsf{S} \vdash \mathsf{I} \to \mathsf{I} \to \mathsf{I} \prec: \mathsf{I}}\;\textsc{A-arrR}}{\{l\}, \mathsf{I}\,\&\,\mathsf{S}, \mathsf{I}\,\&\,\mathsf{S} \vdash \{l : \mathsf{I} \to \mathsf{I} \to \mathsf{I}\} \prec: \mathsf{I}}\;\textsc{A-rcdR}}{\{l\}, \mathsf{I}\,\&\,\mathsf{S}, \mathsf{I}\,\&\,\mathsf{S} \vdash \{l : \mathsf{I} \to \mathsf{I} \to \mathsf{I}\} \,\&\, \{l : \mathsf{S} \to \mathsf{S} \to \mathsf{S}\} \prec: \mathsf{I}}\;\textsc{A-andR1}$$

$$D' = \dfrac{\dfrac{\dfrac{\dfrac{[] \vdash \mathsf{S} \prec: \mathsf{S}}{[] \vdash \mathsf{I}\,\&\,\mathsf{S} \prec: \mathsf{S}} \qquad \dfrac{\dfrac{[] \vdash \mathsf{S} \prec: \mathsf{S}}{[] \vdash \mathsf{I}\,\&\,\mathsf{S} \prec: \mathsf{S}} \qquad [] \vdash \mathsf{S} \prec: \mathsf{S}}{\mathsf{I}\,\&\,\mathsf{S} \vdash \mathsf{S} \to \mathsf{S} \prec: \mathsf{S}}}{\mathsf{I}\,\&\,\mathsf{S}, \mathsf{I}\,\&\,\mathsf{S} \vdash \mathsf{S} \to \mathsf{S} \to \mathsf{S} \prec: \mathsf{S}}\;\textsc{A-arrR}}{\{l\}, \mathsf{I}\,\&\,\mathsf{S}, \mathsf{I}\,\&\,\mathsf{S} \vdash \{l : \mathsf{S} \to \mathsf{S} \to \mathsf{S}\} \prec: \mathsf{S}}\;\textsc{A-rcdR}}{\{l\}, \mathsf{I}\,\&\,\mathsf{S}, \mathsf{I}\,\&\,\mathsf{S} \vdash \{l : \mathsf{I} \to \mathsf{I} \to \mathsf{I}\} \,\&\, \{l : \mathsf{S} \to \mathsf{S} \to \mathsf{S}\} \prec: \mathsf{S}}\;\textsc{A-andR2}$$

Figure 3.10: Example derivation

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

Figure 3.11: Meta-functions of coercions

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 lemmas connect the declarative subtyping with the meta-functions.

Lemma 3 (☞). $\top <: \mathcal{L} \Rightarrow \top \rightsquigarrow [\![\mathcal{L}]\!]_\top$

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

Lemma 4 (☞). $(\mathcal{L} \Rightarrow A) \,\&\, (\mathcal{L} \Rightarrow B) <: \mathcal{L} \Rightarrow (A \,\&\, B) \rightsquigarrow [\![\mathcal{L}]\!]_\&$

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

The proof of soundness is straightforward.

Theorem 3 (☞ Soundness). *If* $\mathcal{L} \vdash A \prec: B \rightsquigarrow c$ *then* $A <: \mathcal{L} \Rightarrow B \rightsquigarrow c$.

Proof. By induction on the derivation of the algorithmic subtyping and applying Lemmas 3 and 4 where appropriate. □

Completeness of the Algorithm. Completeness, however, is much harder. The reason is that, due to the use of $\mathcal{L}$, 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 5 (☞). $A \in \mathcal{U}(A)$

Proof. By induction on the structure of A. □

Lemma 6 (☞). *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 6 is not found in Pierce's proofs [Pierce 1989], which is crucial in Lemma 7, from which reflexivity (Lemma 8) follows immediately.

Lemma 7 (☞). *If* $\mathcal{L} \Rightarrow B \in \mathcal{U}(A)$ *then there exists* c *such that* $\mathcal{L} \vdash A \prec: B \rightsquigarrow c$.

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

Now it immediately follows that the algorithmic subtyping is reflexive.

Lemma 8 (☞ Reflexivity). *For every* A *there exists* c *such that* $[] \vdash A \prec: A \rightsquigarrow c$.

Proof. Immediate from Lemma 5 and Lemma 7. □

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 9 (☞ Transitivity). *If* $[] \vdash A_1 \prec: A_2 \rightsquigarrow c_1$ *and* $[] \vdash A_2 \prec: A_3 \rightsquigarrow c_2$*, then there exists* c *such that* $[] \vdash A_1 \prec: A_3 \rightsquigarrow c$*.*

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

Theorem 4 (☞ Completeness). *If* $A <: B \rightsquigarrow c$ *then there exists* c' *such that* $[] \vdash A \prec: B \rightsquigarrow c'$.

Proof. By induction on the derivation of the declarative subtyping and applying Lemmas 8 and 9 where appropriate. □

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

$$\frac{B_1 <: A_1 \rightsquigarrow c_1 \qquad A_2 <: B_2 \rightsquigarrow c_2}{A_1 \to A_2 <: B_1 \to B_2 \rightsquigarrow c_1 \to c_2}\ \textsc{S-arr}$$

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

Remark. It is worth pointing out that the two coercions c and c' in Theorem 4 are *contextually equivalent* (the precise definition is found in Chapter 5), which follows from Theorem 3 and Corollary 4.

4 Semantics of the F_i^+ Calculus

In this chapter, we are going to enrich λ_i^+ with parametric polymorphism. As we will see in later chapters, the combination is very expressive, able to express sophisticated concepts such as mixins/traits and a highly modular form of Visitors [Oliveira 2009; Torgersen 2004]. The combination is also highly challenging in that coherence becomes even harder to prove than in the simply typed setting. We present F_i^+, the first typed calculus combining disjoint polymorphism with BCD subtyping. F_i^+ is a variant of Leivant's predicative System F [Leivant 1991]. The choice of predicativity is due to the simplicity of the coherence proof. We will further discuss this in Chapter 6. F_i^+ serves as the theoretical foundation of typed first-class traits, which will be introduced in Chapter 7.

4.1 Motivation

Parametric polymorphism [Reynolds 1983] is a well-beloved (and well-studied) programming feature. It enables a single piece of code to be reused on data of different types. So it is quite natural and theoretically interesting to study combining parametric polymorphism with disjoint intersection types, especially how it affects disjointness and coherence. On a more pragmatic note, the combination of parametric polymorphism and disjoint intersection types also reveals new insights into practical applications. Dynamically-typed languages (such as JavaScript) usually embrace quite flexible mechanisms for class/object composition, e.g., mixin composition where objects can be composed at run time, and their structures are not necessarily statically known. The use of intersection types in TypeScript is inspired by such flexible programming patterns. For example, an important use of intersection types in TypeScript is the following function for mixin composition:

```
function extend<T, U>(first: T, second : U) : T & U {...}
```

which is analogous to our merge operator in that it takes two objects and produces an object with the intersection of the types of the argument objects. However, the types of the two objects are not known, i.e., they are generic. An important point is that, while it is possible to define such function in TypeScript (albeit using some low-level (and type-unsafe) features of JavaScript), it can also cause, as pointed out by Alpuim et al. [2017], run-time type errors! Clearly a well-defined meaning for intersection types with type variables is needed.

Disjoint Polymorphism. Motivated by the above two points, Alpuim et al. [2017] proposed disjoint polymorphism, a variant of parametric polymorphism. The main novelty is *disjoint (universal) quantification* of the form $\forall(\alpha * A). B$. Inspired by bounded quantification [Cardelli et al. 1994] where a type variable is constrained by a type bound, disjoint quantification allows type variables to be associated with *disjointness constraints*. Correspondingly, a term construct of the form $\Lambda(\alpha * A). E$ is used to create values of disjoint quantification. We have seen some examples of disjoint polymorphism at work in Section 2.1.4. With disjointness constraints and a built-in merge operator, a type-safe and conflict-free `extend` function can be naturally defined as follows:

```
extend T [U * T] (first : T) (second : U) : T & U = first ,, second;
```

The disjointness constraint on the type variable `U` ensures that no conflicts can occur when composing two objects, which is quite similar to trait-based approach [Schärli et al. 2003] in object-orientated programming. We shall devote a whole chapter (Chapter 7) to further this point.

Adding BCD Subtyping. While Alpuim et al. [2017] studied the combination of disjoint intersection types and parametric polymorphism, they follow the then-standard approach of Oliveira et al. [2016] to ensure coherence, thus excluding the possibility of adding BCD subtyping. The combination of BCD subtyping and disjoint polymorphism opens doors for more expressiveness. For example, we can define the following function

```
combine A [B * A] (f : {foo : Int → A})
                  (g : {foo : Int → B}) : {foo : Int → A & B} = f ,, g;
```

which "combines" two singleton records with parts of types unknown and returns another singleton record containing an intersection type. A variant of this function plays a fundamental role in defining Object Algebra combinators (cf. Chapter 8).

In what follows, we first present the syntax and semantics (subtyping and typing) of F_i^+. We then discuss the disjointness judgment in detail, in particular, the disjointness relation between type variables and arbitrary types. Finally we talk about the elaboration semantics of F_i^+ and its target calculus F_{co}, a variant of System F with explicit coercions.

Types	A, B, C	$::=$	$\rho \mid \top \mid A \to B \mid A \,\&\, B \mid \{l : A\} \mid \alpha \mid \forall(\alpha * A).\, B$
Monotypes	t	$::=$	$\rho \mid \top \mid t_1 \to t_2 \mid t_1 \,\&\, t_2 \mid \alpha \mid \{l : t\}$
Expressions	E	$::=$	$x \mid i \mid \top \mid \lambda x.\, E \mid E_1\, E_2 \mid E_1 ,, E_2 \mid E : A$
		$\mid$	$\{l = E\} \mid E.l \mid \Lambda(\alpha * A).\, E \mid E\, A$
Value Context	Γ	$::=$	$\cdot \mid \Gamma, x : A$
Type Context	Δ	$::=$	$\cdot \mid \Delta, \alpha * A$

Figure 4.1: Syntax of F_i^+

4.2 Syntax and Semantics

Figure 4.1 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 $\forall(\alpha * A).\, B$. Monotypes t are the same, minus the universal quantification. Metavariable E ranges over expressions. We extend λ_i^+ expressions with two standard constructs in 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.

Contexts. In the traditional formulation of System F, there is a single context that is used to keep track of both type variables and term variables. Here we use another style of presentation [Harper 2016, chap. 16] where contexts are split into *value contexts* Γ and *type contexts* Δ. The former track bound term variables x with their types A; and the latter track bound type variables α with their disjointness constraints A. This formulation is also convenient for the presentation of logical relations in Chapter 6.

Well-formedness. The well-formedness judgments for types and contexts in Fig. 4.2, though quite standard in System F, is the key difference from F_i. The F_i calculus follows λ_i where it has a disjointness condition $A * B$ in rule swft-and. This is crucial to ensure coherence due to the syntactic proof method, but it also complicates the metatheory a lot with extra effort required to show that all types in F_i are well-formed. In particular, they only have a weaker version of the substitution lemma, shown in Proposition 2: the general substitution lemma as shown in Lemma 10 does not hold in F_i, but holds in F_i^+.

Proposition 2 (Types are stable under substitution in F_i). *If $\Delta \vdash A$ and $\Delta \vdash B$ and $(\alpha * C) \in \Delta$ and $\Delta \vdash B * C$ and well-formed context $[B/\alpha]\Delta$, then $[B/\alpha]\Delta \vdash [B/\alpha]A$.*

Lemma 10 (☞ 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 $\Delta \vdash A$. □

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

$$\text{SWFT-TOP}\quad \frac{}{\Delta \vdash \top} \qquad \text{SWFT-INT}\quad \frac{}{\Delta \vdash \rho} \qquad \text{SWFT-VAR}\quad \frac{(\alpha * A) \in \Delta}{\Delta \vdash \alpha} \qquad \text{SWFT-ARROW}\quad \frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \to B}$$

$$\text{SWFT-ALL}\quad \frac{\Delta \vdash A \qquad \Delta, \alpha * A \vdash B}{\Delta \vdash \forall(\alpha * A).\, B} \qquad \text{SWFT-AND}\quad \frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \,\&\, B} \qquad \text{SWFT-RCD}\quad \frac{\Delta \vdash A}{\Delta \vdash \{l : A\}}$$

$\boxed{\Delta \vdash \Gamma}$ *(Well-formedness of value context)*

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

$\boxed{\vdash \Delta}$ *(Well-formedness of type context)*

$$\text{SWFTE-EMPTY}\quad \frac{}{\vdash \cdot} \qquad \text{SWFTE-VAR}\quad \frac{\vdash \Delta \qquad \Delta \vdash A}{\vdash \Delta, \alpha * A}$$

Figure 4.2: Well-formedness of types and contexts

Declarative Subtyping. We naturally extend the subtyping rules of λ_i^+ with only one rule S-forall, highlighted in Fig. 4.3, which specifies the subtyping relation between two universal quantifiers. In rule S-forall, a universal quantifier is covariant in its body, and contravariant in its disjointness constraint. We also need a new coercion $c_\forall$ to expression the conversion between two polymorphic types. 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. As with λ_i^+ subtyping, the subtyping relation of F_i^+ is trivially *reflexive* and *transitive*.

Remark. In our Coq formalization, we require that the two types A and B are well-formed relative to some type context, resulting in the subtyping judgment $\Delta \vdash A <: B$. But this is not very important for the purpose of presentation, thus we omit contexts.

Typing. The bidirectional type system of F_i^+ follows that of λ_i^+, as shown in Fig. 4.4. 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

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

$$\text{S-REFL}\quad \frac{}{A <: A \rightsquigarrow \mathsf{id}} \qquad \text{S-TRANS}\quad \frac{A_2 <: A_3 \rightsquigarrow c_1 \qquad A_1 <: A_2 \rightsquigarrow c_2}{A_1 <: A_3 \rightsquigarrow c_1 \circ c_2} \qquad \text{S-TOP}\quad \frac{}{A <: \top \rightsquigarrow \mathsf{top}}$$

$$\text{S-RCD}\quad \frac{A <: B \rightsquigarrow c}{\{l : A\} <: \{l : B\} \rightsquigarrow c} \qquad \text{S-ARR}\quad \frac{B_1 <: A_1 \rightsquigarrow c_1 \qquad A_2 <: B_2 \rightsquigarrow c_2}{A_1 \to A_2 <: B_1 \to B_2 \rightsquigarrow c_1 \to c_2}$$

$$\text{S-ANDL}\quad \frac{}{A_1 \,\&\, A_2 <: A_1 \rightsquigarrow \pi_1} \qquad \text{S-ANDR}\quad \frac{}{A_1 \,\&\, A_2 <: A_2 \rightsquigarrow \pi_2} \qquad \text{S-AND}\quad \frac{A_1 <: A_2 \rightsquigarrow c_1 \qquad A_1 <: A_3 \rightsquigarrow c_2}{A_1 <: A_2 \,\&\, A_3 \rightsquigarrow \langle c_1, c_2 \rangle}$$

$$\text{S-DISTARR}\quad \frac{}{(A_1 \to A_2) \,\&\, (A_1 \to A_3) <: A_1 \to A_2 \,\&\, A_3 \rightsquigarrow \mathsf{dist}_{\to}} \qquad \text{S-TOPARR}\quad \frac{}{\top <: \top \to \top \rightsquigarrow \mathsf{top}_{\to}}$$

$$\text{S-DISTRCD}\quad \frac{}{\{l : A\} \,\&\, \{l : B\} <: \{l : A \,\&\, B\} \rightsquigarrow \mathsf{id}} \qquad \text{S-TOPRCD}\quad \frac{}{\top <: \{l : \top\} \rightsquigarrow \mathsf{id}}$$

$$\text{S-FORALL}\quad \frac{B_1 <: B_2 \rightsquigarrow c \qquad A_2 <: A_1}{\forall(\alpha * A_1).\, B_1 <: \forall(\alpha * A_2).\, B_2 \rightsquigarrow c_{\forall}}$$

Figure 4.3: Declarative subtyping of F_i^+

checking judgment $\Delta; \Gamma \vdash E \Leftarrow A$ asserts that E checks against the type A in the contexts Δ and Γ. The rules directly ported from λ_i^+ are inferring rules FT-TOP for top values, FT-INT for integers, FT-VAR for variables, FT-APP for applications, FT-MERGE for merges, FT-ANNO for annotated terms, FT-RCD and FT-PROJ for records; checking rules FT-ABS for term abstractions, and the subsumption rule FT-SUB. Note that in rule FT-MERGE, the disjointness judgment has an extra type context, which will be explained in Section 4.3.

Disjoint quantification. The new rules are the inferring rules for type abstractions FT-TABS and type applications FT-TAPP. In rule FT-TABS, the disjointness constraint is added to the type context. During a type application in rule FT-TAPP, the type system checks that the type argument agrees with the disjointness constraint; this is the rule where we diverge from System F and F_i, and allow only monotype instantiations (t rather than A). We shall return

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

$$\textsc{FT-top}\quad \frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash \top \Rightarrow \top \rightsquigarrow \langle\rangle} \qquad \textsc{FT-int}\quad \frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash i \Rightarrow \rho \rightsquigarrow i} \qquad \textsc{FT-var}\quad \frac{\vdash \Delta \qquad \Delta \vdash \Gamma \qquad (x : A) \in \Gamma}{\Delta; \Gamma \vdash x \Rightarrow A \rightsquigarrow x}$$

$$\textsc{FT-app}\quad \frac{\Delta; \Gamma \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e_1 \qquad \Delta; \Gamma \vdash E_2 \Leftarrow A_1 \rightsquigarrow e_2}{\Delta; \Gamma \vdash E_1\, E_2 \Rightarrow A_2 \rightsquigarrow e_1\, e_2}$$

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

$$\textsc{FT-anno}\quad \frac{\Delta; \Gamma \vdash E \Leftarrow A \rightsquigarrow e}{\Delta; \Gamma \vdash E : A \Rightarrow A \rightsquigarrow e} \qquad \textsc{FT-tabs}\quad \frac{\Delta \vdash A \qquad \Delta, \alpha * A; \Gamma \vdash E \Rightarrow B \rightsquigarrow e}{\Delta; \Gamma \vdash \Lambda(\alpha * A).\, E \Rightarrow \forall(\alpha * A).\, B \rightsquigarrow \Lambda\alpha.\, e}$$

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

$$\textsc{FT-rcd}\quad \frac{\Delta; \Gamma \vdash E \Rightarrow A \rightsquigarrow e}{\Delta; \Gamma \vdash \{l = E\} \Rightarrow \{l : A\} \rightsquigarrow e} \qquad \textsc{FT-proj}\quad \frac{\Delta; \Gamma \vdash E \Rightarrow \{l : A\} \rightsquigarrow e}{\Delta; \Gamma \vdash E.l \Rightarrow A \rightsquigarrow e}$$

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

$$\textsc{FT-abs}\quad \frac{\Delta \vdash A \qquad \Delta; \Gamma, x : A \vdash E \Leftarrow B \rightsquigarrow e}{\Delta; \Gamma \vdash \lambda x.\, E \Leftarrow A \to B \rightsquigarrow \lambda x.\, e}$$

$$\textsc{FT-sub}\quad \frac{\Delta; \Gamma \vdash E \Rightarrow B \rightsquigarrow e \qquad \Delta \vdash A \qquad B <: A \rightsquigarrow c}{\Delta; \Gamma \vdash E \Leftarrow A \rightsquigarrow c\, e}$$

Figure 4.4: Bidirectional type system of F_i^+

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

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

$$\text{FD-andL}\quad \frac{\Delta \vdash A_1 * B \qquad \Delta \vdash A_2 * B}{\Delta \vdash A_1 \,\&\, A_2 * B} \qquad \text{FD-andR}\quad \frac{\Delta \vdash A * B_1 \qquad \Delta \vdash A * B_2}{\Delta \vdash A * B_1 \,\&\, B_2} \qquad \text{FD-rcdEq}\quad \frac{\Delta \vdash A * B}{\Delta \vdash \{l : A\} * \{l : B\}}$$

$$\text{FD-rcdNeq}\quad \frac{l_1 \neq l_2}{\Delta \vdash \{l_1 : A\} * \{l_2 : B\}} \qquad \text{FD-tvarL}\quad \frac{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash \alpha * B} \qquad \text{FD-tvarR}\quad \frac{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash B * \alpha}$$

$$\text{FD-forall}\quad \frac{\Delta, \alpha * A_1 \,\&\, A_2 \vdash B_1 * B_2}{\Delta \vdash \forall(\alpha * A_1).\, B_1 * \forall(\alpha * A_2).\, B_2} \qquad \text{FD-ax}\quad \frac{A *_{ax} B}{\Delta \vdash A * B}$$

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

$$\text{Dax-sym}\quad \frac{B *_{ax} A}{A *_{ax} B} \qquad \text{Dax-intArr}\quad \frac{}{\rho *_{ax} A_1 \to A_2} \qquad \text{Dax-intRcd}\quad \frac{}{\rho *_{ax} \{l : A\}} \qquad \text{Dax-intAll}\quad \frac{}{\rho *_{ax} \forall(\alpha * B_1).\, B_2}$$

$$\text{Dax-arrAll}\quad \frac{}{A_1 \to A_2 *_{ax} \forall(\alpha * B_1).\, B_2} \qquad \text{Dax-arrRcd}\quad \frac{}{A_1 \to A_2 *_{ax} \{l : B\}} \qquad \text{Dax-allRcd}\quad \frac{}{\forall(\alpha * A_1).\, A_2 *_{ax} \{l : B\}}$$

Figure 4.5: Disjointness of F_i^+

to this point in Chapter 6. Rules FT-merge and FT-tapp are the only two rules that use the disjointness checking.

4.3 Disjointness

In this section we present the inference rules of disjointness, as show in Fig. 4.5. The disjointness rules of F_i^+ are directly inherited from F_i [Alpuim et al. 2017], which consists of two judgments.

Main judgment. The main judgment $\Delta \vdash A * B$ says that the two types A and B are disjoint in the context Δ. As a precondition, A and B are required to be both well-formed in the context Δ. Most of the rules are similar to those of λ_i^+. The major additions are the two rules FD-tvarL and FD-tvarR for type variables, and rule FD-forall for disjoint quantification. Rule FD-tvarL and the symmetric one FD-tvarR state that a type variable α is disjoint with some type B if its disjointness constraint (i.e., A) in the context Δ 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. In a more precise sense, we have the following:

Lemma 11 (☞ 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 12. □

Lemma 12 (☞ 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. □

An intuition of the following may help better understanding Lemma 11. Another way to interpret two types being disjoint is that their least upper bound is (isomorphic to) $\top$ (cf. Section 10.1). Following this interpretation, it is obvious that if the least upper bound of two given types is already $\top$, a supertype of one of them will not change this fact.

We now turn to rule FD-forall. To illustrate this rule, consider the following two types:

$$\forall(\alpha * \mathsf{Int}).\, \alpha \,\&\, \mathsf{Int} \qquad\qquad \forall(\alpha * \mathsf{Char}).\, \alpha \,\&\, \mathsf{Char}$$

Under what conditions are the two types disjoint? In the first type, α cannot be instantiated to Int (among others) and in the second type α cannot be instantiated to Char. Therefore for both bodies to be disjoint, α can only be instantiated to types that are disjoint with both Int and Char. More formally, in rule FD-forall, we add to the context a new constraint $A_1 \,\&\, A_2$ by intersecting the two constraints A_1 and A_2, and check for disjointness of the bodies under the extended context.

Disjointness axioms. Disjointness axioms $A *_{ax} B$ take care of two types with different type constructs, except for when one of them is $\top$, an intersection type or a type variable, which are all dealt with by the main judgment.

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

Types	τ	::=	$\rho \mid \langle\rangle \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \boxed{\alpha} \mid \boxed{\forall \alpha.\, \tau}$
Expressions	e	::=	$x \mid i \mid \langle\rangle \mid \lambda x.\, e \mid e_1\, e_2 \mid \langle e_1, e_2 \rangle \mid c\, e \mid \boxed{\Lambda \alpha.\, e} \mid \boxed{e\, \tau}$
Coercions	c	::=	$\mathsf{id} \mid c_1 \circ c_2 \mid \mathsf{top} \mid c_1 \to c_2 \mid \langle c_1, c_2 \rangle \mid \pi_1 \mid \pi_2$
		$\mid$	$\mathsf{dist}_{\to} \mid \mathsf{top}_{\to} \mid \boxed{c_\forall}$
Values	v	::=	$i \mid \langle\rangle \mid \lambda x.\, e \mid \langle v_1, v_2 \rangle \mid (c_1 \to c_2)\, v \mid \mathsf{dist}_{\to}\, v \mid \mathsf{top}_{\to}\, v$
		$\mid$	$\boxed{\Lambda \alpha.\, e} \mid \boxed{c_\forall\, v}$
Value Context	Ψ	::=	$\cdot \mid \Psi, x : \tau$
Type Context	Φ	::=	$\cdot \mid \Phi, \alpha$
Evaluation Context	$\mathcal{E}$	::=	$[\cdot] \mid \mathcal{E}\, e \mid v\, \mathcal{E} \mid \langle \mathcal{E}, e \rangle \mid \langle v, \mathcal{E} \rangle \mid c\, \mathcal{E} \mid \boxed{\mathcal{E}\, \tau}$

Figure 4.6: Syntax of F_{co}

Lemma 13 (☞ Symmetry of disjointness). *If $\Delta \vdash A * B$, then $\Delta \vdash B * A$.*

Proof. By induction on the disjointness derivation. In the case for rule FD-FORALL, apply Lemma 12. □

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. 4.6, with the differences from λ_{co} highlighted. We naturally extend the type translation function $|\cdot|$ to cover type variables and disjoint quantification as shown in Definition 8. For disjoint quantification, we simply erase the disjointness constraints and translate the body.

Definition 8 (Type translation from F_i^+ to F_{co}).

$$|\rho| = \rho$$
$$|\top| = \langle\rangle$$
$$|A \to B| = |A| \to |B|$$
$$|A \,\&\, B| = |A| \times |B|$$
$$|\alpha| = \alpha$$
$$|\forall(\alpha * A).\, B| = \forall \alpha.\, |B|$$

Coercions and Coercive Subtyping. As shown in Fig. 4.6, we extend the coercions of λ_{co} with a new coercion form $c_\forall$, which expresses the transformation between two universal quantifiers. Now we go back to the coercion part in rule S-FORALL. Since the disjointness

Coercion	Term	Coercion	Term
id	$\lambda x.\, x$	$c_1 \circ c_2$	$\lambda x.\, c_1\,(c_2\, x)$
top	$\lambda x.\, \langle\rangle$	$c_1 \to c_2$	$\lambda f.\, \lambda x.\, c_2\,(f\,(c_1\, x))$
π_1	$\lambda x.\, \pi_1\, x$	π_2	$\lambda x.\, \pi_2\, x$
$\langle c_1, c_2\rangle$	$\lambda x.\, \langle c_1\, x, c_2\, x\rangle$	$\mathsf{dist}_{\to}$	$\lambda x.\, \lambda y.\, \langle(\pi_1\, x)\, y, (\pi_2\, x)\, y\rangle$
$\mathsf{top}_{\to}$	$\lambda x.\, \lambda y.\, \langle\rangle$	$c_{\forall}$	$\lambda f.\, \Lambda\alpha.\, c\,(f\,\alpha)$

Table 4.1: Correspondence between coercions and terms, extended

constraint is erased during elaboration, it does not contribute to the overall coercion; we only need the coercion generated by the subtyping of the bodies B_1 and B_2. As a cognitive aid, it is instructive to mentally "desugar" the coercion $c_{\forall}$ to the regular term $\lambda f.\, \Lambda\alpha.\, c\,(f\,\alpha)$, as shown in Table 4.1, then the expression $c_{\forall}\, v$ is "equal" to $\Lambda\alpha.\, c\,(v\,\alpha)$, which is why we can treat $c_{\forall}\, v$ as a value.

F_{co} Static Semantics. Figure 4.7 presents the typing rules of F_{co}. Most of the rules are quite standard. Rule FT-CAPP uses the coercion typing judgment $c \vdash \tau_1 \rhd \tau_2$. We extend the coercion typing of λ_{co} in Fig. 3.7 with one new rule CT-FORALL as shown below:

$$\textsc{ct-forall}\quad \frac{c \vdash \tau_1 \rhd \tau_2}{c_{\forall} \vdash \forall\alpha.\, \tau_1 \rhd \forall\alpha.\, \tau_2}$$

F_{co} Dynamic Semantics. We extend the evaluation context with one new form $\mathcal{E}\,\tau$ for type applications, as shown in Fig. 4.6. The set of reduction rules for F_{co} in Fig. 4.8 is a straightforward extension of λ_{co}. We have a new reduction rule R-FORALL for the new coercion. This rule might look strange at first. To explain, let us use our old trick of treating the coercion $c_{\forall}$ as the term $\lambda f.\, \Lambda\alpha.\, c\,(f\,\alpha)$, then the application $(\lambda f.\, \Lambda\alpha.\, c\,(f\,\alpha))\, v\, \tau$ reduces to $c\,(v\,\tau)$. 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 5 (☞ Preservation of F_{co}). *If* $\cdot;\cdot \vdash e : \tau$ *and* $e \longrightarrow e'$, *then* $\cdot;\cdot \vdash e' : \tau$.

Theorem 6 (☞ Progress of F_{co}). *If* $\cdot;\cdot \vdash e : \tau$, *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. 4.4. 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

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

$$\textsc{wfe-empty}\quad \frac{}{\Phi \vdash \cdot} \qquad\qquad \textsc{wfe-var}\quad \frac{\Phi \vdash \tau \qquad \Phi \vdash \Psi}{\Phi \vdash \Psi, x : \tau}$$

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

$$\textsc{wft-int}\quad \frac{}{\Phi \vdash \mathsf{Int}} \qquad \textsc{wft-var}\quad \frac{\alpha \in \Phi}{\Phi \vdash \alpha} \qquad \textsc{wft-arrow}\quad \frac{\Phi \vdash \tau_1 \qquad \Phi \vdash \tau_2}{\Phi \vdash \tau_1 \to \tau_2} \qquad \textsc{wft-prod}\quad \frac{\Phi \vdash \tau_1 \qquad \Phi \vdash \tau_2}{\Phi \vdash \tau_1 \times \tau_2} \qquad \textsc{wft-all}\quad \frac{\Phi, \alpha \vdash \tau_2}{\Phi \vdash \forall \alpha.\, \tau_2}$$

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

$$\textsc{Ft-unit}\quad \frac{\Phi \vdash \Psi}{\Phi; \Psi \vdash \langle\rangle : \langle\rangle} \qquad \textsc{Ft-int}\quad \frac{\Phi \vdash \Psi}{\Phi; \Psi \vdash i : \rho} \qquad \textsc{Ft-var}\quad \frac{\Phi \vdash \Psi \qquad (x : \tau) \in \Psi}{\Phi; \Psi \vdash x : \tau}$$

$$\textsc{Ft-abs}\quad \frac{\Phi; \Psi, x : \tau_1 \vdash e : \tau_2 \qquad \Phi \vdash \tau_1}{\Phi; \Psi \vdash \lambda x.\, e : \tau_1 \to \tau_2} \qquad \textsc{Ft-app}\quad \frac{\Phi; \Psi \vdash e_1 : \tau_1 \to \tau_2 \qquad \Phi; \Psi \vdash e_2 : \tau_1}{\Phi; \Psi \vdash e_1\, e_2 : \tau_2}$$

$$\textsc{Ft-tabs}\quad \frac{\Phi, \alpha; \Psi \vdash e : \tau \qquad \Phi \vdash \Psi}{\Phi; \Psi \vdash \Lambda \alpha.\, e : \forall \alpha.\, \tau} \qquad \textsc{Ft-tapp}\quad \frac{\Phi; \Psi \vdash e : \forall \alpha.\, \tau' \qquad \Phi \vdash \tau}{\Phi; \Psi \vdash e\, \tau : [\tau/\alpha]\tau'}$$

$$\textsc{Ft-pair}\quad \frac{\Phi; \Psi \vdash e_1 : \tau_1 \qquad \Phi; \Psi \vdash e_2 : \tau_2}{\Phi; \Psi \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \qquad \textsc{Ft-capp}\quad \frac{\Phi; \Psi \vdash e : \tau_1 \qquad c \vdash \tau_1 \rhd \tau_2 \qquad \Phi \vdash \tau_2}{\Phi; \Psi \vdash c\, e : \tau_2}$$

Figure 4.7: Typing rules of F_{co}

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

$$\frac{}{\mathsf{id}\; v \longrightarrow v}\;\textsc{R-Id} \qquad \frac{}{(c_1 \circ c_2)\; v \longrightarrow c_1\,(c_2\; v)}\;\textsc{R-Trans} \qquad \frac{}{\mathsf{top}\; v \longrightarrow \langle\rangle}\;\textsc{R-Top} \qquad \frac{}{(\mathsf{top}_{\rightarrow}\,\langle\rangle)\,\langle\rangle \longrightarrow \langle\rangle}\;\textsc{R-TopArr}$$

$$\frac{}{\langle c_1, c_2\rangle\; v \longrightarrow \langle c_1\; v, c_2\; v\rangle}\;\textsc{R-Pair} \qquad \frac{}{((c_1 \to c_2)\; v_1)\; v_2 \longrightarrow c_2\,(v_1\,(c_1\; v_2))}\;\textsc{R-Arr}$$

$$\frac{}{(\mathsf{dist}_{\rightarrow}\,\langle v_1, v_2\rangle)\; v_3 \longrightarrow \langle v_1\; v_3, v_2\; v_3\rangle}\;\textsc{R-DistArr} \qquad \frac{}{\pi_1\,\langle v_1, v_2\rangle \longrightarrow v_1}\;\textsc{R-ProjL} \qquad \frac{}{\pi_2\,\langle v_1, v_2\rangle \longrightarrow v_2}\;\textsc{R-ProjR}$$

$$\frac{}{(c_\forall\; v)\;\tau \longrightarrow c\,(v\;\tau)}\;\textsc{R-Forall} \qquad \frac{}{(\Lambda\alpha.\, e)\;\tau \longrightarrow [\tau/\alpha]e}\;\textsc{R-TApp} \qquad \frac{}{(\lambda x.\, e)\; v \longrightarrow [v/x]e}\;\textsc{R-App}$$

$$\frac{e \longrightarrow e'}{\mathcal{E}[e] \longrightarrow \mathcal{E}[e']}\;\textsc{R-Ctxt}$$

Figure 4.8: Dynamic semantics of F_{co}

quantification and type applications, respectively. For rules FT-Rcd and FT-Proj we simply erase the labels and translate the corresponding underlying term. All the remaining rules are ported from λ_i^+. To conclude, we show an example translation:

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

As with λ_i^+, we show two lemmas that relate F_i^+ to F_{co}.

Lemma 14 (☞ Coercions preserve types)**.** *If $A <: B \rightsquigarrow c$, then $c \vdash |A| \rhd |B|$.*

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

Lemma 15 (☞ 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. □

$\boxed{\mathcal{L} \vdash A \prec: B \rightsquigarrow c}$ *(Algorithmic subtyping)*

$$\textsc{A-prim}\quad \frac{}{[] \vdash \rho \prec: \rho \rightsquigarrow \mathsf{id}}$$

$$\textsc{A-and}\quad \frac{\mathcal{L} \vdash A \prec: B_1 \rightsquigarrow c_1 \qquad \mathcal{L} \vdash A \prec: B_2 \rightsquigarrow c_2}{\mathcal{L} \vdash A \prec: B_1 \,\&\, B_2 \rightsquigarrow [\![\mathcal{L}]\!]_{\&} \circ \langle c_1, c_2 \rangle}$$

$$\textsc{A-arr}\quad \frac{\mathcal{L}, B_1 \vdash A \prec: B_2 \rightsquigarrow c}{\mathcal{L} \vdash A \prec: B_1 \to B_2 \rightsquigarrow c}$$

$$\textsc{A-rcd}\quad \frac{\mathcal{L}, \{l\} \vdash A \prec: B \rightsquigarrow c}{\mathcal{L} \vdash A \prec: \{l : B\} \rightsquigarrow c}$$

$$\textsc{A-top}\quad \frac{}{\mathcal{L} \vdash A \prec: \top \rightsquigarrow [\![\mathcal{L}]\!]_{\top} \circ \mathsf{top}}$$

$$\textsc{A-forall}\quad \frac{[] \vdash A_2 \prec: A_1 \qquad [] \vdash B_1 \prec: B_2 \rightsquigarrow c}{[] \vdash \forall(\alpha * A_1).\, B_1 \prec: \forall(\alpha * A_2).\, B_2 \rightsquigarrow c_{\forall}}$$

$$\textsc{A-var}\quad \frac{}{[] \vdash \alpha \prec: \alpha \rightsquigarrow \mathsf{id}}$$

$$\textsc{A-arrR}\quad \frac{[] \vdash A \prec: A_1 \rightsquigarrow c_1 \qquad \mathcal{L} \vdash A_2 \prec: B \rightsquigarrow c_2 \qquad B \text{ rigid}}{A, \mathcal{L} \vdash A_1 \to A_2 \prec: B \rightsquigarrow c_1 \to c_2}$$

$$\textsc{A-rcdR}\quad \frac{\mathcal{L} \vdash A \prec: B \rightsquigarrow c \qquad B \text{ rigid}}{\{l\}, \mathcal{L} \vdash \{l : A\} \prec: B \rightsquigarrow c}$$

$$\textsc{A-andR1}\quad \frac{\mathcal{L} \vdash A_1 \prec: B \rightsquigarrow c \qquad B \text{ rigid}}{\mathcal{L} \vdash A_1 \,\&\, A_2 \prec: B \rightsquigarrow c \circ \pi_1}$$

$$\textsc{A-andR2}\quad \frac{\mathcal{L} \vdash A_2 \prec: B \rightsquigarrow c \qquad B \text{ rigid}}{\mathcal{L} \vdash A_1 \,\&\, A_2 \prec: B \rightsquigarrow c \circ \pi_2}$$

Figure 4.9: Algorithmic subtyping of F_i^+

Algorithmic subtyping. We extend the algorithmic subtyping for λ_i^+ with two rules A-forall and A-var, as shown in Fig. 4.9. They are simple adaptions from their declarative counterparts. Note that they all have empty $\mathcal{L}$ because neither polymorphic types nor type variables have subtyping relations with function types. We also need to extend the definition of rigid types to include type variables and disjoint quantification, shown in Definition 9, because as we can see in Fig. 4.3, they do not have distributivity rules.

Definition 9 (Rigid types, extended).

$$\rho \text{ rigid} \qquad \alpha \text{ rigid} \qquad \forall(\alpha * A).\, B \text{ rigid}$$

Finally we show the correctness of the algorithmic subtyping:

Conjecture 1 (Soundness). *If $\mathcal{L} \vdash A \prec: B \rightsquigarrow c$ then $A <: \mathcal{L} \Rightarrow B \rightsquigarrow c$.*

Conjecture 2 (Completeness). *If $A <: B \rightsquigarrow c$ then there exists c' such that $[] \vdash A \prec: B \rightsquigarrow c'$.*

Part II

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 $\mathsf{Int}\,\&\,\mathsf{Int}$ 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 $1 : \mathsf{Int}\,\&\,\mathsf{Int}$ 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, $\mathsf{Int}\,\&\,\mathsf{Int}$ 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 $\mathsf{Int} \,\&\, \mathsf{Int}$ 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 observational 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 want 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	$\mathcal{D}$	$::=$	$[\cdot] \mid \lambda x.\, \mathcal{D} \mid \mathcal{D}\, e \mid e\, \mathcal{D} \mid \langle \mathcal{D}, e\rangle \mid \langle e, \mathcal{D}\rangle \mid c\, \mathcal{D}$
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, $\mathsf{id}\ 1$ and $(\mathsf{id} \circ \mathsf{id})\ 1$, 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 so 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)**.** Two complete programs, e and e', are Kleene equal, written $e \backsimeq e'$, if there exists i such that $e \longrightarrow^* i$ and $e' \longrightarrow^* i$.

Definition 4 (Contextual Equivalence)**.**

$$\begin{aligned}\Psi \vdash e_1 \backsimeq_{ctx} e_2 : \tau \triangleq\ & \Psi \vdash e_1 : \tau \wedge \Psi \vdash e_2 : \tau \wedge \\ & (\forall \mathcal{D}.\, \mathcal{D} : (\Psi \vdash \tau) \mapsto (\cdot \vdash \mathsf{Int}) \Longrightarrow \mathcal{D}\{e_1\} \backsimeq \mathcal{D}\{e_2\})\end{aligned}$$

Regarding Example 1, it seems adequate to say that 3 and $\mathsf{id}\ 3$ 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: $(\lambda x.\, x) : \mathsf{Int}\,\&\,\mathsf{Int} \to \mathsf{Int}$. 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 λ_{co} context $\mathcal{D}$ so that $\mathcal{D} : (|\Gamma| \vdash |A|) \mapsto (|\Gamma'| \vdash |A'|)$ holds by construction. The full typing rules appear in Appendix B. Now we are ready to refine Definition 4's contextual equivalence to take into consideration both λ_i^+ and λ_{co} contexts.

Definition 10 (☞ λ_i^+ Contextual Equivalence)**.**

$$\begin{aligned}\Gamma \vdash E_1 \backsimeq_{ctx} E_2 : A \triangleq \forall e_1, e_2.\, \Gamma \vdash E_1 \Rightarrow A \rightsquigarrow e_1 \wedge \Gamma \vdash E_2 \Rightarrow A \rightsquigarrow e_2 \,\wedge \\ (\forall C, \mathcal{D}.\, \mathcal{C} : (\Gamma \;\Rightarrow\; A) \mapsto (\cdot \;\Rightarrow\; \mathsf{Int}) \rightsquigarrow \mathcal{D} \Longrightarrow \mathcal{D}\{e_1\} \backsimeq \mathcal{D}\{e_2\})\end{aligned}$$

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

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

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 $\backsimeq_{ctx}$ to refer to contextual equivalence in Definition 10. With Definition 10 we can formally state that λ_i^+ is coherent in the following sense:

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

$$\frac{\text{CTyp-empty1}}{[\cdot] : (\Gamma \Rightarrow A) \mapsto (\Gamma \Rightarrow A) \rightsquigarrow [\cdot]}$$

$$\text{CTyp-appL1}\ \frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1 \to A_2) \rightsquigarrow \mathcal{D} \qquad \Gamma' \vdash E_2 \Leftarrow A_1 \rightsquigarrow e}{\mathcal{C}\, E_2 : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D}\, e}$$

$$\text{CTyp-appR1}\ \frac{\Gamma' \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e \qquad \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Leftarrow A_1) \rightsquigarrow \mathcal{D}}{E_1\, \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow e\, \mathcal{D}}$$

$$\text{CTyp-mergeL1}\ \frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1) \rightsquigarrow \mathcal{D} \qquad \Gamma' \vdash E_2 \Rightarrow A_2 \rightsquigarrow e \qquad A_1 * A_2}{\mathcal{C},, E_2 : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle \mathcal{D}, e \rangle}$$

$$\text{CTyp-mergeR1}\ \frac{\Gamma' \vdash E_1 \Rightarrow A_1 \rightsquigarrow e \qquad \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D} \qquad A_1 * A_2}{E_1 ,, \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle e, \mathcal{D} \rangle}$$

$$\text{CTyp-rcd1}\ \frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}{\{l = \mathcal{C}\} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}} \qquad \text{CTyp-proj1}\ \frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}}{\mathcal{C}.l : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}$$

$$\text{CTyp-anno1}\ \frac{\mathcal{C} : (\Gamma \Rightarrow B) \mapsto (\Gamma' \Leftarrow A) \rightsquigarrow \mathcal{D}}{\mathcal{C} : A : (\Gamma \Rightarrow B) \mapsto (\Gamma' \Rightarrow A) \rightsquigarrow \mathcal{D}}$$

Figure 5.2: λ_i^+ context typing (excerpt)

Theorem 7 (☞ Coherence)**.** *We have that*

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

That is, coherence is just a special case of Definition 10 where we set E_1 and E_2 to be the same source expression. 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 10, it is not E itself but all its translations e that should behave the same! The rest of the chapter is devoted to proving the validity of Theorem 7.

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

Figure 5.3: ☞ The canonicity relation for λ_i^+

5.3 The Canonicity Relation, Formally Defined

As intuitive as Definition 10 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 7, 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].

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 originating from disjoint intersection types. Those values are essentially translated from merges, and since rule T-merge ensures disjointness, they are unambiguous. For example, two values of types Int and $\{l : \mathsf{Int}\}$ can always be 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 of disjoint types. 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 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.

[1] The logical relation is slightly different form 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.

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

$$\textsc{D-andL}\ \frac{A_1 * B \qquad A_2 * B}{A_1 \,\&\, A_2 * B} \qquad\qquad \textsc{D-andR}\ \frac{A * B_1 \qquad A * B_2}{A * B_1 \,\&\, B_2}$$

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

Lemma 16 (☞ 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 $\Gamma = x_1 : A_1, \dots, x_n : A_n$ is a finite function assigning closed values $v_1 : |A_1|, \dots, v_n : |A_n|$ to $x_1, \dots, x_n$, respectively. We write $\gamma(e)$ for the substitution $[v_1, \dots, v_n / x_1, \dots, x_n]e$. The interruption of typing contexts, written $(\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!]$ is inductively defined as follows:

Definition 11 (☞ Interpretation of value contexts).

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

Two open terms are related if every pair of related closing substitutions makes them related:

Definition 12 (☞ Logical equivalence).

$$\begin{aligned}\Gamma \vdash e_1 \backsimeq_{log} e_2 : A; B \triangleq{}& |\Gamma| \vdash e_1 : |A| \wedge |\Gamma| \vdash e_2 : |B| \wedge \\ & (\forall \gamma_1, \gamma_2.\, (\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!] \Longrightarrow (\gamma_1(e_1), \gamma_2(e_2)) \in \mathcal{E}[\![A; B]\!])\end{aligned}$$

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

5.4 Establishing Coherence

With all the machinery in place, we are now ready to prove Theorem 7. 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 17 (☞ Coercion Compatibility). *Suppose that* $A_1 <: A_2 \rightsquigarrow c$,

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

Proof. By induction on the subtyping derivation. □

Lemma 18 (☞ Merge compatibility). *If* $\Gamma \vdash e_1 \backsimeq_{log} e_1' : A$, $\Gamma \vdash e_2 \backsimeq_{log} e_2' : B$ *and* $A * B$, *then* $\Gamma \vdash \langle e_1, e_2 \rangle \backsimeq_{log} \langle e_1', e_2' \rangle : A \,\&\, B$.

Proof. By the definition of logical relation and Lemma 16. □

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 8.

Theorem 8 (☞ Inference Uniqueness). *If* $\Gamma \vdash E \Rightarrow A_1$ *and* $\Gamma \vdash E \Rightarrow A_2$, *then* $A_1 \equiv_\alpha A_2$.

Theorem 9 (☞ Fundamental Property). *We have that:*

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

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

$$\textsc{T-sub}\quad \frac{\Gamma \vdash E \Rightarrow A \rightsquigarrow e \qquad A <: B \rightsquigarrow c}{\Gamma \vdash E \Leftarrow B \rightsquigarrow c\, e}$$

where we need Theorem 8 to be able to apply the induction hypothesis. Then we apply Lemma 17 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 19 (☞ 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* $\Gamma \vdash e_1 \backsimeq_{log} e_2 : A$, *then* $\Gamma' \vdash \mathcal{D}\{e_1\} \backsimeq_{log} \mathcal{D}\{e_2\} : A'$.

Proof. By induction on the typing derivation of the context $\mathcal{C}$, and applying the compatibility lemmas where appropriate. □

Lemma 20 (☞ Adequacy). *If* $\cdot \vdash e_1 \backsimeq_{log} e_2 : \mathsf{Int}$ *then* $e_1 \backsimeq e_2$.

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 10 (Soundness w.r.t. Contextual Equivalence). *Given* $\Gamma \vdash e_1 \backsimeq_{log} e_2 : A$, *we have*

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

Proof. From Definition 10, we are given a context $\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\cdot \Rightarrow \mathsf{Int}) \rightsquigarrow \mathcal{D}$. By Lemma 19 we have $\cdot \vdash \mathcal{D}\{e_1\} \backsimeq_{log} \mathcal{D}\{e_2\} : \mathsf{Int}$, thus $\mathcal{D}\{e_1\} \backsimeq \mathcal{D}\{e_2\}$ by Lemma 20. □

Armed with Theorem 9 and Theorem 10, coherence follows directly.

Theorem 7 (☞ Coherence). *We have that*

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

Proof. Immediate from Theorem 9 and Theorem 10. □

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 1 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 2 and Corollary 3 express that merges are commutative and associative, respectively. Corollary 4 states that coercions from the same types are "coherent", i.e., they can be used interchangeably.

Corollary 1 (Neutrality). *If* $\Gamma \vdash E_1 \Rightarrow A$ *and* $\Gamma \vdash E_1 ,, E_2 \Rightarrow A$, *then* $\Gamma \vdash E_1 \backsimeq_{ctx} E_1 ,, E_2 : A$

Corollary 2 (Commutativity). *If* $\Gamma \vdash E_1 ,, E_2 \Rightarrow A$ *and* $\Gamma \vdash E_2 ,, E_1 \Rightarrow A$, *then* $\Gamma \vdash E_1 ,, E_2 \backsimeq_{ctx} E_2 ,, E_1 : A$.

Corollary 3 (Associativity). *If* $\Gamma \vdash (E_1 ,, E_2) ,, E_3 \Rightarrow A$ *and* $\Gamma \vdash E_1 ,, (E_2 ,, E_3) \Rightarrow A$, *then* $\Gamma \vdash (E_1 ,, E_2) ,, E_3 \backsimeq_{ctx} E_1 ,, (E_2 ,, E_3) : A$.

Corollary 4 (Coercions Preserve Semantics). *If* $A <: B \rightsquigarrow c_1$ *and* $A <: B \rightsquigarrow c_2$, *then* $\Gamma \vdash \lambda x.\, c_1\, x \backsimeq_{log} \lambda x.\, c_2\, x : A \to B$.

6 Coherence for F_i^+

In this chapter, we extend the canonicity relation introduced in Chapter 5 to prove coherence for F_i^+. Firstly in Section 6.1 we discuss why adding BCD subtyping to disjoint polymorphism makes establishing coherence even challenging. We then review the parametric logical relation for System F [Reynolds 1983] in Section 6.2, and talk about a failed attempt on a natural extension to deal with disjoint polymorphism in Section 6.3. The technical difficulty is *well-foundedness*, stemming from the interaction between impredicativity and disjointness. Finally in Section 6.4, we present our (predicative) logical relation that is specially crafted to prove coherence for F_i^+, and allude briefly to a potential solution to lift the predicativity restriction.

6.1 The Challenges

To have a better understanding at why adding BCD subtyping to disjoint polymorphism poses difficulties in terms of proving coherence, let us first understand how F_i retains coherence. In F_i it is of great importance to show that type system only produces *well-formed* types. Recall the well-formedness rule for intersection types (rule WF-AND) in Fig. 2.1, which has a deep impact on the metatheory. For example, they need extra effort to prove a (weaker) substitution lemma, and also that disjointness between two types is preserved after substitution. To understand why the former is required, consider the judgment $\alpha * \mathsf{Int} \vdash \alpha \,\&\, \mathsf{Int}$. The type variable α cannot be instantiated with arbitrary types, e.g., substituting α with Int would lead to an ill-formed intersection type $\mathsf{Int} \,\&\, \mathsf{Int}$. Therefore the range of types is narrowed down to those that respect the disjointness constraints—we have essentially a weaker substitution lemma. To motivate the latter, which is closely related to the former, consider the judgment $\alpha * \mathsf{Int} \vdash \alpha * \mathsf{Int}$. Obviously α cannot be instantiated with Int, either. Generally speaking, substitution-related lemmas are all weakened to account for disjointness conditions in F_i. All of these contribute to the single most important theorem: subtyping of *well-formed* types produces *unique* coercions.

However for F_i^+, the well-formedness of intersection types (rule SWFT-AND) in Fig. 4.2 does not demand a disjointness premise. This implies that we now have a general substi-

$$
\begin{array}{lcl}
(v_1, v_2) \in \mathcal{V}[\![\alpha]\!]_\rho & \triangleq & (v_1, v_2) \in \rho(\alpha) \\
(v_1, v_2) \in \mathcal{V}[\![\mathsf{Int}]\!]_\rho & \triangleq & \exists i.\, v_1 = v_2 = i \\
(v_1, v_2) \in \mathcal{V}[\![\tau_1 \to \tau_2]\!]_\rho & \triangleq & \forall (v_1', v_2') \in \mathcal{V}[\![\tau_1]\!]_\rho.\, (v_1\, v_1', v_2\, v_2') \in \mathcal{E}[\![\tau_2]\!]_\rho \\
(v_1, v_2) \in \mathcal{V}[\![\tau_1 \times \tau_2]\!]_\rho & \triangleq & (\pi_1\, v_1, \pi_1\, v_2) \in \mathcal{E}[\![\tau_1]\!]_\rho \wedge (\pi_2\, v_1, \pi_2\, v_2) \in \mathcal{E}[\![\tau_2]\!]_\rho \\
(v_1, v_2) \in \mathcal{V}[\![\forall \alpha.\, \tau]\!]_\rho & \triangleq & \forall \tau_1, \tau_2, \mathsf{R} \subseteq \tau_1 \times \tau_2.\, (v_1\, \tau_1, v_2\, \tau_2) \in \mathcal{E}[\![\tau]\!]_{\rho[\alpha \mapsto \mathsf{R}]} \\
\\
(e_1, e_2) \in \mathcal{E}[\![\tau]\!]_\rho & \triangleq & \exists v_1, v_2.\, e_1 \longrightarrow^* v_1 \wedge e_2 \longrightarrow^* v_2 \wedge (v_1, v_2) \in \mathcal{V}[\![\tau]\!]_\rho
\end{array}
$$

Figure 6.1: A logical relation for System F

tution lemma, but also that the avenue taken by Alpuim et al. [2017] to prove coherence does not work anymore. In particular, subtyping does not necessarily produces unique coercions. For example, consider the possible coercions generated by $\forall(\alpha * \mathsf{Int}).\, \alpha \,\&\, \alpha \;<:\; \forall(\alpha * \mathsf{Int} \,\&\, \mathsf{Int}).\, \alpha$. Neither of the types is "well-formed" in the sense of F_i. Two possible coercions are $\lambda f.\, \Lambda\alpha.\, \pi_1\, (f\alpha)$ and $\lambda f.\, \Lambda\alpha.\, \pi_2\, (f\alpha)$. It is not entirely obvious that these two are equivalent in some sense. Moreover, the addition of BCD subtyping aggravates the matter—i.e., the subtyping relation becomes even more non-deterministic, producing more syntactically different coercions that are harder to argue to be equivalent. Clearly a better way to prove coherence is called for! Parametricity à la System F might give us some intuitions, as we will discuss below.

6.2 Review of the Parametric Logical Relation

System F provides a reasoning principle called *relational parametricity* [Reynolds 1983] for establishing when two expressions of the same type have identical behavior. The principle is expressed in terms of a logical relation. It is well-known that the logical relation of System F induces the *abstraction theorem* (also called the *parametricity theorem*) of Reynolds [1983], which roughly says that every well-typed expression behaves the same as itself according to its type. The stringency of parametricity ensures that a polymorphic type has very few inhabitants, so few that we can deduce the behavior of a program by just looking at its type, which Wadler [1989] christened *theorems for free*.

Consider an expression e of type $\forall \alpha.\, \alpha \to \mathsf{Int}$. The following "free" theorem completely determines the behavior of e and any other function of the same type—i.e., all expressions with this type have to be constant functions.

Proposition 3 (A free theorem of $\forall \alpha.\, \alpha \to \mathsf{Int}$)**.** *Suppose e is any expression of type $\forall \alpha.\, \alpha \to \mathsf{Int}$. Let τ_1 and τ_2 be arbitrary types. For any v_1 of type τ_1 and v_2 of type τ_2, if $e\, \tau_1\, v_1 \longrightarrow^* v_3$ and $e\, \tau_2\, v_2 \longrightarrow^* v_4$ then $v_3 = v_4$.*

Figure 6.1 defines the logical relation for call-by-value System F. Compared with the logical relation for simple types, it is parameterized by a relational mapping ρ, which plays a central role for type abstractions and type variables. Intuitively, for two values v_1 and v_2 of type $\forall\alpha.\,\tau$ to behave the same, their instantiations must behave the same. However, because v_1 and v_2 do not manipulate the values of type α, it is not required that they should behave the same at the *same* instantiation. Indeed, as we saw in Proposition 3, we shall be able to consider *separately* instances of v_1 and v_2 by types τ_1 and τ_2, and treat the type variable α as standing for any relation R between τ_1 and τ_2. Now it is apparent that the relational mapping ρ maps each type variable α to a relation R that says how values at type α should be related. This explains why for type variables, values of type α are related according to the relation associated with α in the mapping ρ.

6.3 Impredicativity and Disjointness at Odds

Based on the parametric logical relation of System F, it is not hard to come up with a heterogeneous version, similar to Fig. 5.3, but with a relational mapping.

$$\begin{aligned}
(v_1, v_2) \in \mathcal{V}[\![A_1 \to B_1; A_2 \to B_2]\!]_\rho &\triangleq \forall (v_1', v_2') \in \mathcal{V}[\![A_1; A_2]\!]_\rho.\, (v_1\, v_1', v_2\, v_2') \in \mathcal{E}[\![B_1; B_2]\!]_\rho \\
(\langle v_1, v_2\rangle, v_3) \in \mathcal{V}[\![A \,\&\, B; C]\!]_\rho &\triangleq (v_1, v_3) \in \mathcal{V}[\![A; C]\!]_\rho \wedge (v_2, v_3) \in \mathcal{V}[\![B; C]\!]_\rho \\
(v_3, \langle v_1, v_2\rangle) \in \mathcal{V}[\![C; A \,\&\, B]\!]_\rho &\triangleq (v_3, v_1) \in \mathcal{V}[\![C; A]\!]_\rho \wedge (v_3, v_2) \in \mathcal{V}[\![C; B]\!]_\rho \\
(v_1, v_2) \in \mathcal{V}[\![\alpha; \alpha]\!]_\rho &\triangleq (v_1, v_2) \in \rho(\alpha)
\end{aligned}$$

The first three cases do not manipulate ρ and are directly ported from Fig. 5.3. The case for type variables is directly copied from Fig. 6.1. For type abstractions, we follow the definition for System F, but also take care of disjointness constraints, as shown below.

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

Compared with parametricity à la System F, here we cannot pick arbitrary two types except those that respect the disjointness constraints, due to the typing rule for type applications (rule FT-tapp). Let us see an example in action to get a taste of how this definition works.

Suppose e is any expression corresponding to type $\forall(\alpha * \mathsf{Bool}).\,\forall(\beta * \alpha).\,\alpha \,\&\, \beta \to \beta \,\&\, \alpha$, for any integer v_1 and character v_2, it can be shown that

$$e\,\mathsf{Int}\,\mathsf{Char}\,\langle v_1, v_2\rangle \longrightarrow^* \langle v_2, v_1\rangle$$

First, we have $(e, e) \in \mathcal{E}[\![\forall(\alpha * \mathsf{Bool}).\,\forall(\beta * \alpha).\,\alpha \,\&\, \beta \to \beta \,\&\, \alpha]\!]$. Choose $\mathsf{R}_1 = \{(v_1, v_1)\}$ and $\mathsf{R}_2 = \{(v_2, v_2)\}$. This is allowed because Int and Char both respect the disjointness constraints. Suppose $e\,\mathsf{Int}\,\mathsf{Char} \longrightarrow^* v$ for some v and we have

$$(v, v) \in \mathcal{V}[\![\alpha \,\&\, \beta \to \beta \,\&\, \alpha]\!]_{\emptyset[\alpha\mapsto\mathsf{R}_1][\beta\mapsto\mathsf{R}_2]}$$

Note that the input $\langle v_1, v_2\rangle$ is related to itself at $\alpha \,\&\, \beta$

$$(\langle v_1, v_2\rangle, \langle v_1, v_2\rangle) \in \mathcal{V}[\![\alpha \,\&\, \beta]\!]_{\emptyset[\alpha\mapsto\mathsf{R}_1][\beta\mapsto\mathsf{R}_2]}$$

So the outputs of v are related at $\beta \,\&\, \alpha$

$$(v\,\langle v_1, v_2\rangle, v\,\langle v_1, v_2\rangle) \in \mathcal{E}[\![\beta \,\&\, \alpha]\!]_{\emptyset[\alpha\mapsto\mathsf{R}_1][\beta\mapsto\mathsf{R}_2]}$$

Suppose $v\,\langle v_1, v_2\rangle \longrightarrow^* \langle v_3, v_4\rangle$, we have

$$(\langle v_3, v_4\rangle, \langle v_3, v_4\rangle) \in \mathcal{V}[\![\beta \,\&\, \alpha]\!]_{\emptyset[\alpha\mapsto\mathsf{R}_1][\beta\mapsto\mathsf{R}_2]}$$

We have $(v_3, v_3) \in \mathsf{R}_2$ and $(v_4, v_4) \in \mathsf{R}_1$, which means $v_3 = v_2$ and $v_4 = v_1$. So we have shown

$$e\,\mathsf{Int}\,\mathsf{Char}\,\langle v_1, v_2\rangle \longrightarrow^* v\,\langle v_1, v_2\rangle \longrightarrow^* \langle v_2, v_1\rangle$$

The Problem with Type Variables. Heterogeneity forces us to consider the case for $\mathcal{V}[\![\alpha; A]\!]$ where $A \neq \alpha$. A seemingly innocuous definition is as follows:

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

That is, given v_1 of type α and v_2 of type A, they are related at α and A if and only if they are related at $\rho(\alpha)$ and $\rho(A)$. Let us see another example to motivate this definition. Suppose

$$E = \Lambda(\alpha * \mathsf{Bool}).\,(\lambda x.\,x) : \alpha \,\&\, \mathsf{Int} \to \alpha \,\&\, \mathsf{Int}$$

we should expect the following to hold:[1]

$$E \,\mathsf{Int}\, 1 \longrightarrow^* \langle 1, 1\rangle$$

It boils down to verifying the following:

$$(1, 1) \in \mathcal{V}[\![\alpha; \mathsf{Int}]\!]_\rho$$

where the mapping ρ maps α to a relation over integers. If we replace α with Int then the above holds obviously. Unfortunately, the seemingly innocuous definition for type variables has a serious issue with impredicativity—in other words, the relation in question is *not well-founded*. We could pick a bad instantiation to make the relation "go into a loop". For example, suppose ρ only contains a mapping from α to $\forall\alpha.\, \forall\beta.\, \beta$, we then have the following infinite chain of equational reasoning:

$$\mathcal{V}[\![\alpha; \forall\beta.\, \beta]\!]_\rho = \mathcal{V}[\![\forall\alpha.\, \forall\beta.\, \beta; \forall\beta.\, \beta]\!]_\emptyset = \mathcal{E}[\![\forall\beta.\, \beta; \alpha]\!]_\rho = \mathcal{V}[\![\forall\beta.\, \beta; \alpha]\!]_\rho = \dots$$

The culprit is the polymorphic instantiation $\forall\alpha.\, \forall\beta.\, \beta$, which is larger than $\forall\beta.\, \beta$. This is the exact circumstance where in Fig. 6.1 substitution is deliberately avoided for type abstractions.

6.4 Predicative Logical Relation

In light of the fact that substitution in the logical relation seems unavoidable in our setting, and that impredicativity conflicts with substitution, we turn to, for the lack of a better logical relation, *predicativity*. The restriction to predicativity, though reducing the expressiveness in theory, does not cost much in practice. Languages based on the Hindley–Milner type system [Hindley 1969; Milner 1978], such as Haskell and ML, have such restriction. We also plan to study a variant of F_i^+ with implicit polymorphism, as briefly discussed in Section 10.2.

Substitution with predicative instantiations does not prevent well-foundedness. Figure 6.2 presents the logical relation for F_i^+.[2] We extend the canonicity relation in Fig. 5.3 with a new clause (highlighted) for disjoint quantification. Notice that it does not quantify over arbitrary relations, which means that our logical relation *does not* imply parametricity, and in particular, Proposition 3 is impossible to prove using our logical relation. However, it suffices for our purposes to prove coherence. Also notice that we directly substitute α with t in both B_1 and B_2. It can be shown that the relation is well-founded.

[1] The reader is advised to try it out in our prototype interpreter.

[2] In the Coq formalization, due to some technical difficulties, it contains one "Admit" about well-foundedness, for which we provide a manual proof (Lemma 21).

$$
\begin{array}{lcl}
(v_1, v_2) \in \mathcal{V}[\![\mathsf{Int}; \mathsf{Int}]\!] & \triangleq & \exists i.\, v_1 = v_2 = i \\
(v_1, v_2) \in \mathcal{V}[\![\{l : A\}; \{l : B\}]\!] & \triangleq & (v_1, v_2) \in \mathcal{V}[\![A; B]\!] \\
(v_1, v_2) \in \mathcal{V}[\![A_1 \to B_1; A_2 \to B_2]\!] & \triangleq & \forall (v_2', v_1') \in \mathcal{V}[\![A_2; A_1]\!].\, (v_1\, v_1', v_2\, v_2') \in \mathcal{E}[\![B_1; B_2]\!] \\
(v_1, v_2) \in \mathcal{V}[\![\forall (\alpha * A_1).\, B_1; \forall (\alpha * A_2).\, B_2]\!] & \triangleq & \forall \cdot \vdash t * A_1 \,\&\, A_2. \\
 & & (v_1\, |t|, v_2\, |t|) \in \mathcal{E}[\![[t/\alpha]B_1; [t/\alpha]B_2]\!] \\
(\langle v_1, v_2 \rangle, v_3) \in \mathcal{V}[\![A \,\&\, B; C]\!] & \triangleq & (v_1, v_3) \in \mathcal{V}[\![A; C]\!] \wedge (v_2, v_3) \in \mathcal{V}[\![B; C]\!] \\
(v_3, \langle v_1, v_2 \rangle) \in \mathcal{V}[\![C; A \,\&\, B]\!] & \triangleq & (v_3, v_1) \in \mathcal{V}[\![C; A]\!] \wedge (v_3, v_2) \in \mathcal{V}[\![C; B]\!] \\
(v_1, v_2) \in \mathcal{V}[\![A; B]\!] & \triangleq & \mathsf{true} \quad \mathsf{otherwise} \\
 & & \\
(e_1, e_2) \in \mathcal{E}[\![A; B]\!] & \triangleq & \exists v_1, v_2.\, e_1 \longrightarrow^* v_1 \wedge e_2 \longrightarrow^* v_2 \wedge \\
 & & (v_1, v_2) \in \mathcal{V}[\![A; B]\!]
\end{array}
$$

Figure 6.2: ☞ The canonicity relation for F_i^+

Lemma 21 (Well-foundedness). *The logical relation as defined in Fig. 6.2 is well-founded.*

Proof. Let $|\cdot|_\forall$ and $|\cdot|_s$ denote the number of $\forall$-quantifies and the size of types, respectively. The following measure suffices to prove well-foundedness:

$$\langle |\cdot|_\forall, |\cdot|_s \rangle$$

where $\langle \dots \rangle$ denotes lexicographic order. We can verify that for the clause of disjoint quantification, the number of $\forall$-quantifies decreases, because the monotype t does not contain $\forall$-quantifies. For the other clauses, either the measure of $|\cdot|_\forall$ decreases, or it remains the same but the measure of $|\cdot|_s$ decreases. □

Another noticeable thing is that in the value relation $\mathcal{V}[\![A; B]\!]$ (and $\mathcal{E}[\![A; B]\!]$), we keep the invariant that A and B are closed types. This is why we do not need to consider type variables in the logical relation, which simplifies the proof a lot. We show that the logical relation is symmetric.

Lemma 22 (☞ Symmetry of logical relation). *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. 6.2, the proof proceeds by first induction on $|A|_\forall$ then simultaneous induction on the structures of A and B. □

6.5 Establishing Coherence

We are now ready to prove coherence for F_i^+. The proof of coherence basically follows that in Chapter 5. We first give the interpretations of type and value contexts (ρ is a mapping from type variables to monotypes).

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

Figure 6.3: Expression contexts of F_{co} and F_i^+

Definition 13 (☞ Interpretation of type contexts).

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

Definition 14 (☞ Interpretation of value contexts).

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

Logical Equivalence. Logical equivalence is defined in terms of logical relation by considering all possible interpretations of free type and term variables.

Definition 15 (☞ Logical equivalence).

$$\begin{aligned}\Delta; \Gamma \vdash e_1 \backsimeq_{log} e_2 : A; B \triangleq\ & |\Delta|; |\Gamma| \vdash e_1 : |A| \wedge |\Delta|; |\Gamma| \vdash e_2 : |B| \wedge \\ & (\forall \rho, \gamma_1, \gamma_2.\ (\rho \in \mathcal{D}[\![\Delta]\!] \wedge (\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!]_\rho) \\ & \Longrightarrow (\gamma_1(\rho_1(e_1)), \gamma_2(\rho_2(e_2))) \in \mathcal{E}[\![\rho(A); \rho(B)]\!])\end{aligned}$$

Contextual Equivalence. To define the contextual equivalence, we must define the expression contexts for F_i^+ and F_{co}, which are shown in Fig. 6.3 (highlighted for the differences from λ_i^+ contexts). The typing judgment of F_i^+ contexts has 4 different forms:

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

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

Figure 6.4 shows one form of rules. The full typing rules appear in Appendix C. Now we may give the definition of contextual equivalence for F_i as follows:

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

FCTyp-empty1
$$\frac{}{[\cdot] : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta; \Gamma \Rightarrow A) \rightsquigarrow [\cdot]}$$

FCTyp-appL1
$$\frac{\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_1 \to A_2) \rightsquigarrow \mathcal{D} \qquad \Delta'; \Gamma' \vdash E_2 \Leftarrow A_1 \rightsquigarrow e}{\mathcal{C}\, E_2 : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D}\, e}$$

FCTyp-appR1
$$\frac{\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Leftarrow A_1) \rightsquigarrow \mathcal{D} \qquad \Delta'; \Gamma' \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e}{E_1\, \mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_2) \rightsquigarrow e\, \mathcal{D}}$$

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
$$\frac{\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta', \alpha * B; \Gamma' \Rightarrow B') \rightsquigarrow \mathcal{D} \qquad \Delta' \vdash B}{\Lambda(\alpha * B).\, \mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow \forall(\alpha * B).\, B') \rightsquigarrow \Lambda\alpha.\, \mathcal{D}}$$

FCTyp-tapp1
$$\frac{\begin{array}{c}\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow \forall(\alpha * A_1).\, A_2) \rightsquigarrow \mathcal{D} \\ \Delta' \vdash t \qquad \Delta' \vdash t * A_1\end{array}}{\mathcal{C}\, B : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow [t/\alpha]A_2) \rightsquigarrow \mathcal{D}\, |t|}$$

Figure 6.4: F_i^+ context typing (excerpt)

Definition 16 (☞ F_i^+ Contextual Equivalence)**.**

$$\begin{aligned}\Delta;\Gamma \vdash E_1 \backsimeq_{ctx} E_2 : A \triangleq \forall e_1, e_2.\ &\Delta;\Gamma \vdash E_1 \Rightarrow A \rightsquigarrow e_1 \land \Delta;\Gamma \vdash E_2 \Rightarrow A \rightsquigarrow e_2 \land \\ &(\forall \mathcal{C}, \mathcal{D}.\ \mathcal{C} : (\Delta;\Gamma \Rightarrow A) \mapsto (\cdot;\cdot \Rightarrow \mathsf{Int}) \rightsquigarrow \mathcal{D} \\ &\implies \mathcal{D}\{e_1\} \backsimeq \mathcal{D}\{e_2\})\end{aligned}$$

The connection between disjointness and the value relation becomes a bit complicated due to the addition of type variables. We first prove that disjoint values of monotypes are related.

Lemma 23 (☞ Disjoint values of monotypes are related)**.** *If* $\cdot \vdash t_1 * t_2$*,* $\cdot;\cdot \vdash v_1 : |t_1|$ *and* $\cdot;\cdot \vdash v_2 : |t_2|$ *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 24 (☞ Disjoint values are related)**.** *If* $\Delta \vdash A * B$*,* $\rho \in \mathcal{D}[\![\Delta]\!]$*,* $\cdot;\cdot \vdash v_1 : |\rho(A)|$ *and* $\cdot;\cdot \vdash v_2 : |\rho(B)|$ *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:

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

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, we know $\rho(\alpha)$ and $\rho(A)$ are disjoint by definition. Then by Lemma 11 and $A <: B$, we have $\rho(\alpha)$ and $\rho(B)$ are also disjoint. Finally we apply Lemma 23. □

Next up are the compatibility lemmas. They are essentially the same as in Chapter 5. So we skip them and state the last two important theorems without giving proofs. The proofs have mostly the same structures as in the simply-typed case, but with more cases. The interested reader can refer to our Coq formalization.

Theorem 11 (☞ Fundamental Property)**.** *We have that:*

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

Theorem 12 (☞ Congruence). *If* $\mathcal{C} : (\Delta; \Gamma \Leftrightarrow A) \mapsto (\Delta'; \Gamma' \Leftrightarrow' A') \rightsquigarrow \mathcal{D}$, $\Delta; \Gamma \vdash E_1 \Leftrightarrow A \rightsquigarrow e_1$, $\Delta; \Gamma \vdash E_2 \Leftrightarrow A \rightsquigarrow e_2$ *and* $\Delta; \Gamma \vdash e_1 \backsimeq_{log} e_2 : A$, *then* $\Delta'; \Gamma' \vdash \mathcal{D}\{e_1\} \backsimeq_{log} \mathcal{D}\{e_2\} : A'$.

Finally F_i^+ is coherent.

Theorem 13 (☞ Coherence of F_i^+). *We have that*

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

Final remarks. It would be interesting to study parametricity of F_i^+. As we have seen, it is not obvious how to extend the parametric logical relation as defined in Fig. 6.1 to account for disjointness, and avoid potential circularity due to impredicativity. A possible solution is to use step-indexed logical relations. We have yet investigated further on that direction.

Part III

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* 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 of modularizing 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 OO 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. In what follows, we present SEDEL: the first design of typed first-class traits.

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:

```
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.

We wish to bring the readers' 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 an untyped 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 OO 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 fails 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 readers 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 = "comm∧";
    }
    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 in a typed setting:

- 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 version 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, we can justify the invocation of `version` on **self** 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 version, 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 OO 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 cousin `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.

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`, we are 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.

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 = "comm∧";
    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 behaviour 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 $\top$ to instantiate `A` because the `editor` trait provides exactly those method specified by `Editor` and nothing more (hence $\top$). 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 (expected) behavior.

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 find-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, 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

[2] In fact, as we will see in Section 7.3, we can still access `on_key` in `editor` by the forwarding operator.

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 OO languages, the keyword **new** is used to create an object. A difference to other OO 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 error. This behaviour 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 behaviour 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 ($\top$). 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` definition. In many other OO 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 $\top$. This is different from typical OO 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. `IDEEditor` is defined as an intersection type (`Editor & Version & Spelling & ModalEdit`). 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 behaviour 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 ++ " ∧ " ++ (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 "comm∧");
main = a_editor2.do_cut
-- Output:
-- "Process C-x on modal editor ∧ 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:

```
merge A [B * A] (x : Trait[A]) (y : Trait[B]) = new[A & B] x & y;
```

The `merge` 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, `merge` 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 and Palsberg's work on the denotational semantics for inheritance [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 A.

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.

$$
\begin{array}{llcl}
\text{Types} & \mathcal{A}, \mathcal{B}, \mathcal{C} & ::= & \top \mid \rho \mid \mathcal{A} \to \mathcal{B} \mid \mathcal{A} \,\&\, \mathcal{B} \mid \{l : \mathcal{A}\} \mid \alpha \mid \forall(\alpha * \mathcal{A}).\, \mathcal{B} \mid \boxed{\textbf{Trait}\, [\mathcal{A}, \mathcal{B}]} \\
\text{Expressions} & \mathcal{T} & ::= & \top \mid i \mid x \mid \lambda x. \mathcal{T} \mid \mathcal{T}_1\, \mathcal{T}_2 \mid \Lambda(\alpha * \mathcal{A}).\, \mathcal{T} \mid \mathcal{T}\, \mathcal{A} \mid \mathcal{T}_1 ,, \mathcal{T}_2 \mid \mathcal{T} : \mathcal{A} \\
 & & \mid & \{l = \mathcal{T}\} \mid \mathcal{T}.l \mid \textbf{letrec}\, x : \mathcal{A} = \mathcal{T}_1\, \textbf{in}\, \mathcal{T}_2 \mid \boxed{\textbf{new}\, [\mathcal{A}](\, \overline{\mathcal{T}_i}^{\,i}\,)} \mid \boxed{\mathcal{T}_1 \wedge \mathcal{T}_2} \\
 & & \mid & \boxed{\textbf{trait}\, [\mathsf{self} : \mathcal{B}]\, \textbf{inherits}\, \overline{\mathcal{T}_i}^{\,i}\, \{\, \overline{l_j = \mathcal{T}_j'}^{\,j}\, \} : \mathcal{A}} \\
\text{Value contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \mathcal{A} \\
\text{Type contexts} & \Delta & ::= & \cdot \mid \Delta, \alpha * \mathcal{A}
\end{array}
$$

$$
\begin{array}{llcl}
\text{Record types} & \{l_1 : \mathcal{A}_1, \ldots, l_n : \mathcal{A}_n\} & \triangleq & \{l_1 : \mathcal{A}_1\} \,\&\, \ldots \,\&\, \{l_n : \mathcal{A}_n\} \\
\text{Records} & \{l_1 = \mathcal{T}_1, \ldots, l_n = \mathcal{T}_n\} & \triangleq & \{l_1 = \mathcal{T}_1\} ,, \ldots ,, \{l_n = \mathcal{T}_n\}
\end{array}
$$

Figure 7.1: SEDEL core syntax and syntactic abbreviations

Types. Metavariables $\mathcal{A}, \mathcal{B}, \mathcal{C}$ range over types. Types include a top type $\top$, primitive types ρ, function types $\mathcal{A} \to \mathcal{B}$, 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 **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 $\textbf{letrec}\, x : \mathcal{A} = \mathcal{T}_1\, \textbf{in}\, \mathcal{T}_2$, 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 $\textbf{trait}\, [\mathsf{self} : \mathcal{B}]\, \textbf{inherits}\, \overline{\mathcal{T}_i}^{\,i}\, \{\, \overline{l_j = \mathcal{T}_j'}^{\,j}\, \} : \mathcal{A}$, which specifies a list of trait expressions $\overline{\mathcal{T}_i}$ in the **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 **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 $\textbf{new}\, [\mathcal{A}](\, \overline{\mathcal{T}_i}^{\,i}\,)$ 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.

Abbreviations. For ease of programming, multiple-field record types are merely syntactic sugar for intersections of single-field record types. Similarly, multi-field record expressions are syntactic sugar for merges of single-field records.

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

$$\frac{}{\Delta \vdash \top}\ \text{WF-TOP} \qquad \frac{}{\Delta \vdash \rho}\ \text{WF-INT} \qquad \frac{\Delta \vdash \mathcal{A} \qquad \Delta \vdash \mathcal{B}}{\Delta \vdash \mathcal{A} \to \mathcal{B}}\ \text{WF-ARR} \qquad \frac{\Delta \vdash \mathcal{A}}{\Delta \vdash \{l : \mathcal{A}\}}\ \text{WF-RCD} \qquad \frac{(\alpha * \mathcal{A}) \in \Delta}{\Delta \vdash \alpha}\ \text{WF-VAR}$$

$$\frac{\Delta \vdash \mathcal{A} \qquad \Delta \vdash \mathcal{B}}{\Delta \vdash \mathcal{A} \,\&\, \mathcal{B}}\ \text{WF-AND} \qquad \frac{\Delta \vdash \mathcal{A} \qquad \Delta, \alpha * \mathcal{A} \vdash \mathcal{B}}{\Delta \vdash \forall(\alpha * \mathcal{A}).\, \mathcal{B}}\ \text{WF-FORALL} \qquad \frac{\Delta \vdash \mathcal{A} \qquad \Delta \vdash \mathcal{B}}{\Delta \vdash \textbf{Trait}\,[\mathcal{A}, \mathcal{B}]}\ \text{WF-TRAIT}$$

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

$$\frac{}{\mathcal{A} <: \mathcal{A}}\ \text{TS-REFL} \qquad \frac{\mathcal{A}_2 <: \mathcal{A}_3 \qquad \mathcal{A}_1 <: \mathcal{A}_2}{\mathcal{A}_1 <: \mathcal{A}_3}\ \text{TS-TRANS} \qquad \frac{}{\mathcal{A} <: \top}\ \text{TS-TOP} \qquad \frac{\mathcal{A} <: \mathcal{B}}{\{l : \mathcal{A}\} <: \{l : \mathcal{B}\}}\ \text{TS-RCD}$$

$$\frac{\mathcal{B}_1 <: \mathcal{A}_1 \qquad \mathcal{A}_2 <: \mathcal{B}_2}{\mathcal{A}_1 \to \mathcal{A}_2 <: \mathcal{B}_1 \to \mathcal{B}_2}\ \text{TS-ARR} \qquad \frac{}{\mathcal{A}_1 \,\&\, \mathcal{A}_2 <: \mathcal{A}_1}\ \text{TS-ANDL} \qquad \frac{}{\mathcal{A}_1 \,\&\, \mathcal{A}_2 <: \mathcal{A}_2}\ \text{TS-ANDR}$$

$$\frac{\mathcal{A}_1 <: \mathcal{A}_2 \qquad \mathcal{A}_1 <: \mathcal{A}_3}{\mathcal{A}_1 <: \mathcal{A}_2 \,\&\, \mathcal{A}_3}\ \text{TS-AND} \qquad \frac{}{(\mathcal{A}_1 \to \mathcal{A}_2) \,\&\, (\mathcal{A}_1 \to \mathcal{A}_3) <: \mathcal{A}_1 \to \mathcal{A}_2 \,\&\, \mathcal{A}_3}\ \text{TS-DISTARR} \qquad \frac{}{\top <: \top \to \top}\ \text{TS-TOPARR}$$

$$\frac{}{\{l : \mathcal{A}\} \,\&\, \{l : \mathcal{B}\} <: \{l : \mathcal{A} \,\&\, \mathcal{B}\}}\ \text{TS-DISTRCD} \qquad \frac{}{\top <: \{l : \top\}}\ \text{TS-TOPRCD} \qquad \frac{\mathcal{B}_1 <: \mathcal{A}_1 \qquad \mathcal{A}_2 <: \mathcal{B}_2}{\textbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] <: \textbf{Trait}\,[\mathcal{B}_1, \mathcal{B}_2]}\ \text{TS-TRAIT}$$

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)*

$$\frac{}{\Delta \vdash \top * \mathcal{A}}\ \textsc{SD-topL} \qquad \frac{}{\Delta \vdash \mathcal{A} * \top}\ \textsc{SD-topR} \qquad \frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \mathcal{A}_1 \to \mathcal{A}_2 * \mathcal{B}_1 \to \mathcal{B}_2}\ \textsc{SD-arr}$$

$$\frac{\Delta \vdash \mathcal{A}_1 * \mathcal{B} \qquad \Delta \vdash \mathcal{A}_2 * \mathcal{B}}{\Delta \vdash \mathcal{A}_1 \,\&\, \mathcal{A}_2 * \mathcal{B}}\ \textsc{SD-andL} \qquad \frac{\Delta \vdash \mathcal{A} * \mathcal{B}_1 \qquad \Delta \vdash \mathcal{A} * \mathcal{B}_2}{\Delta \vdash \mathcal{A} * \mathcal{B}_1 \,\&\, \mathcal{B}_2}\ \textsc{SD-andR}$$

$$\frac{\Delta \vdash \mathcal{A} * \mathcal{B}}{\Delta \vdash \{l : \mathcal{A}\} * \{l : \mathcal{B}\}}\ \textsc{SD-rcdEq} \qquad \frac{l_1 \neq l_2}{\Delta \vdash \{l_1 : \mathcal{A}\} * \{l_2 : \mathcal{B}\}}\ \textsc{SD-rcdNeq} \qquad \frac{(\alpha * \mathcal{A}) \in \Delta \qquad \mathcal{A} <: \mathcal{B}}{\Delta \vdash \alpha * \mathcal{B}}\ \textsc{SD-tvarL}$$

$$\frac{(\alpha * \mathcal{A}) \in \Delta \qquad \mathcal{A} <: \mathcal{B}}{\Delta \vdash \mathcal{B} * \alpha}\ \textsc{SD-tvarR} \qquad \frac{\Delta, \alpha * \mathcal{A}_1 \,\&\, \mathcal{A}_2 \vdash \mathcal{B}_1 * \mathcal{B}_2}{\Delta \vdash \forall(\alpha * \mathcal{A}_1).\, \mathcal{B}_1 * \forall(\alpha * \mathcal{A}_2).\, \mathcal{B}_2}\ \textsc{SD-forall}$$

$$\frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \mathbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] * \mathbf{Trait}\,[\mathcal{B}_1, \mathcal{B}_2]}\ \textsc{SD-trait} \qquad \frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \mathbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] * \mathcal{B}_1 \to \mathcal{B}_2}\ \textsc{SD-traitArr1}$$

$$\frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \mathcal{A}_1 \to \mathcal{A}_2 * \mathbf{Trait}\,[\mathcal{B}_1, \mathcal{B}_2]}\ \textsc{SD-traitArr2} \qquad \frac{\mathcal{A} *_{ax} \mathcal{B}}{\Delta \vdash \mathcal{A} * \mathcal{B}}\ \textsc{SD-ax}$$

$\boxed{\mathcal{A} *_{ax} \mathcal{B}}$ *(Disjointness axiom)*

$$\frac{\mathcal{B} *_{ax} \mathcal{A}}{\mathcal{A} *_{ax} \mathcal{B}}\ \textsc{SDax-sym} \qquad \frac{}{\rho *_{ax} \mathcal{A}_1 \to \mathcal{A}_2}\ \textsc{SDax-intArr} \qquad \frac{}{\rho *_{ax} \{l : \mathcal{A}\}}\ \textsc{SDax-intRcd} \qquad \frac{}{\rho *_{ax} \forall(\alpha * \mathcal{B}_1).\, \mathcal{B}_2}\ \textsc{SDax-intAll}$$

$$\frac{}{\mathcal{A}_1 \to \mathcal{A}_2 *_{ax} \forall(\alpha * \mathcal{B}_1).\, \mathcal{B}_2}\ \textsc{SDax-arrAll} \qquad \frac{}{\mathcal{A}_1 \to \mathcal{A}_2 *_{ax} \{l : \mathcal{B}\}}\ \textsc{SDax-arrRcd} \qquad \frac{}{\forall(\alpha * \mathcal{A}_1).\, \mathcal{A}_2 *_{ax} \{l : \mathcal{B}\}}\ \textsc{SDax-allRcd}$$

$$\frac{}{\rho *_{ax} \mathbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2]}\ \textsc{SDax-intTrait} \qquad \frac{}{\mathbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] *_{ax} \forall(\alpha * \mathcal{B}_1).\, \mathcal{B}_2}\ \textsc{SDax-traitAll} \qquad \frac{}{\mathbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] *_{ax} \{l : \mathcal{B}\}}\ \textsc{SDax-traitRcd}$$

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 ($\rightsquigarrow E$) 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

$$\frac{\begin{array}{c}\overline{\Delta; \Gamma, \mathsf{self} : \mathcal{B} \vdash \mathcal{T}_i \Rightarrow \textbf{Trait}\,[\mathcal{B}_i, \mathcal{C}_i] \rightsquigarrow E_i}^{i \in 1..n} \\ \Delta; \Gamma, \mathsf{self} : \mathcal{B}, \mathsf{super} : \mathcal{C}_1 \,\&\, .. \,\&\, \mathcal{C}_n \vdash \{\, \overline{l_j = \mathcal{T}'_j}^{j \in 1..m} \,\} \Rightarrow \mathcal{C} \rightsquigarrow E \\ \overline{\mathcal{B} <: \mathcal{B}_i}^{i \in 1..n} \qquad \Delta \vdash \mathcal{C}_1 * \,..\, * \mathcal{C}_n * \mathcal{C} \qquad \mathcal{C}_1 \,\&\, .. \,\&\, \mathcal{C}_n \,\&\, \mathcal{C} <: \mathcal{A}\end{array}}{\begin{array}{c}\Delta; \Gamma \vdash \textbf{trait}\,[\mathsf{self} : \mathcal{B}]\,\textbf{inherits}\,\overline{\mathcal{T}_i}^{i \in 1..n}\,\{\, \overline{l_j = \mathcal{T}'_j}^{j \in 1..m} \,\} : \mathcal{A} \Rightarrow \textbf{Trait}\,[\mathcal{B}, \mathcal{A}] \rightsquigarrow \\ \lambda \mathsf{self} : |\mathcal{B}|.\,(\textbf{let}\,\mathsf{super} = \overline{(E_i\,\mathsf{self})}^{i \in 1..n}\,\textbf{in}\,\mathsf{super}, , E)\end{array}}$$

$\boxed{\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \mathcal{A} \rightsquigarrow E}$ *(Inference)*

$$\textsc{ST-top}\quad \frac{}{\Delta; \Gamma \vdash \top \Rightarrow \top \rightsquigarrow \top} \qquad \textsc{ST-int}\quad \frac{}{\Delta; \Gamma \vdash i \Rightarrow \rho \rightsquigarrow i} \qquad \textsc{ST-var}\quad \frac{(x : \mathcal{A}) \in \Delta}{\Delta; \Gamma \vdash x \Rightarrow \mathcal{A} \rightsquigarrow x}$$

$$\textsc{ST-app}\quad \frac{\Delta; \Gamma \vdash \mathcal{T}_1 \Rightarrow \mathcal{A}_1 \to \mathcal{A}_2 \rightsquigarrow E_1 \qquad \Delta; \Gamma \vdash \mathcal{T}_2 \Leftarrow \mathcal{A}_1 \rightsquigarrow E_2}{\Delta; \Gamma \vdash \mathcal{T}_1\, \mathcal{T}_2 \Rightarrow \mathcal{A}_2 \rightsquigarrow E_1\, E_2}$$

$$\textsc{ST-merge}\quad \frac{\Delta; \Gamma \vdash \mathcal{T}_1 \Rightarrow \mathcal{A} \rightsquigarrow E_1 \qquad \Delta; \Gamma \vdash \mathcal{T}_2 \Rightarrow \mathcal{B} \rightsquigarrow E_2 \qquad \Delta \vdash \mathcal{A} * \mathcal{B}}{\Delta; \Gamma \vdash \mathcal{T}_1 ,, \mathcal{T}_2 \Rightarrow \mathcal{A} \,\&\, \mathcal{B} \rightsquigarrow E_1 ,, E_2}$$

$$\textsc{ST-anno}\quad \frac{\Delta; \Gamma \vdash \mathcal{T} \Leftarrow \mathcal{A} \rightsquigarrow E}{\Delta; \Gamma \vdash \mathcal{T} : \mathcal{A} \Rightarrow \mathcal{A} \rightsquigarrow E : |\mathcal{A}|} \qquad \textsc{ST-tabs}\quad \frac{\Delta \vdash \mathcal{A} \qquad \Delta, \alpha * \mathcal{A}; \Gamma \vdash \mathcal{T} \Rightarrow \mathcal{B} \rightsquigarrow E}{\Delta; \Gamma \vdash \Lambda(\alpha * \mathcal{A}).\, \mathcal{T} \Rightarrow \forall(\alpha * \mathcal{A}).\, \mathcal{B} \rightsquigarrow \Lambda(\alpha * |\mathcal{A}|).\, E}$$

$$\textsc{ST-tapp}\quad \frac{\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \forall(\alpha * \mathcal{B}_1).\, \mathcal{B}_2 \rightsquigarrow E \qquad \Delta \vdash \mathcal{A} \qquad \Delta \vdash \mathcal{A} * \mathcal{B}_1}{\Delta; \Gamma \vdash \mathcal{T}\, \mathcal{A} \Rightarrow [\mathcal{A}/\alpha]\mathcal{B}_2 \rightsquigarrow E\, |\mathcal{A}|}$$

$$\textsc{ST-rcd}\quad \frac{\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \mathcal{A} \rightsquigarrow E}{\Delta; \Gamma \vdash \{l = \mathcal{T}\} \Rightarrow \{l : \mathcal{A}\} \rightsquigarrow \{l = E\}} \qquad \textsc{ST-proj}\quad \frac{\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \{l : \mathcal{A}\} \rightsquigarrow E}{\Delta; \Gamma \vdash \mathcal{T}.l \Rightarrow \mathcal{A} \rightsquigarrow E.l}$$

$$\textsc{ST-trait}\quad \frac{\begin{array}{c} \overline{\Delta; \Gamma, \mathsf{self} : \mathcal{B} \vdash \mathcal{T}_i \Rightarrow \textbf{Trait}\,[\mathcal{B}_i, \mathcal{C}_i] \rightsquigarrow E_i}^{i \in 1..n} \\ \Delta; \Gamma, \mathsf{self} : \mathcal{B} \vdash \{\, \overline{l_j = \mathcal{T}_j'}^{j \in 1..m} \,\} \Rightarrow \mathcal{C} \rightsquigarrow E \\ \overline{\mathcal{B} <: \mathcal{B}_i}^{i \in 1..n} \qquad \Delta \vdash \mathcal{C}_1 * \,..\, * \mathcal{C}_n * \mathcal{C} \qquad \mathcal{C}_1 \,\&\, ..\, \&\, \mathcal{C}_n \,\&\, \mathcal{C} <: \mathcal{A} \end{array}}{\begin{array}{c} \Delta; \Gamma \vdash \textbf{trait}\,[\mathsf{self} : \mathcal{B}]\ \textbf{inherits}\, \overline{\mathcal{T}_i}^{i \in 1..n} \{\, \overline{l_j = \mathcal{T}_j'}^{j \in 1..m} \,\} : \mathcal{A} \Rightarrow \textbf{Trait}\,[\mathcal{B}, \mathcal{A}] \rightsquigarrow \\ \lambda \mathsf{self} : |\mathcal{B}|.\, (((\overline{(E_i\, \mathsf{self})}^{i \in 1..n}) ,, E) \end{array}}$$

$$\textsc{ST-new}\quad \frac{\begin{array}{c} \overline{\Delta; \Gamma \vdash \mathcal{T}_i \Rightarrow \textbf{Trait}\,[\mathcal{A}_i, \mathcal{B}_i] \rightsquigarrow E_i}^{i \in 1..n} \\ \overline{\mathcal{A} <: \mathcal{A}_i}^{i \in 1..n} \qquad \Delta \vdash \mathcal{B}_1 * \,..\, * \mathcal{B}_n \qquad \mathcal{B}_1 \,\&\, ..\, \&\, \mathcal{B}_n <: \mathcal{A} \end{array}}{\Delta; \Gamma \vdash \textbf{new}\,[\mathcal{A}](\, \overline{\mathcal{T}_i}^{i \in 1..n}\,) \Rightarrow \mathcal{A} \rightsquigarrow \textbf{letrec}\, \mathsf{self} : |\mathcal{A}| = \overline{(E_i\, \mathsf{self})}^{i \in 1..n}\ \textbf{in}\, \mathsf{self}}$$

$$\textsc{ST-forward}\quad \frac{\Delta; \Gamma \vdash \mathcal{T}_1 \Rightarrow \textbf{Trait}\,[\mathcal{A}, \mathcal{B}] \rightsquigarrow E_1 \qquad \Delta; \Gamma \vdash \mathcal{T}_2 \Leftarrow \mathcal{A} \rightsquigarrow E_2}{\Delta; \Gamma \vdash \mathcal{T}_1 \mathbin{\char`\^} \mathcal{T}_2 \Rightarrow \mathcal{B} \rightsquigarrow E_1\, E_2}$$

$\boxed{\Delta; \Gamma \vdash \mathcal{T} \Leftarrow \mathcal{A} \rightsquigarrow E}$ *(Checking)*

$$\textsc{ST-abs}\quad \frac{\Delta \vdash \mathcal{A} \qquad \Delta; \Gamma, x : \mathcal{A} \vdash \mathcal{T} \Leftarrow \mathcal{B} \rightsquigarrow E}{\Delta; \Gamma \vdash \lambda x. \mathcal{T} \Leftarrow \mathcal{A} \to \mathcal{B} \rightsquigarrow \lambda x.\, E} \qquad \textsc{ST-sub}\quad \frac{\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \mathcal{A} \rightsquigarrow E \qquad \mathcal{A} <: \mathcal{B} \qquad \Delta \vdash \mathcal{B}}{\Delta; \Gamma \vdash \mathcal{T} \Leftarrow \mathcal{B} \rightsquigarrow E}$$

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 $\overline{(E_i\ \mathsf{self})}^{i\in 1..n}$ means $E_1\ \mathsf{self}, , \ldots, , E_n\ \mathsf{self}$. 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 $\overline{(E_i\ \mathsf{self})}^{i\in 1..n}$, 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 and Palsberg'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 and Palsberg'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 Corollary 3 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 A 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:

$$|\textbf{Trait}\,[\mathcal{A},\mathcal{B}]| = |\mathcal{A}| \to |\mathcal{B}|$$

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 25. *If* $\Delta \vdash \mathcal{A}$ *then* $|\Delta| \vdash |\mathcal{A}|$.

Lemma 26. *If* $\mathcal{A} <: \mathcal{B}$ *then* $|\mathcal{A}| <: |\mathcal{B}|$.

Lemma 27. *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 14 (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 15 (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 13), 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 OO 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 and supports 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 from a textbook on 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 [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`, modelled 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 [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 OO 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:

$$\mathsf{combine} \in F[A] \times F[B] \rightarrow F[A \,\&\, B]$$

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`:

Types	τ	::=	$\mathsf{int} \mid \mathsf{bool}$	
Expressions	e	::=	$i \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e_1 \div e_2$	*natF*
		$\mid$	$\mathbb{B} \mid \textbf{if}\, e_1 \,\textbf{then}\, e_2 \,\textbf{else}\, e_3$	*boolF*
		$\mid$	$e_1 == e_2 \mid e_1 < e_2$	*compF*
		$\mid$	$e_1 \,\&\&\, e_2 \mid e_1 \,\vert\vert\, e_2$	*logicF*
		$\mid$	$x \mid \textbf{var}\, x = e_1; e_2$	*varF*
		$\mid$	$e_1\, e_2$	*funcF*
Programs	pgm	::=	$decl_1 \ldots decl_n\, e$	*funcF*
Functions	$decl$	::=	$\textbf{function}\, f(x : \tau)\{e\}$	*funcF*
Values	v	::=	$i \mid \mathbb{B}$	

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 the expressions, values and types provided by the features; 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.

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. Besides, we also define a combined algebra with the combined behavior of 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	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.

Finally, we acknowledge the limitation of our case study in that SLOC is just one metric and we have not measured any other metrics. 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.

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.

As far as we know, our work is the first to use logical relations to show the coherence for intersection types and the merge operator. The 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. They also did not study parametric polymorphism, which requires extra effort in the proof.

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:

$$\frac{}{\rho \vdash \rho} \qquad \frac{}{\vdash \top} \qquad \frac{\vdash B}{\vdash A \to B} \qquad \frac{\vdash A}{\vdash \{l : A\}} \qquad \frac{\Gamma, \Delta \vdash C}{\Gamma, \rho, \Delta \vdash C} \qquad \frac{\Gamma, \Delta \vdash C}{\Gamma, \top, \Delta \vdash C}$$

$$\frac{\Gamma, \Delta \vdash C}{\Gamma, A \to B, \Delta \vdash C} \qquad \frac{\Gamma, \Delta \vdash C}{\Gamma, \{l : B\}, \Delta \vdash C} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B} \qquad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, A \,\&\, B, \Delta \vdash C}$$

$$\frac{A \vdash A_1 \qquad \ldots \qquad A \vdash A_n \qquad B_1, \ldots, B_n \vdash B}{A_1 \to B_1, \ldots, A_n \to B_n \vdash A \to B} \qquad \frac{A_1, \ldots, A_n \vdash B}{\{l : A_1\}, \ldots, \{l : A_n\} \vdash \{l : B\}}$$

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 coercions and records. In the course of our mechanization we identified several mistakes in Pierce's proofs, as well as some important missing lemmas. 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 and the Merge Operator

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):

$$\frac{\Gamma \vdash \lambda x.\, p_2 : \theta_1 \to \theta_2}{\Gamma \vdash (p_1,, \lambda x.\, p_2) : \theta_1 \to \theta_2}$$

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 overloading functions. Interestingly, $\lambda\&$ has a special merge operator that works on functions only. Like ours, they also impose well-formedness conditions on the formation of a (functional) merge. However, those conditions operate on function types only, and it is not clear how to generalize them to arbitrary types. Dunfield [2014] shows significant expressiveness of type systems with (unrestricted) intersection types and a merge operator. However, his calculus lacks coherence. The limitation was addressed by Oliveira et al. [2016], who introduced disjointness to ensure coherence.

Compared to prior work on disjoint intersection types, the approach in this thesis simplifies type systems with disjoint intersection types by removing several restrictions. Furthermore, our calculi adopt a more powerful subtyping relation based on BCD subtyping, which in turn requires the use of a more sophisticated proof method for proving coherence. On a pragmatic note, dynamic inheritance, self-references and abstract methods are all missing from prior work, but, as shown in this thesis, they can be encoded using an elaboration that employs ideas from the denotational model of inheritance [Cook and Palsberg 1989].

9.4 Intersection Types and Polymorphism

Pierce [1991] proposed $\mathsf{F}_\wedge$, a calculus combining intersection types and bounded quantification. $\mathsf{F}_\wedge$ also adopts a BCD-like subtyping relation. Moreover, since the $\forall$-quantifier behaves somewhat like an arrow constructor, Pierce added a new rule which allows intersections to be distributed over quantifiers on the right-hand side.

$$\frac{}{(\forall(\alpha <: A).\, B_1) \,\&\, (\forall(\alpha <: A).\, B_2) <: \forall(\alpha <: A).\, B_1 \,\&\, B_2}$$

In F_i^+, we do not have this distributivity rule, but are thinking to add a similar rule, as shown below:

$$\textsc{distPoly}\quad \frac{}{(\forall(\alpha * A).\, B_1) \,\&\, (\forall(\alpha * A).\, B_2) <: \forall(\alpha * A).\, B_1 \,\&\, B_2}$$

This rule will add extra expressiveness to F_i^+, e.g., we can compose algebras with polymorphic components:

$$\{l : \forall\alpha.\, \alpha \to A\} \,\&\, \{l : \forall\alpha.\, \alpha \to B\} <: \{l : \forall\alpha.\, \alpha \to A \,\&\, B\}$$

We expect this rule would not pose any difficulties in terms of the coherence proof. Like the elaboration semantics of F_i^+, 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. Compared to F_i^+, their intersections are used between function types, allowing overloading of types, as shown below (altering their syntax slightly):

```
even :: (Int → Bool) & ((a \ Int) → (a \ Int))
even x = case x of
           | Int → x `mod` 2 == 0
           | _   → x
```

The above function operates differently according to the type of the argument: it checks whether an argument is an integer; if it is so it returns whether the integer is even or not, otherwise it returns the argument as it received. Note that type difference (i.e., `a \ Int`) is crucial to ensure no ambiguity in the domain types of two functions. In F_i^+, we cannot express this kind of intersections. However, F_i^+ allows some other intersections (e.g., $(\mathsf{Int} \to \mathsf{Bool}) \,\&\, (\mathsf{Int} \to \mathsf{Int})$) that are not allowed in their system. Nevertheless, both systems need to express negative information about type variables: in their system type difference (e.g., `a \ Int`) achieves this, whereas in F_i^+ we use disjointness constraints (e.g., `a * Int`). On a more theoretical note, Castagna et al. [2014] adopt the *semantic* approach for defining the subtyping relation, where one first chooses a model, and an interpretation of types as subsets of the model, then the subtyping relation can be defined as the inclusion of denoted sets. The benefit of this approach, compared with the more used *syntactic* approach, is that the subtyping relation is by definition *complete*. In that regard, their subtyping relation thus completely subsumes BCD subtyping.

The combination of intersection types, a merge operator and parametric polymorphism, while achieving coherence was first studied in the F_i calculus [Alpuim et al. 2017], which servers as a foundation for our F_i^+ calculus. Compared to F_i, the essential novelty is a BCD subtyping, and a more powerful proof method for proving coherence. As far as we know, we are the first to study the metatheory of the combination of BCD subtyping, parametric polymorphism and the merge operator.

9.5 Intersection Types and Multiple Inheritance

Compagnoni and Pierce [1996] proposed a λ-calculus $\mathsf{F}_\wedge^\omega$, an extension of System F^ω with intersection types to model multiple inheritance. $\mathsf{F}_\wedge^\omega$ allows arbitrary finite intersections, where all the type members must have the same kind. On the language side, modern object-oriented languages such as Scala, TypeScript, Flow, Ceylon, and Grace have adopted some form of intersection types. Notably, the DOT calculus [Amin et al. 2012; Rompf and Amin 2016]—a new type-theoretic foundation for Scala—has a native support for intersection types. Generally speaking, the most significant difference between our calculi and those languages/calculi is that they do not have an explicit introduction form of intersection types, like our merge operator. The lack of a native merge operator leads to some awkward and type-unsafe solutions for defining a merge operator in those languages. As noted by Alpuim et al. [2017], one important use of intersection types in TypeScript is the following function:

```
function extend<T, U>(first: T, second : U) : T & U {...}
```

which is analogous to our merge operator in that it takes two objects and produces an object with the intersection of the types of the argument objects. The implementation of `extend` relies on low-level (and type-unsafe) features of JavaScript. Similar encodings have also been proposed for Scala to enable applications where the merge operator plays a fundamental role [Oliveira et al. 2013; Rendel et al. 2014]. As we have shown in Section 4.1, with disjointness constraints and a built-in merge operator, a type-safe and conflict-free `extend` function can be naturally defined.

9.6 Row Polymorphism and Extensible Records

Row polymorphism, first proposed by Wand [1987], was intended 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]. Cardelli and Mitchell [1989] define three primitive operations on records: *selection*, *restriction* and *extension*. In our calculi, the merge operator can be regarded as a generalization of record extension/concatenation, and selection is also supported natively. In contrast to most record systems, restriction is not a primitive operation in our calculi, but can be simulated via subtyping. According to Leijen [2005], when it comes to extension, record calculi can be divided into those that support *free* extension, and those that support *strict* extension. The former allows duplicate labels to coexist, whereas the latter does not. As pointed out by Alpuim et al. [2017], our calculi can be thought as a hybrid of these two approaches: we allow duplicate labels as long as their types are disjoint. This is more flexible then strict extension, but less expressive than Leijen's system where it also accepts duplicate fields even when their types are overlapping. We refer to Alpuim et al. [2017] for a detailed account of encodings of polymorphic extensible records using disjoint polymorphism.

Row polymorphism alone cannot express the merge operator, as it only operates on records (possibly with statically unknown fields). This essentially limits its applications to extensible designs (such as defining Object Algebra combinators in Chapter 8). In this sense, we believe disjoint polymorphism is more expressive than row polymorphism. It would be interesting to rigorously study the relationship between disjoint polymorphism and row polymorphism, whether the former subsumes the latter. We have some further discussion about this point in Section 10.3.

9.7 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. Unlike disjoint polymorphism, row polymorphism prohibits forgetting class members. While this is reasonable in the setting of mixins, in some cases, a function taking one class as an argument can return another class that has fewer methods. For example, in SEDEL we can write:

```
foo [A * {bar : String}] (t : Trait[{bar : String} & A]) : Trait[A] = t;
```

where `foo` drops `bar` from its argument trait `t`, which is impossible to express using row polymorphism. 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.

9.8 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 OO 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.

9.9 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 traditional 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.

9.10 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. Although 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 *object 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 unsound. 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.

9.11 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 constructors*. 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.

9.12 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 *implementor-side* data abstraction by allowing the implementor 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 are 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 On 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]:

$$\begin{tikzcd}
[\![\Gamma]\!] \arrow[r, bend left, "{[\![D_1 :: \Gamma \vdash E : A]\!]}"] \arrow[r, bend right, "{[\![D_2 :: \Gamma \vdash E : A]\!]}"'] & [\![A]\!]
\end{tikzcd}$$

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 17 (Type Equivalence)**.** Two types A and B are equivalent, written $A \approx B$, when $A <: B$ and $B <: A$.

Definition 18 (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 4.

$$
\begin{aligned}
A_1 \,\&\, (A_2 \,\&\, A_3) &\approx (A_1 \,\&\, A_2) \,\&\, A_3 \\
\top \,\&\, A &\approx A \\
A \,\&\, \top &\approx A \\
A_1 \,\&\, A_2 &\approx A_2 \,\&\, A_1 \\
A \,\&\, A &\approx A \\
\{l : A_1 \,\&\, A_2\} &\approx \{l : A_1\} \,\&\, \{l : A_2\} \\
A \to A_1 \,\&\, A_2 &\approx (A \to A_1) \,\&\, (A \to A_2) \\
\{l : \top\} &\approx \top \\
A \to \top &\approx \top
\end{aligned}
$$

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 5.

$$\begin{aligned}
A \sqcup B &= B \sqcup A \\
A \sqcup \top &= \top \\
A_1 \sqcup (A_2 \,\&\, A_3) &= (A_1 \sqcup A_2) \,\&\, (A_1 \sqcup A_3) \\
\rho \sqcup \{l : A\} &= \top \\
\rho \sqcup (A_1 \to A_2) &= \top \\
\{l : A\} \sqcup (A_1 \to A_2) &= \top \\
\{l : A_1\} \sqcup \{l : A_2\} &= \{l : A_1 \sqcup A_2\} \\
\{l_1 : A_1\} \sqcup \{l_2 : A_2\} &= \top \quad \textit{when } l_1 \neq l_2 \\
(A_1 \to A_1') \sqcup (A_2 \to A_2') &= (A_1 \,\&\, A_2) \to (A_1' \sqcup A_2')
\end{aligned}$$

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 16. *$A * B$ if and only if $A \sqcup B \approx \top$.*

Proof. By induction on the derivation of disjointness. An interesting case is rule D-ARR

$$\textsc{D-arr}\quad \frac{A_2 * B_2}{A_1 \to A_2 * B_1 \to B_2}$$

$A_2 \sqcup B_2 \approx \top$	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 5
$(A_1 \to A_2) \sqcup (B_1 \to B_2) \approx (A_1 \,\&\, B_1) \to \top$	By above equality
$(A_1 \,\&\, B_1) \to \top \approx \top$	By Proposition 4
$(A_1 \to A_2) \sqcup (B_1 \to B_2) \approx \top$	By above equality

□

Remark. We can view Theorem 16 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 $\pi_2 : [\![A_1 \,\&\, A_2]\!] \to [\![A_2]\!]$

$$
\begin{tikzcd}
{[\![A_1]\!]} & & {[\![A_2]\!]} \\
& {[\![A_1 \,\&\, A_2]\!]} \arrow[ul, "\pi_1"] \arrow[ur, "\pi_2"'] &
\end{tikzcd}
$$

2. For any types A_1 and A_2, there exists a least upper bound $A_1 \sqcup A_2$ (Proposition 5), 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:

$$
\begin{tikzcd}
& {[\![A_1 \sqcup A_2]\!]} & \\
{[\![A_1]\!]} \arrow[ur, "{[\![A_1 <: A_1 \sqcup A_2]\!]}"] & & {[\![A_2]\!]} \arrow[ul, "{[\![A_2 <: A_1 \sqcup A_2]\!]}"'] \\
& {[\![A_1 \,\&\, A_2]\!]} \arrow[ul, "\pi_1"] \arrow[ur, "\pi_2"'] &
\end{tikzcd}
$$

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 $\llbracket A \rrbracket$ to $\llbracket A_1 \,\&\, A_2 \rrbracket$. Call this μ_0. The following diagram should commute:

$$
\begin{tikzcd}
& \llbracket A_1 \sqcup A_2 \rrbracket & \\
\llbracket A_1 \rrbracket \arrow[ur, "\llbracket A_1 <: A_1 \sqcup A_2 \rrbracket"] & & \llbracket A_2 \rrbracket \arrow[ul, "\llbracket A_2 <: A_1 \sqcup A_2 \rrbracket"'] \\
& \llbracket A_1 \,\&\, A_2 \rrbracket \arrow[ul, "\pi_1"'] \arrow[ur, "\pi_2"] & \\
& \llbracket A \rrbracket \arrow[u, "\mu_0"] \arrow[uul, "\llbracket A <: A_1 \rrbracket"] \arrow[uur, "\llbracket A <: A_2 \rrbracket"'] &
\end{tikzcd}
$$

4. Furthermore, in the above diagram, we replace $\llbracket A \rrbracket$ by an arbitrary object s and $\llbracket A <: A_1 \rrbracket$ and $\llbracket A <: A_1 \rrbracket$ 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 $\llbracket A_1 \,\&\, A_2 \rrbracket$ to be unique. Specifically, we define $\llbracket A_1 \,\&\, A_2 \rrbracket$ by requiring the following diagram must commute:

$$
\begin{tikzcd}
& \llbracket A_1 \sqcup A_2 \rrbracket & \\
\llbracket A_1 \rrbracket \arrow[ur, "\llbracket A_1 <: A_1 \sqcup A_2 \rrbracket"] & & \llbracket A_2 \rrbracket \arrow[ul, "\llbracket A_2 <: A_1 \sqcup A_2 \rrbracket"'] \\
& \llbracket A_1 \,\&\, A_2 \rrbracket \arrow[ul, "\pi_1"'] \arrow[ur, "\pi_2"] & \\
& s \arrow[u, dotted, "\mu_0"] \arrow[uul, "f_1"] \arrow[uur, "f_2"'] &
\end{tikzcd}
$$

Thus we have defined $\llbracket A_1 \,\&\, A_2 \rrbracket$ to be the *pullback* of $\llbracket A_1 \rrbracket$, $\llbracket A_2 \rrbracket$ and $\llbracket A_1 \sqcup A_2 \rrbracket$.

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 16). Also we have $\llbracket \top \rrbracket = 1$—i.e., the terminal object. By specializing $\llbracket A_1 \sqcup A_2 \rrbracket$ to be the terminal object ($\llbracket A_1 <: A_1 \sqcup A_2 \rrbracket$ and $\llbracket A_2 <: A_1 \sqcup A_2 \rrbracket$ are then uniquely determined), then the pullback "degenerates" to the *product* of $\llbracket A_1 \rrbracket$ and $\llbracket A_2 \rrbracket$. In other words, the interpretation of disjoint intersection types is given by the following theorem:

Theorem 17. *If* $A_1 * A_2$ *then* $[\![A_1 \,\&\, A_2]\!] \cong [\![A_1]\!] \times [\![A_2]\!]$.

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 On 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:

$$\textsc{IS-allL}\quad \frac{\Delta \vdash t * A_1 \qquad \Delta \vdash [t/\alpha]A_2 <: B}{\Delta \vdash \forall (\alpha * A_1).\, A_2 <: B} \qquad\qquad \textsc{IS-allR}\quad \frac{\Delta, \alpha * B_1 \vdash A <: B_2}{\Delta \vdash A <: \forall (\alpha * B_1).\, B_2}$$

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:

$$\dfrac{\dfrac{\dfrac{A_2 <: A_1}{\alpha * A_2 \vdash \alpha * A_1}\ \text{FD-tvarL} \qquad \alpha * A_2 \vdash B_1 <: B_2}{\alpha * A_2 \vdash \forall(\alpha * A_1).\, B_1 <: B_2}\ \text{IS-allL}}{\cdot \vdash \forall(\alpha * A_1).\, B_1 <: \forall(\alpha * A_2).\, B_2}\ \text{IS-allR}$$

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:

$$\text{FD-implicit}\quad \dfrac{\Delta \vdash t_1 * A_1 \qquad \Delta \vdash [t_1/\alpha]A_2 * B_2}{\Delta \vdash \forall(\alpha * A_1).\, A_2 * B_2}$$

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:

$$\text{FD-implicit}\quad \dfrac{\forall t_1.\, \Delta \vdash t_1 * A_1 \Longrightarrow \Delta \vdash [t_1/\alpha]A_2 * B_2}{\Delta \vdash \forall(\alpha * A_1).\, A_2 * B_2}$$

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:

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

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:

$$\frac{\Delta ; \Gamma \vdash E_1 \Rightarrow A \rightsquigarrow e_1 \qquad \Delta \vdash A \rhd A_1 \to A_2 \qquad \Delta ; \Gamma \vdash E_2 \Leftarrow A_1 \rightsquigarrow e_2}{\Delta ; \Gamma \vdash E_1\, E_2 \Rightarrow A_2 \rightsquigarrow e_1\, e_2} \;\textsc{FT-appI}$$

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:

$$\frac{\Delta \vdash t * A_1 \qquad \Delta \vdash [t/\alpha] A_1 \rhd B_1 \to B_2}{\Delta \vdash \forall (\alpha * A_1) .\, A_2 \rhd B_1 \to B_2} \;\textsc{m-forall} \qquad \frac{}{\Delta \vdash A_1 \to A_2 \rhd A_1 \to A_2} \;\textsc{m-arr}$$

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 $A_1 \to A_2$ 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 9.6, row polymorphism alone cannot express the merge operator. 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):

$$\begin{aligned}\Lambda\alpha_1\#(\{l_1 : \mathsf{Int}\}, \{l_2 : \mathsf{Int}\}).\ &\Lambda\alpha_2\#(\alpha_1, \{l_1 : \mathsf{Int}\}, \{l_2 : \mathsf{Int}\}).\\ &\lambda x_1 : (\alpha_1 \| \{l_1 : \mathsf{Int}\}).\ \lambda x_2 : (\alpha_2 \| \{l_2 : \mathsf{Int}\}).\ x_1 \| x_2\end{aligned}$$

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 associated 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:

$$\begin{aligned} &\Lambda(\alpha_1 * \{l_1 : \mathsf{Int}\} \,\&\, \{l_2 : \mathsf{Int}\}).\ \Lambda(\alpha_2 * \alpha_1 \,\&\, \{l_1 : \mathsf{Int}\} \,\&\, \{l_2 : \mathsf{Int}\}). \\ &\qquad\qquad \lambda x_1 : \alpha_1 \,\&\, \{l_1 : \mathsf{Int}\}.\ \lambda x_2 : \alpha_2 \,\&\, \{l_2 : \mathsf{Int}\}.\ x_1 ,, x_2 \end{aligned}$$

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" $\mathsf{mergeAny}$ 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.

We believe F_i^+ completely subsumes $\lambda^{\|}$. To support this claim, we may show a type directed translation from $\lambda^{\|}$ to F_i^+. However, there could be many feasible translations, some of which may even discard important information. For example, we could easily translate $\lambda^{\|}$ to System F, ignoring all compatibility conditions. Thus we need a stronger argument: a bisimulation property between $\lambda^{\|}$ and (a subset of) F_i^+: (1) the translation from $\lambda^{\|}$ to F_i^+ is type-safe (i.e., it type check in F_i^+), (2) and the translation from a subset of F_i^+ to $\lambda^{\|}$ is type-safe (i.e., it type check in $\lambda^{\|}$).

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].

$$\frac{\Sigma, X <: Y \vdash A <: B}{\Sigma \vdash \mu X.\, A <: \mu Y.\, B}\ \textsc{RS-amber} \qquad \frac{(X <: Y) \in \Sigma}{\Sigma \vdash X <: Y}\ \textsc{RS-var}$$

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. 4.3, 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 $\mu X.\, X \to \mathsf{Int}$ and $\mu Y.\, Y \to \mathsf{Int}$. 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 $X \to \mathsf{Int}$ and $Y \to \mathsf{Int}$ 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

$X \to \mathsf{Int}$ and $Y \to \mathsf{Int}$ are not disjoint. So we have $\mu X.\, X \to \mathsf{Int}$ and $\mu Y.\, Y \to \mathsf{Int}$ 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 $A_1 \to A_2$, 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:

D-rec
$$\frac{A * B}{\mu X.\, A * \mu Y.\, B}$$

We also need a few more disjointness axioms:

Dax-intRec
$$\frac{}{\rho *_{ax} \mu X.\, B}$$

Dax-rcdRec
$$\frac{}{\{l : A\} *_{ax} \mu X.\, B}$$

Dax-arrRec
$$\frac{}{A_1 \to A_2 *_{ax} \mu X.\, B}$$

Dax-intRVar
$$\frac{}{\rho *_{ax} X}$$

Dax-rcdRVar
$$\frac{}{\{l : A\} *_{ax} X}$$

Dax-arrRVar
$$\frac{}{A_1 \to A_2 *_{ax} X}$$

Dax-recRVar
$$\frac{}{\mu Y.\, A *_{ax} X}$$

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.

$$\textsc{union}\ \frac{A <: C \qquad B <: C}{A|B <: C} \qquad \textsc{unionL}\ \frac{}{A <: A|B} \qquad \textsc{unionR}\ \frac{}{B <: A|B}$$

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, $\mathsf{Int} \,\&\, \mathsf{Bool}$ can reach to $\mathsf{Int}|\mathsf{Bool}$ via two paths, as shown below, each leading to semantically different translations.

$$\begin{tikzcd}
 & \mathsf{Int} \arrow[dr, "<:"] & \\
\mathsf{Int} \,\&\, \mathsf{Bool} \arrow[ur, "<:"] \arrow[dr, "<:"'] & & \mathsf{Int}|\mathsf{Bool} \\
 & \mathsf{Bool} \arrow[ur, "<:"'] &
\end{tikzcd}$$

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]).

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, enabling a simple solution to the expression problem. In order to prove coherence, we have introduced a powerful proof method called the canonicity relation based on logical relations.
- The F_i^+ calculus, building on λ_i^+, supports parametric polymorphism. We have shown that it can express a very dynamic (and conflict-free) form of composition, which could serve as a foundation for more sophisticated compositional models. 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. Through a case study, we have shown the usefulness of F_i^+ in building highly reusable software components using a improved form of Object Algebras. The case study demonstrates that the state-of-art encodings of extensible designs are greatly improved by F_i^+.

Throughout the thesis, we have demonstrated 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!

A Proofs about SEDEL

Lemma 25. *If* $\Delta \vdash \mathcal{A}$ *then* $|\Delta| \vdash |\mathcal{A}|$.

Proof. By simple induction on the derivation of the judgment. □

Lemma 26. *If* $\mathcal{A} <: \mathcal{B}$ *then* $|\mathcal{A}| <: |\mathcal{B}|$.

Proof. Most of them are straightforward. We only show rule TS-TRAIT.

- $$\textsc{TS-trait}\quad \frac{\mathcal{B}_1 <: \mathcal{A}_1 \qquad \mathcal{A}_2 <: \mathcal{B}_2}{\textbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] <: \textbf{Trait}\,[\mathcal{B}_1, \mathcal{B}_2]}$$

$\lvert\mathcal{B}_1\rvert <: \lvert\mathcal{A}_1\rvert$	By i.h.
$\lvert\mathcal{A}_2\rvert <: \lvert\mathcal{B}_2\rvert$	By i.h.
$\lvert\mathcal{A}_1\rvert \to \lvert\mathcal{A}_2\rvert <: \lvert\mathcal{B}_1\rvert \to \lvert\mathcal{B}_2\rvert$	By rule S-ARR

□

Lemma 28. *If* $\mathcal{A} *_{ax} \mathcal{B}$ *then* $|\mathcal{A}| *_{ax} |\mathcal{B}|$.

Proof. Note that $|\textbf{Trait}\,[\mathcal{A}, \mathcal{B}]| = |\mathcal{A}| \to |\mathcal{B}|$, the rest are immediate. □

Lemma 27. *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.
- $$\textsc{SD-tvarL}\quad \frac{(\alpha * \mathcal{A}) \in \Delta \qquad \mathcal{A} <: \mathcal{B}}{\Delta \vdash \alpha * \mathcal{B}}$$

$\lvert\mathcal{A}\rvert <: \lvert\mathcal{B}\rvert$	By Lemma 26
$(\alpha * \mathcal{A}) \in \Delta$	Given
$(\alpha * \lvert\mathcal{A}\rvert) \in \lvert\Delta\rvert$	Above
$\lvert\Delta\rvert \vdash \alpha * \lvert\mathcal{B}\rvert$	By rule FD-TVARL

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

$\lvert\mathcal{A}\rvert <: \lvert\mathcal{B}\rvert$	By Lemma 26
$(\alpha * \mathcal{A}) \in \Delta$	Given
$(\alpha * \lvert\mathcal{A}\rvert) \in \lvert\Delta\rvert$	Above
$\lvert\Delta\rvert \vdash \lvert\mathcal{B}\rvert * \alpha$	By rule FD-TVARR

- $$\frac{\Delta, \alpha * \mathcal{A}_1 \,\&\, \mathcal{A}_2 \vdash \mathcal{B}_1 * \mathcal{B}_2}{\Delta \vdash \forall(\alpha * \mathcal{A}_1).\, \mathcal{B}_1 * \forall(\alpha * \mathcal{A}_2).\, \mathcal{B}_2}\ \textsc{SD-forall}$$

$\lvert\Delta\rvert, \alpha * \lvert\mathcal{A}_1\rvert \,\&\, \lvert\mathcal{A}_2\rvert \vdash \lvert\mathcal{B}_1\rvert * \lvert\mathcal{B}_2\rvert$	By i.h.
$\lvert\Delta\rvert \vdash \forall(\alpha * \lvert\mathcal{A}_1\rvert).\, \lvert\mathcal{B}_1\rvert * \forall(\alpha * \lvert\mathcal{A}_2\rvert).\, \lvert\mathcal{B}_2\rvert$	By rule FD-FORALL

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

$\lvert\Delta\rvert \vdash \lvert\mathcal{A}\rvert * \lvert\mathcal{B}\rvert$	By i.h.
$\lvert\Delta\rvert \vdash \{l : \lvert\mathcal{A}\rvert\} * \{l : \lvert\mathcal{B}\rvert\}$	By rule FD-RCDEQ

- $$\frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \mathcal{A}_1 \to \mathcal{A}_2 * \mathcal{B}_1 \to \mathcal{B}_2}\ \textsc{SD-arr}$$

$\lvert\Delta\rvert \vdash \lvert\mathcal{A}_2\rvert * \lvert\mathcal{B}_2\rvert$	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

- $$\frac{\Delta \vdash \mathcal{A}_1 * \mathcal{B} \qquad \Delta \vdash \mathcal{A}_2 * \mathcal{B}}{\Delta \vdash \mathcal{A}_1 \,\&\, \mathcal{A}_2 * \mathcal{B}}\ \textsc{SD-andL}$$

$\lvert\Delta\rvert \vdash \lvert\mathcal{A}_1\rvert * \lvert\mathcal{B}\rvert$	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

- $$\frac{\Delta \vdash \mathcal{A} * \mathcal{B}_1 \qquad \Delta \vdash \mathcal{A} * \mathcal{B}_2}{\Delta \vdash \mathcal{A} * \mathcal{B}_1 \,\&\, \mathcal{B}_2}\ \textsc{SD-andR}$$

$\lvert\Delta\rvert \vdash \lvert\mathcal{A}\rvert * \lvert\mathcal{B}_1\rvert$	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

- $$\frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \textbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] * \textbf{Trait}\,[\mathcal{B}_1, \mathcal{B}_2]}\ \textsc{SD-trait}$$

$\lvert\Delta\rvert \vdash \lvert\mathcal{A}_2\rvert * \lvert\mathcal{B}_2\rvert$	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

- $$\frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \textbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] * \mathcal{B}_1 \to \mathcal{B}_2}\ \textsc{SD-traitArr1}$$

$\lvert\Delta\rvert \vdash \lvert\mathcal{A}_2\rvert * \lvert\mathcal{B}_2\rvert$	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

- $$\frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \mathcal{A}_1 \to \mathcal{A}_2 * \textbf{Trait}\,[\mathcal{B}_1, \mathcal{B}_2]}\ \textsc{SD-traitArr2}$$

$\lvert\Delta\rvert \vdash \lvert\mathcal{A}_2\rvert * \lvert\mathcal{B}_2\rvert$	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

- $$\frac{\mathcal{A} *_{ax} \mathcal{B}}{\Delta \vdash \mathcal{A} * \mathcal{B}}\ \text{SD-AX}$$

$\lvert\mathcal{A}\rvert *_{ax} \lvert\mathcal{B}\rvert$	By Lemma 28
$\lvert\Delta\rvert \vdash \lvert\mathcal{A}\rvert * \lvert\mathcal{B}\rvert$	By rule FD-AX

□

Theorem 14 (Type-safe translation). *We have that:*

- *If* $\Delta; \Gamma \vdash \mathcal{T} \Rightarrow \mathcal{A} \rightsquigarrow E$ *then* $\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash E \Rightarrow \lvert\mathcal{A}\rvert$.
- *If* $\Delta; \Gamma \vdash \mathcal{T} \Leftarrow \mathcal{A} \rightsquigarrow E$ *then* $\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash E \Leftarrow \lvert\mathcal{A}\rvert$.

Proof. By induction on the typing judgment.

- Rules ST-TOP, ST-INT, and ST-VAR are immediate.
- $$\frac{\Delta; \Gamma \vdash \mathcal{T} \Leftarrow \mathcal{A} \rightsquigarrow E}{\Delta; \Gamma \vdash \mathcal{T} : \mathcal{A} \Rightarrow \mathcal{A} \rightsquigarrow E : \lvert\mathcal{A}\rvert}\ \text{ST-ANNO}$$

$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash E \Leftarrow \lvert\mathcal{A}\rvert$	By i.h.
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash E : \lvert\mathcal{A}\rvert \Rightarrow \lvert\mathcal{A}\rvert$	By rule FT-ANNO

- $$\frac{\Delta; \Gamma \vdash \mathcal{T}_1 \Rightarrow \mathcal{A}_1 \to \mathcal{A}_2 \rightsquigarrow E_1 \qquad \Delta; \Gamma \vdash \mathcal{T}_2 \Leftarrow \mathcal{A}_1 \rightsquigarrow E_2}{\Delta; \Gamma \vdash \mathcal{T}_1\, \mathcal{T}_2 \Rightarrow \mathcal{A}_2 \rightsquigarrow E_1\, E_2}\ \text{ST-APP}$$

$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash E_1 \Rightarrow \lvert\mathcal{A}_1\rvert \to \lvert\mathcal{A}_2\rvert$	By i.h.
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash E_2 \Leftarrow \lvert\mathcal{A}_1\rvert$	By i.h.
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash E_1\, E_2 \Rightarrow \lvert\mathcal{A}_2\rvert$	By rule FT-APP

- $$\frac{\Delta;\Gamma\vdash\mathcal{T}\ \Rightarrow\ \forall(\alpha*\mathcal{B}_1).\,\mathcal{B}_2\rightsquigarrow E \qquad \Delta\vdash\mathcal{A} \qquad \Delta\vdash\mathcal{A}*\mathcal{B}_1}{\Delta;\Gamma\vdash\mathcal{T}\,\mathcal{A}\ \Rightarrow\ [\mathcal{A}/\alpha]\mathcal{B}_2\rightsquigarrow E\,|\mathcal{A}|}\ \textsc{ST-tapp}$$

$\lvert\Delta\rvert;\lvert\Gamma\rvert\vdash E\ \Rightarrow\ \forall(\alpha*\lvert\mathcal{B}_1\rvert).\,\lvert\mathcal{B}_2\rvert$	By i.h.
$\lvert\Delta\rvert\vdash\lvert\mathcal{A}\rvert$	By Lemma 25
$\lvert\Delta\rvert\vdash\lvert\mathcal{A}\rvert*\lvert\mathcal{B}_1\rvert$	By Lemma 27
$\lvert\Delta\rvert;\lvert\Gamma\rvert\vdash E\,\lvert\mathcal{A}\rvert\ \Rightarrow\ [\lvert\mathcal{A}\rvert/\alpha]\lvert\mathcal{B}_2\rvert$	By rule FT-tapp

- $$\frac{\Delta;\Gamma\vdash\mathcal{T}_1\ \Rightarrow\ \mathcal{A}\rightsquigarrow E_1 \qquad \Delta;\Gamma\vdash\mathcal{T}_2\ \Rightarrow\ \mathcal{B}\rightsquigarrow E_2 \qquad \Delta\vdash\mathcal{A}*\mathcal{B}}{\Delta;\Gamma\vdash\mathcal{T}_1\,,,\,\mathcal{T}_2\ \Rightarrow\ \mathcal{A}\,\&\,\mathcal{B}\rightsquigarrow E_1\,,,\,E_2}\ \textsc{ST-merge}$$

$\lvert\Delta\rvert;\lvert\Gamma\rvert\vdash E_1\ \Rightarrow\ \lvert\mathcal{A}\rvert$	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 27
$\lvert\Delta\rvert;\lvert\Gamma\rvert\vdash E_1\,,,\,E_2\ \Rightarrow\ \lvert\mathcal{A}\rvert\,\&\,\lvert\mathcal{B}\rvert$	By rule FT-merge

- $$\frac{\Delta;\Gamma\vdash\mathcal{T}\ \Rightarrow\ \mathcal{A}\rightsquigarrow E}{\Delta;\Gamma\vdash\{l=\mathcal{T}\}\ \Rightarrow\ \{l:\mathcal{A}\}\rightsquigarrow\{l=E\}}\ \textsc{ST-rcd}$$

$\lvert\Delta\rvert;\lvert\Gamma\rvert\vdash E\ \Rightarrow\ \lvert\mathcal{A}\rvert$	By i.h.
$\lvert\Delta\rvert;\lvert\Gamma\rvert\vdash\{l=E\}\ \Rightarrow\ \{l:\lvert\mathcal{A}\rvert\}$	By rule FT-rcd

- $$\frac{\Delta;\Gamma\vdash\mathcal{T}\ \Rightarrow\ \{l:\mathcal{A}\}\rightsquigarrow E}{\Delta;\Gamma\vdash\mathcal{T}.l\Rightarrow\ \mathcal{A}\rightsquigarrow E.l}\ \textsc{ST-proj}$$

$\lvert\Delta\rvert;\lvert\Gamma\rvert\vdash E\ \Rightarrow\ \{l:\lvert\mathcal{A}\rvert\}$	By i.h.
$\lvert\Delta\rvert;\lvert\Gamma\rvert\vdash E.l\Rightarrow\ \lvert\mathcal{A}\rvert$	By rule FT-proj

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

$\lvert\Delta\rvert \vdash \lvert\mathcal{A}\rvert$	By Lemma 25
$\lvert\Delta\rvert, \alpha * \lvert\mathcal{A}\rvert; \lvert\Gamma\rvert \vdash E \Rightarrow \lvert\mathcal{B}\rvert$	By i.h.
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash \Lambda(\alpha * \lvert\mathcal{A}\rvert).\, E \Rightarrow \forall(\alpha * \lvert\mathcal{A}\rvert).\, \lvert\mathcal{B}\rvert$	By rule FT-TABS

- ST-LETREC
$$\frac{\Delta; \Gamma, x : \mathcal{A} \vdash \mathcal{T}_1 \Leftarrow \mathcal{A} \rightsquigarrow E_1 \qquad \Delta; \Gamma, x : \mathcal{A} \vdash \mathcal{T}_2 \Rightarrow \mathcal{B} \rightsquigarrow E_2}{\Delta; \Gamma \vdash \mathbf{letrec}\, x : \mathcal{A} = \mathcal{T}_1 \,\mathbf{in}\, \mathcal{T}_2 \Rightarrow \mathcal{B} \rightsquigarrow \mathbf{letrec}\, x : |\mathcal{A}| = E_1 \,\mathbf{in}\, E_2}$$

$\lvert\Delta\rvert; \lvert\Gamma\rvert, x : \lvert\mathcal{A}\rvert \vdash E_1 \Leftarrow \lvert\mathcal{A}\rvert$	By i.h.
$\lvert\Delta\rvert; \lvert\Gamma\rvert, x : \lvert\mathcal{A}\rvert \vdash E_2 \Rightarrow \lvert\mathcal{B}\rvert$	By i.h.
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash \mathbf{letrec}\, x : \lvert\mathcal{A}\rvert = E_1 \,\mathbf{in}\, E_2 \Rightarrow \lvert\mathcal{B}\rvert$	

- ST-NEW
$$\frac{\begin{array}{c}\overline{\Delta; \Gamma \vdash \mathcal{T}_i \Rightarrow \mathbf{Trait}\,[\mathcal{A}_i, \mathcal{B}_i] \rightsquigarrow E_i}^{i \in 1..n} \\ \overline{\mathcal{A} <: \mathcal{A}_i}^{i \in 1..n} \qquad \Delta \vdash \mathcal{B}_1 * .. * \mathcal{B}_n \qquad \mathcal{B}_1 \,\&\, .. \,\&\, \mathcal{B}_n <: \mathcal{A}\end{array}}{\Delta; \Gamma \vdash \mathbf{new}\,[\mathcal{A}](\,\overline{\mathcal{T}_i}^{i \in 1..n}\,) \Rightarrow \mathcal{A} \rightsquigarrow \mathbf{letrec}\,\mathsf{self} : |\mathcal{A}| = \overline{(E_i\,\mathsf{self})}^{i \in 1..n} \,\mathbf{in}\,\mathsf{self}}$$

$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash E_i \Rightarrow \lvert\mathcal{A}_i\rvert \to \lvert\mathcal{B}_i\rvert$	By i.h.
$\lvert\mathcal{A}\rvert <: \lvert\mathcal{A}_i\rvert$	By Lemma 26
$\lvert\Delta\rvert \vdash \lvert\mathcal{B}_1\rvert * .. * \lvert\mathcal{B}_n\rvert$	By Lemma 27
$\lvert\mathcal{B}_1\rvert \,\&\, .. \,\&\, \lvert\mathcal{B}_n\rvert <: \lvert\mathcal{A}\rvert$	By Lemma 26
$\lvert\Delta\rvert; \lvert\Gamma\rvert, \mathsf{self} : \lvert\mathcal{A}\rvert \vdash \mathsf{self} \Rightarrow \lvert\mathcal{A}\rvert$	By rule FT-VAR
$\lvert\Delta\rvert; \lvert\Gamma\rvert, \mathsf{self} : \lvert\mathcal{A}\rvert \vdash \mathsf{self} \Leftarrow \lvert\mathcal{A}_i\rvert$	By rule FT-SUB
$\lvert\Delta\rvert; \lvert\Gamma\rvert, \mathsf{self} : \lvert\mathcal{A}\rvert \vdash E_i\,\mathsf{self} \Rightarrow \lvert\mathcal{B}_i\rvert$	By rule FT-APP
$\lvert\Delta\rvert; \lvert\Gamma\rvert, \mathsf{self} : \lvert\mathcal{A}\rvert \vdash (E_1\,\mathsf{self}),, ..,, (E_n\,\mathsf{self}) \Rightarrow \lvert\mathcal{B}_1\rvert \,\&\, .. \,\&\, \lvert\mathcal{B}_n\rvert$	By rule FT-MERGE
$\lvert\Delta\rvert; \lvert\Gamma\rvert, \mathsf{self} : \lvert\mathcal{A}\rvert \vdash (E_1\,\mathsf{self}),, ..,, (E_n\,\mathsf{self}) \Leftarrow \lvert\mathcal{A}\rvert$	By rule FT-SUB
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash \mathbf{letrec}\,\mathsf{self} : \lvert\mathcal{A}\rvert = (E_1\,\mathsf{self}),, ..,, (E_n\,\mathsf{self})\,\mathbf{in}\,\mathsf{self} \Rightarrow \lvert\mathcal{A}\rvert$	

- ST-TRAIT

$$\frac{\overline{\Delta;\Gamma,\mathsf{self}:\mathcal{B}\vdash\mathcal{T}_i\ \Rightarrow\ \textbf{Trait}\,[\mathcal{B}_i,\mathcal{C}_i]\rightsquigarrow E_i}^{i\in 1..n}\qquad \Delta;\Gamma,\mathsf{self}:\mathcal{B}\vdash\{\overline{l_j=\mathcal{T}_j'}^{j\in 1..m}\}\ \Rightarrow\ \mathcal{C}\rightsquigarrow E\qquad \overline{\mathcal{B}<:\mathcal{B}_i}^{i\in 1..n}\qquad \Delta\vdash\mathcal{C}_1*\,..\,*\mathcal{C}_n*\mathcal{C}\qquad \mathcal{C}_1\,\&\,..\,\&\,\mathcal{C}_n\,\&\,\mathcal{C}<:\mathcal{A}}{\Delta;\Gamma\vdash\textbf{trait}\,[\mathsf{self}:\mathcal{B}]\ \textbf{inherits}\,\overline{\mathcal{T}_i}^{i\in 1..n}\,\{\overline{l_j=\mathcal{T}_j'}^{j\in 1..m}\}:\mathcal{A}\ \Rightarrow\ \textbf{Trait}\,[\mathcal{B},\mathcal{A}]\rightsquigarrow \lambda\mathsf{self}:|\mathcal{B}|.\,((\overline{(E_i\,\mathsf{self})}^{i\in 1..n}),,E)}$$

$\lvert\Delta\rvert;\lvert\Gamma\rvert,\mathsf{self}:\lvert\mathcal{B}\rvert\vdash E_i\ \Rightarrow\ \lvert\mathcal{B}_i\rvert\to\lvert\mathcal{C}_i\rvert$	By i.h.
$\lvert\Delta\rvert;\lvert\Gamma\rvert,\mathsf{self}:\lvert\mathcal{B}\rvert\vdash E\ \Rightarrow\ \lvert\mathcal{C}\rvert$	By i.h.
$\lvert\mathcal{B}\rvert<:\lvert\mathcal{B}_i\rvert$	By Lemma 26
$\lvert\Delta\rvert\vdash\lvert\mathcal{C}_1\rvert*\,..\,*\lvert\mathcal{C}_n\rvert*\lvert\mathcal{C}\rvert$	By Lemma 25
$\lvert\mathcal{C}_1\rvert\,\&\,..\,\&\,\lvert\mathcal{C}_n\rvert\,\&\,\lvert\mathcal{C}\rvert<:\lvert\mathcal{A}\rvert$	By Lemma 26
$\lvert\Delta\rvert;\lvert\Gamma\rvert,\mathsf{self}:\lvert\mathcal{B}\rvert\vdash\mathsf{self}\ \Rightarrow\ \lvert\mathcal{B}\rvert$	By rule FT-VAR
$\lvert\Delta\rvert;\lvert\Gamma\rvert,\mathsf{self}:\lvert\mathcal{B}\rvert\vdash\mathsf{self}\ \Leftarrow\ \lvert\mathcal{B}_i\rvert$	By rule FT-SUB
$\lvert\Delta\rvert;\lvert\Gamma\rvert,\mathsf{self}:\lvert\mathcal{B}\rvert\vdash E_i\,\mathsf{self}\ \Rightarrow\ \lvert\mathcal{C}_i\rvert$	By rule TI-APP
$\lvert\Delta\rvert;\lvert\Gamma\rvert,\mathsf{self}:\lvert\mathcal{B}\rvert\vdash(E_1\,\mathsf{self}),,\,..\,,,(E_n\,\mathsf{self}),,E\ \Rightarrow\ \lvert\mathcal{C}_1\rvert\,\&\,..\,\&\,\lvert\mathcal{C}_n\rvert\,\&\,\lvert\mathcal{C}\rvert$	By rule FT-MERGE
$\lvert\Delta\rvert;\lvert\Gamma\rvert,\mathsf{self}:\lvert\mathcal{B}\rvert\vdash(E_1\,\mathsf{self}),,\,..\,,,(E_n\,\mathsf{self}),,E\ \Rightarrow\ \lvert\mathcal{A}\rvert$	By rule FT-SUB
$\lvert\Delta\rvert;\lvert\Gamma\rvert\vdash\lambda\mathsf{self}:\lvert\mathcal{B}\rvert.\,(E_1\,\mathsf{self}),,\,..\,,,(E_n\,\mathsf{self}),,E\ \Rightarrow\ \lvert\mathcal{B}\rvert\to\lvert\mathcal{A}\rvert$	

- ST-FORWARD

$$\frac{\Delta;\Gamma\vdash\mathcal{T}_1\ \Rightarrow\ \textbf{Trait}\,[\mathcal{A},\mathcal{B}]\rightsquigarrow E_1\qquad \Delta;\Gamma\vdash\mathcal{T}_2\ \Leftarrow\ \mathcal{A}\rightsquigarrow E_2}{\Delta;\Gamma\vdash\mathcal{T}_1\,\hat{}\,\mathcal{T}_2\ \Rightarrow\ \mathcal{B}\rightsquigarrow E_1\,E_2}$$

$\lvert\Delta\rvert;\lvert\Gamma\rvert\vdash E_1\ \Rightarrow\ \lvert\mathcal{A}\rvert\to\lvert\mathcal{B}\rvert$	By i.h.
$\lvert\Delta\rvert;\lvert\Gamma\rvert\vdash E_2\ \Leftarrow\ \lvert\mathcal{A}\rvert$	By i.h.
$\lvert\Delta\rvert;\lvert\Gamma\rvert\vdash E_1\,E_2\ \Rightarrow\ \lvert\mathcal{B}\rvert$	By rule FT-APP

- ST-ABS

$$\frac{\Delta\vdash\mathcal{A}\qquad \Delta;\Gamma,x:\mathcal{A}\vdash\mathcal{T}\ \Leftarrow\ \mathcal{B}\rightsquigarrow E}{\Delta;\Gamma\vdash\lambda x.\mathcal{T}\ \Leftarrow\ \mathcal{A}\to\mathcal{B}\rightsquigarrow\lambda x.\,E}$$

$\lvert\Delta\rvert\vdash\lvert\mathcal{A}\rvert$	By Lemma 25
$\lvert\Delta\rvert;\lvert\Gamma\rvert,x:\lvert\mathcal{A}\rvert\vdash E\ \Leftarrow\ \lvert\mathcal{B}\rvert$	By i.h.

$|\Delta|; |\Gamma| \vdash \lambda x.\, E \;\Leftarrow\; |\mathcal{A}| \to |\mathcal{B}|$ By rule FT-ABS

- $$\frac{\Delta; \Gamma \vdash \mathcal{T} \;\Rightarrow\; \mathcal{A} \rightsquigarrow E \qquad \mathcal{A} <: \mathcal{B} \qquad \Delta \vdash \mathcal{B}}{\Delta; \Gamma \vdash \mathcal{T} \;\Leftarrow\; \mathcal{B} \rightsquigarrow E}\;\textsc{ST-sub}$$

$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash E \;\Rightarrow\; \lvert\mathcal{A}\rvert$	By i.h.
$\lvert\mathcal{A}\rvert <: \lvert\mathcal{B}\rvert$	By Lemma 26
$\lvert\Delta\rvert \vdash \lvert\mathcal{B}\rvert$	By Lemma 25
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash E \;\Leftarrow\; \lvert\mathcal{B}\rvert$	By rule FT-SUB

□

B λ_i^+ Typing Rules, in Full

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

$$\frac{}{A <: A \rightsquigarrow \mathsf{id}}\ \textsc{S-refl} \qquad \frac{A_2 <: A_3 \rightsquigarrow c_1 \qquad A_1 <: A_2 \rightsquigarrow c_2}{A_1 <: A_3 \rightsquigarrow c_1 \circ c_2}\ \textsc{S-trans} \qquad \frac{}{A <: \top \rightsquigarrow \mathsf{top}}\ \textsc{S-top}$$

$$\frac{A <: B \rightsquigarrow c}{\{l : A\} <: \{l : B\} \rightsquigarrow c}\ \textsc{S-rcd} \qquad \frac{B_1 <: A_1 \rightsquigarrow c_1 \qquad A_2 <: B_2 \rightsquigarrow c_2}{A_1 \to A_2 <: B_1 \to B_2 \rightsquigarrow c_1 \to c_2}\ \textsc{S-arr}$$

$$\frac{}{A_1 \,\&\, A_2 <: A_1 \rightsquigarrow \pi_1}\ \textsc{S-andL} \qquad \frac{}{A_1 \,\&\, A_2 <: A_2 \rightsquigarrow \pi_2}\ \textsc{S-andR} \qquad \frac{A_1 <: A_2 \rightsquigarrow c_1 \qquad A_1 <: A_3 \rightsquigarrow c_2}{A_1 <: A_2 \,\&\, A_3 \rightsquigarrow \langle c_1, c_2 \rangle}\ \textsc{S-and}$$

$$\frac{}{\top <: \top \to \top \rightsquigarrow \mathsf{top}_{\to}}\ \textsc{S-topArr} \qquad \frac{}{\top <: \{l : \top\} \rightsquigarrow \mathsf{id}}\ \textsc{S-topRcd} \qquad \frac{}{\{l : A\} \,\&\, \{l : B\} <: \{l : A \,\&\, B\} \rightsquigarrow \mathsf{id}}\ \textsc{S-distRcd}$$

$$\frac{}{(A_1 \to A_2) \,\&\, (A_1 \to A_3) <: A_1 \to A_2 \,\&\, A_3 \rightsquigarrow \mathsf{dist}_{\to}}\ \textsc{S-distArr}$$

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

$$\frac{}{\top * A}\ \textsc{D-topL} \qquad \frac{}{A * \top}\ \textsc{D-topR} \qquad \frac{A_2 * B_2}{A_1 \to A_2 * B_1 \to B_2}\ \textsc{D-arr} \qquad \frac{A_1 * B \qquad A_2 * B}{A_1 \,\&\, A_2 * B}\ \textsc{D-andL}$$

$$\frac{A * B_1 \qquad A * B_2}{A * B_1 \,\&\, B_2}\ \textsc{D-andR} \qquad \frac{A * B}{\{l : A\} * \{l : B\}}\ \textsc{D-rcdEq} \qquad \frac{l_1 \neq l_2}{\{l_1 : A\} * \{l_2 : B\}}\ \textsc{D-rcdNeq} \qquad \frac{A *_{ax} B}{A * B}\ \textsc{D-ax}$$

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

$$\textsc{Dax-sym}\ \frac{B *_{ax} A}{A *_{ax} B} \qquad \textsc{Dax-intArr}\ \frac{}{\rho *_{ax} A_1 \to A_2} \qquad \textsc{Dax-intRcd}\ \frac{}{\rho *_{ax} \{l : A\}} \qquad \textsc{Dax-arrRcd}\ \frac{}{A_1 \to A_2 *_{ax} \{l : B\}}$$

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

$$\textsc{T-top}\ \frac{}{\Gamma \vdash \top \Rightarrow \top \rightsquigarrow \langle\rangle} \qquad \textsc{T-lit}\ \frac{}{\Gamma \vdash i \Rightarrow \rho \rightsquigarrow i} \qquad \textsc{T-var}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A \rightsquigarrow x}$$

$$\textsc{T-app}\ \frac{\Gamma \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e_1 \qquad \Gamma \vdash E_2 \Leftarrow A_1 \rightsquigarrow e_2}{\Gamma \vdash E_1\, E_2 \Rightarrow A_2 \rightsquigarrow e_1\, e_2} \qquad \textsc{T-anno}\ \frac{\Gamma \vdash E \Leftarrow A \rightsquigarrow e}{\Gamma \vdash E : A \Rightarrow A \rightsquigarrow e}$$

$$\textsc{T-proj}\ \frac{\Gamma \vdash E \Rightarrow \{l : A\} \rightsquigarrow e}{\Gamma \vdash E.l \Rightarrow A \rightsquigarrow e} \qquad \textsc{T-merge}\ \frac{\Gamma \vdash E_1 \Rightarrow A_1 \rightsquigarrow e_1 \qquad \Gamma \vdash E_2 \Rightarrow A_2 \rightsquigarrow e_2 \qquad A_1 * A_2}{\Gamma \vdash E_1 ,, E_2 \Rightarrow A_1 \,\&\, A_2 \rightsquigarrow \langle e_1, e_2\rangle}$$

$$\textsc{T-rcd}\ \frac{\Gamma \vdash E \Rightarrow A \rightsquigarrow e}{\Gamma \vdash \{l = E\} \Rightarrow \{l : A\} \rightsquigarrow e}$$

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

$$\textsc{T-abs}\ \frac{\Gamma, x : A \vdash E \Leftarrow B \rightsquigarrow e}{\Gamma \vdash \lambda x.\, E \Leftarrow A \to B \rightsquigarrow \lambda x.\, e} \qquad \textsc{T-sub}\ \frac{\Gamma \vdash E \Rightarrow A \rightsquigarrow e \qquad A <: B \rightsquigarrow c}{\Gamma \vdash E \Leftarrow B \rightsquigarrow c\, e}$$

$\boxed{\mathcal{L} \vdash A \prec: B \rightsquigarrow c}$ *(Algorithmic subtyping)*

$$\textsc{A-prim}\ \frac{}{[] \vdash \rho \prec: \rho \rightsquigarrow \mathsf{id}} \qquad \textsc{A-and}\ \frac{\mathcal{L} \vdash A \prec: B_1 \rightsquigarrow c_1 \qquad \mathcal{L} \vdash A \prec: B_2 \rightsquigarrow c_2}{\mathcal{L} \vdash A \prec: B_1 \,\&\, B_2 \rightsquigarrow [\![\mathcal{L}]\!]_{\&} \circ \langle c_1, c_2\rangle}$$

$$\textsc{A-arr}\ \frac{\mathcal{L}, B_1 \vdash A \prec: B_2 \rightsquigarrow c}{\mathcal{L} \vdash A \prec: B_1 \to B_2 \rightsquigarrow c} \qquad \textsc{A-rcd}\ \frac{\mathcal{L}, \{l\} \vdash A \prec: B \rightsquigarrow c}{\mathcal{L} \vdash A \prec: \{l : B\} \rightsquigarrow c} \qquad \textsc{A-top}\ \frac{}{\mathcal{L} \vdash A \prec: \top \rightsquigarrow [\![\mathcal{L}]\!]_{\top} \circ \mathsf{top}}$$

$$\frac{[] \vdash A \prec: A_1 \rightsquigarrow c_1 \qquad \mathcal{L} \vdash A_2 \prec: B \rightsquigarrow c_2 \qquad B \text{ rigid}}{A, \mathcal{L} \vdash A_1 \to A_2 \prec: B \rightsquigarrow c_1 \to c_2}\ \textsc{A-arrR}$$

$$\frac{\mathcal{L} \vdash A \prec: B \rightsquigarrow c \qquad B \text{ rigid}}{\{l\}, \mathcal{L} \vdash \{l : A\} \prec: B \rightsquigarrow c}\ \textsc{A-rcdR}$$

$$\frac{\mathcal{L} \vdash A_1 \prec: B \rightsquigarrow c \qquad B \text{ rigid}}{\mathcal{L} \vdash A_1 \,\&\, A_2 \prec: B \rightsquigarrow c \circ \pi_1}\ \textsc{A-andR1}$$

$$\frac{\mathcal{L} \vdash A_2 \prec: B \rightsquigarrow c \qquad B \text{ rigid}}{\mathcal{L} \vdash A_1 \,\&\, A_2 \prec: B \rightsquigarrow c \circ \pi_2}\ \textsc{A-andR2}$$

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

$$\frac{}{[\cdot] : (\Gamma \Rightarrow A) \mapsto (\Gamma \Rightarrow A) \rightsquigarrow [\cdot]}\ \textsc{CTyp-empty1}$$

$$\frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1 \to A_2) \rightsquigarrow \mathcal{D} \qquad \Gamma' \vdash E_2 \Leftarrow A_1 \rightsquigarrow e}{\mathcal{C}\, E_2 : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D}\, e}\ \textsc{CTyp-appL1}$$

$$\frac{\Gamma' \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e \qquad \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Leftarrow A_1) \rightsquigarrow \mathcal{D}}{E_1\, \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow e\, \mathcal{D}}\ \textsc{CTyp-appR1}$$

$$\frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1) \rightsquigarrow \mathcal{D} \qquad \Gamma' \vdash E_2 \Rightarrow A_2 \rightsquigarrow e \qquad A_1 * A_2}{\mathcal{C} ,, E_2 : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle \mathcal{D}, e \rangle}\ \textsc{CTyp-mergeL1}$$

$$\frac{\Gamma' \vdash E_1 \Rightarrow A_1 \rightsquigarrow e \qquad \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D} \qquad A_1 * A_2}{E_1 ,, \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle e, \mathcal{D} \rangle}\ \textsc{CTyp-mergeR1}$$

$$\frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}{\{l = \mathcal{C}\} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}}\ \textsc{CTyp-rcd1}$$

$$\frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}}{\mathcal{C}.l : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}\ \textsc{CTyp-proj1}$$

$$\frac{\mathcal{C} : (\Gamma \Rightarrow B) \mapsto (\Gamma' \Leftarrow A) \rightsquigarrow \mathcal{D}}{\mathcal{C} : A : (\Gamma \Rightarrow B) \mapsto (\Gamma' \Rightarrow A) \rightsquigarrow \mathcal{D}}\ \textsc{CTyp-anno1}$$

$\boxed{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Leftarrow B) \rightsquigarrow \mathcal{D}}$ *(Context typing II)*

$$\text{CTyp-empty2}\ \frac{}{[\cdot] : (\Gamma \Leftarrow A) \mapsto (\Gamma \Leftarrow A) \rightsquigarrow [\cdot]}$$

$$\text{CTyp-abs2}\ \frac{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma', x : A_1 \Leftarrow A_2) \rightsquigarrow \mathcal{D} \qquad x \notin \Gamma'}{\lambda x.\, \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Leftarrow A_1 \to A_2) \rightsquigarrow \lambda x.\, \mathcal{D}}$$

$\boxed{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}$ *(Context typing III)*

$$\text{CTyp-appL2}\ \frac{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_1 \to A_2) \rightsquigarrow \mathcal{D} \qquad \Gamma' \vdash E_2 \Leftarrow A_1 \rightsquigarrow e}{\mathcal{C}\, E_2 : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D}\, e}$$

$$\text{CTyp-appR2}\ \frac{\Gamma' \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e \qquad \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Leftarrow A_1) \rightsquigarrow \mathcal{D}}{E_1\, \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow e\, \mathcal{D}}$$

$$\text{CTyp-mergeL2}\ \frac{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_1) \rightsquigarrow \mathcal{D} \qquad \Gamma' \vdash E_2 \Rightarrow A_2 \rightsquigarrow e \qquad A_1 * A_2}{\mathcal{C},,E_2 : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle \mathcal{D}, e \rangle}$$

$$\text{CTyp-mergeR2}\ \frac{\Gamma' \vdash E_1 \Rightarrow A_1 \rightsquigarrow e \qquad \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D} \qquad A_1 * A_2}{E_1,,\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle e, \mathcal{D} \rangle}$$

$$\text{CTyp-rcd2}\ \frac{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}{\{l = \mathcal{C}\} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}}$$

$$\text{CTyp-proj2}\ \frac{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}}{\mathcal{C}.l : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}$$

$$\text{CTyp-anno2}\ \frac{\mathcal{C} : (\Gamma \Leftarrow B) \mapsto (\Gamma' \Leftarrow A) \rightsquigarrow \mathcal{D}}{\mathcal{C} : A : (\Gamma \Leftarrow B) \mapsto (\Gamma' \Rightarrow A) \rightsquigarrow \mathcal{D}}$$

$\boxed{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Leftarrow B) \rightsquigarrow \mathcal{D}}$ *(Context typing IV)*

$$\text{CTyp-abs1}\ \frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma', x : A_1 \Leftarrow A_2) \rightsquigarrow \mathcal{D} \qquad x \notin \Gamma'}{\lambda x.\, \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Leftarrow A_1 \to A_2) \rightsquigarrow \lambda x.\, \mathcal{D}}$$

B.1 λ_{co}

$\boxed{c \vdash \tau_1 \rhd \tau_2}$ *(Coercion typing)*

$$\frac{}{\mathsf{id} \vdash \tau \rhd \tau}\ \textsc{ct-refl} \qquad \frac{c_1 \vdash \tau_2 \rhd \tau_3 \qquad c_2 \vdash \tau_1 \rhd \tau_2}{c_1 \circ c_2 \vdash \tau_1 \rhd \tau_3}\ \textsc{ct-trans} \qquad \frac{}{\mathsf{top} \vdash \tau \rhd \langle\rangle}\ \textsc{ct-top}$$

$$\frac{}{\mathsf{top}_{\rightarrow} \vdash \langle\rangle \rhd \langle\rangle \rightarrow \langle\rangle}\ \textsc{ct-topArr} \qquad \frac{c_1 \vdash \tau_1' \rhd \tau_1 \qquad c_2 \vdash \tau_2 \rhd \tau_2'}{c_1 \rightarrow c_2 \vdash \tau_1 \rightarrow \tau_2 \rhd \tau_1' \rightarrow \tau_2'}\ \textsc{ct-arr}$$

$$\frac{c_1 \vdash \tau_1 \rhd \tau_2 \qquad c_2 \vdash \tau_1 \rhd \tau_3}{\langle c_1, c_2\rangle \vdash \tau_1 \rhd \tau_2 \times \tau_3}\ \textsc{ct-pair} \qquad \frac{}{\pi_1 \vdash \tau_1 \times \tau_2 \rhd \tau_1}\ \textsc{ct-projL} \qquad \frac{}{\pi_2 \vdash \tau_1 \times \tau_2 \rhd \tau_2}\ \textsc{ct-projR}$$

$$\frac{}{\mathsf{dist}_{\rightarrow} \vdash (\tau_1 \rightarrow \tau_2) \times (\tau_1 \rightarrow \tau_3) \rhd \tau_1 \rightarrow \tau_2 \times \tau_3}\ \textsc{ct-distArr}$$

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

$$\frac{}{\mathsf{id}\ v \longrightarrow v}\ \textsc{r-id} \qquad \frac{}{(c_1 \circ c_2)\ v \longrightarrow c_1\ (c_2\ v)}\ \textsc{r-trans} \qquad \frac{}{\mathsf{top}\ v \longrightarrow \langle\rangle}\ \textsc{r-top} \qquad \frac{}{(\mathsf{top}_{\rightarrow}\ \langle\rangle)\ \langle\rangle \longrightarrow \langle\rangle}\ \textsc{r-topArr}$$

$$\frac{}{\langle c_1, c_2\rangle\ v \longrightarrow \langle c_1\ v, c_2\ v\rangle}\ \textsc{r-pair} \qquad \frac{}{((c_1 \rightarrow c_2)\ v_1)\ v_2 \longrightarrow c_2\ (v_1\ (c_1\ v_2))}\ \textsc{r-arr}$$

$$\frac{}{(\mathsf{dist}_{\rightarrow}\ \langle v_1, v_2\rangle)\ v_3 \longrightarrow \langle v_1\ v_3, v_2\ v_3\rangle}\ \textsc{r-distArr} \qquad \frac{}{\pi_1\ \langle v_1, v_2\rangle \longrightarrow v_1}\ \textsc{r-projL} \qquad \frac{}{\pi_2\ \langle v_1, v_2\rangle \longrightarrow v_2}\ \textsc{r-projR}$$

$$\frac{}{(\lambda x.\ e)\ v \longrightarrow [v/x]e}\ \textsc{r-app} \qquad \frac{e \longrightarrow e'}{\mathcal{E}[e] \longrightarrow \mathcal{E}[e']}\ \textsc{r-ctxt}$$

C F_i^+ Typing Rules, in Full

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

$$\frac{}{\Delta \vdash \top}\ \textsc{swft-top} \qquad \frac{}{\Delta \vdash \rho}\ \textsc{swft-int} \qquad \frac{(\alpha * A) \in \Delta}{\Delta \vdash \alpha}\ \textsc{swft-var} \qquad \frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \to B}\ \textsc{swft-arrow}$$

$$\frac{\Delta \vdash A \qquad \Delta, \alpha * A \vdash B}{\Delta \vdash \forall (\alpha * A).\, B}\ \textsc{swft-all} \qquad \frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \,\&\, B}\ \textsc{swft-and} \qquad \frac{\Delta \vdash A}{\Delta \vdash \{l : A\}}\ \textsc{swft-rcd}$$

$\boxed{\Delta \vdash \Gamma}$ *(Well-formedness of value context)*

$$\frac{}{\Delta \vdash \cdot}\ \textsc{swfe-empty} \qquad \frac{\Delta \vdash \Gamma \qquad \Delta \vdash A}{\Delta \vdash \Gamma, x : A}\ \textsc{swfe-var}$$

$\boxed{\vdash \Delta}$ *(Well-formedness of type context)*

$$\frac{}{\vdash \cdot}\ \textsc{swfte-empty} \qquad \frac{\vdash \Delta \qquad \Delta \vdash A}{\vdash \Delta, \alpha * A}\ \textsc{swfte-var}$$

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

$$\frac{}{A <: A \rightsquigarrow \mathsf{id}}\ \textsc{S-refl} \qquad \frac{A_2 <: A_3 \rightsquigarrow c_1 \qquad A_1 <: A_2 \rightsquigarrow c_2}{A_1 <: A_3 \rightsquigarrow c_1 \circ c_2}\ \textsc{S-trans} \qquad \frac{}{A <: \top \rightsquigarrow \mathsf{top}}\ \textsc{S-top}$$

$$\frac{A <: B \rightsquigarrow c}{\{l : A\} <: \{l : B\} \rightsquigarrow c}\ \textsc{S-rcd} \qquad \frac{B_1 <: A_1 \rightsquigarrow c_1 \qquad A_2 <: B_2 \rightsquigarrow c_2}{A_1 \to A_2 <: B_1 \to B_2 \rightsquigarrow c_1 \to c_2}\ \textsc{S-arr}$$

$$\textsc{S-andL}\quad \frac{}{A_1 \,\&\, A_2 <: A_1 \rightsquigarrow \pi_1} \qquad \textsc{S-andR}\quad \frac{}{A_1 \,\&\, A_2 <: A_2 \rightsquigarrow \pi_2} \qquad \textsc{S-and}\quad \frac{A_1 <: A_2 \rightsquigarrow c_1 \qquad A_1 <: A_3 \rightsquigarrow c_2}{A_1 <: A_2 \,\&\, A_3 \rightsquigarrow \langle c_1, c_2 \rangle}$$

$$\textsc{S-topArr}\quad \frac{}{\top <: \top \to \top \rightsquigarrow \mathsf{top}_{\to}} \qquad \textsc{S-topRcd}\quad \frac{}{\top <: \{l : \top\} \rightsquigarrow \mathsf{id}}$$

$$\textsc{S-distArr}\quad \frac{}{(A_1 \to A_2) \,\&\, (A_1 \to A_3) <: A_1 \to A_2 \,\&\, A_3 \rightsquigarrow \mathsf{dist}_{\to}}$$

$$\textsc{S-distRcd}\quad \frac{}{\{l : A\} \,\&\, \{l : B\} <: \{l : A \,\&\, B\} \rightsquigarrow \mathsf{id}} \qquad \textsc{S-forall}\quad \frac{B_1 <: B_2 \rightsquigarrow c \qquad A_2 <: A_1}{\forall(\alpha * A_1).\, B_1 <: \forall(\alpha * A_2).\, B_2 \rightsquigarrow c_{\forall}}$$

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

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

$$\textsc{FD-andL}\quad \frac{\Delta \vdash A_1 * B \qquad \Delta \vdash A_2 * B}{\Delta \vdash A_1 \,\&\, A_2 * B} \qquad \textsc{FD-andR}\quad \frac{\Delta \vdash A * B_1 \qquad \Delta \vdash A * B_2}{\Delta \vdash A * B_1 \,\&\, B_2} \qquad \textsc{FD-rcdEq}\quad \frac{\Delta \vdash A * B}{\Delta \vdash \{l : A\} * \{l : B\}}$$

$$\textsc{FD-rcdNeq}\quad \frac{l_1 \neq l_2}{\Delta \vdash \{l_1 : A\} * \{l_2 : B\}} \qquad \textsc{FD-tvarL}\quad \frac{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash \alpha * B} \qquad \textsc{FD-tvarR}\quad \frac{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash B * \alpha}$$

$$\textsc{FD-forall}\quad \frac{\Delta, \alpha * A_1 \,\&\, A_2 \vdash B_1 * B_2}{\Delta \vdash \forall(\alpha * A_1).\, B_1 * \forall(\alpha * A_2).\, B_2} \qquad \textsc{FD-ax}\quad \frac{A *_{ax} B}{\Delta \vdash A * B}$$

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

$$\textsc{Dax-sym}\quad \frac{B *_{ax} A}{A *_{ax} B} \qquad \textsc{Dax-intArr}\quad \frac{}{\rho *_{ax} A_1 \to A_2} \qquad \textsc{Dax-intRcd}\quad \frac{}{\rho *_{ax} \{l : A\}} \qquad \textsc{Dax-intAll}\quad \frac{}{\rho *_{ax} \forall(\alpha * B_1).\, B_2}$$

$$\textsc{Dax-arrAll}\quad \frac{}{A_1 \to A_2 *_{ax} \forall(\alpha * B_1).\, B_2} \qquad \textsc{Dax-arrRcd}\quad \frac{}{A_1 \to A_2 *_{ax} \{l : B\}} \qquad \textsc{Dax-allRcd}\quad \frac{}{\forall(\alpha * A_1).\, A_2 *_{ax} \{l : B\}}$$

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

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

$$\frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta;\Gamma \vdash i \Rightarrow \rho \rightsquigarrow i}\ \textsc{FT-int}$$

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

$$\frac{\Delta;\Gamma \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e_1 \qquad \Delta;\Gamma \vdash E_2 \Leftarrow A_1 \rightsquigarrow e_2}{\Delta;\Gamma \vdash E_1\,E_2 \Rightarrow A_2 \rightsquigarrow e_1\,e_2}\ \textsc{FT-app}$$

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

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

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

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

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

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

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

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

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

$\boxed{\mathcal{L} \vdash A \prec: B \rightsquigarrow c}$ *(Algorithmic subtyping)*

$$\textsc{A-prim}\quad \frac{}{[\,] \vdash \rho \prec: \rho \rightsquigarrow \mathsf{id}}$$

$$\textsc{A-and}\quad \frac{\mathcal{L} \vdash A \prec: B_1 \rightsquigarrow c_1 \qquad \mathcal{L} \vdash A \prec: B_2 \rightsquigarrow c_2}{\mathcal{L} \vdash A \prec: B_1 \,\&\, B_2 \rightsquigarrow [\![\mathcal{L}]\!]_\& \circ \langle c_1, c_2 \rangle}$$

$$\textsc{A-arr}\quad \frac{\mathcal{L}, B_1 \vdash A \prec: B_2 \rightsquigarrow c}{\mathcal{L} \vdash A \prec: B_1 \to B_2 \rightsquigarrow c}$$

$$\textsc{A-rcd}\quad \frac{\mathcal{L}, \{l\} \vdash A \prec: B \rightsquigarrow c}{\mathcal{L} \vdash A \prec: \{l : B\} \rightsquigarrow c}$$

$$\textsc{A-top}\quad \frac{}{\mathcal{L} \vdash A \prec: \top \rightsquigarrow [\![\mathcal{L}]\!]_\top \circ \mathsf{top}}$$

$$\textsc{A-forall}\quad \frac{[\,] \vdash A_2 \prec: A_1 \qquad [\,] \vdash B_1 \prec: B_2 \rightsquigarrow c}{[\,] \vdash \forall(\alpha * A_1).\, B_1 \prec: \forall(\alpha * A_2).\, B_2 \rightsquigarrow c_\forall}$$

$$\textsc{A-var}\quad \frac{}{[\,] \vdash \alpha \prec: \alpha \rightsquigarrow \mathsf{id}}$$

$$\textsc{A-arrR}\quad \frac{[\,] \vdash A \prec: A_1 \rightsquigarrow c_1 \qquad \mathcal{L} \vdash A_2 \prec: B \rightsquigarrow c_2 \qquad B \text{ rigid}}{A, \mathcal{L} \vdash A_1 \to A_2 \prec: B \rightsquigarrow c_1 \to c_2}$$

$$\textsc{A-rcdR}\quad \frac{\mathcal{L} \vdash A \prec: B \rightsquigarrow c \qquad B \text{ rigid}}{\{l\}, \mathcal{L} \vdash \{l : A\} \prec: B \rightsquigarrow c}$$

$$\textsc{A-andR1}\quad \frac{\mathcal{L} \vdash A_1 \prec: B \rightsquigarrow c \qquad B \text{ rigid}}{\mathcal{L} \vdash A_1 \,\&\, A_2 \prec: B \rightsquigarrow c \circ \pi_1}$$

$$\textsc{A-andR2}\quad \frac{\mathcal{L} \vdash A_2 \prec: B \rightsquigarrow c \qquad B \text{ rigid}}{\mathcal{L} \vdash A_1 \,\&\, A_2 \prec: B \rightsquigarrow c \circ \pi_2}$$

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

$$\textsc{FCTyp-empty1}\quad \frac{}{[\cdot] : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta; \Gamma \Rightarrow A) \rightsquigarrow [\cdot]}$$

$$\textsc{FCTyp-appL1}\quad \frac{\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_1 \to A_2) \rightsquigarrow \mathcal{D} \qquad \Delta'; \Gamma' \vdash E_2 \Leftarrow A_1 \rightsquigarrow e}{\mathcal{C}\, E_2 : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D}\, e}$$

$$\textsc{FCTyp-appR1}\quad \frac{\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Leftarrow A_1) \rightsquigarrow \mathcal{D} \qquad \Delta'; \Gamma' \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e}{E_1\, \mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_2) \rightsquigarrow e\, \mathcal{D}}$$

$$\textsc{FCTyp-mergeL1}\quad \frac{\begin{array}{c}\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_1) \rightsquigarrow \mathcal{D} \\ \Delta'; \Gamma' \vdash E_2 \Rightarrow A_2 \rightsquigarrow e \qquad \Delta' \vdash A_1 * A_2\end{array}}{\mathcal{C},, E_2 : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle \mathcal{D}, e \rangle}$$

$$\textsc{FCTyp-mergeR1}\quad \frac{\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D} \qquad \Delta'; \Gamma' \vdash E_1 \Rightarrow A_1 \rightsquigarrow e \qquad \Delta' \vdash A_1 * A_2}{E_1 ,, \mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_1 \& A_2) \rightsquigarrow \langle e, \mathcal{D} \rangle}$$

$$\textsc{FCTyp-rcd1}\quad \frac{\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}{\{l = \mathcal{C}\} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}}$$

$$\textsc{FCTyp-proj1}\quad \frac{\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}}{\mathcal{C}.l : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}$$

$$\textsc{FCTyp-anno1}\quad \frac{\mathcal{C} : (\Delta; \Gamma \Rightarrow B) \mapsto (\Delta'; \Gamma' \Leftarrow A) \rightsquigarrow \mathcal{D}}{\mathcal{C} : A : (\Delta; \Gamma \Rightarrow B) \mapsto (\Delta'; \Gamma' \Rightarrow A) \rightsquigarrow \mathcal{D}}$$

$$\textsc{FCTyp-tabs1}\quad \frac{\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta', \alpha * B; \Gamma' \Rightarrow B') \rightsquigarrow \mathcal{D} \qquad \Delta' \vdash B}{\Lambda(\alpha * B). \mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow \forall(\alpha * B). B') \rightsquigarrow \Lambda\alpha. \mathcal{D}}$$

$$\textsc{FCTyp-tapp1}\quad \frac{\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow \forall(\alpha * A_1). A_2) \rightsquigarrow \mathcal{D} \qquad \Delta' \vdash t \qquad \Delta' \vdash t * A_1}{\mathcal{C}\, B : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow [t/\alpha]A_2) \rightsquigarrow \mathcal{D}\, |t|}$$

$$\boxed{\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Leftarrow B) \rightsquigarrow \mathcal{D}}$$ *(Context typing II)*

$$\textsc{FCTyp-empty2}\quad \frac{}{[\cdot] : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta; \Gamma \Leftarrow A) \rightsquigarrow [\cdot]}$$

$$\textsc{FCTyp-abs2}\quad \frac{\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma', x : A_1 \Leftarrow A_2) \rightsquigarrow \mathcal{D} \qquad \Delta' \vdash A_1}{\lambda x. \mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Leftarrow A_1 \to A_2) \rightsquigarrow \lambda x. \mathcal{D}}$$

$\boxed{\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}$ *(Context typing III)*

$$\frac{\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_1 \to A_2) \rightsquigarrow \mathcal{D} \qquad \Delta'; \Gamma' \vdash E_2 \Leftarrow A_1 \rightsquigarrow e}{\mathcal{C}\, E_2 : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D}\, e} \text{ FCTyp-appL2}$$

$$\frac{\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Leftarrow A_1) \rightsquigarrow \mathcal{D} \qquad \Delta'; \Gamma' \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e}{E_1\, \mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_2) \rightsquigarrow e\, \mathcal{D}} \text{ FCTyp-appR2}$$

$$\frac{\begin{array}{c}\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_1) \rightsquigarrow \mathcal{D} \\ \Delta'; \Gamma' \vdash E_2 \Rightarrow A_2 \rightsquigarrow e \qquad \Delta' \vdash A_1 * A_2\end{array}}{\mathcal{C},, E_2 : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle \mathcal{D}, e \rangle} \text{ FCTyp-mergeL2}$$

$$\frac{\begin{array}{c}\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D} \\ \Delta'; \Gamma' \vdash E_1 \Rightarrow A_1 \rightsquigarrow e \qquad \Delta' \vdash A_1 * A_2\end{array}}{E_1,, \mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle e, \mathcal{D} \rangle} \text{ FCTyp-mergeR2}$$

$$\frac{\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}{\{l = \mathcal{C}\} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}} \text{ FCTyp-rcd2}$$

$$\frac{\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}}{\mathcal{C}.l : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}} \text{ FCTyp-proj2}$$

$$\frac{\mathcal{C} : (\Delta; \Gamma \Leftarrow B) \mapsto (\Delta'; \Gamma' \Leftarrow A) \rightsquigarrow \mathcal{D}}{\mathcal{C} : A : (\Delta; \Gamma \Leftarrow B) \mapsto (\Delta'; \Gamma' \Rightarrow A) \rightsquigarrow \mathcal{D}} \text{ FCTyp-anno2}$$

$$\frac{\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta', \alpha * B; \Gamma' \Rightarrow B') \rightsquigarrow \mathcal{D} \qquad \Delta' \vdash B}{\Lambda(\alpha * B). \mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow \forall(\alpha * B). B') \rightsquigarrow \Lambda\alpha. \mathcal{D}} \text{ FCTyp-tabs2}$$

$$\frac{\begin{array}{c}\mathcal{C} : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow \forall(\alpha * A_1). A_2) \rightsquigarrow \mathcal{D} \\ \Delta' \vdash t \qquad \Delta' \vdash t * A_1\end{array}}{\mathcal{C}\, B : (\Delta; \Gamma \Leftarrow A) \mapsto (\Delta'; \Gamma' \Rightarrow [t/\alpha]A_2) \rightsquigarrow \mathcal{D}\, |t|} \text{ FCTyp-tapp2}$$

$\boxed{\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Leftarrow B) \rightsquigarrow \mathcal{D}}$ *(Context typing IV)*

$$\frac{\mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma', x : A_1 \Leftarrow A_2) \rightsquigarrow \mathcal{D} \qquad \Delta' \vdash A_1}{\lambda x.\, \mathcal{C} : (\Delta; \Gamma \Rightarrow A) \mapsto (\Delta'; \Gamma' \Leftarrow A_1 \to A_2) \rightsquigarrow \lambda x.\, \mathcal{D}}\ \textsc{FCTyp-abs1}$$

C.1 F_{co}

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

$$\frac{}{\Phi \vdash \cdot}\ \textsc{wfe-empty} \qquad \frac{\Phi \vdash \tau \qquad \Phi \vdash \Psi}{\Phi \vdash \Psi, x : \tau}\ \textsc{wfe-var}$$

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

$$\frac{}{\Phi \vdash \mathsf{Int}}\ \textsc{wft-int} \qquad \frac{\alpha \in \Phi}{\Phi \vdash \alpha}\ \textsc{wft-var} \qquad \frac{\Phi \vdash \tau_1 \qquad \Phi \vdash \tau_2}{\Phi \vdash \tau_1 \to \tau_2}\ \textsc{wft-arrow} \qquad \frac{\Phi \vdash \tau_1 \qquad \Phi \vdash \tau_2}{\Phi \vdash \tau_1 \times \tau_2}\ \textsc{wft-prod} \qquad \frac{\Phi, \alpha \vdash \tau_2}{\Phi \vdash \forall \alpha.\, \tau_2}\ \textsc{wft-all}$$

$\boxed{c \vdash \tau_1 \rhd \tau_2}$ *(Coercion typing)*

$$\frac{}{\mathsf{id} \vdash \tau \rhd \tau}\ \textsc{ct-refl} \qquad \frac{c_1 \vdash \tau_2 \rhd \tau_3 \qquad c_2 \vdash \tau_1 \rhd \tau_2}{c_1 \circ c_2 \vdash \tau_1 \rhd \tau_3}\ \textsc{ct-trans} \qquad \frac{}{\mathsf{top} \vdash \tau \rhd \langle\rangle}\ \textsc{ct-top}$$

$$\frac{}{\mathsf{top}_{\to} \vdash \langle\rangle \rhd \langle\rangle \to \langle\rangle}\ \textsc{ct-topArr} \qquad \frac{c_1 \vdash \tau_1' \rhd \tau_1 \qquad c_2 \vdash \tau_2 \rhd \tau_2'}{c_1 \to c_2 \vdash \tau_1 \to \tau_2 \rhd \tau_1' \to \tau_2'}\ \textsc{ct-arr}$$

$$\frac{c_1 \vdash \tau_1 \rhd \tau_2 \qquad c_2 \vdash \tau_1 \rhd \tau_3}{\langle c_1, c_2 \rangle \vdash \tau_1 \rhd \tau_2 \times \tau_3}\ \textsc{ct-pair} \qquad \frac{}{\pi_1 \vdash \tau_1 \times \tau_2 \rhd \tau_1}\ \textsc{ct-projl} \qquad \frac{}{\pi_2 \vdash \tau_1 \times \tau_2 \rhd \tau_2}\ \textsc{ct-projr}$$

$$\frac{c \vdash \tau_1 \rhd \tau_2}{c_{\forall} \vdash \forall \alpha.\, \tau_1 \rhd \forall \alpha.\, \tau_2}\ \textsc{ct-forall} \qquad \frac{}{\mathsf{dist}_{\to} \vdash (\tau_1 \to \tau_2) \times (\tau_1 \to \tau_3) \rhd \tau_1 \to \tau_2 \times \tau_3}\ \textsc{ct-distArr}$$

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

$$\frac{\Phi \vdash \Psi}{\Phi; \Psi \vdash \langle\rangle : \langle\rangle}\ \textsc{Ft-unit} \qquad \frac{\Phi \vdash \Psi}{\Phi; \Psi \vdash i : \rho}\ \textsc{Ft-int} \qquad \frac{\Phi \vdash \Psi \qquad (x : \tau) \in \Psi}{\Phi; \Psi \vdash x : \tau}\ \textsc{Ft-var}$$

$$\frac{\Phi; \Psi, x : \tau_1 \vdash e : \tau_2 \qquad \Phi \vdash \tau_1}{\Phi; \Psi \vdash \lambda x.\, e : \tau_1 \to \tau_2}\ \textsc{Ft-abs} \qquad \frac{\Phi; \Psi \vdash e_1 : \tau_1 \to \tau_2 \qquad \Phi; \Psi \vdash e_2 : \tau_1}{\Phi; \Psi \vdash e_1\, e_2 : \tau_2}\ \textsc{Ft-app}$$

$$\frac{\Phi, \alpha; \Psi \vdash e : \tau \qquad \Phi \vdash \Psi}{\Phi; \Psi \vdash \Lambda\alpha.\, e : \forall\alpha.\, \tau}\ \textsc{Ft-tabs} \qquad \frac{\Phi; \Psi \vdash e : \forall\alpha.\, \tau' \qquad \Phi \vdash \tau}{\Phi; \Psi \vdash e\, \tau : [\tau/\alpha]\tau'}\ \textsc{Ft-tapp}$$

$$\frac{\Phi; \Psi \vdash e_1 : \tau_1 \qquad \Phi; \Psi \vdash e_2 : \tau_2}{\Phi; \Psi \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2}\ \textsc{Ft-pair} \qquad \frac{\Phi; \Psi \vdash e : \tau_1 \qquad c \vdash \tau_1 \rhd \tau_2 \qquad \Phi \vdash \tau_2}{\Phi; \Psi \vdash c\, e : \tau_2}\ \textsc{Ft-capp}$$

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

$$\frac{}{\mathsf{id}\, v \longrightarrow v}\ \textsc{r-id} \qquad \frac{}{(c_1 \circ c_2)\, v \longrightarrow c_1\, (c_2\, v)}\ \textsc{r-trans} \qquad \frac{}{\mathsf{top}\, v \longrightarrow \langle\rangle}\ \textsc{r-top} \qquad \frac{}{(\mathsf{top}_{\to}\, \langle\rangle)\, \langle\rangle \longrightarrow \langle\rangle}\ \textsc{r-topArr}$$

$$\frac{}{\langle c_1, c_2\rangle\, v \longrightarrow \langle c_1\, v, c_2\, v\rangle}\ \textsc{r-pair} \qquad \frac{}{((c_1 \to c_2)\, v_1)\, v_2 \longrightarrow c_2\, (v_1\, (c_1\, v_2))}\ \textsc{r-arr}$$

$$\frac{}{(\mathsf{dist}_{\to}\, \langle v_1, v_2\rangle)\, v_3 \longrightarrow \langle v_1\, v_3, v_2\, v_3\rangle}\ \textsc{r-distArr} \qquad \frac{}{\pi_1\, \langle v_1, v_2\rangle \longrightarrow v_1}\ \textsc{r-projl} \qquad \frac{}{\pi_2\, \langle v_1, v_2\rangle \longrightarrow v_2}\ \textsc{r-projr}$$

$$\frac{}{(c_{\forall}\, v)\, \tau \longrightarrow c\, (v\, \tau)}\ \textsc{r-forall} \qquad \frac{}{(\Lambda\alpha.\, e)\, \tau \longrightarrow [\tau/\alpha]e}\ \textsc{r-tapp} \qquad \frac{}{(\lambda x.\, e)\, v \longrightarrow [v/x]e}\ \textsc{r-app} \qquad \frac{e \longrightarrow e'}{\mathcal{E}[e] \longrightarrow \mathcal{E}[e']}\ \textsc{r-ctxt}$$

D SEDEL Typing Rules, in Full

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

$$\text{WF-TOP}\ \frac{}{\Delta \vdash \top} \qquad \text{WF-INT}\ \frac{}{\Delta \vdash \rho} \qquad \text{WF-ARR}\ \frac{\Delta \vdash \mathcal{A} \qquad \Delta \vdash \mathcal{B}}{\Delta \vdash \mathcal{A} \to \mathcal{B}} \qquad \text{WF-RCD}\ \frac{\Delta \vdash \mathcal{A}}{\Delta \vdash \{l : \mathcal{A}\}} \qquad \text{WF-VAR}\ \frac{(\alpha * \mathcal{A}) \in \Delta}{\Delta \vdash \alpha}$$

$$\text{WF-AND}\ \frac{\Delta \vdash \mathcal{A} \qquad \Delta \vdash \mathcal{B}}{\Delta \vdash \mathcal{A} \,\&\, \mathcal{B}} \qquad \text{WF-FORALL}\ \frac{\Delta \vdash \mathcal{A} \qquad \Delta, \alpha * \mathcal{A} \vdash \mathcal{B}}{\Delta \vdash \forall(\alpha * \mathcal{A}).\, \mathcal{B}} \qquad \text{WF-TRAIT}\ \frac{\Delta \vdash \mathcal{A} \qquad \Delta \vdash \mathcal{B}}{\Delta \vdash \textbf{Trait}\,[\mathcal{A}, \mathcal{B}]}$$

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

$$\text{TS-REFL}\ \frac{}{\mathcal{A} <: \mathcal{A}} \qquad \text{TS-TRANS}\ \frac{\mathcal{A}_2 <: \mathcal{A}_3 \qquad \mathcal{A}_1 <: \mathcal{A}_2}{\mathcal{A}_1 <: \mathcal{A}_3} \qquad \text{TS-TOP}\ \frac{}{\mathcal{A} <: \top} \qquad \text{TS-RCD}\ \frac{\mathcal{A} <: \mathcal{B}}{\{l : \mathcal{A}\} <: \{l : \mathcal{B}\}}$$

$$\text{TS-ARR}\ \frac{\mathcal{B}_1 <: \mathcal{A}_1 \qquad \mathcal{A}_2 <: \mathcal{B}_2}{\mathcal{A}_1 \to \mathcal{A}_2 <: \mathcal{B}_1 \to \mathcal{B}_2} \qquad \text{TS-ANDL}\ \frac{}{\mathcal{A}_1 \,\&\, \mathcal{A}_2 <: \mathcal{A}_1} \qquad \text{TS-ANDR}\ \frac{}{\mathcal{A}_1 \,\&\, \mathcal{A}_2 <: \mathcal{A}_2}$$

$$\text{TS-AND}\ \frac{\mathcal{A}_1 <: \mathcal{A}_2 \qquad \mathcal{A}_1 <: \mathcal{A}_3}{\mathcal{A}_1 <: \mathcal{A}_2 \,\&\, \mathcal{A}_3} \qquad \text{TS-DISTARR}\ \frac{}{(\mathcal{A}_1 \to \mathcal{A}_2) \,\&\, (\mathcal{A}_1 \to \mathcal{A}_3) <: \mathcal{A}_1 \to \mathcal{A}_2 \,\&\, \mathcal{A}_3}$$

$$\text{TS-TOPARR}\ \frac{}{\top <: \top \to \top} \qquad \text{TS-DISTRCD}\ \frac{}{\{l : \mathcal{A}\} \,\&\, \{l : \mathcal{B}\} <: \{l : \mathcal{A} \,\&\, \mathcal{B}\}} \qquad \text{TS-TOPRCD}\ \frac{}{\top <: \{l : \top\}}$$

$$\text{TS-TRAIT}\ \frac{\mathcal{B}_1 <: \mathcal{A}_1 \qquad \mathcal{A}_2 <: \mathcal{B}_2}{\textbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] <: \textbf{Trait}\,[\mathcal{B}_1, \mathcal{B}_2]}$$

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

$$\textsc{SD-topL}\quad \frac{}{\Delta \vdash \top * \mathcal{A}}$$

$$\textsc{SD-topR}\quad \frac{}{\Delta \vdash \mathcal{A} * \top}$$

$$\textsc{SD-arr}\quad \frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \mathcal{A}_1 \to \mathcal{A}_2 * \mathcal{B}_1 \to \mathcal{B}_2}$$

$$\textsc{SD-andL}\quad \frac{\Delta \vdash \mathcal{A}_1 * \mathcal{B} \qquad \Delta \vdash \mathcal{A}_2 * \mathcal{B}}{\Delta \vdash \mathcal{A}_1 \,\&\, \mathcal{A}_2 * \mathcal{B}}$$

$$\textsc{SD-andR}\quad \frac{\Delta \vdash \mathcal{A} * \mathcal{B}_1 \qquad \Delta \vdash \mathcal{A} * \mathcal{B}_2}{\Delta \vdash \mathcal{A} * \mathcal{B}_1 \,\&\, \mathcal{B}_2}$$

$$\textsc{SD-rcdEq}\quad \frac{\Delta \vdash \mathcal{A} * \mathcal{B}}{\Delta \vdash \{l : \mathcal{A}\} * \{l : \mathcal{B}\}}$$

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

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

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

$$\textsc{SD-forall}\quad \frac{\Delta, \alpha * \mathcal{A}_1 \,\&\, \mathcal{A}_2 \vdash \mathcal{B}_1 * \mathcal{B}_2}{\Delta \vdash \forall(\alpha * \mathcal{A}_1).\, \mathcal{B}_1 * \forall(\alpha * \mathcal{A}_2).\, \mathcal{B}_2}$$

$$\textsc{SD-trait}\quad \frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \textbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] * \textbf{Trait}\,[\mathcal{B}_1, \mathcal{B}_2]}$$

$$\textsc{SD-traitArr1}\quad \frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \textbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] * \mathcal{B}_1 \to \mathcal{B}_2}$$

$$\textsc{SD-traitArr2}\quad \frac{\Delta \vdash \mathcal{A}_2 * \mathcal{B}_2}{\Delta \vdash \mathcal{A}_1 \to \mathcal{A}_2 * \textbf{Trait}\,[\mathcal{B}_1, \mathcal{B}_2]}$$

$$\textsc{SD-ax}\quad \frac{\mathcal{A} *_{ax} \mathcal{B}}{\Delta \vdash \mathcal{A} * \mathcal{B}}$$

$\boxed{\mathcal{A} *_{ax} \mathcal{B}}$ *(Disjointness axiom)*

$$\textsc{SDax-sym}\quad \frac{\mathcal{B} *_{ax} \mathcal{A}}{\mathcal{A} *_{ax} \mathcal{B}}$$

$$\textsc{SDax-intArr}\quad \frac{}{\rho *_{ax} \mathcal{A}_1 \to \mathcal{A}_2}$$

$$\textsc{SDax-intRcd}\quad \frac{}{\rho *_{ax} \{l : \mathcal{A}\}}$$

$$\textsc{SDax-intAll}\quad \frac{}{\rho *_{ax} \forall(\alpha * \mathcal{B}_1).\, \mathcal{B}_2}$$

$$\textsc{SDax-arrAll}\quad \frac{}{\mathcal{A}_1 \to \mathcal{A}_2 *_{ax} \forall(\alpha * \mathcal{B}_1).\, \mathcal{B}_2}$$

$$\textsc{SDax-arrRcd}\quad \frac{}{\mathcal{A}_1 \to \mathcal{A}_2 *_{ax} \{l : \mathcal{B}\}}$$

$$\textsc{SDax-allRcd}\quad \frac{}{\forall(\alpha * \mathcal{A}_1).\, \mathcal{A}_2 *_{ax} \{l : \mathcal{B}\}}$$

$$\textsc{SDax-intTrait}\quad \frac{}{\rho *_{ax} \textbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2]}$$

$$\textsc{SDax-traitAll}\quad \frac{}{\textbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] *_{ax} \forall(\alpha * \mathcal{B}_1).\, \mathcal{B}_2}$$

$$\textsc{SDax-traitRcd}\quad \frac{}{\textbf{Trait}\,[\mathcal{A}_1, \mathcal{A}_2] *_{ax} \{l : \mathcal{B}\}}$$

$\boxed{\Delta; \Gamma \vdash \mathcal{T} \;\Rightarrow\; \mathcal{A} \rightsquigarrow E}$ *(Inference)*

$$\textsc{ST-top}\quad \frac{}{\Delta; \Gamma \vdash \top \;\Rightarrow\; \top \rightsquigarrow \top} \qquad \textsc{ST-int}\quad \frac{}{\Delta; \Gamma \vdash i \;\Rightarrow\; \rho \rightsquigarrow i} \qquad \textsc{ST-var}\quad \frac{(x : \mathcal{A}) \in \Delta}{\Delta; \Gamma \vdash x \;\Rightarrow\; \mathcal{A} \rightsquigarrow x}$$

$$\textsc{ST-app}\quad \frac{\Delta; \Gamma \vdash \mathcal{T}_1 \;\Rightarrow\; \mathcal{A}_1 \to \mathcal{A}_2 \rightsquigarrow E_1 \qquad \Delta; \Gamma \vdash \mathcal{T}_2 \;\Leftarrow\; \mathcal{A}_1 \rightsquigarrow E_2}{\Delta; \Gamma \vdash \mathcal{T}_1\, \mathcal{T}_2 \;\Rightarrow\; \mathcal{A}_2 \rightsquigarrow E_1\, E_2}$$

$$\textsc{ST-merge}\quad \frac{\Delta; \Gamma \vdash \mathcal{T}_1 \;\Rightarrow\; \mathcal{A} \rightsquigarrow E_1 \qquad \Delta; \Gamma \vdash \mathcal{T}_2 \;\Rightarrow\; \mathcal{B} \rightsquigarrow E_2 \qquad \Delta \vdash \mathcal{A} * \mathcal{B}}{\Delta; \Gamma \vdash \mathcal{T}_1 ,, \mathcal{T}_2 \;\Rightarrow\; \mathcal{A} \,\&\, \mathcal{B} \rightsquigarrow E_1 ,, E_2}$$

$$\textsc{ST-anno}\quad \frac{\Delta; \Gamma \vdash \mathcal{T} \;\Leftarrow\; \mathcal{A} \rightsquigarrow E}{\Delta; \Gamma \vdash \mathcal{T} : \mathcal{A} \;\Rightarrow\; \mathcal{A} \rightsquigarrow E : |\mathcal{A}|}$$

$$\textsc{ST-tabs}\quad \frac{\Delta \vdash \mathcal{A} \qquad \Delta, \alpha * \mathcal{A}; \Gamma \vdash \mathcal{T} \;\Rightarrow\; \mathcal{B} \rightsquigarrow E}{\Delta; \Gamma \vdash \Lambda(\alpha * \mathcal{A}).\, \mathcal{T} \;\Rightarrow\; \forall(\alpha * \mathcal{A}).\, \mathcal{B} \rightsquigarrow \Lambda(\alpha * |\mathcal{A}|).\, E}$$

$$\textsc{ST-tapp}\quad \frac{\Delta; \Gamma \vdash \mathcal{T} \;\Rightarrow\; \forall(\alpha * \mathcal{B}_1).\, \mathcal{B}_2 \rightsquigarrow E \qquad \Delta \vdash \mathcal{A} \qquad \Delta \vdash \mathcal{A} * \mathcal{B}_1}{\Delta; \Gamma \vdash \mathcal{T}\, \mathcal{A} \;\Rightarrow\; [\mathcal{A}/\alpha]\mathcal{B}_2 \rightsquigarrow E\, |\mathcal{A}|}$$

$$\textsc{ST-rcd}\quad \frac{\Delta; \Gamma \vdash \mathcal{T} \;\Rightarrow\; \mathcal{A} \rightsquigarrow E}{\Delta; \Gamma \vdash \{l = \mathcal{T}\} \;\Rightarrow\; \{l : \mathcal{A}\} \rightsquigarrow \{l = E\}} \qquad \textsc{ST-proj}\quad \frac{\Delta; \Gamma \vdash \mathcal{T} \;\Rightarrow\; \{l : \mathcal{A}\} \rightsquigarrow E}{\Delta; \Gamma \vdash \mathcal{T}.l \;\Rightarrow\; \mathcal{A} \rightsquigarrow E.l}$$

$$\textsc{ST-trait}\quad \frac{\begin{array}{c} \overline{\Delta; \Gamma, \mathsf{self} : \mathcal{B} \vdash \mathcal{T}_i \;\Rightarrow\; \textbf{Trait}\,[\mathcal{B}_i, \mathcal{C}_i] \rightsquigarrow E_i}^{i \in 1..n} \\ \Delta; \Gamma, \mathsf{self} : \mathcal{B} \vdash \{\, \overline{l_j = \mathcal{T}_j'}^{j \in 1..m} \,\} \;\Rightarrow\; \mathcal{C} \rightsquigarrow E \\ \overline{\mathcal{B} <: \mathcal{B}_i}^{i \in 1..n} \qquad \Delta \vdash \mathcal{C}_1 * \,..\, * \mathcal{C}_n * \mathcal{C} \qquad \mathcal{C}_1 \,\&\, ..\, \&\, \mathcal{C}_n \,\&\, \mathcal{C} <: \mathcal{A} \end{array}}{\begin{array}{c} \Delta; \Gamma \vdash \textbf{trait}\,[\mathsf{self} : \mathcal{B}]\ \textbf{inherits}\, \overline{\mathcal{T}_i}^{i \in 1..n} \{\, \overline{l_j = \mathcal{T}_j'}^{j \in 1..m} \,\} : \mathcal{A} \;\Rightarrow\; \textbf{Trait}\,[\mathcal{B}, \mathcal{A}] \rightsquigarrow \\ \lambda \mathsf{self} : |\mathcal{B}|.\, (((\overline{(E_i\, \mathsf{self})}^{i \in 1..n}) ,, E) \end{array}}$$

ST-NEW
$$\frac{\overline{\Delta;\Gamma \vdash \mathcal{T}_i \;\Rightarrow\; \textbf{Trait}\,[\mathcal{A}_i,\mathcal{B}_i] \rightsquigarrow E_i}^{i\in 1..n} \qquad \overline{\mathcal{A} <: \mathcal{A}_i}^{i\in 1..n} \qquad \Delta \vdash \mathcal{B}_1 * \,..\, * \mathcal{B}_n \qquad \mathcal{B}_1 \,\&\, ..\, \& \mathcal{B}_n <: \mathcal{A}}{\Delta;\Gamma \vdash \textbf{new}\,[\mathcal{A}](\overline{\mathcal{T}_i}^{i\in 1..n}) \;\Rightarrow\; \mathcal{A} \rightsquigarrow \textbf{letrec}\,\mathsf{self} : |\mathcal{A}| = \overline{(E_i\,\mathsf{self})}^{i\in 1..n}\,\textbf{in}\,\mathsf{self}}$$

ST-FORWARD
$$\frac{\Delta;\Gamma \vdash \mathcal{T}_1 \;\Rightarrow\; \textbf{Trait}\,[\mathcal{A},\mathcal{B}] \rightsquigarrow E_1 \qquad \Delta;\Gamma \vdash \mathcal{T}_2 \;\Leftarrow\; \mathcal{A} \rightsquigarrow E_2}{\Delta;\Gamma \vdash \mathcal{T}_1 \,\hat{}\, \mathcal{T}_2 \;\Rightarrow\; \mathcal{B} \rightsquigarrow E_1\,E_2}$$

$\boxed{\Delta;\Gamma \vdash \mathcal{T} \;\Leftarrow\; \mathcal{A} \rightsquigarrow E}$ *(Checking)*

ST-ABS
$$\frac{\Delta \vdash \mathcal{A} \qquad \Delta;\Gamma, x:\mathcal{A} \vdash \mathcal{T} \;\Leftarrow\; \mathcal{B} \rightsquigarrow E}{\Delta;\Gamma \vdash \lambda x.\mathcal{T} \;\Leftarrow\; \mathcal{A} \to \mathcal{B} \rightsquigarrow \lambda x.\,E}$$

ST-SUB
$$\frac{\Delta;\Gamma \vdash \mathcal{T} \;\Rightarrow\; \mathcal{A} \rightsquigarrow E \qquad \mathcal{A} <: \mathcal{B} \qquad \Delta \vdash \mathcal{B}}{\Delta;\Gamma \vdash \mathcal{T} \;\Leftarrow\; \mathcal{B} \rightsquigarrow E}$$

Bibliography

Amal Ahmed. 2006. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming*. Springer, 69–83.

Amal Jamil Ahmed. 2004. *Semantics of types for mutable state*. Ph.D. Dissertation. Princeton University.

Eric E. Allen, Jonathan Bannet, and Robert Cartwright. 2003. A first-class approach to genericity. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*.

João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. 2017. Disjoint Polymorphism. In *European Symposium on Programming*.

Nada Amin, Adriaan Moors, and Martin Odersky. 2012. Dependent object types. In *Workshop on Foundations of Object-Oriented Languages*.

Davide Ancona, Giovanni Lagorio, and Elena Zucca. 2003. Jam–designing a Java extension with mixins. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*.

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.

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)*.

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.

Lorenzo Bettini and Ferruccio Damiani. 2017. Xtraitj: Traits for the Java platform. *Journal of Systems and Software* 131 (2017), 419 – 441.

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.

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

Dariusz Biernacki and Piotr Polesiuk. 2015. Logical relations for coherence of effect subtyping. In *LIPIcs*.

Viviana Bono, Amit Patel, and Vitaly Shmatikov. 1999. A core calculus of classes and mixins. In *European Conference on Object-Oriented Programming (ECOOP)*.

Gilad Bracha. 1992. *The programming language jigsaw: mixins, modularity and multiple inheritance*. Ph.D. Dissertation. Dept. of Computer Science, University of Utah.

Gilad Bracha and William R. Cook. 1990. Mixin-based inheritance. In *Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*.

Gilad Bracha and Gary Lindstrom. 1992. Modularity meets Inheritance. In *International Conference on Computer Languages*. IEEE Computer Society, 282–290.

Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. 1991. Inheritance as implicit coercion. *Information and Computation* 93, 1 (1991), 172–221.

Kim Bruce, Luca Cardelli, Giuseppe Castagna, Gary T Leavens, Benjamin Pierce, et al. 1996. On binary methods. In *Theory and Practice of Object Systems*.

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*.

Luca Cardelli. 1984. A semantics of multiple inheritance. *Semantics of data types* (1984), 51–67.

Luca Cardelli. 1985. Amber. In *Combinators and Functional Programming Languages, Thirteenth Spring School of the LITP, Val d'Ajol, France, May 6-10, 1985, Proceedings*. 21–47.

Luca Cardelli, Simone Martini, John C Mitchell, and Andre Scedrov. 1994. An extension of system F with subtyping. *Information and Computation* 109, 1-2 (1994), 4–56.

Luca Cardelli and John C Mitchell. 1989. Operations on records. In *International Conference on Mathematical Foundations of Programming Semantics*. Springer, 22–52.

Luca Cardelli and Peter Wegner. 1985. On understanding types, data abstraction, and polymorphism. *Comput. Surveys* 17, 4 (1985), 471–523.

Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. 1992. A calculus for overloaded functions with subtyping. In *LFP*.

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)*.

David Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. 2007. Tribe: More Types for Virtual Classes. In *AOSD*.

Adriana B Compagnoni and Benjamin C Pierce. 1996. Higher-order intersection types and multiple inheritance. *MSCS* 6, 5 (1996), 469–501.

Steve Cook. 1987. Varieties of inheritance. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*. 35–40.

William R. Cook. 2013. *Anatomy of Programming Languages*. http://www.cs.utexas.edu/~wcook/anatomy/

William R. Cook, Walter Hill, and Peter S Canning. 1989. Inheritance is not subtyping. In *Principles of Programming Languages (POPL)*.

William R. Cook and Jens Palsberg. 1989. A denotational semantics of inheritance and its correctness. In *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*.

Mario Coppo and Mariangiola Dezani-Ciancaglini. 1978. A new type assignment for λ-terms. 19 (01 1978), 139–156.

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 1981. Functional Characters of Solvable Terms. *Mathematical Logic Quarterly* 27, 2-6 (1981), 45–58.

Andrea Corradi, Marco Servetto, and Elena Zucca. 2012. DeepFJig — Modular composition of nested classes. *The Journal of Object Technology* 11, 2 (2012), 1:1.

Karl Crary, Robert Harper, and Sidd Puri. 1999. What is a recursive module?. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation - PLDI '99*. ACM Press.

Pierre-Louis Curien and Giorgio Ghelli. 1992. Coherence of subsumption, minimum typing and type-checking in F $\leq$. *MSCS* 2, 01 (1992), 55.

Haskell B. Curry and Robert Feys. 1958. *Combinatory Logic*. Vol. 1. North Holland. Second edition, 1968.

Rowan Davies and Frank Pfenning. 2000. Intersection types and computational effects. In *International Conference on Functional Programming*.

Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. 2006. Traits: A Mechanism for Fine-grained Reuse. *ACM Transactions on Programming Languages and Systems* 28, 2 (3 2006), 331–388.

Joshua Dunfield. 2014. Elaborating intersection and union types. *Journal of Functional Programming* 24, 2-3 (2014), 133–165.

Joshua Dunfield and Neelakantan R Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism. In *International Conference on Functional Programming (ICFP)*.

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)*.

Ecma International. 2015. *ECMAScript 2015 Language Specification* (6th ed.). Geneva. http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf

Erik Ernst. 2000. gbeta-a language with virtual attributes, Block Structure, and Propagating, Dynamic Inheritance. *DAIMI Report Series* 29, 549 (2000).

Erik Ernst. 2001. Family Polymorphism. In *European Conference on Object-Oriented Programming*.

Erik Ernst. 2003. Higher-Order Hierarchies. In *European Conference on Object-Oriented Programming*.

Erik Ernst. 2004. The expression problem, Scandinavian style. *On Mechanisms For Specialization* (2004), 27.

Erik Ernst, Klaus Ostermann, and William R. Cook. 2006. A Virtual Class Calculus. In *Symposium on Principles of Programming Languages*.

Facebook. 2014. Flow. `https://flow.org/`. (2014).

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. 2002. DrScheme: a programming environment for Scheme. *J. Funct. Program.* 12, 2 (2002), 159–182.

Kathleen Fisher and John Reppy. 2000. Extending Moby with inheritance-based subtyping. In *European Conference on Object-Oriented Programming*. 83–107.

Kathleen Fisher and John Reppy. 2004. A typed calculus of traits. In *Workshop on Foundations of Object-Oriented Languages*.

Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. 2006. Scheme with Classes, Mixins, and Traits. In *Programming Languages and Systems (APLAS)*.

Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1998. Classes and mixins. In *Principles of Programming Languages (POPL)*.

Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Conference on Programming Language Design and Implementation*.

Benedict R Gaster and Mark P Jones. 1996. *A polymorphic type system for extensible records and variants*. Technical Report. University of Nottingham.

Robert Harper. 2016. *Practical Foundations for Programming Languages*. Cambridge University Press.

Robert Harper and Benjamin Pierce. 1991. A Record Calculus Based on Symmetric Concatenation. In *Principles of Programming Languages (POPL)*.

Fritz Henglein. 1994. Dynamic typing: syntax and proof theory. *Science of Computer Programming* 22, 3 (6 1994), 197–230.

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Symbolic Computation* 23, 2 (2010), 167.

Roger Hindley. 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society* 146 (1969), 29–60.

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)*.

Timothy Jones, Michael Homer, James Noble, and Kim B. Bruce. 2016. Object Inheritance Without Classes. In *European Conference on Object-Oriented Programming (ECOOP)*.

Toshihiko Kurata and Masako Takahashi. 1995. Decidable properties of intersection type systems. *Typed Lambda Calculi and Applications* (1995), 297–311.

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.

Joachim Lambek. 1985. Cartesian closed categories and typed λ-calculi. In *LITP Spring School on Theoretical Computer Science*. 136–175.

Olivier Laurent. 2012a. Intersection types with subtyping by means of cut elimination. *Fundamenta Informaticae* 121, 1-4 (2012), 203–226.

Olivier Laurent. 2012b. A syntactic introduction to intersection types. (2012). Unpublished note.

Olivier Laurent. 2018. Intersection Subtyping with Constructors. In *Proceedings of the Ninth Workshop on Intersection Types and Related Systems*.

Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin. 2015. A Theory of Tagged Objects. In *European Conference on Object-Oriented Programming (ECOOP)*.

Ole Lehrmann Madsen, Birger Mø-Pedersen, and Kristen Nygaard. 1993. *Object-oriented Programming in the BETA Programming Language*. ACM Press/Addison-Wesley Publishing Co.

Daan Leijen. 2005. Extensible records with scoped labels. *Trends in Functional Programming* 5 (2005), 297–312.

Daniel Leivant. 1991. Finitely stratified polymorphism. *Information and Computation* 93, 1 (1991), 93–113.

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)*.

David MacQueen. 1984. Modules for standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming - LFP '84*.

O. L. Madsen and B. Moller-Pedersen. 1989. Virtual classes: a powerful mechanism in object-oriented programming. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Bertrand Meyer. 1987. Eiffel: programming for reusability and extendibility. *ACM Sigplan Notices* 22, 2 (1987), 85–94.

Microsoft. 2012. TypeScript. https://www.typescriptlang.org/. (2012).

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

Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92.

James Hiram Morris Jr. 1969. *Lambda-calculus models of programming languages*. Ph.D. Dissertation. Massachusetts Institute of Technology.

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*.

Oscar Nierstrasz, Stéphane Ducasse, and Nathanael Schärli. 2006. Flattening Traits. *Journal of Object Technology* 5, 4 (May 2006), 129–148.

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.

Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. 2004. Scalable extensibility via nested inheritance. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

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)*.

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.

Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *Symposium on Principles of Programming Languages (POPL)*.

Martin Odersky and Matthias Zenger. 2005. Scalable component abstractions. In *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA 2005)*.

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)*.

Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses. In *European Conference on Object-Oriented Programming (ECOOP)*.

Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint intersection types. In *International Conference on Functional Programming*.

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)*.

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)*.

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.

Benjamin C Pierce. 1989. *A decision procedure for the subtype relation on intersection types with bounded variables*. Technical Report. Carnegie Mellon University.

Benjamin C Pierce. 1991. *Programming with intersection types and bounded polymorphism*. Ph.D. Dissertation. University of Pennsylvania.

Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press.

Gordon Plotkin. 1973. *Lambda-definability and logical relations*. Edinburgh University.

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.

Xin Qi and Andrew C. Myers. 2009. Sharing Classes Between Families. In *Conference on Programming Language Design and Implementation*.

Redhat. 2011. Ceylon. `https://ceylon-lang.org/`. (2011).

Jakob Rehof and Paweł Urzyczyn. 2011. Finite combinatory logic with intersection types. In *International Conference on Typed Lambda Calculi and Applications*.

Tillmann Rendel, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2014. From Object Algebras to Attribute Grammars. In *Object Oriented Programming, Systems Languages and Applications (OOPSLA)*.

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *IFIP*.

John C Reynolds. 1988. *Preliminary design of the programming language Forsythe*. Technical Report. Carnegie Mellon University.

John C. Reynolds. 1991. The coherence of languages with intersection types. In *Lecture Notes in Computer Science (LNCS)*. Springer Berlin Heidelberg, 675–700.

John C Reynolds. 1997. Design of the programming language Forsythe. In *ALGOL-like languages*. 173–233.

Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Andreas Rossberg. 2015. 1ML – core and modules united (F-ing first-class modules). In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming - ICFP 2015*. ACM Press.

Andreas Rossberg and Derek Dreyer. 2013. Mixin' Up the ML Module System. *ACM Transactions on Programming Languages and Systems* 35, 1 (4 2013), 1–84.

Andreas Rossberg, Claudio Russo, and Derek Dreyer. 2014. F-ing modules. *Journal of Functional Programming* 24, 05 (2014), 529–607.

Claudio V. Russo. 2001. Recursive structures for standard ML. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming - ICFP '01*. ACM Press.

Chieri Saito and Atsushi Igarashi. 2009. Matching *ThisType* to subtyping. In *Symposium on Applied Computing (SAC)*.

Chieri Saito, Atsushi Igarashi, and Mirko Viroli. 2007. Lightweight family polymorphism. *Journal of Functional Programming* 18, 03 (2007).

Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. 2003. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming (ECOOP)*.

Nathanael Scharli, St Ducasse, Roel Wuyts, Andrew Black, et al. 2003. Traits: The formal model. (2003).

Jan Schwinghammer. 2008. Coherence of subsumption for monadic types. *Journal of Functional Programming* 19, 02 (2008), 157.

Marco Servetto and Elena Zucca. 2014. A meta-circular language for active libraries. *Science of Computer Programming* 95 (2014), 219 – 253.

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*.

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*.

Yannis Smaragdakis and Don S. Batory. 2000. Mixin-Based Programming in C++. In *Generative and Component-Based Software Engineering (GCSE)*.

Charles Smith and Sophia Drossopoulou. 2005. Chai: Traits for Java-Like Languages. In *European Conference on Object-Oriented Programming (ECOOP)*, Andrew P. Black (Ed.).

Richard Statman. 1985. Logical relations and the typed λ-calculus. *Information and Control* 65, 2-3 (1985), 85–97.

Rick Statman. 2015. A Finite Model Property for Intersection Types. *Electronic Proceedings in Theoretical Computer Science* 177 (2015), 1–9.

Wouter Swierstra. 2008. Data types à la carte. *Journal of Functional Programming* 18, 4 (2008), 423–436.

W. W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *The Journal of symbolic logic* 32, 2 (1967), 198–212.

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)*.

Mads Torgersen. 2004. The Expression Problem Revisited. In *European Conference on Object-Oriented Programming (ECOOP)*.

David Ungar and Randall B Smith. 1988. SELF: the power of simplicity (object-oriented language). In *Compcon Spring'88. Thirty-Third IEEE Computer Society International Conference, Digest of Papers*.

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*.

Philip Wadler. 1998. The expression problem. *Java-genericity mailing list* (1998).

Mitchell Wand. 1987. Complete type inference for simple objects. In *Symposium on Logic in Computer Science (LICS)*.

Mitchell Wand. 1994. Type inference for objects with instance variables and inheritance. *Theoretical aspects of object-oriented programming* (1994), 97–120.

Yanlin Wang and Bruno C. d. S. Oliveira. 2016. The expression problem, trivially!. In *Proceedings of the 15th International Conference on Modularity*.

Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira. 2018. Consistent Subtyping for All. In *European Symposium on Programming*. 3–30.

Pamela Zave. 1999. Faq sheet on feature interaction. *Link: http://www. research. att. com/~ pamela/faq. html* (1999).

Mathhias Zenger and Martin Odersky. 2005. Independently Extensible Solutions to the Expression Problem. In *Foundations of Object-Oriented Languages*.

Yizhou Zhang and Andrew C. Myers. 2017. Familia: unifying interfaces, type classes, and family polymorphism. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*.