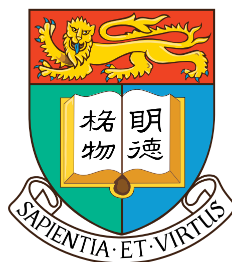
A Blend of Intersection Types and Union Types

by

Baber Rehman



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

August 2023

Abstract of thesis entitled

"A Blend of Intersection Types and Union Types"

Submitted by

Baber Rehman

for the degree of Doctor of Philosophy
at The University of Hong Kong
in August 2023

Languages define the constructs and protocols of communication. Programming languages, in particular, define the constructs that assist in communicating with machines. With the growing complexity and need of software applications, it has become a challenge to design an expressive, yet simple programming language. The complexity in a language design mostly occurs due to the interaction of disparate features. This thesis is an effort in simplifying and formally studying the design of intersection and union types in programming languages.

Intersection and union types are advance and powerful features available in many modern programming languages. Intersection types provide an interface to introduce an expression of multiple types. Union types, on the other hand, provide an interface to express an expression of variant types. Interaction of intersection and union types is known to be non-trivial in theory. This thesis examines the interaction of intersection and union types. Our study starts with a deterministic type-based switch expression for the elimination of union types. Disjointness plays an integral role in keeping the calculus deterministic, which ensures that no two branches of a type-based switch expression overlap. Thus the scrutinee falls in a maximum of one branch. The resulting calculus is called λ_u. We further extend λ_u with powerful and advance features of intersection types, subtyping distributivity, nominal types, and polymorphism. Moreover, we study λ_u with the merge operator. An extension with the merge operator poses novel challenges in determinism.

Intersection and union types are well-known dual features. We also examine the duality of intersection and union types formally in this thesis. The duality unifies some of the subtyping rules and reduces the theoretical complexity of a system with dual features. The benefits include the reduction in number of subtyping rules and simplified proofs for certain theo-

rems such as subtyping transitivity. All of the metatheory of this thesis has been formalized in the Coq theorem prover.

An abstract of 310 words

To my late father...

Declaration

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

. . . .

Baber Rehman

August 2023

Acknowledgments

I have no words to show the gratitude towards my advisor Dr. Bruno C. d. S. Oliveira and I think it certainly cannot be expressed in words. The study of programming languages and specifically the type systems was an alien universe for me before starting the PhD. I have programmed in a few programming languages but never really thought about the theoretical complexity involved behind the design of a sound programming language. All credits due to Bruno's guidance which made me able to learn a glimpse of this once an alien universe. I am grateful to Bruno for his guidance and support throughout my doctoral studies.

I would like to extend my sincere gratitude towards the examiners, reviewers, and the referees. The comments, suggestions, and recommendations from Prof. Sukyoung Ryu, Dr. H.F. Ting, Dr. Chenshu Wu, Dr. Sergio D.N. Lourenco, Dr. Lionel Parreaux, Dr. Marco Campion, Dr. Khurram Shahzad, and Dr. Imran Farid Khan have been imperative.

Afterwards, the support from the Snow (Xuejing Huang) has been incredible. Thank you for showing me a way out when I was lost in in my research. Moving on, the insightful philosophical discussions with Alvin (Mingqi Xue) have been inspiring. This has been a great pleasure for me to meet with Alvin over a meal and discuss various social topics. Perhaps the homo sapiens may learn from their recurring mistakes someday and the world would be a better place. My sincere thanks to all of my colleagues, Xuan Bi (Jeremy), Haoyuan Zhang, Yanlin Wang (Grace), Yanpeng Yang (Linus), Weixin Zhang, Ningning Xie, Jinxu Zhao (Jimmy), Yaoda Zhou, Yaozhu Sun, Wenjia Ye, Xu Xue, Chen Cui, Jinhao Tan, Shengyi Jiang, Litao Zhou (Tony), and Han Xu.

Hong Kong blessed me with the amazing experiences and people. I am grateful to the New College tutorial team and specifically to Prof. CHOW Y.M. Amy for her support and guidance. Sustainable learning from the New College would be an integral part of my career. I am thankful to the beautiful souls I met in Hong Kong. Sharas, Sabir, Niraj, Shujahat, Huzaifa, Mahram, Adeel, Abdullah, Feroz, Atta, Najam, Nadeem and Hassan. I would like to take the opportunity and express my gratitude to my friends and colleagues at CloudPlex PVT LTD, where I worked after my Bachelor's degree. Most of all, Asad Faizi and Haseeb Humayun, thank you for your support. Importantly, I am grateful to Asad Faizi for showing me around Seattle post OPLSS'19.

Being a recipient of the University Postgraduate Fellowship (UPF), I am much obliged to the Lee Shau Kee Postgraduate Fellowship donors for the award. Last but not the least, thanks to my wife, mother and siblings for their incredible support. This would not have been possible without your support. A supporting partner can make the hard times easier to bear. I have felt that through thick and thin with my adorable wife.

Contents

Declaration i

Acknowledgments iii

List of Figures ix

List of Tables xi

I Preface 1

1 Introduction 3
- 1.1 Union types and disjoint switch expressions 6
- 1.2 Duality of subtyping 10
- 1.3 Outline and contributions 13

2 Background 17
- 2.1 Intersection Types 17
- 2.2 Merge Operator 19
- 2.3 Tagged Union Types 21
- 2.4 Un-tagged Union Types 22
- 2.5 Type-directed Elimination forms for Union Types 23
- 2.6 Union Types and Disjoint Switches in Ceylon 25
- 2.7 Nullable Types 26
- 2.8 Polymorphism 28
- 2.9 Duality 28

II Deterministic Union Elimination (λ_u) 31

3 Union Types with Disjoint Switches 33
- 3.1 Overview 33

3.2 The Union Calculus λ_u 37
3.2.1 Syntax 37
3.2.2 Subtyping 38
3.2.3 Disjointness 38
3.2.4 Typing 40
3.2.5 Operational Semantics 42
3.2.6 Type Soundness and Determinism 43
3.2.7 An Alternative Specification for Disjointness 44
3.3 λ_u with Intersections, Distributive Subtyping and Nominal Types 44
3.3.1 Syntax, Well-formedness and Ordinary Types 45
3.3.2 Distributive Subtyping 46
3.3.3 Disjointness Specification 49
3.3.4 Algorithmic Disjointness 51
3.3.5 Typing, Semantics and Metatheory 53
3.4 Switches with Disjoint Polymorphism and Empty Types 54
3.4.1 Disjoint Polymorphism 55
3.4.2 A More General Subtyping Rule for Bottom Types 59
3.4.3 Implementation of Disjoint Switches 60

4 Revisiting Disjointness 61
4.1 Disjointness with Intersection and Union Types 62
4.1.1 Disjointness, Union Ordinary, and Union Splittable Types 63
4.1.2 Essence of Union Ordinary Types 65
4.1.3 Essence of Union Splittable Types 67
4.1.4 Metatheory with Union Ordinary and Union Splittable Types 71
4.2 Redesigning Disjoint Polymorphism 73
4.2.1 Disjointness 74
4.2.2 Subtyping, typing, and operational semantics 76

5 Togetherness: Switches and Merges 83
5.1 Overview 83
5.2 λ_{um} Calculus 85
5.2.1 Syntax, subtyping, and typing 85
5.2.2 Type casting 87
5.2.3 Operational semantics 89
5.2.4 Applicative dispatch and rule step-dispatch 90
5.2.5 Metatheory of λ_{um} 92

5.3 λ_{um} and Dunfield's system 93
5.3.1 Dunfield's calculus 94
5.3.2 Completeness with respect to Dunfield's calculus 96
5.4 Stumbling block: non-determinism 97
5.4.1 Non-determinism in the presence of merge operator 97
5.4.2 Non-determinism in the presence of switch expression 98
5.4.3 Non-determinism with merge operator and switch expression . . . 99

III Duality of Intersection and Union Types 101

6 The Duality of Subtyping (Duotyping) 103
6.1 Overview 103
6.1.1 Subtyping with union and intersection types 103
6.1.2 Subtyping Specifications using Duotyping 104
6.1.3 Implementations using Duotyping 107
6.1.4 Discovering new features 109
6.1.5 New proof techniques 112
6.2 The $\lambda_{\Diamond}^{\wedge\vee}$ calculus 114
6.2.1 Syntax and Duotyping 114
6.2.2 Semantics and type soundness 116
6.2.3 Summary and Comparison 117
6.3 The $F_{k\Diamond}^{\wedge\vee}$ calculus 118
6.3.1 Syntax and Duotyping 118
6.3.2 Semantics and type soundness 121
6.3.3 Summary and Comparison 122
6.4 A Case Study on Duotyping 123
6.4.1 Case Study 123
6.4.2 Does Duotyping provide shorter specifications? 125
6.4.3 Does Duotyping increase the complexity of the formalization and metatheory of the language? 127
6.4.4 Does Duotyping make transitivity proofs simpler? 128
6.4.5 Is Duotyping a generally applicable technique? 128

IV Related Work 131

7 Related Work 133
7.1 Union Types 133
7.2 Disjoint Intersection Types 135
7.3 Overloading and Dynamic Dispatch 136
7.4 Duality in Logic and Programming Language Theory 137
7.5 Generalizations in Type Systems and Type Theory 138

V Epilogue 141

8 Conclusion and Future Work 143
8.1 Conclusion 143
8.2 Future Work 143
8.2.1 Determinism for λ_{um} 143
8.2.2 Multiple Interface Inheritance 147
8.2.3 Explicit Disjointness of Nominal Types 148
8.2.4 Gradual Typing 149

Bibliography 151

VI Technical Appendix 163

A Alternative Disjointness 165
A.1 Common Ordinary Subtypes (COST) 165
A.2 Disjointness 168
A.3 Typing, Operational Semantics, and Type-safety 168

List of Figures

2.1 Ceylon's subtyping hierarchy. Note that `Null` only has `Nothing` as its subtype. 27

3.1 Syntax and subtyping for λ_u. 37
3.2 Bottom-like types, algorithmic disjointness and typing for λ_u. 41
3.3 Operational semantics and approximate type definitions for λ_u. 42
3.4 Syntax and well-formedness. 45
3.5 Distributive subtyping for λ_u with intersection types and nominal types. 47
3.6 Algorithmic subtyping for λ_u with distributivity, intersection and nominal types. 48
3.7 Lowest ordinary subtypes function and additional typing rule for λ_u with intersection types and nominal types. 52
3.8 Syntax, additional typing, subtyping, and reduction rules for λ_u with polymorphism. 55
3.9 Subtyping for polymorphic λ_u. 57
3.10 Typing for polymorphic λ_u. 58
3.11 Operational Semantics for polymorphic λ_u. 59

4.1 Syntax and subtyping for λ_u with intersection types. 62
4.2 Syntax, union ordinary, and union splittable types. 63
4.3 Disjointness based on splittable types for λ_u. 64
4.4 Disjointness **without** union ordinary and union splittable types for λ_u. 66
4.5 Typing and operational semantics for λ_u. 72
4.6 Syntax, union ordinary, and union splittable types for polymorphic λ_u. 73
4.7 Disjointness with union splittable types for polymorphic λ_u. 75
4.8 Subtyping for λ_u. 77
4.9 Operational Semantics for λ_u. 78
4.10 Operational Semantics for λ_u. 79

5.1 Syntax and subtyping for λ_{um}. 85
5.2 Typing for λ_{um}. 86

5.3 Type casting for λ_{um}. 88
5.4 Operational semantics for λ_{um}. 90
5.5 Dynamic dispatch, dynamic type and input type relation for λ_{um}. 91
5.6 Source syntax and source typing of Dunfield's calculus. 95

6.1 Subtyping for union and intersection types. 104
6.2 The Duotyping relation for a calculus with union and intersection types. . . 106
6.3 Haskell code for implementing an algorithmic formulation of Duotyping rules. 108
6.4 The Duotyping relation for simply typed lambda calculus calculus. 112
6.5 Syntax and Duotyping relation for union and intersection types. 114
6.6 Typing and reduction for $\lambda_{\Diamond}^{\wedge\vee}$. 116
6.7 Syntax and additional rules for Duotyping in $F_{k\Diamond}^{\wedge\vee}$. 119
6.8 Typing and reduction of the duotyped kernel $F_{<:}$. 121

A.1 Syntax, subtyping, ordinary, union ordinary and union splittable types for λ_u with intersection types. 166
A.2 Common ordinary subtypes based on union splittable types for λ_u. 167
A.3 Typing for λ_u. 169

List of Tables

6.1 Description of all systems. 124
6.2 Declarative Duotyping rules of all systems. 126
6.3 Comparing the features and number of rules with subtyping and Duotyping. 127
6.4 SLOC of traditional subtyping and Duotyping systems. 127
6.5 SLOC for transitivity proofs. 128

Part I

Preface

1 Introduction

Programming languages are the first mode of communication between humans and computers. As the name indicates, programming languages are primarily used to program computers and machines in general. The robustness of a program largely depends upon the programming language used to build that program. Programming languages have evolved over the decades. Researches are working in various areas in the programming languages including security, parallel computing, domain specific languages, compilers and quantum computing among others.

Type systems [Cardelli 1996; Pierce 2002b] is one such area in the study of programming languages. Type systems received significant attention in programming languages since they were first introduced. Programming languages can roughly be categorized into statically typed and dynamically typed languages considering the type system. C++ [Stroustrup 1986], Java [Gosling et al. 2000] and Scala [Odersky et al. 2004] are some of the examples of statically typed programming languages. Whereas, JavaScript [Crockford 2008], Python [Van Rossum et al. 1995] and most scripting languages are classified as dynamically typed programming languages. Our focus will mainly revolve around statically typed programming languages in this thesis. Statically typed languages perform type-checking at compile time. Whereas, dynamically typed languages may perform type-checking at runtime.

Nevertheless, *types* play an essential role both in statically and dynamically typed languages. Types provide a robust mechanism to introduce and eliminate program expressions by classifying the expressions in various categories or sets. A sound type system ensures that a well-typed expression is used in the program where expected. In statically typed languages, the type system raises an error at compile time if the expressions are ill-typed. The following program illustrates the use of types in a program:

```
Int add (x : Int, y : Int) {
  return x + y
}

result = add (1, 2)   //accepted
```

The above program will run fine because `add` is expecting two integers. The arguments 1 and 2 being passed to the `add` function are integers. Whereas, the following call to `add` will be rejected at compile time:

```
add (1, "Hello")    //rejected
```

The problem here is that we cannot and should not be adding an integer and a string together. In this case, type system statically detects that `"Hello"` is not an integer and restrains from the compilation. If not detected before compilation, such errors may cause the program to terminate abnormally during execution. On the other hand, in an untyped programming language, following error may go undetected during compilation:

```
add (x, y) {
  return x + y
}

add (1, "Hello")   //undetected error
```

Notice that no types are involved in this program. There is no static guarantee that either we are adding two integers, strings or booleans. Such a program may crash at runtime.

Type systems further categorize programs into well-typed and ill-typed programs. Ill-typed programs may contain a runtime error thus rejected at compile time. Whereas, well-typed programs do not go wrong because of typing errors. Thus *type system guarantees the absence of certain runtime behaviors where program may go wrong*. Types also provide a form of program documentation. For example, by looking at the return type of a function one can learn what should be expected from the function.

While type systems come with benefits, the study of type systems is hard. Adding a new feature with sound theoretical standings is non-trivial. Normally, the theoretical complexity of the type system increases with the number of features and how various features interact. Interacting features are hard to deal with compared to the orthogonal features. The type system becomes sophisticated with more advance features resulting in significant increase of complexity in the metatheory. Intersection and union types [Barbanera et al. 1995; Dunfield 2014; Muehlboeck and Tate 2018; Pierce 2002a] are such sophisticated advance features which can significantly increase the theoretical complexity of a sound type system.

Intersection types allow to construct a term of multiple (maybe) non-overlapping types. Union types allow a term to have alternative types. Many modern programming languages support intersection and union types, such as Scala [Odersky 2021] and Ceylon [King 2013]. This illustrates the significance of these features in today's programming universe. The intersection and union types are useful to encode various programming constructs such as function overloading [Cardelli and Wegner 1985; Castagna et al. 1995; Wadler and Blott

1989] and multiple inheritance [Compagnoni and Pierce 1996]. The following Scala code demonstrates the use of intersection and union types in practice:

```
trait Person () {}
trait Robot  () {}

//illustrating intersection types
class Hybrid () extends Person with Robot {}

//illustrating union types
def speak(x: Person | Robot): String = {
  x match
    case p: Person => "I am person"
    case r: Robot  => "I am robot"
}
```

The first two lines of the code declare two traits i.e. `Person` and `Robot`. We then create a class named `Hybrid`. Note that the class `Hybrid` extends two traits, the `Person` and `Robot` using the keyword `with` . In Scala, the keyword `with` uses intersection types for multiple interface inheritance. Next, we define a function `speak`. It takes an input of type `Person` or `Robot` (`Person | Robot`). The function `speak` then returns different value depending upon the specific type of parameter `x`. Such a type-based case analysis on an expression of union of types is analogous to function overloading [Cardelli and Wegner 1985; Castagna et al. 1995; Wadler and Blott 1989]. Similar to function overloading, the type-based case analysis provides the flexibility of executing disparate code depending on the dynamic (specific) type of the expression. Another Scala example[1] illustrating the usability of intersection and union types is:

```
trait Fish {
  def swim(): Unit = {}
}

trait Bird {
  def fly(): Unit = {}
}

//illustrating intersection types
class FlyingFish() extends Fish with Bird {
  override def swim(): Unit = {}
  override def fly(): Unit = {}
}

var flyingFish: Fish & Bird = FlyingFish()
```

[1] https://medium.com/@Methrat0n/intersection-type-in-scala-5320dedf5cf.

```
flyingFish.fly()  //safe
flyingFish.swim() //safe

//illustrating union types
def move(FishOrBird: Fish | Bird): Unit =
  FishOrBird match
    case fish: Fish ⇒ fish.swim()
    case bird: Bird ⇒ bird.fly()
```

Although, intersection and union types can encode such powerful and advance programming features. The interaction of intersection and union types is known to be non-trivial in theory. For example, the challenges involved in the subtyping distributivity of intersection and union types [Barendregt et al. 1983]. Huang and Oliveira [2021] elegantly explain such challenges. This thesis examines the interaction of intersection and union types in various theoretical and practical settings. It consists of two main technical parts:

1. In part 1 (Chapters 3 to 5), we discuss the constructs to introduce and eliminate intersection types and union types respectively in a type-safe manner. Specifically, we start with a type-based switch expression for union elimination based upon the disjointness called λ_u calculus. λ_u is designed to keep the branches of a switch expression disjoint. Disjointness prevents a value to fall in multiple branches and ensures the determinism of the calculus. This work is dual to disjoint intersection types [Oliveira et al. 2016] in its simplified form. We then extend λ_u with advanced features including intersection types, nominal types, subtyping distributivity and disjoint polymorphism. Moreover, we study λ_u with the merge operator which results in loss of determinism still retaining type soundness.

2. Part 2 (Chapter 6) consists of a study of subtyping relation which exploits the duality of intersection and union types. We show that the theoretical complexity of subtyping decreases by exploiting the duality of intersection types and union types. We also explore extra features that come for free with the duality.

1.1 Union types and disjoint switch expressions

Most programming languages support some mechanism to express terms with alternative types. Algol 68 [Van Wijngaarden et al. 1969; van Wijngaarden et al. 2012] included a form of *tagged* unions for this purpose. With tagged unions an explicit tag distinguishes between different cases in the union type. Such an approach has been adopted by functional languages, like Haskell, ML, or Ocaml, which allow tagged unions (or sum types [Pierce 2002b]),

typically via either *algebraic datatypes* [Burstall et al. 1981] or *variant types* [Garrigue 1998]. Languages like C or C++ support *untagged* union types where values of the alternative types are simply stored at the same memory location. However, there is no checking of types when accessing values of such untagged types. It is up to the programmer to ensure that the proper values are accessed correctly in different contexts; otherwise the program may produce errors by accessing the value at the incorrect type.

Modern OOP languages, such as Scala 3 [Odersky 2021], Flow [Chaudhuri 2015], TypeScript [Bierman et al. 2014], and Ceylon [King 2013], support a form of untagged union types. In such languages a union type $A \vee B$ denotes expressions which can have type A or type B. Union types have grown to be quite popular in some of these languages. A simple Google search on questions regarding union types on StackOverflow returns around 6620 results (at the time of writing), many of which arising from TypeScript programmers. Union types can be useful in many situations. For instance, union types provide an alternative to some forms of overloading and they enable an approach to *nullable types* (or explicit nulls) [Gunnerson 2012; Nieto et al. 2020].

Elimination construct for union types. To safely access values with union types, some form of *elimination construct* is needed. Many programming languages often employ a language construct that checks the types of the values at runtime for this purpose. Several elimination constructs for (untagged) union types have also been studied in the research literature [Benzaken et al. 2003; Castagna et al. 2014a; Dunfield 2014]. Typically, such constructs take the form of a type-based case analysis expression. An example of the elimination construct for union types in Scala is shown next:

```
var str: String | Null = null

str match
  case s: String ⇒ s.length
  case n: Null   ⇒ -1
```

The code starts by declaring a variable `str` of type `String | Null`. It then does a type-based case analysis on `str` to safely access the `length` function. Without such a type-based case analysis, a call to length function (`s.length`) may result in a runtime error because the length function is not defined on `null` values. This code uses a recently proposed explicit nulls [Nieto et al. 2020] feature of the Scala programming language. The core of this feature lies in revising the subtyping hierarchy of Scala. Interested reader may refer to the original publication for details. When enabled, explicit nulls allow to explicitly mark the nullable values. For example, `str` in this example stores a `null` value, therefore, it must be having the

type `Null` explicitly. In other words, the following program will be rejected which stores a `null` value but does not explicitly marks it nullable with the static type of `str`.

```
var str: String = null //rejected
```

Complication with subtyping. A complication is that the presence of subtyping introduces the possibility of *overlapping types*. For instance, we may have a `Student` and a `Person`, where every student is a person (but not vice-versa). If we try to eliminate a union using such types we can run into situations where the type in one branch can cover a type in a different branch (for instance `Person` can cover `Student`). More generally, types can *partially overlap* and for some values two branches with such types can apply, whereas for some other values only one branch applies. Therefore, the design of such elimination constructs has to consider what to do in situations where overlapping types arise. A first possibility is to have a *non-deterministic semantics*, where any of the branches that matches can be taken. However, in practice determinism is a desirable property, so this option is not practical. A second possibility, which is commonly used for overloading, is to employ a *best-match semantics*, where we attempt to find the case with the type that best matches the value. Yet another option is to use a *first-match semantics*, which employs the order of the branches in the case. Various existing elimination constructs for unions [Benzaken et al. 2003; Castagna et al. 2014a] employ a first-match approach. All of these three options have been explored and studied in the literature.

They Ceylon programming language. The Ceylon language [King 2013] is a JVM-based language that aims to provide an alternative to Java. The type system is interesting in that it departs from existing language designs, in particular with respect to union types and method overloading. The Ceylon designers had a few different reasons for this. They wanted to have a fairly rich type system supporting, among others: *subtyping*; *generics with bounded quantification*; *union and intersection types*; and *type-inference*. The aim was to support most features available in Java, as well as a few new ones. However the Ceylon designers wanted to do this in a principled way, where all the features interacted nicely. A stumbling block towards this goal was Java-style method overloading [Tate 2011]. The interaction of overloading with other features was found to be challenging. Additionally, overloaded methods with overlapping types make reasoning about the code hard for both tools and humans. Algorithms for finding the best match for an overloaded method in the presence of rich type system features (such as those in Ceylon) are challenging, and not necessarily well-studied in the existing literature. Moreover allowing overlapping methods can make the code harder to reason for humans: without a clear knowledge of how overloading resolution works, pro-

grammers may incorrectly assume that a different overloaded method is invoked. Or worse, overloading can happen silently, by simply reusing the same name for a new method. These problems can lead to subtle bugs. For these reasons, the Ceylon designers decided not to support Java-style method overloading.

To counter the absence of overloading, the Ceylon designers turned to union types instead, but in a way that differs from existing approaches. Ceylon includes a type-based *switch construct* where all the cases must be *disjoint*. If two types are found to be overlapping, then the program is statically rejected. For example, the following program will statically be rejected in Ceylon (where Student is a subtype of Person):

```
// Student <: Person, PG <: Student
  Student | Person human = PG();

  switch (human)
      case (is Person)  {}
      case (is Student) {}
```

Whereas, the following program is accepted:

```
  Person | Robot hybrid = Robot();

  switch (hybrid)
      case (is Person)  {}
      case (is Robot)   {}
```

Many common cases of method overloading, which are clearly not ambiguous, can be modelled using union types and disjoint switches. By using an approach based on disjointness, some use cases for overloading that involve Java-style overloading with overlapping types are forbidden. However, programmers can still resort to creating non-overloaded methods in such a case, which arguably results in code easier to reason about. Disjointness ensures that it is always clear which implementation is selected for an "overloaded" method, and only in such cases overloading is allowed[2]. In the switch construct, the order of the cases does not matter and reordering the cases has no impact on the semantics, which can also aid program understanding and refactoring. Finally, from the language design point of view, it would be strange to support two mechanisms (method overloading and union types), which greatly overlap in terms of functionality.

Theoretical study of disjoint switches. While implemented in the Ceylon language, disjoint switches have not been studied formally. To our knowledge, the work by Muehlboeck

[2] Ceylon does allow dynamic type tests, which in combination with switches can simulate some overlapping.

and Tate [2018] is the only work where Ceylon's switch construct and disjointness are mentioned. However, their focus is on algorithmic formulations of distributive subtyping with unions and intersection types. No semantics of the switch construct is given. Disjointness is informally defined in various sentences in the Ceylon documentation. It involves a set of 14 rules described in English [King 2016]. Some of the rules are relatively generic, while others are quite language specific. Interestingly, a notion of disjointness has already been studied in the literature for intersection types [Oliveira et al. 2016]. That line of work studies calculi with intersection types and a *merge operator* [Reynolds 1988]. Disjointness is used to prevent ambiguity in merges, which can create values with types such as $\mathsf{Int} \wedge \mathsf{Bool}$. Only values with disjoint types can be used in a merge.

1.2 Duality of subtyping

Subtyping is a concept frequently encountered in many programming languages and calculi. It is also a pervasive and fundamental feature in Object-Oriented Programming (OOP). Various forms of subtyping exist for different type system features, including *intersection types* [Barbanera et al. 1995], *union types* [Barbanera et al. 1995] or *bounded quantification* [Canning et al. 1989]. Modern OOP languages such as Scala [Odersky et al. 2004], Ceylon [King 2013], Flow [Chaudhuri 2015] or TypeScript [Bierman et al. 2014] all support the aforementioned type system features.

As programming languages evolve, new features are added. This requires that subtyping for these new features is developed and also integrated with existing features. However, the design and implementation of subtyping for new features is quite often non-trivial. There are several, well-documented issues in the literature. These include finding algorithmic forms for subtyping (for instance doing transitivity elimination) [Steffen and Pierce 1994] or proving metatheoretical properties such as transitivity or narrowing [Abel and Rodriguez 2008]. Such issues occur, for instance, in some of the latest developments for OOP languages, such as the DOT calculi (which model the essence of Scala) [Amin et al. 2012]. One possible way to reduce the non-trivial amount of work needed to develop new features, would be if two related features could be develop at once with a coherent design. This thesis explores a new methodology that enables such benefits.

Normally programming language features are designed independently of each other. However there are features that are closely related to each other, and can be viewed as *dual features*. Various programming language features are known to be dual in programming language theory. For instance sum and product types are well-known to be duals [Bird and de Moor 1996]. Similarly universal and existential quantification are dual concepts as well [Barwise

and Cooper 1981]. Moreover duality is a key concept in category theory [Lane 1998] and many abstractions widely used in functional programming (such as Monads and CoMonads [Uustalu and Vene 2008]) are also known to be duals.

In OOP type systems dual features are also common. For instance all OOP languages contain a *top type* (called `Object` in Java or `Any` in Scala), which is the supertype of all types. Many OOP languages also contain a *bottom type*, which is a subtype of all types. Top and bottom types can be viewed as dual features, mirroring the functionality of each other. *Intersection* and *union* types are another example of dual features. The intersection of two types `A` and `B` can be used to type a value that implements *both* `A` *and* `B`. The union of two types `A` and `B` can be used to type a value that implements *either* `A` *or* `B`.

Duality in OOP and subtyping is often only informally observed by humans. For instance, by simply understanding the behaviour of the features and observing their complementary roles, as we just did in the previous paragraph. At best duality is more precisely observed by looking at the rules for the language constructs and their duals and observing a certain symmetry between those rules. However existing formalisms and language designs for type systems and subtyping relations do not directly incorporate duality. Unfortunately this means that an opportunity to exploit obvious similarities between features is lost.

This thesis proposes a novel methodology for designing subtyping relations that exploits duality between features directly in the formalism. At the core of our methodology is a generalization of subtyping relations, which we call *Duotyping*. Duotyping is parameterized by the mode of the relation. One of these modes is the usual subtyping, while another mode is supertyping (the dual of subtyping). Using the mode it is possible to generalize the usual rules of subtyping to account not only for the intended behaviour of one particular language construct, but also of its dual. This means that the behaviour of the language construct and its dual is modelled by a single, common set of rules. In turn this ensures that the behaviour of the two features is modelled consistently. Moreover it also enables various theorems/properties of subtyping to be generalized to account for the dual features. Therefore, Duotyping offers similar benefits to the how duality is exploited in category theory. More concretely, Duotyping brings multiple benefits for the design of subtyping relations, which are discussed next.

Shorter specifications. When duality is exploited in specifications of subtyping it leads to shorter specifications because rules for dual features are shared. This also ensures a consistent design of the rules between the dual features directly in the formalism. Such consistency is not enforceable in traditional formulations of subtyping where the rules are designed separately, and thus their design is completely unconstrained with respect to the dual feature. A

concrete example that illustrates shorter specifications is a traditional subtyping relation with top, bottom, union and intersection types, which would normally have 8 subtyping rules for those constructs. In a design with Duotyping we only need 5 subtyping rules. Basically we need only *half of the rules* (4 in this case) to model the feature-specific rules, plus an additional *duality rule* which is generic (and plays a similar role to *reflexivity* and *transitivity*).

"Buy" one feature get one feature for free! Duality can lead to the discovery of new features. While top and bottom types, or intersection and union types are well-known in the literature (and understood to be duals), other features in languages with subtyping do not have a known dual feature in the literature. This is partly because, when a language designer employs traditional formulations of subtyping, he/she is often only interested in the design of a feature (but not necessarily of its dual). Even for the case of union and intersection types, intersection types were developed first and the development of union types occurred years later. Because the dual feature is often also useful, the traditional way to design subtyping rules represents a loss of opportunity to get another language feature essentially for free.

One well-known example of a language feature that has been widely exploited in the literature, but its dual feature has received much less attention is *bounded quantification* [Cardelli and Wegner 1985]. Bounded quantification allows type variables to be defined with *upper bounds*. However *lower bounds* are also useful. One can think of universal quantification with lower bounds as a dual to universal quantification with upper bounds. The essence of (upper) bounded quantification is captured by the well-known $F_{<:}$ calculus [Cardelli and Wegner 1985]. However, as far as we know, there is no design that extends $F_{<:}$ with lower bounded quantification in the literature. Applying a Duotyping design to $F_{<:}$ gives us, naturally, the two features at once (lower and upper bounded quantification), as illustrated in our Section 6.3. Such generalization of bounded quantification is related to the recent form of universal quantification with type bounds employed in Scala and the DOT family of calculi [Amin et al. 2012]. However, while Scala's type bounds are more expressive than what we propose, they are also much more complex and are in fact one of the key complications in the type systems of languages like Scala. Most DOT calculi require a built-in transitivity rule in subtyping because it is not known how to eliminate transitivity. In contrast, the generalization of $F_{<:}$ proposed by us has a formulation of subtyping where transitivity can be proved as a separate lemma.

New proof techniques. Designs of subtyping with duality also enable new proof techniques that exploit such duality. For instance there are various theorems that can be stated for both a feature and its dual, instead of having separate theorems for both. Some of the proper-

ties of union and intersection types are examples of this. Moreover, Duotyping also enables new proof techniques to prove traditionally hard theorems such as transitivity. Surprisingly to us, for the vast majority of the calculi that we have applied Duotyping to, transitivity proofs have been considerably simpler than their corresponding traditional formulations due to the use of Duotyping!

Shorter implementations. Finally Duotyping also enables for shorter implementations. The benefits of shorter implementations are similar and follow from the benefits of shorter specifications. However there is a complicating factor when moving from a *relational specification* into an implementation: the *duality rule* is non-algorithmic. This is akin to what happens with transitivity, which is often also used in declarative formulations of subtyping. Eliminating transitivity to obtain an algorithmic system can often be a non-trivial challenge (as illustrated, for instance, by the DOT family of calculi [Amin et al. 2012]). However, we show that there is a simple and generally applicable technique that can be used to move from a declarative formulation of Duotyping into an algorithmic version. This contrasts with transitivity, for which there is not a generally applicable transitivity elimination technique.

1.3 Outline and contributions

In this thesis we discuss the introduction and elimination constructs for intersection and union types respectively. We also discuss the challenges involved in the integration of such constructs in a calculus. Moreover, we study the duality of subtyping with intersection and union types. We show that the duality of subtyping decreases the theoretical complexity of a calculus with dual features. All of the metatheory of this thesis has been formalized in Coq[3] theorem prover and is available at:

- **Chapter 3:** https://github.com/baberrehman/disjoint-switches
- **Chapter 4:** https://github.com/baberrehman/phd-thesis-artifact/tree/main/artifact/chap4
- **Chapter 5:** https://github.com/baberrehman/phd-thesis-artifact/tree/main/artifact/chap5
- **Chapter 6:** https://github.com/baberrehman/coq-duotyping
- **Appendix A:** https://github.com/baberrehman/phd-thesis-artifact/tree/main/artifact/appendixA

[3] Coq is an interactive theorem prover where the proofs are written by humans and are machine-checked.

Contributions towards disjoint switches. We study union types with disjoint switches formally and in a language independent way. We present the *union calculus* (λ_u), which includes disjoint switches and union types. The notion of disjointness in λ_u is interesting in the sense that it is the dual notion of disjointness for intersection types. We prove several results, including *type soundness*, *determinism* and the *soundness* and *completeness* of algorithmic formulations of disjointness. We also study several extensions of λ_u. In particular, the first extension (discussed in Section 3.3) adds intersection types, nominal types and distributive subtyping to λ_u. It turns out such extension is non-trivial, as it reveals a challenge that arises for disjointness when combining union and intersection types: the dual notion of disjointness borrowed from disjoint intersection types no longer works, and we must employ a novel, more general, notion instead. Such change also has an impact on the algorithmic formulation of disjointness, which must change as well. We also study two other extensions for parametric polymorphism and a subtyping rule for a class of empty types in this thesis. We prove that all the extensions retain the original properties of λ_u. Furthermore, for our subtyping relation in Section 3.3 we give a *sound*, *complete* and *decidable* algorithmic formulation by extending the algorithmic formulation employing *splittable types* by Huang and Oliveira [2021].

To illustrate the applications of disjoint switches, we show that they provide an alternative to certain forms of *overloading*, and they enable a simple approach to *nullable (or optional) types*. We also study λ_u with the merge operator [Dunfield 2014; Oliveira et al. 2016; Reynolds 1988] and the resulting calculus is called λ_{um}. The merge operator is an introduction form for intersection types. This addition results in another category of challenges for determinism. Therefore, λ_{um} studied in this thesis is non-deterministic but type-safe. All the results about λ_u, its extensions, and λ_{um} have been formalized in the Coq theorem prover.

Contributions towards duality. To evaluate a design based on Duotyping against traditional designs of subtyping, we formalized various calculi with common OOP features (including union types, intersection types and bounded quantification) in Coq in both styles. Our results show that the metatheory when using Duotyping has similar complexity and size compared to traditional designs. However, the Duotyping formalizations come with more features (for instance lower-bounded quantification) that dualize other well-known features (upper-bounded quantification). Finally, we also show that Duotyping can significantly simplify transitivity proofs for many of the calculi studied by us.

Outline. The outline of the thesis is:

- Chapter 2 provides related background. Specifically, this chapter discusses the background of intersection types, merge operator, union types and elimination constructs for union types. Finally, this chapter discusses the concept of duality in logic and in programming languages.
- Chapter 3 studies a union calculus for deterministic elimination of union types. The calculus is called λ_u. Later parts of Chapter 3 enrich λ_u with advance features including intersection types, subtyping distributivity, nominal types and a restrictive form of disjoint polymorphism based on the ground types restriction.
- Chapter 4 further increases the expressiveness of disjoint polymorphism. In particular, this chapter removes ground type restriction from disjoint polymorphism and studies a novel disjointness algorithm based on splittable types [Huang and Oliveira 2021].
- Chapter 5 studies λ_u with the merge operator. The merge operator [Dunfield 2014; Oliveira et al. 2016; Reynolds 1988] is an introduction form for intersection types. The resulting calculus is called λ_{um}. The integration of the merge operator and disjoint switches result in novel challenges for determinism. Therefore, λ_{um} retains type-safety but lacks determinism. This chapter also studies the completeness of λ_{um} with respect to Dunfield [2014].
- Chapter 6 Studies the duality of subtyping formally with intersection and union types. The duality of subtyping results in various benefits such as reduced complexity in the metatheory and extra features of lower bounds and lower bounded quantification.
- Chapters 7 and 8 discuss the related work and the future work respectively.
- Appendix A studies another variant of algorithmic disjointness based on the so called Common Ordinary Subtypes (COST).

A part of this thesis is based on the following publications:

- **Chapter 3:** Baber Rehman, Xuejing Huang, Ningning Xie and Bruno C. d. S. Oliveira. 2022. "Union Types with Disjoint Switches". In *European Conference on Object-Oriented Programming (ECOOP)*.
- **Chapter 5:** Baber Rehman and Bruno C. d. S. Oliveira. 2023. "Type Soundness with Unrestricted Merges". (In Submission). *Journal of Functional Programming (JFP)*.
- **Chapter 6:** Bruno C. d. S. Oliveira, Cui Shaobo and Baber Rehman. 2020. "The Duality of Subtyping". In *European Conference on Object-Oriented Programming (ECOOP)*.

PREREQUISITES. A background in type systems is assumed to understand the technical contents of this thesis. Interested readers may go over *Types and Programming Languages* [Pierce 2002b] or volume 1 and volume 2 of *Software Foundations* [Pierce et al. 2010] to acquire the relevant technical background.

2 Background

This chapter provides background on intersection types, union types, and common approaches to introduce and eliminate intersection and union types respectively. It then explains the Ceylon approach to union types, and discusses a few applications of union types. Finally, it illustrates the concept of duality in logic and in the subtyping of intersection and union types.

2.1 Intersection Types

Intersection types have a sound history in the study of programming languages and logic in general. They have been studied by multiple researchers [Barbanera et al. 1995; Dunfield 2014; Muehlboeck and Tate 2018; Pierce 2002a] in various settings. Intersection types correspond to product types and conjunction in category theory and logic respectively. In classical logic, types are interpreted as sets and the intersection of types correspond to the intersection of sets. In particular, the intersection of types is viewed as an intersection of values inhabited by the types. The intersection of Int and Bool does not inhabit any values in this view because set-intersection of the values of Int and Bool is an empty set. Therefore, the type $\mathsf{Int} \wedge \mathsf{Bool}$ is an empty type in classical logic. But as per the Curry–Howard correspondence [Curry et al. 1958; Howard 1980], intersection types correspond to conjunctions and conjunctions correspond to product types. The product of type Int and Bool is inhabited by a pair such as $(1, \mathsf{true})$. We follow the later interpretation of the intersection types where a value of the type $\mathsf{Int} \wedge \mathsf{Bool}$ is inhabited.

In programming languages an expression e has an intersection type $A \wedge B$ when e is both of type A and B simultaneously. Therefore, an expression of the type $A \wedge B$ can safely be treated as an expression of the type A or B. In other words, we can extract the value of any of the given types from an expression of intersection of those types. For example:

```
Int&Bool temp (x : Int) {}
```

Function `temp` takes an integer value as an input and returns both an integer and a boolean value. In a calculus with subtyping `temp` can safely be used in the context where an integer or a boolean is expected. This is because the return type of `temp` says that it returns both

an integer and a boolean at the same time. Therefore, it is safe to extract an integer as well as a boolean value from the return value of `temp`. For example, the following `add` function consumes the return value `of temp` function, filters the integer part and performs addition operation only on the integer part:

```
Int add (x : Int, y : Int) {
  return temp(x) + y
}
```

This is because of the inversion of following subtyping rules for intersection types:

$$\frac{A <: C}{A \wedge B <: C}\ \text{S-ANDB} \qquad \frac{B <: C}{A \wedge B <: C}\ \text{S-ANDC}$$

History. Coppo et al. [1981] and Pottinger [1980] initially studied intersection types in programming languages to assign meaningful types to terms. The intersection types for the multiple inheritance are studied by Compagnoni and Pierce [1996]. Extending a class by an intersection of multiple types naturally results in multiple inheritance. Pierce [1991] studied a calculus with intersection types, union types and polymorphism. Pierce has also shown the practicality of intersection types using diverse programming examples. Dunfield [2014] studied a calculus with intersection and union types using an elaboration semantics. Dunfield elaborated intersections to product types and unions to sum types. The intersection types have also been studied in the context of refinement types [Freeman and Pfenning 1991]. Refinement types increase the expressiveness of types. They pose a restriction on the formation of intersections and does not allow intersections of certain types. Specifically, in refinement types, only those intersections are allowed which can be refined. For example:

```
Int temp2 (x : Int&Real) {}
```

is allowed in such calculi because `Int` is a refinement of `Real`. But an intersection of $\mathsf{Int} \wedge \mathsf{Bool}$ is not allowed because none of the types is a refinement of each other.

Introduction and elimination. The traditional typing rule to introduce an expression of intersection types is:

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash e : B}{\Gamma \vdash e : A \wedge B}\ \text{TYP-AND}$$

This rule says that if an expression is of type A and type B, then that expression can safely be treated as of type $A \wedge B$. It is trivial to say by inversion that if an expression is of type $A \wedge B$, then that expression can safely be cast to A and B separately, as illustrated by:

$$\frac{\Gamma \vdash e : A \wedge B}{\Gamma \vdash e : A}\ \text{TYP-ANDL} \qquad \frac{\Gamma \vdash e : A \wedge B}{\Gamma \vdash e : B}\ \text{TYP-ANDR}$$

It is important to notice that the rules TYP-ANDL and TYP-ANDR are subsumed by the subsumption rule in a system with subtyping. A natural question arises at this point is what if we want to construct an expression of two non-overlapping types? Such as Int and Bool. For example, considering the `temp` function:

```
Int&Bool temp (x : Int) {}
```

How would we construct an expression of type $\mathsf{Int} \wedge \mathsf{Bool}$ as the return value of this function? The traditional introduction rule for intersection types does not have enough expressive power to construct such an expression. The so called merge operator [Dunfield 2014; Huang and Oliveira 2020; Oliveira et al. 2016; Reynolds 1988] was introduced for this purpose and is discussed next.

2.2 Merge Operator

The intersection types roughly correspond to the product types in category theory. But classical intersections cannot model all of the expressions of category theory. For example, the product types allow to construct a pair with the following terms:

```
(1, true) : (Int, Bool)
```

But the traditional introduction rule for the intersection types (rule TYP-AND) cannot construct a term having both integer and boolean in it. It is because of the fact that there is no corresponding term level construct which can encode an expression of multiple non-overlapping types. The merge operator [Dunfield 2014; Huang and Oliveira 2020; Oliveira et al. 2016; Reynolds 1988] is studied to overcome this shortcoming. The merge operator significantly increases the term level expressiveness of a calculus. The introduction rule for the intersection types with the merge operator is:

$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash e_1 ,, e_2 : A \wedge B}\ \text{TYP-MERGA} \qquad \text{[Oliveira et al. 2016]}$$

Notice that e_1 and e_2 may not necessarily be the same expressions in rule TYP-MERGA. By exploiting the expressiveness of the merge operator, one can write the following program:

```
x : Int&Bool = 1,,true
```

```
Int&Bool extend(x : Int, y : Bool) {
    return x,,y
  }
```

The merge operator was first introduced in Forsythe programming language by Reynolds [1997]. Dunfield [2014] further studied the merge operator with union types. Dunfield followed an elaboration semantics where intersection types elaborate to product types and merges to pairs. However, the calculus proposed by Dunfield is non-deterministic. For example, an expression `1,,2` may reduce to either 1 or 2. Oliveira et al. [2016] studied disjoint intersection types to overcome the non-deterministic behaviour of the merge operator. Oliveira et al. [2016] proposed a disjointness constraint for the merge operator. The typing rule for the merge operator in their calculus is:

$$\textsc{typ-mergb}\quad \frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B \qquad A * B}{\Gamma \vdash e_1,,e_2 : A \wedge B}$$

The last premise of the rule TYP-MERGB imposes a so called disjointness constraint. In summary, non-overlapping types are disjoint types. For example Int is disjoint to Bool but Int is not disjoint to Int or $\top$. Therefore a merge of `1,,true` is allowed but a merge of `1,,2` is not allowed with such a disjointness constraint.

The calculi studied by Dunfield [2014] and Oliveira et al. [2016] follow an elaboration semantics. Recently, Huang and Oliveira [2020] proposed a direct operational semantics for the merge operator. Alpuim et al. [2017] further studied disjoint intersection types and the merge operator with disjoint polymorphism. It is a variant of the parametric polymorphism [Canning et al. 1989; Cardelli and Wegner 1985] and allows the following program:

```
X&Y extend[X * Y] (x : X, y : Y) {
  return x,,y
}
```

`X` and `Y` are the type variables. The type variable `Y` can be instantiated with any type, whereas `X` can only be instantiated with types that are disjoint with `Y`.

The merge operator is useful to encode multi-field records from single-field records [Reynolds 1988]. A core language only needs to support single-field records and single-field record types. In the presence of the merge operator, multi-field records are simply the merges of single-field records and multi-field records types are the intersections of single-field record types:

```
type Student = {name : String} & {id : Int}    //multi-field record type
s : Student = {name = "John"} ,, {id = 123}    //multi-field record
```

Bi et al. [2018a] further studied merge operator with nested composition and family polymorphism. Bi and Oliveira [2018] studied typed first class traits and dynamic inheritance with static guarantee by exploiting the disjoint intersection types. The first class traits treat classes or traits as first class citizens where they may be passed to or return from a function, thus enabling dynamic inheritance. This enables type-checking highly dynamic object oriented features which are non-trivial to deal with in traditional OOP languages. Xie et al. [2020] show that row and bounded polymorphism can be encoded using disjoint polymorphism. The merge operator can also encode function overloading [Castagna et al. 1995; Reynolds 1988]. This is illustrated by the following example:

```
(succ,,not) 1
(succ,,not) true
```

The `succ` function has type $\mathsf{Int} \to \mathsf{Int}$ and `not` function has type $\mathsf{Bool} \to \mathsf{Bool}$. The merge of these two functions has the type $\mathsf{Int} \to \mathsf{Int} \wedge \mathsf{Bool} \to \mathsf{Bool}$.

```
(succ,,not) : (Int → Int)&(Bool → Bool)
```

The application of `(succ,,not)` to `1` yields `2`, whereas an application on `true` yields `false`. The semantics chooses the right function for application depending upon the dynamic type of the argument. This is further discussed in detail in Chapter 5.

2.3 Tagged Union Types

Many systems model *tagged union types* (also called *sum types* or *variants types*), where explicit *tags* are used to construct terms with union types, as in languages with algebraic datatypes [Burstall et al. 1981] or (polymorphic) variants [Garrigue 1998]. In their basic form, there are two introduction forms: $\mathsf{inj}_1 : A \to A \vee B$ turns the type of an expression from A into $A \vee B$; and $\mathsf{inj}_2 : B \to A \vee B$ turns the type of an expressions from B into $A \vee B$.

For example, we can have:

```
inj1 "foo": String | Int
inj2   1  : String | Int
```

Note that in the code above we write union types as `String | Int` (instead of $\mathsf{String} \vee \mathsf{Int}$), since this is a common notation in many programming languages, including Ceylon.

Using tagged union types, we can implement a safe integer division function, as[1]:

```
String | Int safediv (x : Int) (y : Int) =
  if (y == 0) then inj1 "Divided by zero" else inj2 (x / y)  // uses tags
```

[1] Throughout this thesis, we write union types as `A | B` in code, since this is widely adopted in programming languages (e.g., Ceylon, Scala, and TypeScript), and as $A \vee B$ in the formal calculi, which is more frequently used in the literature.

Here the intention is to have a safe (integer) division operation that detects division by zero errors, and requires clients of this function to handle such errors. The return type `String | Int` denotes that the function can either return an error message (a string), or an integer, when division is performed without errors.

Elimination form for tagged union types. Tagged union types are eliminated by some form of case analysis. For consistency with the rest of the thesis, we use a syntactic form with `switch` expressions for such case analysis. For example, the following program `tostring` has different behaviors depending on the tag of `x`, where `show` takes an `Int` and returns back its string representation.

```
String tostring (x: String | Int) = switch (x)
                                      inj1 str → str
                                      inj2 num → show num
```

Combining union type construction in `safediv` and its elimination in `tostring`, we can easily implement an interface which returns the result of safe division as one `String`.

```
> tostring (safediv 42 2)
"21"
> tostring (safe 42 0)
"Divided by zero"
```

2.4 Un-tagged Union Types

Union types are commonly used to deal with heterogeneous data where we do not know the exact type of data at compile time. However, we know that data can only be among the certain types. For example, we know statically that argument to a function may either be Int or Bool, but we do not know if it's Int or Bool.

As the name suggests, untagged union types do not carry a tag to distinguish between right and left part of the union. Thus with untagged union types, above example will be written as:

```
"foo": String | Int
1    : String | Int
```

Notice that we do not carry explicit tags. Specifically, `inj1` and `inj2` in the example above have been eliminated.

2.5 Type-directed Elimination forms for Union Types

While tags are useful to make it explicit which type a value belongs to, they also add clutter in the programs. On the other hand, in systems with subtyping for union types [Dunfield 2014; Muehlboeck and Tate 2018; Pierce 1991], explicit tags are replaced by implicit coercions represented by the two subtyping rules $A <: A \vee B$ and $B <: A \vee B$. In this thesis we refer to union types where the explicit tags are replaced by implicit coercions as *untagged union types*, or simply *union types*. In those systems, a term of type A or B can be directly used as if it had type $A \vee B$, and thus we can write safe division as:

```
String | Int safediv2 (x : Int) (y : Int) =
  if (y == 0) then "Divided by zero" else (x / y)       // no tags!
```

However, now the elimination form of union types cannot rely on explicit tags anymore, and different systems implement elimination forms differently. The most common alternative is to employ types in the elimination form. We review type-directed union elimination next. Please note that *type-directed elimination* corresponds to *type-directed elimination form for union types*. Generally, when we write tag or type-directed union elimination, we refer to elimination form for tag or type-directed union types.

Type-directed elimination. Some systems [Castagna et al. 2014a] support *type-directed* elimination of union types. For instance, `tostring2` has different behaviors depending on the *type* of `x`.

```
String tostring2 (x: String | Int) = switch (x)
                                       (y : String) → y
                                       (y : Int)    → show y
```

Note that here `show` does not need to be overloaded, as the type-directed elimination *turns* the variable `x` of type `String | Int` into the variable `y` of type `Int`.

However, compared to tag-directed elimination, extra care must be taken with type-directed elimination. In particular, while we can easily distinguish tags, ambiguity may arise when types in a union type overlap for type-directed elimination. For example, consider the type `Person | Student`, where we assume `Student` is a subtype of `Person`. With type-directed elimination, we can write:

```
Bool isstudent (x: Person | Student) = switch (x)
                                         (y : Person)  → false
                                         (y : Student) → true
```

Now it is unclear what happens if we apply `isstudent` to a term of type `Student`, as its type matches both branches. In some calculi [Dunfield 2014], the choice is not determined in the

semantics, in the sense that either branch can be chosen. This leads to a non-deterministic semantics. In some other languages or calculi [Castagna et al. 2014a], branches are inspected from top to bottom, and the first one that matches the type gets chosen. However, in those systems, as `Person` is a supertype of `Student`, the first branch subsumes the second one and will always get chosen, and so the second branch will never get evaluated! This may be unintentional, and similar programs being accepted can lead to subtle bugs. Even if a warning is given to alert programmers that a case can never be executed, there are other situations where two cases overlap, but neither case subsumes the other. For instance we could have `Student` and `Worker` as subtypes of `Person`. For a person that is both a student and a worker, a switch statement that discriminates between workers and students could potentially choose either branch. However for persons that are only students or only workers, only one branch can be chosen.

Best-match and overloading. Some languages support an alternative to typed-based union elimination via method overloading. Such form is used in, for example, Java [Gosling et al. 2021] and Julia [Zappa Nardelli et al. 2018]. In Java, we can encode `isstudent2` as an overloaded method, which has different behaviors when the type of the argument differs.

```
boolean isstudent2 (Person x)  { return false; }
boolean isstudent2 (Student x) { return true; }
```

Java resolves overloading by finding and selecting, from all method implementations, the one with the *best* type signature that describes the argument. If we apply `isstudent2` to a term of type `Student`, the second implementation is chosen, as `Student` is the best type describing the argument. As we can see, such a best-match strategy eliminates the order-sensitive problem, as it always tries to find the best-match despite the order. That is, in Java the method order does not matter: in this case, we have the method for `Person` before the one for `Student`, but Java still finds the one for `Student`.

However, the best-match strategy can also be confusing, especially when the system features implicit upcasting (e.g., by subtyping). If programmers are not very familiar with how overloading resolution works, they may assume that the wrong implementation is called in their code. For instance, in Java we may write:

```
Person p = new Student();
isstudent2(p);
```

In this case Java will pick the `isstudent2` method with the argument `Person`, since Java overloading uses the *static type* (`p` has the static type `Person`) to resolve overloading. But some programmers may assume that the implementation of the method for `Student` would be cho-

sen instead, since the person is indeed a student in this case. This can be confusing and lead to subtle bugs.

Moreover, there are other tricky situations that arise when employing a best-match strategy. For example, suppose that the type `Pegasus` is a subtype of both type `Bird` and type `Horse`. If a method `isbird` is overloaded for `Bird` and `Horse`, then which method implementation should we choose when we apply `isbird` to a term of type `Pegasus`, the one for `Bird`, or the one for `Horse`? In such case, we have an ambiguity. Things get worse when the type system includes more advanced type system features, such as generics, intersections and union types, or type-inference.

2.6 Union Types and Disjoint Switches in Ceylon

The Ceylon language [King 2013] supports type-directed union elimination by a switch expression with branches. The following program is an example with union types using Ceylon's syntax:

```
void print(String|Integer|Float x) {
  switch (x)
    case (is String)        { print("String: ``x``"); }
    case (is Integer|Float) { print("Number: ``x``"); }
}
```

For the switch expression, Ceylon enforces static type checking with two guarantees: *exhaustiveness*, and *disjointness*. First, Ceylon ensures that all cases in a switch expression are *exhaustive*. In the above example, `x` can either be a string, an integer or a floating point number. The types used in the cases do not have to coincide with the types of `x`. Nevertheless, the combination of all cases must be able to handle all possibilities. If the last case only dealt with `Integer` (instead of `Integer|Float`), then the program would be statically rejected, since no case deals with `Float`.

Second, Ceylon enforces that all cases in a switch expression are *disjoint*. That is, unlike the approaches described in Section 2.5, in Ceylon, it is impossible to have two branches that match with the input at the same time. For instance, if the first case used the type `String | Float` instead of `String`, the program would be rejected statically with an error. Indeed, if the program were to be accepted, then the call `print(3.0)` would be ambiguous, since there are two branches that could deal with the floating point number. Note that, since the cases in a switch cannot overlap, their order is irrelevant to the program's behavior and its evaluation result. All of the overlapping examples from the previous section will statically be rejected in similar fashion.

Union types as an alternative to overloading. One motivation for such type-directed union elimination in Ceylon is to model a form of function overloading. The following example, which is adapted from TypeScript's documentation [Microsoft 2023], demonstrates how to define an "overloaded" function `padLeft`, which adds some padding to a string. The idea is that there can be two versions of `padLeft`: one where the second argument is a string; and the other where the second argument is an integer:

```
String space(Integer n){
  if (n==0) { return ""; }
  else      { return " "+space(n-1); }
}
String padLeft(String v, String|Integer x) {
  switch (x)
    case (is String)  { return x+v; }
    case (is Integer) { return space(x)+v; }
}
print(padLeft("?", 5)); // "     ?"
print(padLeft("World", "Hello ")); // "Hello World"
```

In `padLeft`, there are two cases of the switch construct depending on the type of `x`: the first one appends a string to the left of `v`, and the other calls function `space` to generate a string with `x` spaces, and then append that to `v`. Although statically `x` has type `String|Integer`, as a concrete value it can only be a string or an integer. As such, when values with such types are passed to the function, the corresponding branch is chosen and executed.

2.7 Nullable Types

Besides being used for overloading, union types can be used for other purposes too. *Null pointer exceptions (NPEs)* are a well-known and tricky problem in many languages. The problem arises when dereferencing a pointer with the `null` value. For instance, if we have a variable `str`, which is assigned to `null`, the the code `print(str.size)`, in a Java-like language, will raise a null pointer exception. This is because of so-called implicit nulls in Java and other popular languages. With implicit nulls, any variable of a reference type can be `null`.

An interesting application of union types in Ceylon is to encode *nullable types* (or *optional types*) [Gunnerson 2012] in a type-safe way. A similar approach to nullable types has also been recently proposed for Scala [Nieto et al. 2020]. In those languages, there is a special type `Null`, which is inhabited by the `null` value. Note that `Null` differs from `Nothing` (the bottom type in Ceylon), in the sense that `Null` is inhabited while `Nothing` is not. To illustrate the subtle difference, Figure 2.1 presents a part of the subtyping lattice in Ceylon. `Anything`, the

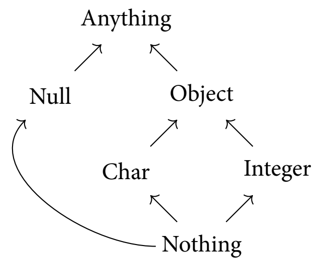


Figure 2.1: Ceylon's subtyping hierarchy. Note that `Null` only has `Nothing` as its subtype.

top type in Ceylon, is an enumerated class. `Anything` is also a supertype of `Object`, which is the root of primitive types, function types, all interfaces and any user-defined class. Notably, `Null` is disjoint to `Object`, and therefore, to all user-defined classes.

In Ceylon the following code:

```
String str = null;
```

is rejected with a type error, since `null` cannot have type `String`. Instead, a type that can have the null value must be defined explicitly in Ceylon using union types:

```
String | Null str = null;
```

Now we cannot call `str.size`, as `str` may be `null`, and `size` is not defined on `null`. To get the `size` of `str`, we must first check whether `str` is `null` or not using disjoint switches:

```
String | Null str = null;
switch (str)
  case (is String) { print(str.size); }
  case (is Null)   { print(); }
```

Other uses of Union Types. Union types are also useful in many other situations. In Section 2.5 we illustrated a `safediv` operation, which can be easily encoded in Ceylon as:

```
String | Integer safediv3 (Integer x, Integer y){
  if (y==0) { return "Divided by zero"; }
  else      { return (x/y); }
}
```

The return value can be a string or an integer, with no explicit tag needed, as union types are implicitly introduced. As long as the declared return type of the function is a supertype of all possible return values, it is valid in Ceylon.

2.8 Polymorphism

Polymorphism is an essential feature supported in almost all the modern programming languages. The word polymorphism can map to three important concepts in programming languages. This includes subtype polymorphism, ad-hoc polymorphism [Cardelli and Wegner 1985; Castagna et al. 1995; Wadler and Blott 1989] and parametric polymorphism [Canning et al. 1989; Cardelli and Wegner 1985]. In our current setting polymorphism refers to parametric polymorphism. It enables generic programming by abstracting over types not just terms. Parametric polymorphism lies at the core of most functional programming languages including Haskell and Ocaml among others. The following programming example illustrates parametric polymorphism:

```
Int length [T] (l Array[T]) {}
```

Length is a generic function which calculates the size of array of any type. It does not depend on the type of the elements in an array, it only cares about the number of elements. In a programming language where parametric polymorphism is not supported one may have to write multiple length functions dealing with one type each. Parametric polymorphism can further be refined using bounded quantification to restrict the instantiation of type variables to be subtype of certain types. For example:

```
Int length′ [T <: Number] (l Array[T]) {}
```

The function *length′* type-checks as long as the type variable T is a subtype of Number. We discuss a variant of parametric polymorphism called the disjoint polymorphism in Section 3.4 and Chapter 4. Disjoint polymorphism poses a disjointness restriction on type variables instead of subtyping restriction. This is illustrated by the following example:

```
Bool isstudent [T * Student] (x: T | Student) = switch (x)
                                                  (y : T)       → false
                                                  (y : Student) → true
```

The function `isstudent` is abstracted over types. Type variable `T` has a disjointness constraint. It can be instantiated by any type given that the type is disjoint to `Student`. Disjoint polymorphism is essential for polymorphic disjoint switches. This will further be discussed in Section 3.4 and Chapter 4.

2.9 Duality

Duality is a common concept in logic. Many features and their duals have been studied. Modal logic [Nanevski et al. 2008; Simpson 1994] studies duality between necessity and possibility. Necessity is dual to possibility in a sense that necessity must always hold, whereas

the possibility may not always hold. Similarly, conjunctions and disjunctions are known to be dual features in logic. Duality of conjunction and disjunction is shown by De Morgan's laws [Copi et al. 2018] as: (where A and B are propositions)

$$A \wedge B \equiv \neg (\neg A \vee \neg B)$$

and

$$A \vee B \equiv \neg (\neg A \wedge \neg B)$$

Similarly, universal and existential quantifiers are known to be dual in logic.

$$\forall \, \mathrm{x} \, . \, A \equiv \neg \exists \, \mathrm{x} \, . \, \neg A$$

Conversely, the other dual equality is written as:

$$\exists \, \mathrm{x} \, . \, A \equiv \neg \forall \, \mathrm{x} \, . \, \neg A$$

There is also a strong connection between conjunction and universal quantifier and disjunction and the existential quantifier. Universal quantifier can be thought as conjunction of infinite propositions. Whereas, existential quantifier can be though as disjunction of infinite propositions. Since propositions are types as per the Curry–Howard correspondence [Curry et al. 1958; Howard 1980]. This points to a study of the duality among types. Intersection types and union types are the dual features in the theory of programming languages. In particular, we study the duality in the subtyping of intersection types and union types in this thesis.

Part II

Deterministic Union Elimination (λ_u)

3 Union Types with Disjoint Switches

3.1 Overview

In this chapter we first introduce a simplified formulation of union calculus (λ_u), which formalizes the basic ideas of union types with disjoint switches similar to those in the Ceylon language. To our best knowledge, there is yet no formalism of disjoint switches, and λ_u studies those features formally and precisely. In particular, λ_u captures the key idea for type-directed elimination of union types in its switch construct in a language independent way, and formally defines disjointness, disjointness and subtyping algorithms, and the operational semantics. The simplified formulation of λ_u is useful to compare with existing calculi with union types in the literature [Barbanera et al. 1995; Castagna and Frisch 2005; Dunfield 2014; Dunfield and Pfenning 2003; MacQueen et al. 1984; Pierce 1991]. Moreover, we study a more fully featured formulation of λ_u that includes practical extensions, such as intersection types, distributive subtyping, nullable types and a simple form of nominal types. λ_u is proved to enjoy many desirable properties, such as type soundness, determinism and the soundness/completeness of disjointness and subtyping definitions. All the Ceylon examples in Sections 2.6 and 2.7 can be encoded in λ_u. Finally, we discuss λ_u with disjoint polymorphism and a more general subtyping rule for bottom type.

The typing rules guarantee that cases in the switch are disjoint and exhaustive. Reduction preserves types and produces deterministic results. We start with an overview of our design and discuss some challenges we met for the calculi designed.

Disjointness. A central concept in the formulation of disjoint switches is disjointness. Our first hurdle was to come up with a suitable formal definition of disjointness. Consider the simple λ_u switch expression:

```
switch x {
  (y : String | Int)  → 0
  (y : Int | Bool)    → 1
}
```

Here we wish to determine whether $\mathsf{String} \vee \mathsf{Int}$ and $\mathsf{Int} \vee \mathsf{Bool}$ are disjoint or not. In other words, we wish to determine whether, for any possible (dynamic) type that x can have, it is

unambiguous which branch to choose. In this case, it turns out that there is ambiguity. For instance, if x is an integer, then either branch can be chosen. Thus λ_u rejects this program with a disjointness error. In this example, the reason to reject the program is basically that $\mathsf{Int} <: \mathsf{String} \vee \mathsf{Int}$ and $\mathsf{Int} <: \mathsf{Int} \vee \mathsf{Bool}$. That is we can find a common subtype (Int) of the types in both branches. Moreover, that subtype can be inhabited by values (integer values in this case). If the only common subtypes of the types in the two branches would be types like $\bot$ (which has no inhabitants), then the switch should be safe because we would not be able to find a value for x that would trigger two branches. This observation leads to the notion of disjointness employed in the first variant of λ_u in Section 3.2. Formally, we have:

Definition 1 (⊥-Disjointness)**.** $A * B ::= \forall\, C$, if $C <: A$ and $C <: B$ then $\rfloor C \lfloor$

Here we use $\rfloor C \lfloor$ to denote that type C is equivalent to type $\bot$, or, bottom-like (i.e. $C <: \bot$). In either definition, Int serves as a counter-example for $\mathsf{String} \vee \mathsf{Int}$ and $\mathsf{Int} \vee \mathsf{Bool}$ to be disjoint. Thus λ_u rejects the program above with a disjointness error. It is worth noting that this first notion of disjointness is essentially dual to a definition of disjointness for intersection types in the literature in terms of top-like common supertypes [Oliveira et al. 2016].

Disjointness for Intersections in the Literature. This is very similar to the related idea of disjointness for intersection types [Oliveira et al. 2016]. In that setting two types are disjoint if the only common supertypes that two types can have are top-like types (i.e. types such as $\top$ or $\top \wedge \top$). While a disjoint switch provides deterministic behavior for downcasting, disjointness in intersection types prevents ambiguity in upcasting. In a type-safe setting, if two values v_1 and v_2 (of type A_1 and A_2) can both be upcasted to type B, then B must be a common supertype of A_1 and A_2. The disjointness restriction of A_1 and A_2 means they cannot have any meaningful common supertype, so when the two values together get upcast to a type like Int, only one of them can contribute to the result. Prior work on disjoint intersection types is also helpful to find an algorithmic formulation of disjointness: essentially we can find an algorithmic formulation that employs dual rules to those for disjoint intersection types. In essence, disjointness definition for disjoint intersection types [Oliveira et al. 2016] is dual to ours and is stated as:

Definition 2 (Disjointness in disjoint intersection types)**.** $A * B ::= \forall\, C$, if $A <: C$ and $B <: C$ then $\rceil C \lceil$

where $\rceil C \lceil$ represents the top-like types: types that are supertype and subtype of top.

Disjointness in the presence of union and intersection types. The variant of λ_u in Section 3.2 does not include intersection types. Unfortunately, the disjointness defini-

tion above does not work in the presence of intersection types. The reason is simple: with intersection types we can always find common subtypes, such as $\mathsf{Int} \land \mathsf{Bool}$, which are not bottom-like, and yet they have no inhabitants. That is, $\mathsf{Int} \land \mathsf{Bool}$ is not a subtype of $\bot$, but no value can have both type Int and type Bool. In other words, when intersection types are added, empty types and bottom-like types no longer coincide. We address this issue by reformulating disjointness in terms of *ordinary types* [Davies and Pfenning 2000], which are basically primitive types (such as integers or functions). Importantly, ordinary types are always inhabited. If we can find common *ordinary* subtypes between two types, we know that they are not disjoint. Thus the disjointness definition used for formulations of λ_u with intersection types is:

Definition 3 ($\land$-Disjointness)**.** $A * B ::= \nexists\, C^\circ, C^\circ <: A$ and $C^\circ <: B$.

Note that here C° is a metavariable denoting ordinary types. Under this definition we can check that Int and Bool are disjoint, since no ordinary type is a subtype of both of these two types. This definition avoids the issue with Definition 1, which would not consider these two types disjoint. Moreover, this definition is a generalization of the previous one, and in the variant with union types only the two definitions coincide.

This new definition requires a different approach to algorithmic disjointness. Our new approach is to use the notion of *lowest ordinary subtypes*: For any given type, we calculate a finite set to represent all the possible values that can match the type. Then we can easily determine whether two types are disjoint by ensuring that the intersection of their lowest ordinary subtypes is empty.

Typing and exhaustiveness In λ_u, a switch expression has two branches. For multiple cases, one can write nested switch expression. We assume the two branches expect A and B. To make sure they exhaust all possible types of the switched term e, there is a premise that e can be checked by the type $A \lor B$. In other words, the dynamic type of e should be a subtype of $A \lor B$, like Int to $\mathsf{Int} \lor \mathsf{Char}$.

TYP-SWITCH

$$\frac{\Gamma \vdash e : A \lor B \qquad \Gamma, x : A \vdash e_1 : C \qquad \Gamma, y : B \vdash e_2 : C \qquad A * B}{\Gamma \vdash \mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} : C}$$

Another premise requires the two cases to be disjoint. Besides, the two branches are typed under different assumption of the bound variable. Although the same type C is used for both of them in the rule, it does not prevent them to return different types. Assuming the type of e_1 is C_1 and the type of e_2 is C_2, we can make C to be $C_1 \lor C_2$.

Distributive Subtyping. In Section 3.3, we study λ_u with an enriched distributive subtyping relation inspired by Ceylon programming language. Distributive subtyping is more expressive than standard subtyping and adds significant complexity in the system, in particular for a formulation of algorithmic subtyping and the completeness proof of the disjointness algorithm. Nevertheless, distributive subtyping does not affect the disjointness definition and its algorithm remains the same with and without distributive subtyping. The following code snippet elaborates on the expressiveness of distributive subtyping:

```
void do (<Integer & String> | Boolean val) { /* do something */ }
```

The function do in above code snippet takes input value of type $(\mathsf{Int} \wedge \mathsf{String}) \vee \mathsf{Bool}$. However, we cannot pass a value of type $(\mathsf{Int} \vee \mathsf{Bool}) \wedge (\mathsf{String} \vee \mathsf{Bool})$ to the function do: we get a type error if we try to do that in a system with standard subtyping (without distributivity), as standard subtyping fails to identify that the value has a subtype of the expected argument type. Distributive rules enable this subtyping relation. With distributivity of unions over intersections (and vice-versa), the type $(\mathsf{Int} \vee \mathsf{Bool}) \wedge (\mathsf{String} \vee \mathsf{Bool})$ is a subtype of $(\mathsf{Int} \wedge \mathsf{String}) \vee \mathsf{Bool}$ (in particular, by rule DS-DISTOR in Figure 3.5). As such with distributive subtyping, the following Ceylon program type-checks:

```
variable <Integer | Boolean> & <String | Boolean> x = true; do(x);
```

Nominal Types and Other Extensions to λ_u. We also study several extensions to λ_u, including nominal types. The extension with nominal types is interesting, since nominal types are highly relevant in practice. We show a sound, complete and decidable algorithmic formulation of subtyping with nominal types by extending an approach by Huang and Oliveira [2021]. We show that disjointness can also be employed in the presence of nominal types. This extension rejects ambiguous programs with overlapping nominal types in different branches of switch construct at compile time. It illustrates that disjointness is practically applicable to structural types as well as the nominal types. For example, the following program will statically be rejected in λ_u with nominal types:

```
Bool isstudent (x: Person | Student) = switch (x)
                                        (y : Person)  → false
                                        (y : Student) → true
```

Whereas, the following program will be accepted if we know that Person and Vehicle are disjoint:

```
Bool isvehicle (x: Person | Vehicle) = switch (x)
                                        (y : Person)  → false
                                        (y : Vehicle) → true
```

Type	A, B, C	$::=$	$\top \mid \bot \mid \mathsf{Int} \mid A \to B \mid A \vee B \mid \mathsf{Null}$
Expr	e	$::=$	$x \mid i \mid \lambda x.e \mid e_1\, e_2 \mid \mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} \mid \mathsf{null}$
Value	v	$::=$	$i \mid \lambda x.e \mid \mathsf{null}$
Context	Γ	$::=$	$\cdot \mid \Gamma, x : A$

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

$$\frac{}{A <: \top}\ \textsc{S-Top} \qquad \frac{}{\mathsf{Null} <: \mathsf{Null}}\ \textsc{S-Null} \qquad \frac{}{\mathsf{Int} <: \mathsf{Int}}\ \textsc{S-Int} \qquad \frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2}\ \textsc{S-Arrow} \qquad \frac{}{\bot <: A}\ \textsc{S-Bot}$$

$$\frac{A <: C \qquad B <: C}{A \vee B <: C}\ \textsc{S-OrA} \qquad \frac{A <: B}{A <: B \vee C}\ \textsc{S-OrB} \qquad \frac{A <: C}{A <: B \vee C}\ \textsc{S-OrC}$$

Figure 3.1: Syntax and subtyping for λ_u.

3.2 The Union Calculus λ_u

This section introduces the simplified union calculus λ_u. The distinctive feature of the λ_u calculus is a type-based switch expression with disjoint cases, which can be used to eliminate values with union types. In this first formulation of λ_u we only include the essential features of a calculus with disjoint switches: union types and disjoint switches. Section 3.3 then presents a richer formulation of λ_u with several extensions of practical relevance. We adapt the notion of disjointness from previous work on *disjoint intersection types* [Oliveira et al. 2016] to λ_u, and show that λ_u is type sound and deterministic.

3.2.1 Syntax

Figure 3.1 shows the syntax for λ_u. Metavariables A, B and C range over types. Types include top ($\top$), bottom ($\bot$), integer types (Int), function types ($A \to B$), union types ($A \vee B$) and null types (Null). Metavariable e ranges over expressions. Expressions include variables (x), integers (i), lambda abstractions ($\lambda x.e$), applications ($e_1\, e_2$), a novel switch expression ($\mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\}$) and the null expression. The `switch` expression evaluates a specific branch by matching the types in the cases. Note that, although the switch expression in λ_u only has two branches, a multi-branch switch can be easily encoded by employing nested switch expressions. We model the two-branch switch for keeping the formalization simple, but no expressive power is lost compared to a multi-branch switch. Metavariable v ranges over values. Values include i, $\lambda x.e$ and null expressions. Finally, a context (Γ) maps variables to their associated types.

3.2.2 Subtyping

The subtyping rules for λ_u are shown at the bottom of Figure 3.1. The rules are standard. Rule S-TOP states that all types are subtypes of the $\top$ type. Rule S-BOT states that $\bot$ type is subtype of all types. Rule S-NULL states that the Null type is a subtype of itself. Rules S-INT and S-ARROW are standard rules for integers and functions respectively. Functions are contravariant in input types and covariant in output types. Rules S-ORA, S-ORB, and S-ORC deal with the subtyping for union types. Rule S-ORA says that the union type of A and B is a subtype of another type C if both A and B are subtypes of C. Rules S-ORB and S-ORC state if a type is subtype of one of the components of a union type, then it is subtype of the union type. The subtyping relation for λ_u is reflexive and transitive.

Lemma 3.1 (Subtyping reflexivity). $A <: A$

Lemma 3.2 (Subtyping transitivity). *If* $A <: B$ *and* $B <: C$ *then* $A <: C$

3.2.3 Disjointness

The motivation for a definition of disjointness based on bottom-like types is basically that in disjoint switches, the selection of branches can be viewed as a type-safe downcast. For instance, recall the example in Section 3.1:

```
switch x {
   (y : String | Int)  → 0
   (y : Int | Bool)    → 1
}
```

Here `x` may have type `Int | String | Bool` and the two branches in the disjoint switch cover two subtypes `String | Int` and `Int | Bool`. When considered together those subtypes cover all possibilities for the value `x` (i.e. `x` can be either an integer, a string or a boolean, and the two cases cover all those possibilities). The *exhaustiveness* of the downcasts is what ensures that the downcasts are type-safe (that is they cannot fail at runtime). However, we also need to ensure that the two cases do not overlap to prevent ambiguity. In essence, in this simple setting of λ_u, checking that two types do not overlap amounts to check that there are no basic types (like `Int` or `Bool`) in common. In other words the only common subtypes should be bottom-like types.

Bottom-Like Types. *Bottom-like* types are types that are equivalent (i.e. both supertypes and subtypes) to $\bot$. In λ_u, there are infinitely many such types, and they all are uninhabited by values. According to the inductive definition shown at the top of Figure 3.2, they include

the bottom type itself (via rule BL-BOT) and unions of two bottom-like types (via rule BL-OR), e.g. $\bot \vee \bot$. The correctness of our definition for bottom-like types is established by the following property:

Lemma 3.3 (Bottom-Like Soundness and Completeness). $\rfloor A \lfloor$ *if and only if* $\forall B$, $A <: B$.

Declarative Disjointness. The declarative definition for disjointness is as follows:

Definition 4 ($\bot$-Disjointness). $A * B ::= \forall$ C, if $C <: A$ and $C <: B$ then $\rfloor C \lfloor$

That is, two types are disjoint if all their common subtypes are *bottom-like*.

We give a few examples next, employing a bold font to highlight the types being compared for disjointness: (Note that $\boldsymbol{A}$ and $\boldsymbol{B}$ are placeholders for actual types)

1. $\boldsymbol{A} = \textbf{Int}$, $\boldsymbol{B} = \textbf{Int} \rightarrow \textbf{Bool}$ **:** Int and Int $\rightarrow$ Bool are disjoint types. All common subtypes of Int and Int $\rightarrow$ Bool are *bottom-like* types, including $\bot$ and unions of $\bot$ types.

2. $\boldsymbol{A} = \textbf{Int} \vee \textbf{Bool}$, $\boldsymbol{B} = \bot$ **:** Int $\vee$ Bool and $\bot$ are disjoint types. All common subtypes are *bottom-like*. In general, the type $\bot$ (or any *bottom-like* type) is disjoint to another type.

3. $\boldsymbol{A} = \textbf{Int}$, $\boldsymbol{B} = \textbf{Int}$ **:**
Int and Int are not disjoint types because they share a common subtype Int which is not *bottom-like*. In general, one type is not disjoint with itself, unless it is bottom-like.

4. $\boldsymbol{A} = \textbf{Int}$, $\boldsymbol{B} = \top$ **:** Int and $\top$ are not disjoint types because they share a common subtype Int which is not *bottom-like*. In general no type is disjoint to $\top$, except for *bottom-like* types. Also, one type is not disjoint with itself, unless it is *bottom-like*.

5. $\boldsymbol{A} = \textbf{Int} \rightarrow \textbf{Bool}$, $\boldsymbol{B} = \textbf{String} \rightarrow \textbf{Char}$ **:** The types Int $\rightarrow$ Bool and String $\rightarrow$ Char are not disjoint, since we can find non-bottom-like types that are subtypes of both types. For instance $\top \rightarrow \bot$ is a subtype of both types. More generally, any two function types can never be disjoint: it is always possible to find a common subtype, which is not *bottom-like*.

Disjointness for Intersection Types. In essence, disjointness for λ_u is dual to the disjointness notion in λ_i [Oliveira et al. 2016], a calculus with disjoint intersection types. In λ_u, two types are disjoint if they do not share any common *subtype* which is not *bottom-like*. While in λ_i, two types are disjoint if they do not share any common *supertype* which is not *top-like* (i.e. equivalent to $\top$). While a disjoint switch provides deterministic behavior for

downcasting, disjointness in intersection types prevents ambiguity in upcasting. In a type-safe setting, if two values v_1 and v_2 (of type A_1 and A_2) can both be upcasted to type B, then B must be a common supertype of A_1 and A_2. The disjointness restriction of A_1 and A_2 means they cannot have any *non-top-like* common supertype, so when the two values together upcasted to a type like Int, only one of them can contribute to the result. Prior work on disjoint intersection types is also helpful to find an algorithmic formulation of disjointness. Declarative disjointness does not directly lead to an algorithm. However, we can find an algorithmic formulation that employs dual rules to those for disjoint intersection types.

Algorithmic Disjointness. We present an algorithmic version of disjointness in the middle of Figure 3.2. Rules AD-BTMR and AD-BTML state that the $\bot$ type is disjoint to all types. Rules AD-INTL and AD-INTR state that Int and $A \to B$ are disjoint types. Algorithmic disjointness can further be scaled to more primitive disjoint types such as Bool and String by adding more rules similar to rules AD-INTL and AD-INTR for additional primitive types. Rules AD-NULL-INTL and AD-NULL-INTR state that Null and Int are disjoint types. Similarly, rules AD-NULL-FUNL and AD-NULL-FUNR state that Null and $A \to B$ are disjoint types. Rules AD-ORL and AD-ORR are two symmetric rules for union types. Any type C is disjoint to an union type $A \vee B$ if C is disjoint to both A and B. We show that algorithmic disjointness is sound and complete with respect to its declarative specification (Definition 4).

Theorem 3.4 (Soundness and Completeness of Algorithmic Disjointness)**.** *$A *_a B$ if and only if $A * B$.*

A natural property of λ_u is that if type A and type B are two disjoint types, then subtypes of A are disjoint to subtypes of B. This property dualises the *covariance of disjointness* property in calculi with disjoint intersection types [Alpuim et al. 2017].

Lemma 3.5 (Disjointness contravariance)**.** *If $A * B$ and $C <: A$ and $D <: B$ then $C * D$.*

Further, disjointness relation is symmetric:

Lemma 3.6 (Disjointness Symmetry)**.** *If $A * B$ then $B * A$.*

3.2.4 Typing

The typing rules are shown at the bottom of Figure 3.2. They are mostly standard. An integer has type Int, null has type Null and variable x gets type from the context. Rule TYP-APP is the standard rule for function application. Similarly, rule TYP-SUB and rule TYP-ABS are standard subsumption and abstraction rules respectively. The most interesting and novel rule is for

$\boxed{\rfloor A \lfloor}$ *(Bottom-Like Types)*

$$\frac{}{\rfloor \bot \lfloor}\ \text{BL-BOT} \qquad \frac{\rfloor A \lfloor \qquad \rfloor B \lfloor}{\rfloor A \vee B \lfloor}\ \text{BL-OR}$$

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

$$\frac{}{A *_a \bot}\ \text{AD-BTMR} \qquad \frac{}{\bot *_a A}\ \text{AD-BTML} \qquad \frac{}{\mathsf{Int} *_a A \to B}\ \text{AD-INTL} \qquad \frac{}{A \to B *_a \mathsf{Int}}\ \text{AD-INTR} \qquad \frac{}{\mathsf{Null} *_a \mathsf{Int}}\ \text{AD-NULL-INTL}$$

$$\frac{}{\mathsf{Int} *_a \mathsf{Null}}\ \text{AD-NULL-INTR} \qquad \frac{}{\mathsf{Null} *_a A \to B}\ \text{AD-NULL-FUNL} \qquad \frac{}{A \to B *_a \mathsf{Null}}\ \text{AD-NULL-FUNR} \qquad \frac{A *_a C \qquad B *_a C}{A \vee B *_a C}\ \text{AD-ORL}$$

$$\frac{A *_a B \qquad A *_a C}{A *_a B \vee C}\ \text{AD-ORR}$$

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

$$\frac{}{\Gamma \vdash i : \mathsf{Int}}\ \text{TYP-INT} \qquad \frac{}{\Gamma \vdash \mathsf{null} : \mathsf{Null}}\ \text{TYP-NULL} \qquad \frac{\Gamma \vdash e : A \qquad A <: B}{\Gamma \vdash e : B}\ \text{TYP-SUB} \qquad \frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B}\ \text{TYP-APP}$$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B}\ \text{TYP-ABS} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \text{TYP-VAR} \qquad \frac{\Gamma \vdash e : A \vee B \qquad \Gamma, x : A \vdash e_1 : C \qquad \Gamma, y : B \vdash e_2 : C \qquad A * B}{\Gamma \vdash \mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} : C}\ \text{TYP-SWITCH}$$

Figure 3.2: Bottom-like types, algorithmic disjointness and typing for λ_u.

switch expressions (rule TYP-SWITCH). It has four conditions. First, $\Gamma \vdash e : A \vee B$ ensures *exhaustiveness* of the cases in the switch: e must check against the types in the branches of the switch. The next two conditions ensure that branches of case expressions are well-typed and have type C, where the input variable is bound to type A and to type B respectively in the two branches. Finally, $A * B$ guarantees the *disjointness* of A and B. This forbids overlapping types for the branches of case expressions to avoid non-deterministic results. Since all the branches have type C, the whole switch expression has type C. Note that the two branches can have different return types. For example, if e_1 and e_2 have type Int and String respectively, the whole expression can have type $\mathsf{Int} \vee \mathsf{String}$.

$\boxed{e \longrightarrow e'}$ *(Operational Semantics)*

$$\text{STEP-APPL}\ \frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2} \qquad \text{STEP-APPR}\ \frac{e \longrightarrow e'}{v\, e \longrightarrow v\, e'} \qquad \text{STEP-BETA}\ \frac{}{(\lambda x.e)\, v \longrightarrow e[x \rightsquigarrow v]}$$

$$\text{STEP-SWITCH}\ \frac{e \longrightarrow e'}{\mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow \mathsf{switch}\ e'\ \{(x : A) \to e_1, (y : B) \to e_2\}}$$

$$\text{STEP-SWITCHL}\ \frac{\lfloor v \rfloor <: A}{\mathsf{switch}\ v\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow e_1[x \rightsquigarrow v]}$$

$$\text{STEP-SWITCHR}\ \frac{\lfloor v \rfloor <: B}{\mathsf{switch}\ v\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow e_2[y \rightsquigarrow v]}$$

Approximate Type $\lfloor v \rfloor$

$\lfloor i \rfloor$	=	Int
$\lfloor \lambda x.e \rfloor$	=	$\top \to \bot$
$\lfloor \mathsf{null} \rfloor$	=	Null

Figure 3.3: Operational semantics and approximate type definitions for λ_u.

3.2.5 Operational Semantics

Now we discuss the small-step operational semantics of λ_u. An important aspect of this semantics is that union elimination is *type-directed*: types are used to pick the branch of the switch expression.

Figure 3.3 shows the operational semantics of λ_u. Rules STEP-APPL, STEP-APPR, and STEP-BETA are the standard call-by-value reduction rules for applications. Of particular interest are rules STEP-SWITCH, STEP-SWITCHL, and STEP-SWITCHR, which reduce the *switch* expressions. First, rule STEP-SWITCH reduces the case expression e, until it becomes a value v, at which point we must choose between the two branches of *switch*. We do so by inspecting the type of v: if the *approximate type* of v is a subtype of type of the left branch, then rule STEP-SWITCHL evaluates the left branch of the *switch* expression, or otherwise if it is a subtype of the type of the right branch, rule STEP-SWITCHR evaluates the right branch.

Note that the approximate type definition gives only a subtype of the actual type for a lambda value. This works, because the approximate type is only employed to allow the selection of a case with a function type, and in λ_u two function types can never be disjoint.

Therefore, if there is a branch with a function type, then that must be the branch that applies to a lambda value. Note also that the program has been type-checked before hand, so we know that the static type of the value is compatible with the types on the branches. The subtyping condition in rules STEP-SWITCHL and STEP-SWITCHR is important, as it provides flexibility for the value to have various subtypes of A and B, instead of strictly having those types. Recall that the typing rule for *switch* (rule TYP-SWITCH) requires that types of left and right branches of a *switch* expression to be disjoint. This ensures that rules STEP-SWITCHL and STEP-SWITCHR cannot overlap, which, as we will see, is important for the operational semantics to be *deterministic*.

Approximate Type. The dynamic semantics employs a simple function that retrieves the dynamic type of a value. The definition is shown at the bottom of Figure 3.3. Int and Null are returned when v is an integer i or a null respectively. Otherwise, for functions, the least function type $\top \to \bot$ is returned.

3.2.6 Type Soundness and Determinism

In this section, we prove that λ_u is type sound and deterministic. Type soundness is established by the type preservation and progress theorems. Type preservation (Theorem 3.7) states that types are preserved during reduction. Progress (Theorem 3.8) states that well typed programs never get stuck: a well typed expression e is either a value or it can reduce to some other expression e'.

Theorem 3.7 (Type Preservation). *If $\Gamma \vdash e : A$ and $e \longrightarrow e'$ then $\Gamma \vdash e' : A$.*

Theorem 3.8 (Progress). *If $\cdot \vdash e : A$ then either e is a value; or $e \longrightarrow e'$ for some e'.*

Determinism of λ_u (Theorem 3.10) ensures that a well-typed program reduces to a unique result. In particular, it guarantees that switch expressions are not order-sensitive: at any time, only one of the rules STEP-SWITCHL and STEP-SWITCHR can apply. The determinism of the switch expression relies on an essential property that a value cannot check against two disjoint types (Lemma 3.9).

Lemma 3.9 (Exclusivity of Disjoint Types). *If $A * B$ then $\nexists\, v$ such that both $\Gamma \vdash v : A$ and $\Gamma \vdash v : B$ holds.*

Theorem 3.10 (Determinism). *If $\Gamma \vdash e : A$ and $e \longrightarrow e_1$ and $e \longrightarrow e_2$ then $e_1 = e_2$.*

3.2.7 An Alternative Specification for Disjointness

The current definition of disjointness (Definition 4) works well for the calculus presented in this section. But it is not the only possible formulation of disjointness. An equivalent formulation of disjointness is:

Definition 5 (∧-Disjointness)**.** $A * B ::= \nexists\, C^{\circ}, C^{\circ} <: A$ and $C^{\circ} <: B$

According to the new definition, two types are disjoint if they do not have common subtypes that are *ordinary*. Ordinary types (denoted by C°) are essentially those types that are primitive, such as integers and functions (see Figure 3.4 for a formal definition).

For the calculus presented in this section, we prove that the new definition is equivalent to the previous definition of disjointness.

Lemma 3.11 (Disjointness Equivalence)**.** *Definition 5 (∧-Disjointness) is sound and complete to Definition 4 (⊥-Disjointness) in λ_u defined in this section.*

Why do we introduce the new definition of disjointness? It turns out that the previous definition is not sufficient when the calculus is extended with intersection types. As we will see, the new definition will play an important role in such variant of the calculus.

3.3 λ_u with Intersections, Distributive Subtyping and Nominal Types

In this section we extend λ_u with intersection types, nominal types and an enriched distributive subtyping relation. The study of an extension of λ_u with intersection types is motivated by the fact that most languages with union types also support intersection types (for example Ceylon, Scala or TypeScript). Furthermore, languages like Ceylon or Scala also support some form of distributive subtyping, as well as nominal types. Therefore it is important to understand whether those extensions can be easily added or whether there are some challenges. As it turns out, adding intersection types does pose a challenge, since the notion of disjointness inspired from disjoint intersection types [Oliveira et al. 2016] no longer works. Moreover subtyping relations with distributive subtyping add significant complexity, and we need an extension that supports nominal types as well. We show that desirable properties, including type soundness and determinism, are preserved in the extended version of λ_u. Moreover we prove that both disjointness and subtyping have sound, complete and decidable algorithms.

$$
\begin{array}{lll}
A, B, C & ::= & \top \mid \bot \mid \mathsf{Int} \mid A \to B \mid \mathsf{Null} \mid A \vee B \mid \boxed{A \wedge B} \mid \boxed{\mathsf{P}} \\
\boxed{A^\circ, B^\circ, C^\circ} & ::= & \mathsf{Int} \mid \mathsf{Null} \mid A \to B \mid \mathsf{P} \\
e & ::= & x \mid i \mid \lambda x.e \mid e_1\, e_2 \mid \mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} \mid \mathsf{null} \mid \boxed{\mathsf{new}\ \mathsf{P}} \\
v & ::= & i \mid \lambda x.e \mid \mathsf{null} \mid \boxed{\mathsf{new}\ \mathsf{P}} \\
\Gamma & ::= & \cdot \mid \Gamma, x : A \\
\boxed{\Delta} & ::= & \cdot \mid \Delta, \mathsf{P}_1 \leq \mathsf{P}_2 \mid \Delta, \mathsf{P} \leq \top
\end{array}
$$

$\boxed{\Delta \vdash A}$ *(Well-formed Types)*

$$
\frac{}{\Delta \vdash \top}\ \text{WFT-TOP} \qquad
\frac{}{\Delta \vdash \bot}\ \text{WFT-BOT} \qquad
\frac{}{\Delta \vdash \mathsf{Int}}\ \text{WFT-INT} \qquad
\frac{}{\Delta \vdash \mathsf{Null}}\ \text{WFT-NULL} \qquad
\frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \to B}\ \text{WFT-ARROW}
$$

$$
\frac{\mathsf{P} \in dom\ \Delta}{\Delta \vdash \mathsf{P}}\ \text{WFT-PRIM} \qquad
\frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \vee B}\ \text{WFT-OR} \qquad
\frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \wedge B}\ \text{WFT-AND}
$$

$\boxed{ok\ \Delta}$ *(Well-formed Nominal Contexts)*

$$
\frac{}{ok\ \cdot}\ \text{OKP-EMPTY} \qquad
\frac{ok\ \Delta \qquad \mathsf{P} \notin dom\ \Delta}{ok\ \Delta, \mathsf{P} \leq \top}\ \text{OKP-CONS} \qquad
\frac{ok\ \Delta \qquad \Delta \vdash \mathsf{P}_2 \qquad \mathsf{P}_1 \notin dom\ \Delta}{ok\ \Delta, \mathsf{P}_1 \leq \mathsf{P}_2}\ \text{OKP-SUB}
$$

Figure 3.4: Syntax and well-formedness.

3.3.1 Syntax, Well-formedness and Ordinary Types

The syntax for this section mostly follows from Section 3.2, with the additional syntax given in Figure 3.4. The most significant difference and novelty in this section is the addition of intersection types $A \wedge B$ and an infinite set of nominal types. We use metavariable P to stand for nominal types. Expressions are extended with a new expression ($\mathsf{new}\ \mathsf{P}$) to create instances of nominal types. The expression $\mathsf{new}\ \mathsf{P}$ is also a value. Context Γ stays the same as in Section 3.2. We add a new context Δ, to track nominal types and their supertypes. For example, adding $\mathsf{P}_1 \leq \mathsf{P}_2$ to Δ declares a new nominal type P_1 that is a subtype of P_2. For a well-formed context, the supertype P_2 has to be declared before P_1. We also allow to declare a new nominal type P_1 with $\top$ as its supertype by adding $\mathsf{P}_1 \leq \top$ to Δ. Metavariable A°, B° and C° ranges over ordinary types [Davies and Pfenning 2000]. There are four kinds of ordinary types: integers, null, function types and nominal types. Well-formed types and well-formedness of ordinary contexts Δ are shown in Figure 3.4.

Remark on Nominal Types. Note that our formulation of nominal types is simplified in two ways compared to languages like Java. Firstly, we do not consider arguments when building new expressions (i.e. we do not allow expressions like `new Person("John")`). Secondly, we also do not introduce class declarations, which would allow nominal types to be associated with method implementations. We follow a design choice for nominal types similar to Featherweight Java [Igarashi et al. 2001]. Featherweight Java uses a fixed size context for nominal types. Diamond inheritance is also not supported in Featherweight Java, and we follow that design choice as well. However, we believe that supporting diamond inheritance in our calculus is relatively easy. These simplifications keep the calculus simple, while capturing the essential features that matter for disjointness and the formalization of disjoint switches. Allowing for a more complete formulation of nominal types can be done in mostly standard ways.

3.3.2 Distributive Subtyping

Another interesting feature of this section is the addition of distributive subtyping to λ_u. Ceylon employs an enriched distributive subtyping relation [Muehlboeck and Tate 2018] that is based on the B+ logic [Routley and Meyer 1972; van Bakel et al. 2000]. To obtain an equivalent algorithmic formulation of subtyping, we employ the idea of *splittable types* [Huang and Oliveira 2021], but extend that algorithm with the `Null` type and nominal types.

Distributive subtyping relation. Figure 3.5 shows a declarative version of distributive subtyping for λ_u with intersection and nominal types. Subtyping includes axioms for reflexivity (rule ds-refl) and transitivity (rule ds-trans). Rules ds-top, ds-bot, ds-arrow, and ds-ora have been discussed in Section 3.2. Rule ds-prim states that a nominal type is a subtype of type A if it is declared as subtype of A in Δ. Note that A can either be a nominal type or $\top$ under a well-formed context Δ. With the help of rule ds-trans, the subtyping of primitive types can also be constructed indirectly, e.g. $\mathsf{P}_1 \leq \top, \mathsf{P}_2 \leq \mathsf{P}_1, \mathsf{P}_3 \leq \mathsf{P}_2 \vdash \mathsf{P}_3 \leq \mathsf{P}_1$. Compared with the algorithmic formulation, having an explicit transitivity rule considerably simplifies the rules for nominal types. Rules ds-orb and ds-orc state that a subpart of a union type is a subtype of whole union type. Rule ds-anda states that a type A is a subtype of the intersection of two types B and C only if A is a subtype of both B and C. Rules ds-andb and ds-andc state that intersection type $A_1 \wedge A_2$ is a subtype of both A_1 and A_2 separately. Rule ds-distarr distributes function types over intersection types. It states that $(A \to B_1) \wedge (A \to B_2)$ is a subtype of $A \to (B_1 \wedge B_2)$. Rule ds-distarru states that $(A_1 \to B) \wedge (A_2 \to B)$ is a subtype of $(A_1 \vee A_2) \to B$ type. Rule ds-distor distributes intersections over unions.

$\boxed{\Delta \vdash A \leq B}$ *(Declarative Subtyping with Distributivity)*

$$\text{DS-REFL}\ \frac{ok\ \Delta \qquad \Delta \vdash A}{\Delta \vdash A \leq A} \qquad \text{DS-TRANS}\ \frac{\Delta \vdash A \leq B \qquad \Delta \vdash B \leq C}{\Delta \vdash A \leq C} \qquad \text{DS-ARROW}\ \frac{\Delta \vdash B_1 \leq A_1 \qquad \Delta \vdash A_2 \leq B_2}{\Delta \vdash A_1 \to A_2 \leq B_1 \to B_2}$$

$$\text{DS-PRIM}\ \frac{ok\ \Delta \qquad \mathsf{P} \leq A \in \Delta}{\Delta \vdash \mathsf{P} \leq A} \qquad \text{DS-ORA}\ \frac{\Delta \vdash A_1 \leq B \qquad \Delta \vdash A_2 \leq B}{\Delta \vdash A_1 \vee A_2 \leq B} \qquad \text{DS-ORB}\ \frac{ok\ \Delta \qquad \Delta \vdash A_1 \qquad \Delta \vdash A_2}{\Delta \vdash A_1 \leq A_1 \vee A_2}$$

$$\text{DS-ORC}\ \frac{ok\ \Delta \qquad \Delta \vdash A_1 \qquad \Delta \vdash A_2}{\Delta \vdash A_2 \leq A_1 \vee A_2} \qquad \text{DS-ANDA}\ \frac{\Delta \vdash B \leq A_1 \qquad \Delta \vdash B \leq A_2}{\Delta \vdash B \leq A_1 \wedge A_2}$$

$$\text{DS-ANDB}\ \frac{ok\ \Delta \qquad \Delta \vdash A_1 \qquad \Delta \vdash A_2}{\Delta \vdash A_1 \wedge A_2 \leq A_1} \qquad \text{DS-ANDC}\ \frac{ok\ \Delta \qquad \Delta \vdash A_1 \qquad \Delta \vdash A_2}{\Delta \vdash A_1 \wedge A_2 \leq A_2}$$

$$\text{DS-DISTARRU}\ \frac{ok\ \Delta \qquad \Delta \vdash A_1 \qquad \Delta \vdash A_2 \qquad \Delta \vdash B}{\Delta \vdash (A_1 \to B) \wedge (A_2 \to B) \leq (A_1 \vee A_2) \to B} \qquad \text{DS-TOP}\ \frac{ok\ \Delta \qquad \Delta \vdash A}{\Delta \vdash A \leq \top}$$

$$\text{DS-DISTOR}\ \frac{ok\ \Delta \qquad \Delta \vdash A_1 \qquad \Delta \vdash A_2 \qquad \Delta \vdash B}{\Delta \vdash (A_1 \vee B) \wedge (A_2 \vee B) \leq (A_1 \wedge A_2) \vee B} \qquad \text{DS-BOT}\ \frac{ok\ \Delta \qquad \Delta \vdash A}{\Delta \vdash \bot \leq A}$$

$$\text{DS-DISTARR}\ \frac{ok\ \Delta \qquad \Delta \vdash A \qquad \Delta \vdash B_1 \qquad \Delta \vdash B_2}{\Delta \vdash (A \to B_1) \wedge (A \to B_2) \leq A \to (B_1 \wedge B_2)}$$

Figure 3.5: Distributive subtyping for λ_u with intersection types and nominal types.

Algorithmic Subtyping. Distributive rules make it hard to eliminate the transitivity rule. Our algorithmic formulation of distributive subtyping is based on a formulation using splittable types by Huang and Oliveira [2021]. The basic idea is to view the distributive rules as some expansion of intersection and union types. For example, rule DS-DISTARR makes $A \to B_1 \wedge B_2$ and $(A \to B_1) \wedge (A \to B_2)$ mutual subtypes. Thus $A \to B_1 \wedge B_2$ is treated like $(A \to B_1) \wedge (A \to B_2)$ in the three intersection-related rules AS-ANDA, AS-ANDB, and AS-ANDC. Here we use $A \approxeq B \wedge C$ to denote that type A can be split into B and C (and therefore, A is equivalent to $B \wedge C$) according to the procedure designed by Huang and Oliveira. Union and union-like types (e.g. $(A_1 \vee A_2) \wedge B \approxeq A_1 \wedge B \vee A_2 \wedge B$) are handled in similar way in rules AS-ORA, AS-ORB, and AS-ORC. For further details of algorithmic subtyping we refer to their paper.

$\boxed{\Delta \vdash A <: B}$ *(Algorithmic Subtyping with Distributivity)*

$$\frac{\Delta \vdash B_1 <: A_1 \qquad \Delta \vdash A_2 <: B_2}{\Delta \vdash A_1 \to A_2 <: B_1 \to B_2}\ \textsc{as-arrow}$$

$$\frac{ok\ (\Delta, \mathsf{P}_1 <: \mathsf{P}_2) \qquad \Delta \vdash \mathsf{P}_2 <: \mathsf{P}_3}{\Delta, \mathsf{P}_1 <: \mathsf{P}_2 \vdash \mathsf{P}_1 <: \mathsf{P}_3}\ \textsc{as-primEq}$$

$$\frac{ok\ \Delta \qquad \Delta \vdash A}{\Delta \vdash A <: A}\ \textsc{as-refl}$$

$$\frac{ok\ (\Delta, \mathsf{P}_2 <: A) \qquad \mathsf{P}_1 \neq \mathsf{P}_2 \qquad \Delta \vdash \mathsf{P}_1 <: \mathsf{P}_3}{\Delta, \mathsf{P}_2 <: A \vdash \mathsf{P}_1 <: \mathsf{P}_3}\ \textsc{as-primNEq}$$

$$\frac{A \approxeq A_1 \vee A_2 \qquad \Delta \vdash A_1 <: B \qquad \Delta \vdash A_2 <: B}{\Delta \vdash A <: B}\ \textsc{as-orA}$$

$$\frac{ok\ \Delta \qquad \Delta \vdash A}{\Delta \vdash A <: \top}\ \textsc{as-top}$$

$$\frac{A \approxeq A_1 \vee A_2 \qquad \Delta \vdash B <: A_1}{\Delta \vdash B <: A}\ \textsc{as-orB}$$

$$\frac{A \approxeq A_1 \vee A_2 \qquad \Delta \vdash B <: A_2}{\Delta \vdash B <: A}\ \textsc{as-orC}$$

$$\frac{A \approxeq A_1 \wedge A_2 \qquad \Delta \vdash B <: A_1 \qquad \Delta \vdash B <: A_2}{\Delta \vdash B <: A}\ \textsc{as-andA}$$

$$\frac{ok\ \Delta \qquad \Delta \vdash A}{\Delta \vdash \bot <: A}\ \textsc{as-bot}$$

$$\frac{A \approxeq A_1 \wedge A_2 \qquad \Delta \vdash A_1 <: B}{\Delta \vdash A <: B}\ \textsc{as-andB}$$

$$\frac{A \approxeq A_1 \wedge A_2 \qquad \Delta \vdash A_2 <: B}{\Delta \vdash A <: B}\ \textsc{as-andC}$$

Figure 3.6: Algorithmic subtyping for λ_u with distributivity, intersection and nominal types.

Subtyping Nominal Types. However, Huang and Oliveira's algorithm does not account for `Null` and nominal types. We add the nominal context Δ in the subtyping judgment and extend the subtyping algorithm with `Null` and nominal types. Nominal types are not splittable, and their subtyping relation is defined by the transitive closure of the context. They are supertypes of $\bot$ and subtypes of $\top$, but not related with other primitive types like Int and Null. So for nominal types, we mainly focus on checking the subtyping relationship among them in our algorithm. Given a well-formed context, any nominal type P appears only once in a subtype position as an explicit declaration for P, and its direct supertype, if is not $\top$, must be declared before P. Thus if $\Delta \vdash \mathsf{P}_1 <: \mathsf{P}_2$ holds, either P_2 is introduced before P_1 in Δ, or they are the same type, in which case the goal can be solved by rule as-refl. For the other cases, we recursively search for P_1 in all subtype positions of the context Δ (rule as-primNEq). When we find P_1, we check its direct supertype. If it is $\top$, no other nominal types can be supertypes of P_1. So in rule as-primEq, we only consider when the direct supertype is another primitive P_2. For P_3 to be a supertype of P_1, it must either equal to P_2, or it is related

to P_2 by the smaller context. In either case, we can prove that P_3 is a supertype of the direct supertype of P_1. We show that the algorithmic subtyping relation for λ_u with intersection types, nominal types and distributivity rules is reflexive and transitive:

Lemma 3.12 (Subtyping reflexivity)**.** $\Delta \vdash A <: A$

Lemma 3.13 (Subtyping transitivity)**.** *If* $\Delta \vdash A <: B$ *and* $B <: C$ *then* $A <: C$

Inversion Lemmas for Type Soundness. Having an algorithmic formulation of subtyping is useful to prove several inversion lemmas that are used in the type soundness proof. For instance, it allows us to prove the following lemma:

Lemma 3.14 (Inversion on Function Types)**.** *If* $\Delta \vdash A_1 \to A_2 <: B_1 \to B_2$ *then* $\Delta \vdash B_1 <: A_1$ *and* $\Delta \vdash A_2 <: B_2$.

While the additional distributive rules make function types more flexible, they retain the contravariance of argument types and covariance of return types. In addition, we show the formulation is sound and complete to the declarative subtyping and it is decidable whether a subtyping judgment holds under a given context.

Lemma 3.15 (Equivalence of subtyping)**.** $\Delta \vdash A \leq B$ *if and only if* $\Delta \vdash A <: B$.

Lemma 3.16 (Decidability of subtyping)**.** $\Delta \vdash A \leq B$ *is decidable.*

3.3.3 Disjointness Specification

Disjointness is another interesting aspect of the extension of λ_u. Unfortunately, Definition 4 does not work with intersection types. In what follows, we first explain why Definition 4 does not work, and then we show how to define disjointness in the presence of intersection types.

Bottom-like types, intersection types and disjointness. Recall that disjointness in Section 3.2 (Definition 4) depends on bottom-like types, where two types are disjoint only if all their common subtypes are bottom-like. However, this definition no longer works with the addition of intersection types. According to the subtyping rule for intersection types, any two types have their intersection as one common subtype. For non-bottom-like types, their intersection is also not bottom-like. For example, type Int and type Bool now have a non-bottom like subtype $\mathsf{Int} \wedge \mathsf{Bool}$. In other words, the disjointness definition fails, since we can always find a common non-bottom-like subtype for any two (non-bottom-like) types.

A POSSIBLE SOLUTION: THE CEYLON APPROACH. A possible solution for this issue is to add a subtyping rule which makes intersections of disjoint types subtypes of $\bot$.

$$\frac{A * B}{A \wedge B <: \bot}\ \textsc{s-disj}$$

This rule is adopted by the Ceylon language [Muehlboeck and Tate 2018]. With the rule S-DISJ now the type $\mathsf{Int} \wedge \mathsf{Bool}$ would be a bottom-like type, and the definition of disjointness used in Section 3.2 could still work. The logic behind this rule is that if we interpret types as sets of values, and intersection as set intersection, then intersecting disjoint sets is the empty set. In other words, we would get a type that has no inhabitants. For instance the set of all integers is disjoint to the set of all booleans, and the intersection of those sets is empty. However we do not adopt the Ceylon solution here for two reasons:

1. Rule S-DISJ complicates the system because it adds a mutual dependency between subtyping and disjointness: disjointness is defined in terms of subtyping, and subtyping now uses disjointness as well in rule S-DISJ. This creates important challenges for the metatheory. In particular, the completeness proof for disjointness becomes quite challenging.

2. Additionally, the assumption that intersections of disjoint types are equivalent to $\bot$ is too strong for some calculi with intersection types. If a merge operator [Reynolds 1988] is allowed in the calculus, intersection types can be inhabited with values (for example, in λ_i Oliveira et al. [2016], the type $\mathsf{Int} \wedge \mathsf{Bool}$ is inhabited by $1\,,,\,\mathsf{true}$). Considering those types bottom-like would lead to a problematic definition of subtyping, since some bottom-like types (those based on disjoint types) would be inhabited.

For those reasons we adopt a different approach in λ_u. Nevertheless, in Section 3.4 we show that it is possible to create an extension of λ_u that includes (and in fact generalizes) the Ceylon-style rule S-DISJ.

DISJOINTNESS BASED ON ORDINARY TYPES TO THE RESCUE. To solve the problem with the disjointness specification, we resort to the alternative definition of disjointness presented in Section 3.2.7. Note that now the disjointness definition also contains Δ as an argument to account for nominal types.

Definition 6 ($\wedge$-Disjointness). $\Delta \vdash A * B ::= \nexists\, C^\circ, \Delta \vdash C^\circ <: A$ and $\Delta \vdash C^\circ <: B$.

Interestingly, while in Section 3.2 such definition was equivalent to the definition using bottom-like types, this is no longer the case for λ_u with intersection types. To see why, consider again the types Int and Bool. Int and Bool do not share any common ordinary subtype. Therefore, Int and Bool are disjoint types according to Definition 6. We further illustrate Definition 6 with a few concrete examples:

1. $\boldsymbol{A = \mathsf{Int},\ B = \mathsf{Int} \rightarrow \mathsf{Bool}}$ **:** Since there is no ordinary type that is a subtype of both Int and Int $\rightarrow$ Bool, the two types are disjoint.

2. $\boldsymbol{A = \mathsf{Int} \vee \mathsf{Bool},\ B = \bot}$ **:** Since there is no ordinary type that is a subtype of both Int $\vee$ Bool and $\bot$, Int $\vee$ Bool and $\bot$ are disjoint types. In general, the $\bot$ type is disjoint to all types because $\bot$ does not have any ordinary subtype.

3. $\boldsymbol{A = \mathsf{Int} \wedge (\mathsf{Int} \rightarrow \mathsf{Bool}),\ B = \mathsf{Int}}$ **:** There is no ordinary type that is a subtype of both Int $\wedge$ (Int $\rightarrow$ Bool) and Int. Therefore, Int $\wedge$ (Int $\rightarrow$ Bool) and Int are disjoint types. In general, an intersection of two disjoint types (Int $\wedge$ (Int $\rightarrow$ Bool) in this case) is always disjoint to all types.

4. $\boldsymbol{A = \mathsf{Int} \wedge \mathsf{Bool},\ B = \mathsf{Int} \vee \mathsf{Bool}}$ **:** There is no ordinary type that is a subtype of both Int $\wedge$ Bool and Int $\vee$ Bool. Therefore, Int $\wedge$ Bool and Int $\vee$ Bool are disjoint types. In general, an intersection of two disjoint types (Int $\wedge$ Bool in this case) is always disjoint to all types.

5. $\boldsymbol{A = \mathsf{Int},\ B = \top}$ **:** In this case, Int and $\top$ share a common ordinary subtype which is Int. Therefore, Int and $\top$ are not disjoint types. $\top$ overlaps with any other types.

3.3.4 Algorithmic Disjointness

The change in the disjointness specification has a significant impact on an algorithmic formulation. In particular, it is not obvious at all how to adapt the algorithmic formulation in Figure 3.2. To obtain a sound, complete and decidable formulation of disjointness, we employ the novel notion of *lowest ordinary subtypes*.

Lowest ordinary subtypes ($|A|_\Delta$). Figure 3.7 shows the definition of *lowest ordinary subtypes* (LOS) ($|A|_\Delta$). LOS is defined as a function which returns a set of ordinary subtypes of the given input type. No ordinary type is a subtype of $\bot$. The only ordinary subtype of Int is Int itself. The function case is interesting. Since no two functions are disjoint in the calculus proposed in this section, the case for function types $A \rightarrow B$ returns $\top \rightarrow \bot$. This type is the least ordinary function type, which is a subtype of all function types. Lowest ordinary

Lowest Ordinary Subtypes (LOS) $|A|_\Delta$

$$
\begin{array}{lcl}
|\top|_\Delta & = & \{\mathsf{Int}, \top \to \bot, \mathsf{Null}\} \cup \mathrm{dom}\ \Delta \\
|\bot|_\Delta & = & \{\} \\
|\mathsf{Int}|_\Delta & = & \{\mathsf{Int}\} \\
|A \to B|_\Delta & = & \{\top \to \bot\} \\
|A \vee B|_\Delta & = & |A|_\Delta \cup |B|_\Delta \\
|A \wedge B|_\Delta & = & |A|_\Delta \cap |B|_\Delta \\
|\mathsf{Null}|_\Delta & = & \{\mathsf{Null}\} \\
|\mathsf{P}|_\Delta & = & \{\mathsf{P}\} \cup \Delta(\mathsf{P})
\end{array}
$$

Nominal Subtypes $\boxed{\Delta(A)}$

$$
\begin{array}{lcl}
\cdot(A) & = & \{\} \\
(\Delta', \mathsf{P} \leq B)(A) & = & \begin{cases} \{\mathsf{P}\} \cup \Delta'(A) \ \textbf{if}\ \mathsf{P} \leq A \in \Delta \\ \Delta'(A)\ \textbf{otherwise} \end{cases}
\end{array}
$$

Figure 3.7: Lowest ordinary subtypes function and additional typing rule for λ_u with intersection types and nominal types.

subtypes of $\top$ are Int, $\top \to \bot$, Null and all the nominal types defined in Δ. In the case of union types $A \vee B$, the algorithm collects the LOS of A and B and returns the union of the two sets. For intersection types $A \wedge B$ the algorithm collects the LOS of A and B and returns the intersection of the two sets. The lowest ordinary subtype of Null is Null itself. Finally, the LOS of P is the union of P itself with all subtypes of P defined in Δ. Note that LOS is defined as a structurally recursive function and therefore its decidability is immediate.

Algorithmic disjointness. With LOS, an algorithmic formulation of disjointness is straightforward:

Definition 7. $\Delta \vdash A *_a B ::= |A|_\Delta \cap |B|_\Delta = \{\}$.

The algorithmic formulation of disjointness in Definition 7 states that two types A and B are disjoint under the context Δ if they do not have any common lowest ordinary subtypes. In other words, the set intersection of $|A|_\Delta$ and $|B|_\Delta$ is the empty set. Note that this algorithm is naturally very close to Definition 6.

Soundness and completeness of algorithmic disjointness. Next, we show that disjointness algorithm is sound and complete with respect to disjointness specifications (The-

orem 3.17). Soundness and completeness of LOS are essential to prove Theorem 3.17. Both of these properties are shown in Lemma 3.18 and Lemma 3.19 respectively.

Theorem 3.17 (Disjointness Equivalence). *$\Delta \vdash A *_a B$ if and only if $\Delta \vdash A * B$.*

Lemma 3.18 (Soundness of $|A|_\Delta$). *$\forall$ well-formed Δ and A and B that are well-formed under Δ, if $B \in |A|_\Delta$, then $\Delta \vdash B <: A$.*

Lemma 3.19 (Completeness of $|A|_\Delta$). *$\forall\ A\ B^\circ$, if $\Delta \vdash B^\circ <: A$, then $B^\circ \in |A|_\Delta$, or B° is an arrow type and $\top \to \bot \in |A|_\Delta$.*

3.3.5 Typing, Semantics and Metatheory

Both typing and the operational semantics are parameterized by the nominal context Δ. The typing rules are extended with a rule for nominal types rule PTYP-PRIM as shown:

PTYP-PRIM
$$\frac{ok\ \Delta \qquad \Delta \vdash \mathsf{P}}{\Delta; \Gamma \vdash \mathsf{new}\ \mathsf{P} : \mathsf{P}}$$

The typing rule PTYP-PRIM states that under a well-formed context Δ and well-formed type P, $\mathsf{new}\ \mathsf{P}$ has type P. No additional reduction rule is required because $\mathsf{new}\ \mathsf{P}$ is a value. However, the rules STEP-SWITCHL and STEP-SWITCHR require Δ because they do a subtyping check. We illustrate the updated rule STEP-SWITCHL next:

NSTEP-SWITCHL
$$\frac{\Delta \vdash \lfloor v \rfloor <: A}{\Delta \vdash \mathsf{switch}\ v\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow e_1[x \rightsquigarrow v]}$$

Rule STEP-SWITCHR is updated similarly as:

NSTEP-SWITCHR
$$\frac{\Delta \vdash \lfloor v \rfloor <: B}{\Delta \vdash \mathsf{switch}\ v\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow e_2[y \rightsquigarrow v]}$$

All the other rules are essentially the same as in Section 3.2, modulo the extra nominal context Δ.

Example. Assuming a context $\Delta = \mathsf{Person} \leq \top, \mathsf{Student} \leq \mathsf{Person}, \mathsf{Robot} \leq \top$, $\mathsf{y} : \mathsf{Person} \mid \mathsf{Robot}$ and $\mathsf{x} : \mathsf{Student}$, we could write the following two switches:

```
switch(y)      // Accepted!
  (z : Person)  → false
  (z : Robot)   → true
```

```
switch (x)     // Rejected!
  (z : Person)  → false
  (z : Student) → true
```

In the above code, the first switch, using `y` is accepted, while the second one (using `x`) is rejected because the types overlap in that case.

Key Properties. We proved that λ_u with intersection types, nominal types and subtyping distributivity preserves type soundness and determinism.

Theorem 3.20 (Type Preservation). *If* $\Delta; \Gamma \vdash e : A$ *and* $\Delta \vdash e \longrightarrow e'$ *then* $\Delta; \Gamma \vdash e' : A$.

Theorem 3.21 (Progress). *If* $\Delta; \cdot \vdash e : A$ *then either* e *is a value; or* e *can take a step to* e'.

Theorem 3.22 (Determinism). *If* $\Delta; \Gamma \vdash e : A$ *and* $\Delta \vdash e \longrightarrow e_1$ *and* $\Delta \vdash e \longrightarrow e_2$ *then* $e_1 = e_2$.

3.4 Switches with Disjoint Polymorphism and Empty Types

The calculus introduced in Section 3.2 is a simple foundational lambda calculus with union types, similar to prior work on union types and their elimination forms [Benzaken et al. 2003; Castagna et al. 2014a; Dunfield 2014]. In Section 3.3 we extend λ_u with various interesting features including intersection types, nominal types and subtyping distributivity, inspired by Ceylon, which has similar features. In this section we discuss two more practical extensions:

- **Disjoint Polymorphism:** The first extension is an extension with a form of *disjoint polymorphism* [Alpuim et al. 2017], which allows the specification of disjointness constraints for type variables. Although Ceylon supports polymorphism, it does not support disjoint polymorphism. The extension with disjoint polymorphism is inspired by the work on disjoint intersection types, where disjoint polymorphism has been proposed to account for disjointness in a polymorphic language.

- **A Special Subtyping Rule for Empty Types:** The second extension that we discuss is an alternative subtyping formulation with a special subtyping rule for empty types, which follows the Ceylon approach.

Note that both extensions above have also been formalized in Coq and proved type-sound and deterministic. In addition, we also have a brief discussion about implementation considerations.

A, B, C	$::=$	$\top \mid \bot \mid \mathsf{Int} \mid A \rightarrow B \mid \mathsf{Null} \mid A \vee B \mid A \wedge B \mid \mathsf{P} \mid \alpha \mid \forall(\alpha * A).B$
e	$::=$	$x \mid i \mid \lambda x.e \mid e_1\, e_2 \mid \mathsf{switch}\ e\ \{(x : A) \rightarrow e_1, (y : B) \rightarrow e_2\} \mid \mathsf{null} \mid \mathsf{new}\ \mathsf{P} \mid e\, A \mid \Lambda(\alpha * A).e$
v	$::=$	$i \mid \lambda x.e \mid \mathsf{null} \mid \mathsf{new}\ \mathsf{P} \mid \Lambda(\alpha * A).e$
Γ	$::=$	$\cdot \mid \Gamma, x : A \mid \Gamma, \alpha * A$
Δ	$::=$	$\cdot \mid \Delta, \mathsf{P} <: A$
G	$::=$	$\top \mid \bot \mid \mathsf{Int} \mid \mathsf{Null} \mid A \rightarrow B \mid G_1 \vee G_2 \mid G_1 \wedge G_2 \mid \forall(\alpha * G).B$

Lowest Ordinary Subtypes (LOS) $\lvert A \rvert_{\Delta;\Gamma}$		
$\lvert\top\rvert_{\Delta;\Gamma}$	=	$\{\mathsf{Int}, \top \rightarrow \bot, \mathsf{Null}\} \cup \mathrm{dom}\ \Delta$
$\lvert\bot\rvert_{\Delta;\Gamma}$	=	$\{\}$
$\lvert\mathsf{Int}\rvert_{\Delta;\Gamma}$	=	$\{\mathsf{Int}\}$
$\lvert A \rightarrow B\rvert_{\Delta;\Gamma}$	=	$\{\top \rightarrow \bot\}$
$\lvert A \vee B\rvert_{\Delta;\Gamma}$	=	$\lvert A\rvert_{\Delta;\Gamma} \cup \lvert B\rvert_{\Delta;\Gamma}$
$\lvert A \wedge B\rvert_{\Delta;\Gamma}$	=	$\lvert A\rvert_{\Delta;\Gamma} \cap \lvert B\rvert_{\Delta;\Gamma}$
$\lvert\mathsf{Null}\rvert_{\Delta;\Gamma}$	=	$\{\mathsf{Null}\}$
$\lvert\mathsf{P}\rvert_{\Delta;\Gamma}$	=	$\{\mathsf{P}\} \cup \Delta(\mathsf{P})$
$\lvert\forall(\alpha * G).B\rvert_{\Delta;\Gamma}$	=	$\{\forall(\alpha * \bot).\bot\}$
$\lvert\alpha\rvert_{\Delta;\Gamma}$	=	$(\lvert\top\rvert_{\Delta;\Gamma})$ - $(\lvert G\rvert_{\Delta;\Gamma})$ where $\alpha * G \in \Gamma$

Figure 3.8: Syntax, additional typing, subtyping, and reduction rules for λ_u with polymorphism.

3.4.1 Disjoint Polymorphism

In this section we discuss an extension of λ_u with parametric polymorphism along with intersection and nominal types. The interesting aspect about this extension is the presence of disjointness constraints. For example, in λ_u with polymorphism a polymorphic disjoint switch such as: $\Gamma, \alpha * \mathsf{Int} \vdash \mathsf{switch}\ e\ \{(x : \mathsf{Int}) \rightarrow \mathsf{true}, (y : \alpha) \rightarrow \mathsf{false}\}$ is accepted. It is safe to use Int and α in alternative branches in a switch in this example. The disjointness constraint in the context ($\Gamma, \alpha * \mathsf{Int}$) on type variable α ensures that α must only be instantiated with types disjoint to Int. Thus an instantiation of α with Null or $A \rightarrow B$ is allowed. Whereas, an instantiation of α with Int is rejected by the type system.

Syntax. Figure 3.8 shows the extension in the syntax of λ_u with polymorphism. Types are extended with type variables α and disjoint quantifiers $\forall(\alpha * G).B$. The reader can think of this extension in the context of bounded quantification [Canning et al. 1989; Cardelli and Wegner 1985] where bounded quantifiers ($\forall(\alpha <: A).B$) are replaced by disjoint quantifiers ($\forall(\alpha * G).B$). Bounded quantification imposes a subtyping restriction on type variables,

whereas disjoint quantification imposes disjointness restriction on type variables. Disjoint quantification only allows the instantiation of disjoint types. For example, $\forall(\alpha <: \mathsf{Int} \vee \mathsf{Bool}).\alpha$ allows α to be instantiated only with subtypes of $\mathsf{Int} \vee \mathsf{Bool}$ and restricts all other types. Whereas, $\forall(\alpha * \mathsf{Int} \vee \mathsf{Bool}).\alpha$ restricts all the instantiations of α which share an ordinary subtype with $\mathsf{Int} \vee \mathsf{Bool}$. In other words, the permitted instantiations of α are the types disjoint to $\mathsf{Int} \vee \mathsf{Bool}$. Null is a valid instantiation in this case, while Int is not a valid instantiation.

Expressions are extended with type application $e\,A$ and type abstraction $\Lambda(\alpha{*}G).e$. A type abstraction is also a value. Additionally, context Γ now also contains type variables with their respective disjointness constraints. The disjointness constraint of type variables is restricted to ground types (G), which includes all the types except type variables. Ground types are shown at the top left of Figure 3.8.

Subtyping. Figure 3.9 shows subtyping relation in the formalization of polymorphic λ_u. Note that subtyping, typing, and reduction relations now have two contexts Δ and Γ. Subtyping is extended for the two newly added types. The subtyping rule for type variables is a special case of reflexivity (rule POLYS-TVAR)). Rule POLYS-ALL is interesting. It says that input and output types of two disjoint quantifiers are covariant in the subtype relation. This contrasts with calculi with bounded quantification and disjoint polymorphism [Alpuim et al. 2017], where the subtyping between the type bounds of the constraints is contravariant, and the subtyping between the types in the universal quantification body is covariant. Note that in the calculus that we formalized in Coq, we study parametric polymorphism without distributive subtyping rules. Subtyping for polymorphic λ_u is reflexive and transitive:

Lemma 3.23 (Subtyping reflexivity). $\Delta;\Gamma \vdash A <: A$

Lemma 3.24 (Subtyping transitivity). *If* $\Delta;\Gamma \vdash A <: B$ *and* $\Delta;\Gamma \vdash B <: C$ *then* $\Delta;\Gamma \vdash A <: C$

Typing and Operational Semantics. Typing is extended to assign the type to two newly added expressions and is shown in Figure 3.10. Rule PTYP-TAP is for type applications and rule PTYP-TABS is for type abstractions. Similarly, Figure 3.11 shows reduction rules for polymorphic λ_u. Rule POLYSTEP-TAPPL is standard reduction rule for type application. Rule POLYSTEP-TAPP replaces α with type B in expression e.

Disjointness. Disjointness has to be updated to accommodate type variables and disjoint quantifiers. The definition of algorithmic disjointness is roughly the same as discussed in

$\boxed{\Delta; \Gamma \vdash A <: B}$ *(Subtyping)*

$$\frac{}{\Delta; \Gamma \vdash A <: \top}\ \textsc{PolyS-Top} \qquad \frac{}{\Delta; \Gamma \vdash \mathsf{Int} <: \mathsf{Int}}\ \textsc{PolyS-Int} \qquad \frac{}{\Delta; \Gamma \vdash \bot <: A}\ \textsc{PolyS-Bot} \qquad \frac{}{\Delta; \Gamma \vdash \mathsf{Null} <: \mathsf{Null}}\ \textsc{PolyS-Unit}$$

$$\frac{\Delta; \Gamma \vdash B_1 <: A_1 \qquad \Delta; \Gamma \vdash A_2 <: B_2}{\Delta; \Gamma \vdash A_1 \to A_2 <: B_1 \to B_2}\ \textsc{PolyS-Arrow} \qquad \frac{\Delta; \Gamma \vdash A <: C \qquad \Delta; \Gamma \vdash B <: C}{\Delta; \Gamma \vdash A \vee B <: C}\ \textsc{PolyS-OrA}$$

$$\frac{\Delta; \Gamma \vdash A <: B}{\Delta; \Gamma \vdash A <: B \vee C}\ \textsc{PolyS-OrB} \qquad \frac{\Delta; \Gamma \vdash A <: C}{\Delta; \Gamma \vdash A <: B \vee C}\ \textsc{PolyS-OrC} \qquad \frac{\Delta; \Gamma \vdash A <: B \qquad \Delta; \Gamma \vdash A <: C}{\Delta; \Gamma \vdash A <: B \wedge C}\ \textsc{PolyS-AndA}$$

$$\frac{\Delta; \Gamma \vdash A <: C}{\Delta; \Gamma \vdash A \wedge B <: C}\ \textsc{PolyS-AndB} \qquad \frac{\Delta; \Gamma \vdash B <: C}{\Delta; \Gamma \vdash A \wedge B <: C}\ \textsc{PolyS-AndC} \qquad \frac{ok\ \Delta \qquad \Delta; \Gamma \vdash \alpha}{\Delta; \Gamma \vdash \alpha <: \alpha}\ \textsc{PolyS-TVar}$$

$$\frac{\Delta; \Gamma \vdash G_1 <: G_2 \qquad \Delta; \Gamma,\ \alpha * G_2 \vdash B_1 <: B_2}{\Delta; \Gamma \vdash \forall(\alpha * G_1).B_1 <: \forall(\alpha * G_2).B_2}\ \textsc{PolyS-All} \qquad \frac{ok\ \Delta \qquad \Delta; \Gamma \vdash \mathsf{P}}{\Delta; \Gamma \vdash \mathsf{P} <: \mathsf{P}}\ \textsc{PolyS-PRefl}$$

$$\frac{ok\ \Delta \qquad \Delta; \Gamma \vdash \mathsf{P}_1 \qquad \mathsf{P}_2 \in \Delta_{\mathsf{P}_1}}{\Delta; \Gamma \vdash \mathsf{P}_2 <: \mathsf{P}_1}\ \textsc{PolyS-PIn}$$

Figure 3.9: Subtyping for polymorphic λ_u.

Section 3.3, except that it takes an additional argument Γ. Context Γ is also an argument of LOS. LOS is extended to handle the additional cases of α and $\forall(\alpha * G).B$ and is shown at the top right of Figure 3.8. LOS returns $\forall(\alpha * \bot).\bot$ as the least ordinary subtype of $\forall(\alpha * G).B$. The type variable case is interesting. It returns the set difference of all ordinary subtypes and LOS of the disjointness constraint of type variable. Note that the disjointness constraint of type variables is restricted to ground types.

Definition 8 (Disjointness)**.** $\Delta; \Gamma \vdash \mathrm{A} * \mathrm{B} ::= |A|_{\Delta;\Gamma} \cap |B|_{\Delta;\Gamma} = \{\}$.

Type-safety and Determinism. The extension with disjoint polymorphism retains the properties of type-soundness and determinism. All the metatheory is formalized in Coq theorem prover.

Theorem 3.25 (Type Preservation)**.** *If* $\Delta; \Gamma \vdash e : A$ *and* $\Delta \vdash e \longrightarrow e'$ *then* $\Delta; \Gamma \vdash e' : A$.

Theorem 3.26 (Progress)**.** *If* $\Delta; \cdot \vdash e : A$ *then either e is a value; or e can take a step to* e'.

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

$$\frac{ok\ \Delta}{\Delta; \Gamma \vdash i : \mathsf{Int}}\ \text{PTYP-INT} \qquad \frac{ok\ \Delta}{\Delta; \Gamma \vdash \mathsf{null} : \mathsf{Null}}\ \text{PTYP-NULL} \qquad \frac{ok\ \Delta \quad \Delta \vdash A \quad x : A \in \Gamma}{\Delta; \Gamma \vdash x : A}\ \text{PTYP-VAR}$$

$$\frac{\begin{array}{c} ok\ \Delta \quad \Delta \vdash A \\ \Delta \vdash B \quad \Delta; \Gamma \vdash e_1 : A \to B \quad \Delta; \Gamma \vdash e_2 : A \end{array}}{\Delta; \Gamma \vdash e_1\, e_2 : B}\ \text{PTYP-APP} \qquad \frac{\Delta; \Gamma \vdash e : A \quad A <: B}{\Delta; \Gamma \vdash e : B}\ \text{PTYP-SUB}$$

$$\frac{ok\ \Delta \quad \Delta \vdash A \quad \Delta \vdash B \quad \Delta; \Gamma, x : A \vdash e : B}{\Delta; \Gamma \vdash \lambda x.e : A \to B}\ \text{PTYP-ABS} \qquad \frac{\Delta; \Gamma \vdash e : A \quad \Delta; \Gamma \vdash e : B}{\Delta; \Gamma \vdash e : A \wedge B}\ \text{PTYP-AND}$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash e : A \vee B \quad \Delta; \Gamma, x : A \vdash e_1 : C \\ \Delta; \Gamma, y : B \vdash e_2 : C \quad \Delta \vdash A *_s B \end{array}}{\Delta; \Gamma \vdash \mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} : C}\ \text{PTYP-SWITCH} \qquad \frac{ok\ \Delta \quad \Delta \vdash \mathsf{P}}{\Delta; \Gamma \vdash \mathsf{new}\ \mathsf{P} : \mathsf{P}}\ \text{PTYP-PRIM}$$

$$\frac{\Delta; \Gamma \vdash e : \forall(\alpha * G).C \quad \Delta; \Gamma \vdash G_1 * G}{\Delta; \Gamma \vdash e\, G_1 : C[\alpha \rightsquigarrow G_1]}\ \text{PTYP-TAP} \qquad \frac{\Delta; \Gamma, \alpha * G \vdash e : B}{\Delta; \Gamma \vdash \Lambda(\alpha * G).e : \forall(\alpha * G).B}\ \text{PTYP-TABS}$$

Figure 3.10: Typing for polymorphic λ_u.

Theorem 3.27 (Determinism). *If* $\Delta; \Gamma \vdash e : A$ *and* $\Delta \vdash e \longrightarrow e_1$ *and* $\Delta \vdash e \longrightarrow e_2$ *then* $e_1 = e_2$.

Progress and determinism does not require significant changes for this extension. Type preservation requires the preservation of disjointness after substitution and disjointness narrowing along with disjointness weakening. Disjointness substitution states that if two types are disjoint before type substitution, they must be disjoint after type substitution as stated in Lemma 3.28. The disjointness narrowing relates disjointness and subtyping. It states that it is safe to change the bounds of type variables from subtypes to supertypes as stated in Lemma 3.29.

Lemma 3.28 (Disjointness Substitution). *If* $\Delta; \Gamma, \alpha * G_1 \vdash B * C$ *and* $\Delta; \Gamma \vdash G_2 * G_1$ *then* $\Delta; \Gamma[\alpha \rightsquigarrow G_2] \vdash B[\alpha \rightsquigarrow G_2] * C[\alpha \rightsquigarrow G_2]$

Lemma 3.29 (Disjointness Narrowing). *If* $\Delta; \Gamma, \alpha * G_1 \vdash B * C$ *and* $\Delta; \Gamma \vdash G_1 <: G_2$ *then* $\Delta; \Gamma, \alpha * G_2 \vdash B * C$

$\boxed{\Delta; \Gamma \vdash e \longrightarrow e'}$ *(Operational Semantics)*

$$\frac{\Delta; \Gamma \vdash e_1 \longrightarrow e_1'}{\Delta; \Gamma \vdash e_1\, e_2 \longrightarrow e_1'\, e_2} \text{ POLYSTEP-APPL} \qquad \frac{\Delta; \Gamma \vdash e \longrightarrow e'}{\Delta; \Gamma \vdash v\, e \longrightarrow v\, e'} \text{ POLYSTEP-APPR} \qquad \frac{}{\Delta; \Gamma \vdash (\lambda x.e)\, v \longrightarrow e[x \rightsquigarrow v]} \text{ POLYSTEP-BETA}$$

$$\frac{\Delta; \Gamma \vdash e \longrightarrow e'}{\Delta; \Gamma \vdash e\, B \longrightarrow e'\, B} \text{ POLYSTEP-TAPPL} \qquad \frac{}{\Delta; \Gamma \vdash (\Lambda(\alpha * G).e)\, B \longrightarrow e[\alpha \rightsquigarrow B]} \text{ POLYSTEP-TAPP}$$

$$\frac{\Delta; \Gamma \vdash e \longrightarrow e'}{\Delta; \Gamma \vdash \mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow \mathsf{switch}\ e'\ \{(x : A) \to e_1, (y : B) \to e_2\}} \text{ POLYSTEP-SWITCH}$$

$$\frac{\textbf{value}\ v \qquad \Delta; \Gamma \vdash \lfloor v \rfloor <: A}{\Delta; \Gamma \vdash \mathsf{switch}\ v\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow e_1[x \rightsquigarrow v]} \text{ POLYSTEP-SWITCHL}$$

$$\frac{\textbf{value}\ v \qquad \Delta; \Gamma \vdash \lfloor v \rfloor <: B}{\Delta; \Gamma \vdash \mathsf{switch}\ v\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow e_2[y \rightsquigarrow v]} \text{ POLYSTEP-SWITCHR}$$

Figure 3.11: Operational Semantics for polymorphic λ_u.

3.4.2 A More General Subtyping Rule for Bottom Types

As discussed in Section 3.3.3, Ceylon includes the following subtyping rule:

$$\frac{A * B}{A \wedge B <: \bot} \text{ S-DISJ}$$

It is possible to support, and in fact generalize, such a rule in λ_u. The idea is to employ our definition of lowest ordinary subtypes, and add the following rule to λ_u with intersection types:

$$\frac{|A| = \{\}}{A <: B} \text{ S-LOS}$$

Rule S-LOS is an interesting addition in subtyping of λ_u. It says that if the LOS returns the empty set for some type A, then A is a subtype of all types. In other words, such type behaves like a *bottom-like* type. Such rule generalizes the rule S-DISJ employed in Ceylon, since when A is an intersection type of two disjoint types, we get the empty set. Moreover, adding rule S-LOS makes rule S-BOT redundant as well, since the LOS for the bottom type is also the empty

set. It is trivial to prove a lemma which says that $\bot$ is a subtype of all types. We drop rule S-BOT from the calculus discussed in Section 3.3 and prove Lemma 3.30 to show this property instead:

Lemma 3.30 (Bottom Type Least Subtype)**.** $\bot <: A$.

A similar lemma can be proved to show that disjoint types are bottom-like (as in rule S-DISJ), when rule S-LOS is added to subtyping:

Lemma 3.31 (Disjont Intersections are Bottom-Like)**.** *If* $A * B$ *then* $A \wedge B <: \bot$.

The use of rule S-LOS instead of rule S-DISJ also has the advantage that it does not create a mutual dependency between disjointness and subtyping. We can have the definition of disjointness, which depends only on subtyping and ordinary types, and the definition of subtyping, which depends on LOS but not on disjointness. Nevertheless, like rule S-DISJ, rule S-LOS would not be an appropriate rule in calculi with a merge operator for the reasons discussed in Section 3.3.

We have formalized and proved all the metatheory, including type soundness, transitivity of subtyping, soundness and completeness of disjointness and determinism for a variant of λ_u with intersection types, nominal types, standard subtyping and rule S-LOS in Coq.

3.4.3 Implementation of Disjoint Switches

Ceylon code runs on the Java Virtual Machine (JVM). A Ceylon program compiles to JVM bytecode. The final bytecode to which a Ceylon program is compiled to erase annotations for types not supported in the JVM. In particular, union types such as String $\vee$ Null are erased into `Object`. Disjoint switches are implemented by type casts. For each branch there is an *instanceof* to test the type of the branch and select a particular branch. An implementation of the λ_u calculus could also use a similar approach for compilation. In essence the use of union types and disjoint switches provides an elegant alternative to type-unsafe idioms, based on *instanceof* tests, that are currently widely used by Java programmers, while keeping comparable runtime performance.

4 Revisiting Disjointness

Polymorphic λ_u discussed in Section 3.4.1 has a ground type restriction on type variable bounds. Ground types constitute of all the types except type variables. This means that a type variable cannot be declared as a bound to another type variable. While this is a common approach in many polymorphic calculi [Dolan and Mycroft 2017], this approach limits the expressiveness of calculus. For example, in our setting, two type variables cannot be declared disjoint in the presence of ground type restriction. This restrains us from writing the following program:

```
Bool isFirstMatch [X * Y] (x : X | Y) = switch (x)
                                          (x:X) → true
                                          (y:Y) → false
```

Since the bound of type variable X is another type variable Y, therefore, this program will not type-check in the presence of ground type restriction on type variable bound. Any type except the type variable can be a bound of a type variable. In contrast, the following program will type-check:

```
Bool isInteger [X * Int] (x : X | Int) = switch (x)
                                          (x:Int)   → true
                                          (y:X)     → false
```

Notice that the bound of type variable X in the program above is a base type i.e. Int. While this approach with ground type restriction is useful in many scenarios, it restrains us from writing some valid programs. In this chapter we study a variant of λ_u with disjoint polymorphism without a ground type restriction on type variable bounds. This makes the current calculus more expressive than the one discussed in Section 3.4 and *isFirstMatch* type-checks in this calculus.

We develop a novel disjointness algorithm for intersection and union types by exploiting union ordinary and union splittable types [Huang and Oliveira 2021]. We study two variants of λ_u with the newly developed disjointness, one without polymorphism and another with polymorphism. The first calculus establishes a connection with the calculi without polymorphism. We show that the disjointness in Section 3.3 is sound and complete with respect to the disjointness in Section 4.1. The second calculus revisits disjoint polymorphism in Sec-

$$
\begin{array}{rcl}
A, B, C & ::= & \top \mid \bot \mid \mathsf{Int} \mid A \to B \mid A \vee B \mid A \wedge B \mid \mathsf{Null} \\
e & ::= & x \mid i \mid \lambda x.e \mid e_1\, e_2 \mid \mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} \mid \mathsf{null} \\
v & ::= & i \mid \lambda x.e \mid \mathsf{null} \\
\Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

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

$$
\frac{}{A <: \top}\ \textsc{s-top} \qquad \frac{}{\mathsf{Int} <: \mathsf{Int}}\ \textsc{s-int} \qquad \frac{}{\bot <: A}\ \textsc{s-bot} \qquad \frac{}{\mathsf{Null} <: \mathsf{Null}}\ \textsc{s-null} \qquad \frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2}\ \textsc{s-arrow}
$$

$$
\frac{A <: C \qquad B <: C}{A \vee B <: C}\ \textsc{s-orA} \qquad \frac{A <: B}{A <: B \vee C}\ \textsc{s-orB} \qquad \frac{A <: C}{A <: B \vee C}\ \textsc{s-orC} \qquad \frac{A <: B \qquad A <: C}{A <: B \wedge C}\ \textsc{s-andA}
$$

$$
\frac{A <: C}{A \wedge B <: C}\ \textsc{s-andB} \qquad \frac{B <: C}{A \wedge B <: C}\ \textsc{s-andC}
$$

Figure 4.1: Syntax and subtyping for λ_u with intersection types.

tion 4.2 and proposes a revised disjointness algorithm without ground types. Appendix A presents another variant of the disjointness algorithm based on Common Ordinary Subtypes (COST).

4.1 Disjointness with Intersection and Union Types

Recall that our first disjointness algorithm introduced in Section 3.2 did not work when we add intersection types. In Section 3.3 we come up with another disjointness algorithm based on Least Ordinary Subtypes (LOS) that accounts for intersection and union types. LOS is a function that computes a set of least ordinary subtypes of the input type. The disjointness algorithm discussed in Section 3.3 states that two types are disjoint if set intersection of LOS of two types is an empty set.

In this section we discuss another variant of the disjointness algorithm which is sound and complete with respect to the standard disjointness specifications discussed earlier in this thesis. We also explain the reason where naive disjointness algorithm fails with intersection and union types in detail in this section.

Syntax and subtyping. Syntax and subtyping for λ_u with intersection types is shown in Figure 4.1. Types, expressions, values and typing context (Γ) stay the same as in Section 3.3.

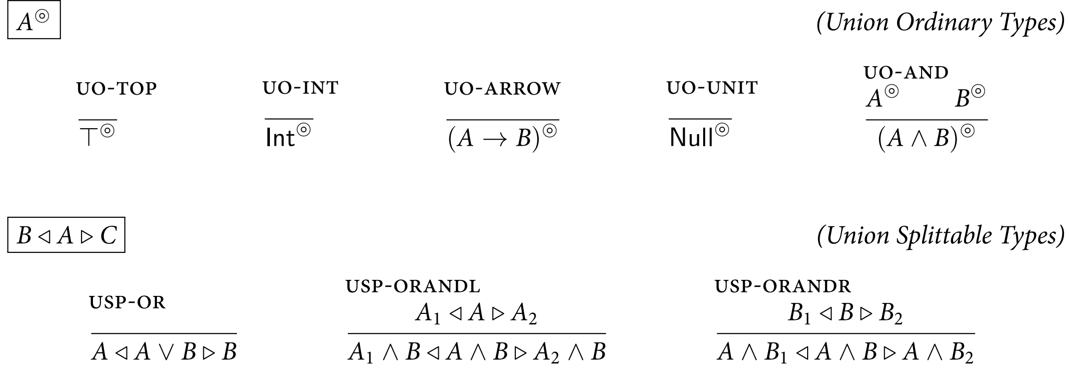


Figure 4.2: Syntax, union ordinary, and union splittable types.

We use conventional subtyping and drop distributive subtyping rules for function, intersection, and union types to emphasize the pivotal concept of disjointness for λ_u.

4.1.1 Disjointness, Union Ordinary, and Union Splittable Types

Union ordinary and union splittable types. Union ordinary and union splittable types play an essential role in the formulation of the novel disjointness algorithm. These types are shown in Figure 4.2. $\top$, Int, $A \to B$ and Null are union ordinary types as shown by rules uo-top, uo-int, uo-arrow, and uo-unit respectively. An intersection type is union ordinary only if both of its parts are union ordinary types as shown in rule uo-and. For example, $\mathsf{Int} \wedge \top$ is a union ordinary type. Whereas, $(\mathsf{Int} \vee A \to B) \wedge \top$ is not union ordinary because left part of the intersection is not union ordinary i.e $\mathsf{Int} \vee A \to B$.

Union types are never union ordinary types. On the contrary, union types are union splittable types by rule usp-or. Intersection types are union splittable if either of the component of the intersection is union splittable by rules usp-andl and usp-andr. We illustrate union splittable types with the help of following examples:

- $\mathsf{Int} \vee \mathsf{Bool}$: $\mathsf{Int} \vee \mathsf{Bool}$ is trivially union splittable into Int and Bool by rule usp-or.

$$\frac{}{\mathsf{Int} \lhd \mathsf{Int} \vee \mathsf{Bool} \rhd \mathsf{Bool}}\ \text{USP-OR}$$

- $(\mathsf{Int} \vee \mathsf{Bool}) \wedge \mathsf{String}$: $(\mathsf{Int} \vee \mathsf{Bool}) \wedge \mathsf{String}$ is splittable into $\mathsf{Int} \wedge \mathsf{String}$ and $\mathsf{Bool} \wedge \mathsf{String}$ by rule usp-orandl and rule usp-or.

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

$$\frac{}{\bot *_a A}\ \text{AD-BTML} \qquad \frac{}{A *_a \bot}\ \text{AD-BTMR} \qquad \frac{}{\mathsf{Int} *_a A \to B}\ \text{AD-INTL} \qquad \frac{}{A \to B *_a \mathsf{Int}}\ \text{AD-INTR} \qquad \frac{}{\mathsf{Null} *_a \mathsf{Int}}\ \text{AD-NULL-INTL}$$

$$\frac{}{\mathsf{Int} *_a \mathsf{Null}}\ \text{AD-NULL-INTR} \qquad \frac{}{\mathsf{Null} *_a A \to B}\ \text{AD-NULL-FUNL} \qquad \frac{}{A \to B *_a \mathsf{Null}}\ \text{AD-NULL-FUNR}$$

$$\frac{A_1 \lhd A \rhd A_2 \qquad A_1 *_a B \qquad A_2 *_a B}{A *_a B}\ \text{AD-ORLL} \qquad \frac{B_1 \lhd B \rhd B_2 \qquad A *_a B_1 \qquad A *_a B_2}{A *_a B}\ \text{AD-ORRR}$$

$$\frac{A_1 *_a B \qquad B^{\circledcirc}}{(A_1 \wedge A_2) *_a B}\ \text{AD-ANDLL} \qquad \frac{A_2 *_a B \qquad B^{\circledcirc}}{(A_1 \wedge A_2) *_a B}\ \text{AD-ANDLSS} \qquad \frac{A *_a B_1 \qquad A^{\circledcirc}}{A *_a (B_1 \wedge B_2)}\ \text{AD-ANDRR} \qquad \frac{A *_a B_2 \qquad A^{\circledcirc}}{A *_a (B_1 \wedge B_2)}\ \text{AD-ANDRSS}$$

$$\frac{A *_a B}{(A \wedge B) *_a C}\ \text{AD-EMPTYL} \qquad \frac{B *_a C}{A *_a (B \wedge C)}\ \text{AD-EMPTYR}$$

Figure 4.3: Disjointness based on splittable types for λ_u.

$$\dfrac{\dfrac{}{\mathsf{Int} \lhd \mathsf{Int} \vee \mathsf{Bool} \rhd \mathsf{Bool}}\ \text{USP-OR}}{\mathsf{Int} \wedge \mathsf{String} \lhd (\mathsf{Int} \vee \mathsf{Bool}) \wedge \mathsf{String} \rhd \mathsf{Bool} \wedge \mathsf{String}}\ \text{USP-ORANDL}$$

- $\mathsf{String} \wedge (\mathsf{Int} \vee \mathsf{Bool})$: $\mathsf{String} \wedge (\mathsf{Int} \vee \mathsf{Bool})$ is splittable into $\mathsf{String} \wedge \mathsf{Int}$ and $\mathsf{String} \wedge \mathsf{Bool}$ by rule USP-ORANDR and rule USP-OR.

$$\dfrac{\dfrac{}{\mathsf{Int} \lhd \mathsf{Int} \vee \mathsf{Bool} \rhd \mathsf{Bool}}\ \text{USP-OR}}{\mathsf{String} \wedge \mathsf{Int} \lhd \mathsf{String} \wedge (\mathsf{Int} \vee \mathsf{Bool}) \rhd \mathsf{String} \wedge \mathsf{Bool}}\ \text{USP-ORANDR}$$

Note that a type is either union ordinary or union splittable (Lemma 4.1):

Lemma 4.1 (Exclusivity of union ordinary and union splittable types). *$\forall A$, A is either union ordinary or union splittable and never both.*

Algorithmic Disjointness. The algorithmic disjointness based on union ordinary and union splittable types is shown in Figure 4.3. Rules AD-BTML, AD-BTMR, AD-INTL, AD-INTR,

AD-NULL-INTL, AD-NULL-INTR, AD-NULL-FUNL, and AD-NULL-FUNR are trivial disjointness axioms. The novelty of the disjointness algorithm lies in the disjointness rules for intersection and union types.

Rule AD-ORLL states that if A is union splittable into A_1 and A_2 then A is disjoint to B only if A_1 and A_2 are disjoint to B. Rule AD-ORRR is symmetric to rule AD-ORLL. Rules AD-ANDLL and AD-ANDLSS state that an intersection type $A_1 \wedge A_2$ is disjoint to another type B when B is union ordinary and either A_1 or A_2 is disjoint to B. Rules AD-ANDRR and AD-ANDRSS are symmetric to rules AD-ANDLL and AD-ANDLSS. Rules AD-EMPTYL and AD-EMPTYR are interesting rules. They state that an intersection of two disjoint types is disjoint with any other type. The intersection of two disjoint types forms an empty type or a bottom-like type, which is disjoint with any other type. The following example illustrates our novel disjointness algorithm:

- $(\mathsf{Int} \vee \mathsf{Bool}) *_a \mathsf{String}$: Int ∨ Bool is disjoint to String by rule AD-ORLL.

$$\dfrac{\dfrac{}{\mathsf{Int} \lhd \mathsf{Int} \vee \mathsf{Bool} \rhd \mathsf{Bool}}\ \text{USP-OR} \qquad \dfrac{}{\mathsf{Int} *_a \mathsf{String}} \qquad \dfrac{}{\mathsf{Bool} *_a \mathsf{String}}}{(\mathsf{Int} \vee \mathsf{Bool}) *_a \mathsf{String}}\ \text{AD-ORLL}$$

4.1.2 Essence of Union Ordinary Types

Union ordinary restriction. Note that the union ordinary premise in rules AD-ANDLL, AD-ANDLSS, AD-ANDRR, and AD-ANDRSS is optional. This premise only makes the rules less overlapping. It allows the application of rules AD-ANDLL, AD-ANDLSS, AD-ANDRR, and AD-ANDRSS only if one type is an intersection type and the other type is a union ordinary type. When the other type is not union ordinary type, the disjointness algorithm falls to the union rules. The algorithm then splits the other type until it becomes union ordinary and then applies either of the rules AD-ANDLL, AD-ANDLSS, AD-ANDRR, and AD-ANDRSS.

Next we explain the naive disjointness algorithm without union ordinary and union splittable types and then emphasize the significance of union ordinary and union splittable types.

Disjointness **without** union ordinary and union splittable types. The naive disjointness algorithm without union ordinary and union splittable types is shown in Figure 4.4. The rules AD-BTML, AD-BTMR, AD-INTL, AD-INTR, AD-NULL-INTL, AD-NULL-INTR, AD-NULL-FUNL, AD-NULL-FUNR, AD-EMPTYL, and AD-EMPTYR are straightforward and already explained. The rules for the intersection types (AD-ANDLA, AD-ANDLB, AD-ANDRA, and AD-ANDRB) state

$\boxed{A *_a B}$ *(Disjointness without union ordinary and union splittable types)*

$$\text{AD-BTML}\ \frac{}{\bot *_a A} \qquad \text{AD-BTMR}\ \frac{}{A *_a \bot} \qquad \text{AD-INTL}\ \frac{}{\mathsf{Int} *_a A \to B} \qquad \text{AD-INTR}\ \frac{}{A \to B *_a \mathsf{Int}} \qquad \text{AD-NULL-INTL}\ \frac{}{\mathsf{Null} *_a \mathsf{Int}}$$

$$\text{AD-NULL-INTR}\ \frac{}{\mathsf{Int} *_a \mathsf{Null}} \qquad \text{AD-NULL-FUNL}\ \frac{}{\mathsf{Null} *_a A \to B} \qquad \text{AD-NULL-FUNR}\ \frac{}{A \to B *_a \mathsf{Null}} \qquad \text{AD-ORL}\ \frac{A *_a C \qquad B *_a C}{A \vee B *_a C}$$

$$\text{AD-ORR}\ \frac{A *_a B \qquad A *_a C}{A *_a B \vee C} \qquad \text{AD-ANDLA}\ \frac{A *_a C}{A \wedge B *_a C} \qquad \text{AD-ANDLB}\ \frac{B *_a C}{A \wedge B *_a C} \qquad \text{AD-ANDRA}\ \frac{A *_a B}{A *_a B \wedge C} \qquad \text{AD-ANDRB}\ \frac{A *_a C}{A *_a B \wedge C}$$

$$\text{AD-EMPTYL}\ \frac{A *_a B}{(A \wedge B) *_a C} \qquad \text{AD-EMPTYR}\ \frac{B *_a C}{A *_a (B \wedge C)}$$

Figure 4.4: Disjointness **without** union ordinary and union splittable types for λ_u.

that an intersection type $A \wedge B$ is disjoint to another type C if either A or B is disjoint to C. Importantly, the rules for the intersection types no longer carry a premise with union ordinary restriction as in Figure 4.3. Similarly, rules for the union types (AD-ORL and AD-ORR) do not depend on union splittable types. The rules AD-ORL and AD-ORR simply state that a union type $A \vee B$ is disjoint to another type C if A disjoint to C and B disjoint to C holds.

Significance of union ordinary types. Recall that the union ordinary types play an essential role to make the rules for the intersection types in Figure 4.3 more algorithmic. We illustrate this with the help of following example. Note that we apply the disjointness rules from Figure 4.4 in the following derivation.

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Int} *_a \mathsf{String}} \quad \dfrac{}{\mathsf{Bool} *_a \mathsf{String}}}{(\mathsf{Int} \vee \mathsf{Bool}) *_a \mathsf{String}}\ \text{AD-ORL} \qquad \dfrac{\dfrac{}{\mathsf{Int} *_a \mathsf{Int}}\ ??? \quad \dfrac{}{\mathsf{Bool} *_a \mathsf{Int}}}{(\mathsf{Int} \vee \mathsf{Bool}) *_a \mathsf{Int}}\ \text{AD-ORL}}{\dfrac{(\mathsf{Int} \vee \mathsf{Bool}) *_a (\mathsf{String} \vee \mathsf{Int})}{(\mathsf{Int} \vee \mathsf{Bool}) \wedge (\mathsf{Bool} \vee \mathsf{String}) *_a (\mathsf{String} \vee \mathsf{Int})}\ \text{AD-ANDLA}}\ \text{AD-ORR}$$

$(\mathsf{Int} \vee \mathsf{Bool}) \wedge (\mathsf{Bool} \vee \mathsf{String})$ and $(\mathsf{String} \vee \mathsf{Int})$ are disjoint types and the algorithm in Figure 4.4 may backtrack to classify them as disjoint types without union ordinary restriction. In the derivation above we apply intersection rules first and then union rules. The derivation

proceeds by applying the rule AD-ANDLA which generates a sub-problem of $(\mathsf{Int} \vee \mathsf{Bool}) *_a (\mathsf{String} \vee \mathsf{Int})$. It then applies rule AD-ORR which further generates two sub-problems $(\mathsf{Int} \vee \mathsf{Bool}) *_a \mathsf{String}$ and $(\mathsf{Int} \vee \mathsf{Bool}) *_a \mathsf{Int}$.

1. Applying rule AD-ORL, $(\mathsf{Int} \vee \mathsf{Bool}) *_a \mathsf{String}$ can be solved since $\mathsf{Int} *_a \mathsf{String}$ and $\mathsf{Bool} *_a \mathsf{String}$ holds.

2. $(\mathsf{Int} \vee \mathsf{Bool}) *_a \mathsf{Int}$ fails to derive. The only option that we have left at this stage is to apply rule AD-ORL which requires $\mathsf{Int} *_a \mathsf{Int}$ and $\mathsf{Bool} *_a \mathsf{Int}$. The highlighted part in the derivation fails to hold i.e. $\mathsf{Int} *_a \mathsf{Int}$ does not hold. Therefore the current derivation fails.

This example also fails to derive if we start by applying rule AD-ANDLB which finally will result in $\mathsf{String} *_a \mathsf{String}$. The algorithm will keep on backtracking with the application of intersection rules at the very start. A solution to the successful derivation in current example is to apply union rules first and then intersection rules. We show a successful derivation below and then explain how union ordinary restriction reduces backtracking and makes disjointness rules less overlapping. Note that we use initials as place-holders instead of full type names for the sake of space. In particular, I stands for Int, B for Bool, C for Char and S for String.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{I *_a S} \quad \dfrac{}{B *_a S}}{(I \vee B) *_a S}\ \text{AD-ORL}}{(I \vee B) \wedge (B \vee S) *_a S}\ \text{AD-ANDLA} \qquad \dfrac{\dfrac{\dfrac{}{B *_a I} \quad \dfrac{}{S *_a I}}{(B \vee S) *_a I}\ \text{AD-ORL}}{(I \vee B) \wedge (B \vee S) *_a I}\ \text{AD-ANDLB}}{(I \vee B) \wedge (B \vee S) *_a (S \vee I)}\ \text{AD-ORR}$$

Notice that we start by applying rule AD-ORR in the derivation above. The key at this point is to apply union rules first and then intersection rules. This reduces the backtracking of the disjointness algorithm. Therefore, we enforce an optional union ordinary restriction in the intersection rules (AD-ANDLL, AD-ANDLSS, AD-ANDRR, and AD-ANDRSS) in Figure 4.3 in contrast to the naive disjointness rules in Figure 4.4. The union ordinary premise restricts the application of the intersection rules unless the union rules have been applied.

4.1.3 Essence of Union Splittable Types

Another issue with the naive disjointness algorithm shown in Figure 4.4 is that it is incomplete. We show the incompleteness with the following example:

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{I * S} \quad \dfrac{\dfrac{}{I * C} \quad \dfrac{}{I * I}\ ???}{I * (C \vee I)}\ \textsc{ad-orr}}{I * (S \vee C \vee I)}\ \textsc{ad-orr} \qquad \dfrac{\dfrac{}{S * S}\ ???}{\vdots \atop (B \vee S) * (S \vee C \vee I)}}{(I \vee B \vee S) * (S \vee C \vee I)}\ \textsc{ad-orl}}{(I \vee B \vee S) * (S \vee C \vee I) \wedge (C \vee I \vee B)}\ \textsc{ad-andra}}{(I \vee B \vee S) \wedge (B \vee S \vee C) * (S \vee C \vee I) \wedge (C \vee I \vee B)}\ \textsc{ad-andla}
$$

(Int ∨ Bool ∨ String) ∧ (Bool ∨ String ∨ Char) and (String ∨ Char ∨ Int) ∧ (Char ∨ Int ∨ Bool) are clearly disjoint types but the algorithm in Figure 4.4 fails to classify them as disjoint types without union splittable types. It does not matter whether we break the left intersection or right intersection first, we cannot make these two types disjoint. Importantly the two types as a whole are disjoint. But if we drop any component from either of the intersection, the smaller types are no longer disjoint as shown in the derivation above.

Rule AD-ANDLA drops a part of intersection from the left type resulting in a sub-problem (Int ∨ Bool ∨ String) ∗ (String ∨ Char ∨ Int) ∧ (Char ∨ Int ∨ Bool) which does not hold because Int is a common ordinary subtype of (Int ∨ Bool ∨ String) and (String ∨ Char ∨ Int) ∧ (Char ∨ Int ∨ Bool). The derivation finally results in the highlighted sub-problems i.e $\mathsf{Int} *_a \mathsf{Int}$ and $\mathsf{String} *_a \mathsf{String}$. Int is not disjoint to Int and so is not String to String. Therefore the naive disjointness algorithm shown in Figure 4.4 fails and is not complete.

Union splittable types to the rescue. Union splittable types come to the rescue in such cases and solve the incompleteness problem of the disjointness algorithm. Note that union ordinary types are optional because union ordinary types just make the rules less overlapping. The disjointness algorithm stays sound and complete without union ordinary types. But union splittable types are essential. The disjointness algorithm will not be complete without union splittable types. The algorithm in Figure 4.3 is sound and complete, whereas the algorithm in Figure 4.4 is sound but incomplete.

Illustration of Completeness with union splittable types. We show how the union splittable types solve the incompleteness problem of the disjointness algorithm in Figure 4.3 using the same example i.e (Int ∨ Bool ∨ String) ∧ (Bool ∨ String ∨ Char) ∗ (String ∨ Char ∨ Int) ∧ (Char ∨ Int ∨ Bool). Since both of the types:

1. A1 = (Int ∨ Bool ∨ String) ∧ (Bool ∨ String ∨ Char)

2. A2 = (String ∨ Char ∨ Int) ∧ (Char ∨ Int ∨ Bool)

are not union ordinary. Therefore we cannot apply intersection rules (AD-ANDLL, AD-ANDLSS, AD-ANDRR, and AD-ANDRSS) from Figure 4.3 at the very start of the derivation. We can apply either of the rule AD-ORLL or rule AD-ORRR since both of the types are union splittable.

$$\dfrac{\dfrac{\vdots}{S \wedge (C \vee I \vee B) \lhd (S \vee C \vee I) \wedge (C \vee I \vee B) \rhd (C \vee I) \wedge (C \vee I \vee B)}\ \text{USP-ORANDL}}{(I \vee B \vee S) \wedge (B \vee S \vee C) * (S \vee C \vee I) \wedge (C \vee I \vee B)}\ \text{AD-ORRR}$$

Splitting goes as follows:

$$\dfrac{\text{USP-OR}\ \dfrac{\vdots}{S \lhd (S \vee C \vee I) \rhd C \vee I} \qquad \vdots}{S \wedge (C \vee I \vee B) \lhd (S \vee C \vee I) \wedge (C \vee I \vee B) \rhd (C \vee I) \wedge (C \vee I \vee B)}\ \text{USP-ORANDL}$$

We split the right side in the disjointness derivation above. The choice of the right or the left type does not affect the outcome of the disjointness algorithm. The union splitting algorithm will keep on splitting the right type unless all the smaller types become union ordinary types. When the splitting algorithm concludes splitting of (String∨Char∨Int)∧(Char∨Int∨Bool), the final result will be a list consisting of the following union ordinary types:

- {String∧Char, String∧Int, String∧Bool, Char∧Char, Char∧Int, Char∧Bool, Int∧Char, Int ∧ Int, Int ∧ Bool}.

The whole disjointness checking problem breaks into the following sub-problems:

1. (Int ∨ Bool ∨ String) ∧ (Bool ∨ String ∨ Char) ∗ String ∧ Char
2. (Int ∨ Bool ∨ String) ∧ (Bool ∨ String ∨ Char) ∗ String ∧ Int
3. (Int ∨ Bool ∨ String) ∧ (Bool ∨ String ∨ Char) ∗ String ∧ Bool
4. (Int ∨ Bool ∨ String) ∧ (Bool ∨ String ∨ Char) ∗ Char ∧ Char

5. $(\mathsf{Int} \vee \mathsf{Bool} \vee \mathsf{String}) \wedge (\mathsf{Bool} \vee \mathsf{String} \vee \mathsf{Char}) * \mathsf{Char} \wedge \mathsf{Int}$

6. $(\mathsf{Int} \vee \mathsf{Bool} \vee \mathsf{String}) \wedge (\mathsf{Bool} \vee \mathsf{String} \vee \mathsf{Char}) * \mathsf{Char} \wedge \mathsf{Bool}$

7. $(\mathsf{Int} \vee \mathsf{Bool} \vee \mathsf{String}) \wedge (\mathsf{Bool} \vee \mathsf{String} \vee \mathsf{Char}) * \mathsf{Int} \wedge \mathsf{Char}$

8. $(\mathsf{Int} \vee \mathsf{Bool} \vee \mathsf{String}) \wedge (\mathsf{Bool} \vee \mathsf{String} \vee \mathsf{Char}) * \mathsf{Int} \wedge \mathsf{Int}$

9. $(\mathsf{Int} \vee \mathsf{Bool} \vee \mathsf{String}) \wedge (\mathsf{Bool} \vee \mathsf{String} \vee \mathsf{Char}) * \mathsf{Int} \wedge \mathsf{Bool}$

Notice that 7 out of 9 sub-problems are trivially solvable by rule EMPTYR except sub-problems 4 and 8. This is because the right type in sub-problems 1, 2, 3, 5, 6, 7 and 9 is an intersection of two disjoint types. For example, the derivation for sub-problem 1 is:

$$\dfrac{\dfrac{}{\mathsf{String} * \mathsf{Char}}}{(\mathsf{Int} \vee \mathsf{Bool} \vee \mathsf{String}) \wedge (\mathsf{Bool} \vee \mathsf{String} \vee \mathsf{Char}) * \mathsf{String} \wedge \mathsf{Char}}\ \textsc{ad-emptyr}$$

Next, we show the derivation trees to solve sub-problem 4 and 8.

DERIVATION FOR (4) $(\mathsf{Int} \vee \mathsf{Bool} \vee \mathsf{String}) \wedge (\mathsf{Bool} \vee \mathsf{String} \vee \mathsf{Char}) * \mathsf{Char} \wedge \mathsf{Char}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Int} * \mathsf{Char}} \quad \dfrac{\dfrac{\dfrac{}{\mathsf{Bool} * \mathsf{Char}} \quad \dfrac{}{\mathsf{String} * \mathsf{Char}}}{\mathsf{Bool} \triangleleft \mathsf{Bool} \vee \mathsf{String} \triangleright \mathsf{String}}}{(\mathsf{Bool} \vee \mathsf{String}) * \mathsf{Char}}\ \textsc{ad-orll}}{\mathsf{Int} \triangleleft (\mathsf{Int} \vee \mathsf{Bool} \vee \mathsf{String}) \triangleright \mathsf{Bool} \vee \mathsf{String}}}{(\mathsf{Int} \vee \mathsf{Bool} \vee \mathsf{String}) * \mathsf{Char}}\ \textsc{ad-orll}}{(\mathsf{Int} \vee \mathsf{Bool} \vee \mathsf{String}) * \mathsf{Char} \wedge \mathsf{Char}}\ \textsc{ad-andrr}}{(\mathsf{Int} \vee \mathsf{Bool} \vee \mathsf{String}) \wedge (\mathsf{Bool} \vee \mathsf{String} \vee \mathsf{Char}) * \mathsf{Char} \wedge \mathsf{Char}}\ \textsc{ad-andll}$$

The derivation for the sub-problem 4 initiates by applying the rule AD-ANDLL which selects the left part of the intersection type and reduces the problem to $(\mathsf{Int} \vee \mathsf{Bool} \vee \mathsf{String}) * \mathsf{Char} \wedge \mathsf{Char}$. Rule AD-ANDRR further reduces the problem to $(\mathsf{Int} \vee \mathsf{Bool} \vee \mathsf{String}) * \mathsf{Char}$. The rule AD-ORLL finally generates the base cases for the derivation. All of the base cases trivially hold which concludes successful derivation for $(\mathsf{Int} \vee \mathsf{Bool} \vee \mathsf{String}) \wedge (\mathsf{Bool} \vee \mathsf{String} \vee \mathsf{Char}) * \mathsf{Char} \wedge \mathsf{Char}$.

Derivation for (8) $(\mathsf{Int} \vee \mathsf{Bool} \vee \mathsf{String}) \wedge (\mathsf{Bool} \vee \mathsf{String} \vee \mathsf{Char}) * \mathsf{Int} \wedge \mathsf{Int}$: The derivation for sub-problem 8 follows a similar path as of sub-problem 4. Complete derivation for sub-problem 8 is shown below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Bool} * \mathsf{Int}} \quad \dfrac{\dfrac{\dfrac{}{\mathsf{String} * \mathsf{Int}} \quad \dfrac{}{\mathsf{Char} * \mathsf{Int}}}{\mathsf{String} \lhd \mathsf{String} \vee \mathsf{Char} \rhd \mathsf{Char}}}{(\mathsf{String} \vee \mathsf{Char}) * \mathsf{Int}}\ \text{AD-ORL}}{\mathsf{Bool} \lhd (\mathsf{Bool} \vee \mathsf{String} \vee \mathsf{Char}) \rhd \mathsf{String} \vee \mathsf{Char}}}{(\mathsf{Bool} \vee \mathsf{String} \vee \mathsf{Char}) * \mathsf{Int}}\ \text{AD-ORL}}{(\mathsf{Bool} \vee \mathsf{String} \vee \mathsf{Char}) * \mathsf{Int} \wedge \mathsf{Int}}\ \text{AD-ANDRR}}{(\mathsf{Int} \vee \mathsf{Bool} \vee \mathsf{String}) \wedge (\mathsf{Bool} \vee \mathsf{String} \vee \mathsf{Char}) * \mathsf{Int} \wedge \mathsf{Int}}\ \text{AD-ANDLSS}$$

Since the newly formulated disjointness algorithm solved all of the sub-problems, therefore $(\mathsf{Int} \vee \mathsf{Bool} \vee \mathsf{String}) \wedge (\mathsf{Bool} \vee \mathsf{String} \vee \mathsf{Char}) * (\mathsf{String} \vee \mathsf{Char} \vee \mathsf{Int}) \wedge (\mathsf{Char} \vee \mathsf{Int} \vee \mathsf{Bool})$ holds. The successful derivation for this example shows that the novel disjointness algorithm with union ordinary and union splittable types shown in Figure 4.3 is able to compute the disjointness of the types on which naive algorithm shown in Figure 4.4 fails.

Soundness and completeness of disjointness. We prove that the novel disjointness algorithm is sound and complete with respect to the disjointness specifications. The disjointness specifications are shown again in Definition 9 for readability.

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

Definition 9 ($\wedge$-Disjointness). $A * B ::= \nexists\, C^\circ, C^\circ <: A$ and $C^\circ <: B$.

Lemma 4.2 (Soundness of disjointness algorithm). $\forall\, A\ B,\ A *_a B \to A * B$.

Lemma 4.3 (Completeness of disjointness algorithm). $\forall\, A\ B,\ A * B \to A *_a B$.

4.1.4 Metatheory with Union Ordinary and Union Splittable Types

Typing and operational semantics. Subtyping, typing and operational semantics essentially stay the same and are shown in Figure 4.5. This calculus preserves the standard properties of subtyping, type-safety and determinism as shown below:

Lemma 4.4 (Subtyping Reflexivity). $A <: A$

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

$$\frac{}{\Gamma \vdash i : \mathsf{Int}}\ \text{TYP-INT} \qquad \frac{}{\Gamma \vdash \mathsf{null} : \mathsf{Null}}\ \text{TYP-NULL} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \text{TYP-VAR} \qquad \frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B}\ \text{TYP-APP}$$

$$\frac{\Gamma \vdash e : A \qquad A <: B}{\Gamma \vdash e : B}\ \text{TYP-SUB} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B}\ \text{TYP-ABS} \qquad \frac{\Gamma \vdash e : A \qquad \Gamma \vdash e : B}{\Gamma \vdash e : A \wedge B}\ \text{TYP-AND}$$

$$\frac{\Gamma \vdash e : A \vee B \qquad \Gamma, x : A \vdash e_1 : C \qquad \Gamma, y : B \vdash e_2 : C \qquad A * B}{\Gamma \vdash \mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} : C}\ \text{TYP-SWITCH}$$

$\boxed{e \longrightarrow e'}$ *(Operational Semantics)*

$$\frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \text{STEP-APPL} \qquad \frac{e \longrightarrow e'}{v\, e \longrightarrow v\, e'}\ \text{STEP-APPR} \qquad \frac{}{(\lambda x.e)\, v \longrightarrow e[x \rightsquigarrow v]}\ \text{STEP-BETA}$$

$$\frac{e \longrightarrow e'}{\mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow \mathsf{switch}\ e'\ \{(x : A) \to e_1, (y : B) \to e_2\}}\ \text{STEP-SWITCH}$$

$$\frac{\lfloor v \rfloor <: A}{\mathsf{switch}\ v\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow e_1[x \rightsquigarrow v]}\ \text{STEP-SWITCHL}$$

$$\frac{\lfloor v \rfloor <: B}{\mathsf{switch}\ v\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow e_2[y \rightsquigarrow v]}\ \text{STEP-SWITCHR}$$

Figure 4.5: Typing and operational semantics for λ_u.

Lemma 4.5 (Subtyping Transitivity). *If $A <: B$ and $B <: C$ then $A <: C$*

Theorem 4.6 (Type Preservation). *If $\Gamma \vdash e : A$ and $e \longrightarrow e'$ then $\Gamma \vdash e' : A$.*

Theorem 4.7 (Progress). *If $\Gamma \vdash e : A$ then either e is a value; or e can take a step to e'.*

Theorem 4.8 (Determinism). *If $\Gamma \vdash e : A$ and $e \longrightarrow e_1$ and $e \longrightarrow e_2$ then $e_1 = e_2$.*

$$
\begin{array}{rcl}
A, B, C & ::= & \top \mid \bot \mid \mathsf{Int} \mid A \rightarrow B \mid \mathsf{Null} \mid A \vee B \mid A \wedge B \mid \mathsf{P} \mid \alpha \mid \forall(\alpha * A).B \\
e & ::= & x \mid i \mid \lambda x.e \mid e_1\, e_2 \mid \mathsf{switch}\ e\ \{(x : A) \rightarrow e_1, (y : B) \rightarrow e_2\} \mid \mathsf{null} \mid \\
 & & \mathsf{new}\ \mathsf{P} \mid e\, A \mid \Lambda(\alpha * A).e \\
v & ::= & i \mid \lambda x.e \mid \mathsf{null} \mid \mathsf{new}\ \mathsf{P} \mid \Lambda(\alpha * A).e \\
\Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, \alpha * A \\
\Delta & ::= & \cdot \mid \Delta, \mathsf{P} <: A
\end{array}
$$

$\boxed{A^{\circledcirc}}$ *(Union Ordinary Types)*

$$
\frac{}{\top^{\circledcirc}}\ \textsc{uo-top}
\qquad
\frac{}{\mathsf{Int}^{\circledcirc}}\ \textsc{uo-int}
\qquad
\frac{}{(A \rightarrow B)^{\circledcirc}}\ \textsc{uo-arrow}
\qquad
\frac{}{\mathsf{Null}^{\circledcirc}}\ \textsc{uo-unit}
\qquad
\frac{A^{\circledcirc} \quad B^{\circledcirc}}{(A \wedge B)^{\circledcirc}}\ \textsc{uo-and}
\qquad
\frac{}{\alpha^{\circledcirc}}\ \textsc{uo-tvar}
$$

$$
\frac{}{\forall(\alpha * A).B^{\circledcirc}}\ \textsc{uo-all}
\qquad
\frac{}{\mathsf{P}^{\circledcirc}}\ \textsc{uo-nom}
$$

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

$$
\frac{}{A \lhd A \vee B \rhd B}\ \textsc{usp-or}
\qquad
\frac{A_1 \lhd A \rhd A_2}{A_1 \wedge B \lhd A \wedge B \rhd A_2 \wedge B}\ \textsc{usp-orandl}
\qquad
\frac{B_1 \lhd B \rhd B_2}{A \wedge B_1 \lhd A \wedge B \rhd A \wedge B_2}\ \textsc{usp-orandr}
$$

Figure 4.6: Syntax, union ordinary, and union splittable types for polymorphic λ_u.

4.2 Redesigning Disjoint Polymorphism

In Section 4.1 we discuss a novel disjointness algorithm by exploiting union ordinary and union splittable types. We show that the disjointness algorithm is sound and complete with respect to the disjointness specifications. In this section we extend the calculus from Section 4.1 with disjoint polymorphism. Importantly, we show that the ground type restriction on type variable bounds is no longer needed with the novel disjointness algorithm.

Syntax, union ordinary, and union splittable types. The syntax for polymorphic λ_u is shown at the top in Figure 4.6. Types are extended with the nominal types P, type variables α, and disjoint quantifiers $\forall(\alpha * A).B$. The syntactic category of expressions now include a new expression ($\mathsf{new}\ \mathsf{P}$) to construct instances of type P. It also includes type applications $e\, A$ and type abstractions $\Lambda(\alpha * A).e$. Expressions $\mathsf{new}\ \mathsf{P}$ and $\Lambda(\alpha * A).e$ are also values. Typing context Γ also has entries for type variables $\Gamma, \alpha * A$. A new context Δ keeps a list and bounds of nominal types.

Union ordinary types are shown in the middle of Figure 4.6. Union ordinary types are extended with type variables (rule UO-TVAR), disjoint quantifiers (rule UO-ALL) and nominal types (rule UO-NOM). Union splittable types stay the same as in Figure 4.2 and are shown at the bottom of Figure 4.6.

4.2.1 Disjointness

The disjointness algorithm with polymorphism is shown in Figure 4.7. In addition to the prior axioms, we also add axioms for universal types and the nominal types. Universal types are disjoint to all the base types and so are the nominal types. Type variables are disjoint to all the subtypes of its bound as shown in rules ADP-VARR and ADP-VARL. For example in a context $[\Gamma,\ \alpha * \top]$, α is essentially disjoint to all the types because all the types are subtype of $\top$. In another context $[\Gamma,\ \alpha * \mathsf{Int} \vee \mathsf{Bool}]$, α is disjoint with all the subtypes of $\mathsf{Int} \vee \mathsf{Bool}$ including Int and Bool but is not disjoint to String.

Note that we scrap the optional union ordinary premise from rules ADP-ANDL, ADP-ANDLS, ADP-ANDR, and ADP-ANDRS. This simplifies the metatheory with the type variables. We also drop rules ADP-EMPTYL and ADP-EMPTYR from disjointness algorithm in Figure 4.7. Dropping rules ADP-EMPTYL and ADP-EMPTYR restricts writing some programs but all the practical programs still type-check. Generally, it does not allow writing empty intersection types in branches. For example, the following program will no longer type-check because of the empty type in the first branch i.e $\mathsf{Int} \wedge \mathsf{Bool}$.

```
Bool isInt (x : Int | Bool) = switch (x)
                                 (x:Int&Bool) → true
                                 (y:Int)      → true
                                 (z:Bool)     → false
```

Since we cannot construct a value of type $\mathsf{Int} \wedge \mathsf{Bool}$ in contemporary system, the first branch in the above program has no practical significance. Therefore not allowing such empty intersections does not affect the programs in practice.

Disjointness for nominal types. The disjointness rule for nominal types (rule ADP-NOM) is interesting. It states that two nominal types P_1 and P_2 are disjoint if the intersection of their subtypes is an empty set. Nominal subtypes ($\Delta(A)$) is a function that finds the subtypes of type A in Δ and returns a list. Note that nominal subtypes is a transitive closure. Nominal subtypes function is shown next:

$\boxed{\Delta; \Gamma \vdash A *_{ax} B}$ *(Disjointness Axioms)*

$$\frac{}{\Delta; \Gamma \vdash \bot *_{ax} A}\ \text{ADPA-BOT} \qquad \frac{}{\Delta; \Gamma \vdash \mathsf{Int} *_{ax} A \to B}\ \text{ADPA-INTARR} \qquad \frac{}{\Delta; \Gamma \vdash \mathsf{Int} *_{ax} \mathsf{Null}}\ \text{ADPA-INTNULL}$$

$$\frac{}{\Delta; \Gamma \vdash \mathsf{Int} *_{ax} \forall(\alpha * A).B}\ \text{ADPA-INTALL} \qquad \frac{}{\Delta; \Gamma \vdash \mathsf{Null} *_{ax} A \to B}\ \text{ADPA-NULLARR} \qquad \frac{}{\Delta; \Gamma \vdash \mathsf{Null} *_{ax} \forall(\alpha * A).B}\ \text{ADPA-NULLALL}$$

$$\frac{}{\Delta; \Gamma \vdash C \to D *_{ax} \forall(\alpha * A).B}\ \text{ADPA-ARRALL} \qquad \frac{}{\Delta; \Gamma \vdash \mathsf{P} *_{ax} \mathsf{Int}}\ \text{ADPA-PINT} \qquad \frac{}{\Delta; \Gamma \vdash \mathsf{P} *_{ax} A \to B}\ \text{ADPA-PARR}$$

$$\frac{}{\Delta; \Gamma \vdash \mathsf{P} *_{ax} \mathsf{Null}}\ \text{ADPA-PNULL} \qquad \frac{}{\Delta; \Gamma \vdash \mathsf{P} *_{ax} \forall(\alpha * A).B}\ \text{ADPA-PALL} \qquad \frac{\Delta; \Gamma \vdash A *_{ax} B}{\Delta; \Gamma \vdash B *_{ax} A}\ \text{ADPA-SYM}$$

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

$$\frac{\alpha * A \in \Gamma \qquad \Delta; \Gamma \vdash B <: A}{\Delta; \Gamma \vdash B * \alpha}\ \text{ADP-VARR} \qquad \frac{A_1 \lhd A \rhd A_2 \qquad \Delta; \Gamma \vdash A_1 * B \qquad \Delta; \Gamma \vdash A_2 * B}{\Delta; \Gamma \vdash A * B}\ \text{ADP-ORL}$$

$$\frac{\alpha * A \in \Gamma \qquad \Delta; \Gamma \vdash B <: A}{\Delta; \Gamma \vdash \alpha * B}\ \text{ADP-VARL} \qquad \frac{B_1 \lhd B \rhd B_2 \qquad \Delta; \Gamma \vdash A * B_1 \qquad \Delta; \Gamma \vdash A * B_2}{\Delta; \Gamma \vdash A * B}\ \text{ADP-ORR}$$

$$\frac{\Delta; \Gamma \vdash A_1 * B}{\Delta; \Gamma \vdash (A_1 \wedge A_2) * B}\ \text{ADP-ANDL} \qquad \frac{\Delta; \Gamma \vdash A_2 * B}{\Delta; \Gamma \vdash (A_1 \wedge A_2) * B}\ \text{ADP-ANDLS} \qquad \frac{\Delta; \Gamma \vdash A * B_1}{\Delta; \Gamma \vdash A * (B_1 \wedge B_2)}\ \text{ADP-ANDR}$$

$$\frac{\Delta; \Gamma \vdash A * B_2}{\Delta; \Gamma \vdash A * (B_1 \wedge B_2)}\ \text{ADP-ANDRS} \qquad \frac{\mathsf{P}_1 :: \Delta(\mathsf{P}_1) \cap \mathsf{P}_2 :: \Delta(\mathsf{P}_2) = \{\}}{\Delta; \Gamma \vdash \mathsf{P}_1 * \mathsf{P}_2}\ \text{ADP-NOM} \qquad \frac{\Delta; \Gamma \vdash A *_{ax} B}{\Delta; \Gamma \vdash A * B}\ \text{ADP-AXIOM}$$

Figure 4.7: Disjointness with union splittable types for polymorphic λ_u.

Nominal Subtypes $\boxed{\Delta(A)}$

$$\begin{array}{lcl} \cdot(A) & = & \{\} \\ (\Delta', \mathsf{P} \leq B)(A) & = & \begin{cases} \{\mathsf{P}\} \cup \Delta'(A) \ \textbf{if}\ \mathsf{P} \leq A \in \Delta \\ \Delta'(A)\ \textbf{otherwise} \end{cases} \end{array}$$

For example, in a context Δ = {Person <: $\top$, Student <: Person, GradStudent <: Student, Robot <: $\top$, OptimumPrime <: Robot}:

- Person is disjoint to Robot as per rule ADP-NOM, because the set intersection of the subtypes of Person and Robot is empty i.e {Person, Student, GradStudent} ∩ {Robot, OptimumPrime} = {}.
- Whereas, Person is not disjoint to GradStudent, because the set intersection of the subtypes of Person and GradStudent is not empty i.e {Person, Student, GradStudent} ∩ {GradStudent} = {GradStudent}.

CONTRAVARIANCE OF DISJOINTNESS. Contravariance of disjointness (Lemma 4.9) states that if two types A and B are disjoint, then the subtypes of A are also disjoint with B. In general subtypes of disjoint types are disjoint as well. For example if $A \to B$ is disjoint to $\mathsf{Int} \vee \mathsf{Null}$, then $A \to B$ is disjoint to both Int and Null among other subtypes of $\mathsf{Int} \vee \mathsf{Bool}$. Similarly if a type A is disjoint to $\top$, then A is disjoint with all the types because all the types are subtypes of $\top$.

Lemma 4.9 (Contravariance of disjointness). *If $\Delta; \Gamma \vdash A * B$ and $\Delta; \Gamma \vdash C <: A$ then $\Delta; \Gamma \vdash C * B$.*

EXPRESSIVENESS OF DISJOINTNESS. The novel disjointness algorithm allows writing the programs that are not allowed in the polymorphic λ_u discussed in Section 3.4.1 due to the ground type restriction. We can write the programs by declaring type variables as bounds of other type variables:

```
Bool isFirstMatch [X * Y] (x : X | Y) = switch (x)
                                          (x:X) → true
                                          (y:Y) → false
```

4.2.2 SUBTYPING, TYPING, AND OPERATIONAL SEMANTICS

Subtyping, typing and operational semantics are altered to lift the ground type restriction on type variable bounds and are shown in Figures 4.8 to 4.10 respectively. These relations have already been explained in Section 3.4. We highlight the major changes next.

MODIFICATIONS IN METATHEORY. The subtyping changes are reflected by rule POLYS-ALLDISJ in Figure 4.8. Note that first premise does not have ground type restriction. A_1 and A_2 can be any types. Typing changes are shown in rules PTYP-TAPDISJ and PTYP-TABSDISJ in Figure 4.9. Similarly, changes for operational semantics are shown in rule POLYSTEP-TAPPDISJ in Figure 4.10. Importantly, we no longer use syntactic category of ground types and the bound of a type variable can be any other type.

$\boxed{\Delta; \Gamma \vdash A <: B}$ *(Subtyping)*

$$\frac{}{\Delta; \Gamma \vdash A <: \top}\ \textsc{PolyS-Top} \qquad \frac{}{\Delta; \Gamma \vdash \mathsf{Int} <: \mathsf{Int}}\ \textsc{PolyS-Int} \qquad \frac{}{\Delta; \Gamma \vdash \bot <: A}\ \textsc{PolyS-Bot} \qquad \frac{}{\Delta; \Gamma \vdash \mathsf{Null} <: \mathsf{Null}}\ \textsc{PolyS-Unit}$$

$$\frac{\Delta; \Gamma \vdash B_1 <: A_1 \qquad \Delta; \Gamma \vdash A_2 <: B_2}{\Delta; \Gamma \vdash A_1 \to A_2 <: B_1 \to B_2}\ \textsc{PolyS-Arrow} \qquad \frac{\Delta; \Gamma \vdash A <: C \qquad \Delta; \Gamma \vdash B <: C}{\Delta; \Gamma \vdash A \vee B <: C}\ \textsc{PolyS-OrA}$$

$$\frac{\Delta; \Gamma \vdash A <: B}{\Delta; \Gamma \vdash A <: B \vee C}\ \textsc{PolyS-OrB} \qquad \frac{\Delta; \Gamma \vdash A <: C}{\Delta; \Gamma \vdash A <: B \vee C}\ \textsc{PolyS-OrC} \qquad \frac{\Delta; \Gamma \vdash A <: B \qquad \Delta; \Gamma \vdash A <: C}{\Delta; \Gamma \vdash A <: B \wedge C}\ \textsc{PolyS-AndA}$$

$$\frac{\Delta; \Gamma \vdash A <: C}{\Delta; \Gamma \vdash A \wedge B <: C}\ \textsc{PolyS-AndB} \qquad \frac{\Delta; \Gamma \vdash B <: C}{\Delta; \Gamma \vdash A \wedge B <: C}\ \textsc{PolyS-AndC} \qquad \frac{ok\ \Delta \qquad \Delta; \Gamma \vdash \alpha}{\Delta; \Gamma \vdash \alpha <: \alpha}\ \textsc{PolyS-TVar}$$

$$\frac{\Delta; \Gamma \vdash A_1 <: A_2 \qquad \Delta; \Gamma,\ \alpha * A_2 \vdash B_1 <: B_2}{\Delta; \Gamma \vdash \forall(\alpha * A_1).B_1 <: \forall(\alpha * A_2).B_2}\ \textsc{PolyS-AllDisj} \qquad \frac{ok\ \Delta \qquad \Delta; \Gamma \vdash \mathsf{P}}{\Delta; \Gamma \vdash \mathsf{P} <: \mathsf{P}}\ \textsc{PolyS-PRefl}$$

$$\frac{ok\ \Delta \qquad \Delta; \Gamma \vdash \mathsf{P}_1 \qquad \mathsf{P}_2 \in \Delta_{\mathsf{P}_1}}{\Delta; \Gamma \vdash \mathsf{P}_2 <: \mathsf{P}_1}\ \textsc{PolyS-PIn}$$

Figure 4.8: Subtyping for λ_u.

Type safety and determinism. Polymorphic λ_u with updated disjointness preserves standard properties of subtyping, type-safety and determinism.

Lemma 4.10 (Subtyping Reflexivity). $\Delta; \Gamma \vdash A <: A$

Lemma 4.11 (Subtyping Transitivity). *If* $\Delta; \Gamma \vdash A <: B$ *and* $\Delta; \Gamma \vdash B <: C$ *then* $\Delta; \Gamma \vdash A <: C$

Theorem 4.12 (Type Preservation). *If* $\Delta; \Gamma \vdash e : A$ *and* $\Delta; \Gamma \vdash e \longrightarrow e'$ *then* $\Delta; \Gamma \vdash e' : A$.

Theorem 4.13 (Progress). *If* $\Delta; \cdot \vdash e : A$ *then either* e *is a value; or* e *can take a step to* e'.

Theorem 4.14 (Determinism). *If* $\Delta; \Gamma \vdash e : A$ *and* $\Delta; \Gamma \vdash e \longrightarrow e_1$ *and* $\Delta; \Gamma \vdash e \longrightarrow e_2$ *then* $e_1 = e_2$.

Substitution lemmas. Type preservation depends on type substitution (Lemma 4.15) which in turn depends on subtyping substitution (Lemma 4.16) and disjointness substitution (Lemma 4.17). Lemma 4.15 states that if in a context where α is disjoint to A_1, an expression

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

$$\frac{ok\ \Delta}{\Delta; \Gamma \vdash i : \mathsf{Int}}\ \textsc{ptyp-int} \qquad \frac{ok\ \Delta}{\Delta; \Gamma \vdash \mathsf{null} : \mathsf{Null}}\ \textsc{ptyp-null} \qquad \frac{ok\ \Delta \quad \Delta \vdash A \quad x : A \in \Gamma}{\Delta; \Gamma \vdash x : A}\ \textsc{ptyp-var}$$

$$\frac{\begin{array}{c} ok\ \Delta \quad \Delta \vdash A \\ \Delta \vdash B \quad \Delta; \Gamma \vdash e_1 : A \to B \quad \Delta; \Gamma \vdash e_2 : A \end{array}}{\Delta; \Gamma \vdash e_1\, e_2 : B}\ \textsc{ptyp-app} \qquad \frac{\Delta; \Gamma \vdash e : A \quad A <: B}{\Delta; \Gamma \vdash e : B}\ \textsc{ptyp-sub}$$

$$\frac{ok\ \Delta \quad \Delta \vdash A \quad \Delta \vdash B \quad \Delta; \Gamma, x : A \vdash e : B}{\Delta; \Gamma \vdash \lambda x.e : A \to B}\ \textsc{ptyp-abs} \qquad \frac{\Delta; \Gamma \vdash e : A \quad \Delta; \Gamma \vdash e : B}{\Delta; \Gamma \vdash e : A \wedge B}\ \textsc{ptyp-and}$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash e : A \vee B \quad \Delta; \Gamma, x : A \vdash e_1 : C \\ \Delta; \Gamma, y : B \vdash e_2 : C \quad \Delta \vdash A *_s B \end{array}}{\Delta; \Gamma \vdash \mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} : C}\ \textsc{ptyp-switch} \qquad \frac{ok\ \Delta \quad \Delta \vdash \mathsf{P}}{\Delta; \Gamma \vdash \mathsf{new}\ \mathsf{P} : \mathsf{P}}\ \textsc{ptyp-prim}$$

$$\frac{\Delta; \Gamma \vdash e : \forall(\alpha * A).C \quad \Delta; \Gamma \vdash B * A}{\Delta; \Gamma \vdash e\, B : C[\alpha \rightsquigarrow B]}\ \textsc{ptyp-tapdisj} \qquad \frac{\Delta; \Gamma,\ \alpha * A \vdash e : B}{\Delta; \Gamma \vdash \Lambda(\alpha * A).e : \forall(\alpha * A).B}\ \textsc{ptyp-tabsdisj}$$

Figure 4.9: Operational Semantics for λ_u.

e has type B and A_2 is disjoint to A_1, then e has type B after substituting α with A_2 in the context Γ, expression e and type B.

Lemma 4.16 states that if in a context where α is disjoint to A_1, type B is subtype of type C and A_2 is disjoint to A_1, then B stays subtype of C after substituting α with A_2 in context Γ, type B and type C. Similarly, Lemma 4.17 states that if in an environment where type variable α is disjoint to A_1, type B is disjoint to type C and A_2 is disjoint to A_1, then B stays disjoint to C after substituting α with A_2 in environment Γ, type B and type C. Notice that substitution lemmas no longer depend upon ground types.

Lemma 4.15 (Typing Substitution)**.** *If* $\Delta; \Gamma,\ \alpha * A_1 \vdash e : B$ *and* $\Delta; \Gamma \vdash A_2 * A_1$ *then* $\Delta; \Gamma[\alpha \rightsquigarrow A_2] \vdash e[\alpha \rightsquigarrow A_2] : B[\alpha \rightsquigarrow A_2]$

Lemma 4.16 (Subtyping Substitution)**.** *If* $\Delta; \Gamma,\ \alpha * A_1 \vdash B <: C$ *and* $\Delta; \Gamma \vdash A_2 * A_1$ *then* $\Delta; \Gamma[\alpha \rightsquigarrow A_2] \vdash B[\alpha \rightsquigarrow A_2] <: C[\alpha \rightsquigarrow A_2]$

Lemma 4.17 (Disjointness Substitution)**.** *If* $\Delta; \Gamma,\ \alpha * A_1 \vdash B * C$ *and* $\Delta; \Gamma \vdash A_2 * A_1$ *then* $\Delta; \Gamma[\alpha \rightsquigarrow A_2] \vdash B[\alpha \rightsquigarrow A_2] * C[\alpha \rightsquigarrow A_2]$

$\boxed{\Delta; \Gamma \vdash e \longrightarrow e'}$ *(Operational Semantics)*

POLYSTEP-APPL
$$\frac{\Delta; \Gamma \vdash e_1 \longrightarrow e_1'}{\Delta; \Gamma \vdash e_1\, e_2 \longrightarrow e_1'\, e_2}$$

POLYSTEP-APPR
$$\frac{\Delta; \Gamma \vdash e \longrightarrow e'}{\Delta; \Gamma \vdash v\, e \longrightarrow v\, e'}$$

POLYSTEP-BETA
$$\frac{}{\Delta; \Gamma \vdash (\lambda x.e)\, v \longrightarrow e[x \rightsquigarrow v]}$$

POLYSTEP-TAPPL
$$\frac{\Delta; \Gamma \vdash e \longrightarrow e'}{\Delta; \Gamma \vdash e\, B \longrightarrow e'\, B}$$

POLYSTEP-TAPPDISJ
$$\frac{}{\Delta; \Gamma \vdash (\Lambda(\alpha * A).e)\, B \longrightarrow e[\alpha \rightsquigarrow B]}$$

POLYSTEP-SWITCH
$$\frac{\Delta; \Gamma \vdash e \longrightarrow e'}{\Delta; \Gamma \vdash \mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow \mathsf{switch}\ e'\ \{(x : A) \to e_1, (y : B) \to e_2\}}$$

POLYSTEP-SWITCHL
$$\frac{\textbf{value}\ v \qquad \Delta; \Gamma \vdash \lfloor v \rfloor <: A}{\Delta; \Gamma \vdash \mathsf{switch}\ v\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow e_1[x \rightsquigarrow v]}$$

POLYSTEP-SWITCHR
$$\frac{\textbf{value}\ v \qquad \Delta; \Gamma \vdash \lfloor v \rfloor <: B}{\Delta; \Gamma \vdash \mathsf{switch}\ v\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow e_2[y \rightsquigarrow v]}$$

Figure 4.10: Operational Semantics for λ_u.

Narrowing lemmas. Narrowing lemmas for polymorphic λ_u are interesting to discuss. Typing narrowing, subtyping narrowing, and disjointness narrowing are stated as Lemma 4.18, Lemma 4.19, and Lemma 4.20 respectively. Narrowing lemmas essentially explain the relation between disjointness and subtyping. In general, they state that it is safe to replace the bound of a type variable with a supertype of it's existing bound. Typing narrowing (Lemma 4.18) states that if in a context where α is disjoint to A_1, an expression e has type B and A_1 is subtype of A_2, then e still has type B after replacing the bound of α with A_2.

Subtyping narrowing (Lemma 4.19) states that if in a context where α is disjoint to A_1, type B is a subtype of type C and A_1 is subtype of A_2, then B stays subtype of C after replacing the bound of α with A_2. Similarly, disjointness narrowing (Lemma 4.20) states that if we update the bound of a type variable with a subtype of its current bound, then it does not affect disjointness. Types A and B stay disjoint if we update the bound of type variable α from A_1 to its subtype A_2.

Lemma 4.18 (Typing narrowing). *If* $\Delta; \Gamma,\ \alpha * A_1 \vdash e : B$ *and* $\Delta; \Gamma \vdash A_1 <: A_2$ *then* $\Delta; \Gamma,\ \alpha * A_2 \vdash e : B$

Lemma 4.19 (Subtyping narrowing). *If* $\Delta; \Gamma,\ \alpha * A_1 \vdash B <: C$ *and* $\Delta; \Gamma \vdash A_1 <: A_2$ *then* $\Delta; \Gamma,\ \alpha * A_2 \vdash B <: C$

Lemma 4.20 (Disjointness narrowing)**.** *If* $\Delta; \Gamma, \alpha * A_1 \vdash B * C$ *and* $\Delta; \Gamma \vdash A_1 <: A_2$ *then* $\Delta; \Gamma, \alpha * A_2 \vdash B * C$

Weakening lemmas. Weakening lemmas for polymorphic λ_u are stated as typing weakening (Lemma 4.21), subtyping weakening (Lemma 4.22), and disjointness weakening (Lemma 4.23). Weakening lemmas state that if a relation is valid in a smaller context, then it stays valid in an enlarged context given that the enlarged context is well-formed. Typing weakening (Lemma 4.21) states that if in a smaller context $[\Gamma_1, \Gamma_2]$, an expression e has type B and an enlarged context $[\Gamma_1, \Gamma_3, \Gamma_2]$ is well-formed, then e still has type B in the extended context $[\Gamma_1, \Gamma_3, \Gamma_2]$.

Subtyping weakening (Lemma 4.22) states that if in a smaller context $[\Gamma_1, \Gamma_2]$, type B is a subtype of type C and an enlarged context $[\Gamma_1, \Gamma_3, \Gamma_2]$ is well-formed, then B stays subtype of C in the extended context $[\Gamma_1, \Gamma_3, \Gamma_2]$. Similarly, disjointness weakening (Lemma 4.23) states that if in a smaller context $[\Gamma_1, \Gamma_2]$, type B is disjoint to type C and an enlarged context $[\Gamma_1, \Gamma_3, \Gamma_2]$ is well-formed, then B stays disjoint to C in the extended context $[\Gamma_1, \Gamma_3, \Gamma_2]$.

Lemma 4.21 (Typing weakening)**.** *If* $\Delta; \Gamma_1, \Gamma_2 \vdash e : B$ *and ok* $\Gamma_1, \Gamma_3, \Gamma_2$ *then* $\Delta; \Gamma_1, \Gamma_3, \Gamma_2 \vdash e : B$

Lemma 4.22 (Subtyping weakening)**.** *If* $\Delta; \Gamma_1, \Gamma_2 \vdash B <: C$ *and ok* $\Gamma_1, \Gamma_3, \Gamma_2$ *then* $\Delta; \Gamma_1, \Gamma_3, \Gamma_2 \vdash B <: C$

Lemma 4.23 (Disjointness weakening)**.** *If* $\Delta; \Gamma_1, \Gamma_2 \vdash B * C$ *and ok* $\Gamma_1, \Gamma_3, \Gamma_2$ *then* $\Delta; \Gamma_1, \Gamma_3, \Gamma_2 \vdash B * C$

More auxiliary lemmas. We discuss a few more interesting auxiliary lemmas in this paragraph. Lemma 4.24, Lemma 4.25, Lemma 4.26 and Lemma 4.27 are essential in proving the metatheory and are shown next. Lemma 4.24 states that if a union ordinary type $A^{\circledcirc}$ is a subtype of union splittable type B ($B_1 \lhd B \rhd B_2$), then A is either subtype of B_1 or B_2. Lemma 4.25 states that if a union splittable type A ($A_1 \lhd A \rhd A_2$) is well-formed, then both A_1 and A_2 are well-formed. Lemma 4.26 states disjointness symmetry. Finally, Lemma 4.27 states that if A and B are disjoint types, then this is not the case that a value v checks against both A and B.

Lemma 4.24 (Subtyping inversion of union ordinary and union splittable types)**.** *If* $\Delta; \Gamma \vdash A <: B$ *and* $B_1 \lhd B \rhd B_2$ *and* $A^{\circledcirc}$ *then* $\Delta; \Gamma \vdash A <: B_1 \vee \Delta; \Gamma \vdash A <: B_2$.

Lemma 4.25 (Well-formedness inversion of union splittable types)**.** *If* $\Delta; \Gamma \vdash A$ *and* $A_1 \lhd A \rhd A_2$ *then* $\Delta; \Gamma \vdash A_1 \wedge \Delta; \Gamma \vdash A_2$.

Lemma 4.26 (Disjointness symmetry). *If* $\Delta; \Gamma \vdash A * B$ *then* $\Delta; \Gamma \vdash B * A$.

Lemma 4.27 (Exclusivity of Disjoint Types). *If* $\Delta; \Gamma \vdash A * B$ *then* $\nexists\, v$ *such that both* $\Delta; \Gamma \vdash v : A$ *and* $\Delta; \Gamma \vdash v : B$ *holds.*

5 Togetherness: Switches and Merges

5.1 Overview

λ_u and its extensions discussed so far mainly revolve around deterministic type-based elimination of union types. Section 3.2 discusses a simple λ_u calculus and introduces the key ideas including disjointness. Section 3.3 enriches λ_u with more advance features including subtyping distributivity, nominal types, and intersection types. Section 3.4.1 and Chapter 4 further study λ_u with disjoint polymorphism. The restriction of overlapping types in alternative branches of a type-based switch expression by employing disjointness has been an integral part of the study so far.

We study λ_u with intersection types in Section 3.3, but so far, none of the calculi in this thesis studies a respective term level construct to introduce intersection types. In this chapter we extend λ_u with a so called merge operator[Dunfield 2014; Reynolds 1988], which acts as a term to introduce intersection types. The resulting calculus is called λ_{um} i.e. union calculus with the merge operator. Intersection types are useful without the merge operator but the merge operator increases term level expressiveness of the calculus. The merge operator has been studied by various researchers in literature [Dunfield 2014; Huang and Oliveira 2020; Oliveira et al. 2016; Reynolds 1988]. Calculi with the merge operator are known to have wide practical applications. The merge operator together with the intersection types is naturally able to encode various advanced programming features such as nested composition [Bi et al. 2018b], multi-field records from single-field records [Reynolds 1988], and function overloading [Castagna et al. 1995; Reynolds 1988] among others.

Essence of the merge operator. The typical introduction rule for the intersection types without the merge operator is:

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash e : B}{\Gamma \vdash e : A \wedge B}\;\textsc{typ-and}$$

Rule typ-and allows to construct certain terms of intersection types, such as:

$$\frac{1 : \mathsf{Int} \qquad 1 : \mathsf{Int}}{1 : \mathsf{Int} \wedge \mathsf{Int}} \text{ TYP-AND}$$

or

$$\frac{1 : \mathsf{Int} \qquad 1 : \top}{1 : \mathsf{Int} \wedge \top} \text{ TYP-AND}$$

Perhaps a more useful program with typical introduction rule for intersection types is:

$$\frac{\lambda x.x : \mathsf{Int} \to \mathsf{Int} \qquad \lambda x.x : \mathsf{Bool} \to \mathsf{Bool}}{\lambda x.x : \mathsf{Int} \to \mathsf{Int} \wedge \mathsf{Bool} \to \mathsf{Bool}} \text{ TYP-AND}$$

However, this rule does not have enough expressiveness to type-check a program consisting of non-overlapping types, for example, Int and Bool. A part of such a program is an integer, and the other part is a boolean. Theoretically, since intersection types correspond to product types, in our work a pair expresses such program with product types as:

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

Such a pair of non-overlapping product types suggests the need of an equivalent expression in programming languages to introduce non-overlapping intersection of types i.e. $\mathsf{Int} \wedge \mathsf{Bool}$. The merge operator ($e_1,,e_2$) [Dunfield 2014; Oliveira et al. 2016; Reynolds 1988] is capable of expressing such programs with intersection types. The introduction rule for intersection types in the presence of merge operator is:

$$\text{TYP-MERGA} \quad \frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash e_1,,e_2 : A \wedge B}$$

The expression $e_1,,e_2$ is called the merge operator. Notice that the two expressions in the premise of rule TYP-MERGA could be two different expressions. This is in contrast to the rule TYP-AND where the expression stays the same. Therefore the introduction rule for intersection types in the presence of the merge operator allows to construct an expression of non-overlapping types, such as:

Type	A, B, C	$::=$	$\top \mid \bot \mid \mathsf{Int} \mid A \to B \mid A \vee B \mid A \wedge B$
Expr	e	$::=$	$x \mid i \mid e : A \mid \lambda x.e : A \to B \mid e_1\, e_2 \mid \mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} \mid$ $e_1,,e_2 \mid \mathsf{fix}\ x.e : A \mid \top$
Value	v	$::=$	$i \mid \lambda x.e : A \to B \mid v_1,,v_2 \mid \top$
Context	Γ	$::=$	$\cdot \mid \Gamma, x : A$

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

$$\text{S-TOP}\ \frac{}{A <: \top} \qquad \text{S-INT}\ \frac{}{\mathsf{Int} <: \mathsf{Int}} \qquad \text{S-ARROW}\ \frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2} \qquad \text{S-BOT}\ \frac{}{\bot <: A} \qquad \text{S-ORA}\ \frac{A <: C \qquad B <: C}{A \vee B <: C}$$

$$\text{S-ORB}\ \frac{A <: B}{A <: B \vee C} \qquad \text{S-ORC}\ \frac{A <: C}{A <: B \vee C} \qquad \text{S-ANDA}\ \frac{A <: B \qquad A <: C}{A <: B \wedge C} \qquad \text{S-ANDB}\ \frac{A <: C}{A \wedge B <: C} \qquad \text{S-ANDC}\ \frac{B <: C}{A \wedge B <: C}$$

Figure 5.1: Syntax and subtyping for λ_{um}.

$$\frac{1 : \mathsf{Int} \qquad \mathsf{true} : \mathsf{Bool}}{1,,\mathsf{true} : \mathsf{Int} \wedge \mathsf{Bool}}\ \text{TYP-MERGA}$$

Elimination form for pairs and merges. Note that the pairs and merges diverge in their elimination form. Pairs are eliminated by explicit elimination constructs such as `fst` and `snd` or explicit tags. Whereas the merge operator has an implicit elimination form usually via subtyping. The direct operational semantics for the merge operator is type-dependent [Huang and Oliveira 2020]. This will be further discussed in later sections.

5.2 λ_{um} Calculus

5.2.1 Syntax, subtyping, and typing

Syntax and subtyping. Figure 5.1 shows the syntax and subtyping for λ_{um}. The notable difference from λ_u is the addition of the merge operator ($e_1,,e_2$) in the syntactic category of expressions. Apart from that, lambda expressions are annotated with the input and output types i.e. $\lambda x.e : A \to B$. We also add a fix point operator ($\mathsf{fix}\ x.e : A$) and a top term ($\top$) in the expressions. Values are constituted of integers (i), lambda expressions ($\lambda x.e : A \to B$), top expression ($\top$) and a merge operator consisting of all the values is also a value ($v_1,,v_2$). Types and the context stay the same as in λ_u with intersection types. Subtyping is also standard

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

$$\text{TYP-VAR}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \text{TYP-INT}\ \frac{}{\Gamma \vdash i : \mathsf{Int}} \qquad \text{TYP-SUB}\ \frac{\Gamma \vdash e : A \quad A <: B}{\Gamma \vdash e : B} \qquad \text{TYP-APP}\ \frac{\Gamma \vdash e_1 : A \to B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B}$$

$$\text{TYP-ABSANN}\ \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash (\lambda x.e : A \to B) : A \to B} \qquad \text{TYP-SWITCHA}\ \frac{\Gamma \vdash e : A \vee B \quad \Gamma, x : A \vdash e_1 : C \quad \Gamma, y : B \vdash e_2 : C}{\Gamma \vdash \mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} : C}$$

$$\text{TYP-ANN}\ \frac{\Gamma \vdash e : A}{\Gamma \vdash (e : A) : A} \qquad \text{TYP-MERGA}\ \frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B}{\Gamma \vdash e_1,,e_2 : A \wedge B} \qquad \text{TYP-FIX}\ \frac{\Gamma \vdash e : A}{\Gamma \vdash (\mathsf{fix}\ x.e : A) : A}$$

Figure 5.2: Typing for λ_{um}.

for a calculus with intersection and union types. Note that subtyping does not deal with distributivity rules. Rules S-TOP, S-INT, S-ARROW, S-BOT, S-ORA, S-ORB, and S-ORC are already explained. Rules S-AND, S-ANDB, and S-ANDC deal with intersection types. The subtyping relation for λ_{um} is reflexive and transitive.

Lemma 5.1 (Subtyping reflexivity). $A <: A$

Lemma 5.2 (Subtyping transitivity). *If $A <: B$ and $B <: C$ then $A <: C$*

Typing. Figure 5.2 shows the typing rules for λ_{um}. A notable difference is the addition of rules TYP-MERGA, TYP-FIX, and TYP-ANN. Rule TYP-MERGA is the typing rule for the newly added merge operator. This rule states that if an expression e_1 has type A and an expression e_2 has type B, then the merge of e_1 and e_2 has type $A \wedge B$. Rule TYP-FIX is a standard typing rule for the fix point operator. Similarly rule TYP-ANN is the standard typing rule for type annotations. The rest of the rules are standard and have already been discussed.

Unlike disjoint intersection types [Oliveira et al. 2016], we do not impose a disjointness restriction in rule TYP-MERGA. Notice that the disjointness restriction has also been scrapped in rule TYP-SWITCHA. This is because the addition of the merge operator makes determinism non-trivial. Therefore we present a simple calculus without disjointness restriction and the calculus is not deterministic. Determinism and its challenges are further discussed in Section 5.4. Using rule TYP-MERGA we can construct the following program which may be a source of non-determinism.

$$\frac{1 : \mathsf{Int} \qquad 2 : \mathsf{Int}}{1,,2 : \mathsf{Int} \wedge \mathsf{Int}} \text{ TYP-MERGA}$$

When applying `succ` function to the argument 1,,2 the result may either be 2 or 3. Another source of non-determinism in this calculus is:

```
Bool isInt (x : Int|Bool) = switch (x)
                             (x:Bool)  → false
                             (y:Int)   → true
```

The above program may return different value depending on the order of the branches if a value of type Int ∧ Bool is passed, such as 1,,true. Moreover, even though Bool and Int are non-overlapping types, the program may fall either in the first or the second branch. This is because the type of x (when x is 1,,true) is a subtype of both Bool and Int. By employing rule TYP-SUB x can be treated as value of both types i.e. Bool and Int.

5.2.2 Type casting

Type casting lies at the core of the calculi with direct operational semantics for the merge operator [Huang and Oliveira 2020]. Generally speaking, type casting makes an expression consistent with the type under which that expression is cast. For example when an expression 1,,true casts under type Int it gives 1 and true when it casts under the type Bool. In summary, type casting enables extracting the value of a specific type from the merge operator. For example:

$$1,,\mathsf{true} \longrightarrow_{\mathsf{Int}} 1 \quad \text{(applying rule CST-MERGL)}$$

In this example ($1,,\mathsf{true} \longrightarrow_{\mathsf{Int}} 1$) when we cast a merge of 1,,true under the type Int it results in 1. This is because of the fact that the only integer we can get from a merge of 1,,true is 1. Similarly true,,1 also yields 1 when casts under the type Int:

$$\mathsf{true},,1 \longrightarrow_{\mathsf{Int}} 1 \quad \text{(applying rule CST-MERGR)}$$

Type casting relation. The type casting relation is shown in Figure 5.3. The relation $v \longrightarrow_A v'$ shows casting of a value v under type A to another value v'. Note that the type casting is only applicable to values. We elaborate each rule next.

When casting an expression under $\top$ type it results in $\top$ expression as stated in rule CST-TOP. Casting an integer under Int returns the same integer. Casting rules for union types

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

$$\frac{}{\mathsf{Ord\ Int}}\ \textsc{ord-int} \qquad \frac{}{\mathsf{Ord}\ A \to B}\ \textsc{ord-arrow}$$

$\boxed{v \longrightarrow_A v'}$ *(Type Casting)*

$$\frac{}{v \longrightarrow_\top \top}\ \textsc{cst-top} \qquad \frac{}{i \longrightarrow_{\mathsf{Int}} i}\ \textsc{cst-int} \qquad \frac{v \longrightarrow_A v'}{v \longrightarrow_{A \vee B} v'}\ \textsc{cst-orl} \qquad \frac{v \longrightarrow_B v'}{v \longrightarrow_{A \vee B} v'}\ \textsc{cst-orr}$$

$$\frac{A_1 \to B_1 <: A_2 \to B_2}{\lambda x.e : A_1 \to B_1 \longrightarrow_{A_2 \to B_2} \lambda x.e : A_1 \to B_1}\ \textsc{cst-arrow} \qquad \frac{\mathsf{Ord}\ A \qquad v_1 \longrightarrow_A v_1'}{v_1 ,, v_2 \longrightarrow_A v_1'}\ \textsc{cst-mergl}$$

$$\frac{\mathsf{Ord}\ A \qquad v_2 \longrightarrow_A v_2'}{v_1 ,, v_2 \longrightarrow_A v_2'}\ \textsc{cst-mergr} \qquad \frac{v \longrightarrow_A v_1 \qquad v \longrightarrow_B v_2}{v \longrightarrow_{A \wedge B} v_1 ,, v_2}\ \textsc{cst-merg}$$

Figure 5.3: Type casting for λ_{um}.

are interesting. If an expression casts under a part of the union type then that expression casts under whole union type as stated in rules CST-ORL and CST-ORR. Casting a lambda expression under a function type returns the same lambda expression as stated in rule CST-ARROW. A merge operator casts under an ordinary type if either part of the merge operator casts under that ordinary type (rules CST-MERGL and CST-MERGL). Casting an expression under an intersection type results in a merge (rule CST-MERG). Type casting preserves the standard properties of type preservation and progress:

- Type casting preservation (Theorem 5.3), which states that if an expression v type casts under type A to some expression v', then v' has type A.
- Type casting progress (Theorem 5.4), which states that if an expression v has type A, then v type casts under A to some v'.

Theorem 5.3 (Type Casting Preservation)**.** *If* $v : A$ *and* $v \longrightarrow_A v'$ *then* $\Gamma \vdash v' : A$.

Theorem 5.4 (Type Casting Progress)**.** *If* $\cdot \vdash v : A$ *then* $v \longrightarrow_A v'$

Theorems 5.3 and 5.4 are vital in proving the type-safety of the calculus. Notice that the type casting relation is non-deterministic which results in non-deterministic semantics. For the illustration purpose, think for a moment what will be the result of the following type cast?

$$\mathsf{true},,1 \longrightarrow_{\mathsf{Int} \vee \mathsf{Bool}} ??$$

It can either result in 1 or true depending upon the type casting rule we apply:

$$1,,\mathsf{true} \longrightarrow_{\mathsf{Int} \vee \mathsf{Bool}} 1 \quad \text{(applying rule CST-ORL)}$$

$$1,,\mathsf{true} \longrightarrow_{\mathsf{Int} \vee \mathsf{Bool}} \mathsf{true} \quad \text{(applying rule CST-ORR)}$$

5.2.3 Operational semantics

The operational semantics for λ_{um} is shown in Figure 5.4. Rules STEP-APPL and STEP-APPR are standard reduction rules. Rule STEP-ABETA is the beta reduction rule. This rule first type casts the argument under the input type and then substitutes the argument in the body of the lambda expression. Type casting drops the unnecessary part from the argument and makes the input value consistent with the input type of applied function. For example:

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

A merge of 1,,true is the input being passed to lambda expression annotated with an input type of Int. Since the dynamic type of 1,,true is $\mathsf{Int} \wedge \mathsf{Bool}$ which is a subtype of Int. The above application type-checks. But before passing 1,,true to the lambda body it will be cast under the input type i.e Int to make input value and the input type consistent:

$$1,,\mathsf{true} \longrightarrow_{\mathsf{Int}} 1$$

The application $(\lambda x.\mathsf{true},,x : \mathsf{Int} \to \mathsf{Bool})\ (1,,\mathsf{true})$ will result in $\mathsf{true},,1 : \mathsf{Bool}$ after beta-reduction. We show the partial derivation next:

$$\dfrac{\dfrac{}{1,,\mathsf{true} \longrightarrow_{\mathsf{Int}} 1}\ \text{CST-MERGL}}{(\lambda x.\mathsf{true},,x : \mathsf{Int} \to \mathsf{Bool})\ (1,,\mathsf{true}) \longrightarrow \mathsf{true},,1 : \mathsf{Bool}}\ \text{STEP-ABETA}$$

Note that the expression $\mathsf{true},,1 : \mathsf{Bool}$ is not a value. Rule STEP-ANNV reduces such annotated values. The expression $(\mathsf{true},,1 : \mathsf{Bool})$ takes one more step following the rule STEP-ANNV and results in true. This is illustrated by the following derivation:

$$\dfrac{\dfrac{\mathsf{true} \longrightarrow_{\mathsf{Bool}} \mathsf{true}}{1,,\mathsf{true} \longrightarrow_{\mathsf{Bool}} \mathsf{true}}\ \text{CST-MERGR}}{\mathsf{true},,1 : \mathsf{Bool} \longrightarrow \mathsf{true}}\ \text{STEP-ANNV}$$

$\boxed{e \longrightarrow e'}$ *(Small-step operational semantics)*

$$\frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \textsc{step-appl} \qquad \frac{e \longrightarrow e'}{v\, e \longrightarrow v\, e'}\ \textsc{step-appr} \qquad \frac{v \longrightarrow_A v'}{(\lambda x.e : A \to B)\, v \longrightarrow (e[x \rightsquigarrow v']) : B}\ \textsc{step-abeta}$$

$$\frac{(v_1,,v_2)\, v \longrightarrow_d e'}{(v_1,,v_2)\, v \longrightarrow e'}\ \textsc{step-dispatch} \qquad \frac{e_1 \longrightarrow e_1'}{e_1,,e_2 \longrightarrow e_1',,e_2}\ \textsc{step-mergl} \qquad \frac{e \longrightarrow e'}{v,,e \longrightarrow v,,e'}\ \textsc{step-mergr} \qquad \frac{e \longrightarrow e'}{e : A \longrightarrow e' : A}\ \textsc{step-ann}$$

$$\frac{v \longrightarrow_A v'}{v : A \longrightarrow v'}\ \textsc{step-annv} \qquad \frac{}{\mathsf{fix}\ x.e : A \longrightarrow e[x \rightsquigarrow \mathsf{fix}\ x.e : A]}\ \textsc{step-fix}$$

$$\frac{e \longrightarrow e'}{\mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow \mathsf{switch}\ e'\ \{(x : A) \to e_1, (y : B) \to e_2\}}\ \textsc{step-switch}$$

$$\frac{v \longrightarrow_A v'}{\mathsf{switch}\ v\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow e_1[x \rightsquigarrow v']}\ \textsc{step-switchlm}$$

$$\frac{v \longrightarrow_B v'}{\mathsf{switch}\ v\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow e_2[y \rightsquigarrow v']}\ \textsc{step-switchrm}$$

Figure 5.4: Operational semantics for λ_{um}.

Rules STEP-MERGL and STEP-MERGR reduce the left and right part of the merge operator. Rule STEP-ANNO reduces an annotated expression. Rules STEPSWITCH, STEP-SWITCHL, and STEP-SWITCHR are the standard reduction rules as in λ_u and are already discussed. Finally, rule STEP-FIX is a standard reduction rule for the fix point operator. It replaces the fix point with x in its own body. The rule STEP-DISPATCH requires detailed explanation and is discussed next.

5.2.4 Applicative dispatch and rule step-dispatch

Sometimes lambda expression maybe hidden inside a merge. The rule STEP-DISPATCH deals with such cases. The rule STEP-DISPATCH uses another relation called applicative dispatch [Xue et al. 2022]. Applicative dispatch relation is shown at the upper part of Figure 5.5. Essentially, applicative dispatch selects the appropriate function for application from the merge

$\boxed{e \longrightarrow_d e'}$ *(Applicative dispatch)*

$$\text{APD-MLEFT}\ \frac{\lfloor v \rfloor <: \lfloor v_1 \rfloor^{\lambda} \qquad \neg \lfloor v \rfloor <: \lfloor v_2 \rfloor^{\lambda}}{(v_1 ,, v_2)\ v \longrightarrow_d v_1\ v} \qquad \text{APD-MRIGHT}\ \frac{\neg \lfloor v \rfloor <: \lfloor v_1 \rfloor^{\lambda} \qquad \lfloor v \rfloor <: \lfloor v_2 \rfloor^{\lambda}}{(v_1 ,, v_2)\ v \longrightarrow_d v_2\ v}$$

$$\text{APD-BOTH}\ \frac{\lfloor v \rfloor <: \lfloor v_1 \rfloor^{\lambda} \qquad \lfloor v \rfloor <: \lfloor v_2 \rfloor^{\lambda}}{(v_1 ,, v_2)\ v \longrightarrow_d v_1\ v ,, v_2\ v}$$

Dynamic Type $\lfloor v \rfloor$

$\lfloor i \rfloor$	=	Int
$\lfloor \top \rfloor$	=	$\top$
$\lfloor \lambda x.e : A \to B \rfloor$	=	$A \to B$
$\lfloor v_1 ,, v_2 \rfloor$	=	$\lfloor v_1 \rfloor \wedge \lfloor v_1 \rfloor$

Input Type $\lfloor v \rfloor^{\lambda}$

$\lfloor \lambda x.e : A \to B \rfloor^{\lambda}$	=	A
$\lfloor v_1 ,, v_2 \rfloor^{\lambda}$	=	$\lfloor v_1 \rfloor^{\lambda} \vee \lfloor v_1 \rfloor^{\lambda}$
$\lfloor i \rfloor^{\lambda}$	=	$\bot$
$\lfloor \top \rfloor^{\lambda}$	=	$\bot$

Figure 5.5: Dynamic dispatch, dynamic type and input type relation for λ_{um}.

depending upon the argument type. Moreover, it provides the adequacy of function overloading. For example:

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

The above application is valid and type-checks. We need to extract $(\lambda x.\mathsf{succ} : \mathsf{Int} \to \mathsf{Int})$ out of this merge and then apply it to the argument 1. Note that it is essential to choose $(\lambda x.\mathsf{succ} : \mathsf{Int} \to \mathsf{Int})$ in this case. The calculus will not be type preserving otherwise. This is due to the fact that $(\mathsf{true}\ 1)$ is not a valid application. The problem gets worst if the merge contains multiple functions. Therefore special care needs to be taken to choose the appropriate function for application. Applicative dispatch compares the argument type with the input type of each expression in the merge. It then applies all of the overlapping functions. The derivation for $((\lambda x.\mathsf{succ} : \mathsf{Int} \to \mathsf{Int}),,\mathsf{true})\ 1$ is shown below:

$$\text{STEP-DISPATCH}\ \frac{\text{APD-MLEFT}\ \dfrac{\mathsf{Int} <: \mathsf{Int} \qquad \neg(\mathsf{Int} <: \bot)}{(\lambda x.\mathsf{succ} : \mathsf{Int} \to \mathsf{Int},,\mathsf{true}) \longrightarrow (\lambda x.\mathsf{succ} : \mathsf{Int} \to \mathsf{Int})\ 1}}{(\lambda x.\mathsf{succ} : \mathsf{Int} \to \mathsf{Int},,\mathsf{true})\ 1 \longrightarrow (\lambda x.\mathsf{succ} : \mathsf{Int} \to \mathsf{Int})\ 1}$$

The expression $((\lambda x.\mathsf{succ} : \mathsf{Int} \to \mathsf{Int})\ 1)$ becomes a standard application with lambda expression at the left side and reduces by following the standard beta reduction.

Applicative dispatch relation. Rule adp-mleft applies the left part of the merge to the argument. It consist of two premises. The first premise states that the dynamic type of the argument is a subtype of the input type of left part of merge. While second premise states that the dynamic type of the argument is not a subtype of the input type of right part of the merge. If both of the conditions succeed, this rule applies left part of the merge to the argument. For example consider the following application:

$$((\lambda x.\mathsf{succ} : \mathsf{Int} \to \mathsf{Int}),,(\lambda x.\mathsf{not} : \mathsf{Bool} \to \mathsf{Bool}))\ 1$$

The functional merge consists of two functions, succ and not. Input type of succ function is Int and the input type of not function is Bool. Note that the dynamic type of argument 1 is Int. It is evident that the only function that can be applied to 1 among succ and not functions is succ. Rule adp-mleft checks that the dynamic type of 1 is a subtype of Int, and is not a subtype of Bool. Therefore succ is applied to 1. Rule adp-mright repeats the same for the right part of the merge.

Rule adp-both is interesting. It applies both the left and the right part of the merge to the argument and returns a merge of the output from both functions. It is applicable in cases where the dynamic type of the argument is a subtype of the left as well as the right part of the merge. Both the succ as well as the pred functions are applicable to 1 in the following application.

$$((\lambda x.\mathsf{succ} : \mathsf{Int} \to \mathsf{Int}),,(\lambda x.\mathsf{pred} : \mathsf{Int} \to \mathsf{Int}))\ 1$$

Dynamic type and input type relations. The dynamic type and the input type relations are shown in Figure 5.5. Dynamic type returns the principal type of a value and has already been discussed. The input type relation returns the input type of a functional value. The functional value is either a lambda expression or a merge containing at least one lambda expression. The first case of the input type deals with lambda expressions. In this case the input type simply returns the input type of the given lambda expression. The second case deals with functional merges. In this case the output is the union of the input types of all the lambda expressions in the merge. The other values cannot appear on the left side of a valid application. Therefore they cannot have an input type. We simply return $\perp$ in such cases.

5.2.5 Metatheory of λ_{um}

The standard properties of type preservation and progress hold in this system. Type preservation (Theorem 5.5) states that the types are preserved during reduction. The progress (Theorem 5.6) states that a well-typed expression is either a value or it reduces until it becomes

a value. Note that the operational semantics is type dependent i.e. operational semantics depends upon the casting relation. Therefore type preservation depends upon type casting preservation (Theorem 5.3) and the progress depends upon type casting progress (Theorem 5.4).

Theorem 5.5 (Type Preservation). *If* $\Gamma \vdash e : A$ *and* $e \longrightarrow e'$ *then* $\Gamma \vdash e' : A$.

Theorem 5.6 (Progress). *If* $\cdot \vdash e : A$ *then either* e *is a value; or* $e \longrightarrow e'$ *for some* e'.

Compelling properties of λ_{um}. We show a few interesting properties of λ_{um} in this section. In particular, Lemma 5.7 is essential in λ_{um}. It states that if a value v casts under a function type then the resulting value must be a lambda expression. Lemma 5.8 states that if a value v checks against a function type $A \to B$ then the input type from the function type (A) is a subtype of the result from input type relation i.e. $A <: \lfloor v \rfloor^{\lambda}$. Lemma 5.9 states that if the dynamic type of value v_1 is a subtype of the input type of v i.e. $\lfloor v_1 \rfloor <: \lfloor v \rfloor^{\lambda}$ then the value v checks against a function type such that the dynamic type of v is the input type of the function. Lemmas 5.8 and 5.9 are essential for the type preservation of λ_{um}. Similarly Lemma 5.10 is essential for the progress. It states that a well-typed application of the merge operator must take a step.

Lemma 5.7 (Casting under a function type). *If* $v \longrightarrow_{A \to B} v'$*, then* $\exists\ e\ A1\ B1,\ v' = \lambda x.e : A_1 \to B_1$.

Lemma 5.8 (Covariance of input type). *If* $\cdot \vdash v : A \to B$ *then* $A <: \lfloor v \rfloor^{\lambda}$.

Lemma 5.9 (Applicative dispatch preservation). *If* $\cdot \vdash v : A$ *and* $\lfloor v_1 \rfloor <: \lfloor v \rfloor^{\lambda}$ *then* $\cdot \vdash v : \lfloor v_1 \rfloor \to \top$

Lemma 5.10 (Applicative dispatch progress). *If* $\cdot \vdash v_1 ,, v_2 : A \to B$ *and* $\cdot \vdash v : A$ *then* $\exists e'$ $(v_1 ,, v_2)\ v \longrightarrow e'$.

5.3 λ_{um} and Dunfield's system

Dunfield [2014] studies a calculus with intersection types, union types, and the merge operator. The merge operator acts as an introduction form for intersection types and elimination form for union types in their calculus. They adopt an elaboration semantics and elaborate the source language to a target language. Dunfield elaborates intersection types to product types and union types to sum types. Similarly they elaborate the merge operator to a pair. The target language is a standard extension of the simply typed lambda calculus with the pairs, sum types, and the product types.

We will explain the Dunfield's type system and prove its completeness with respect to λ_{um} in this section. In particular, we show that all the programs that Dunfield's system type-checks, are type-checked by λ_{um}. In contrast to Dunfield's system, we propose a direct operational semantics which significantly simplifies the theoretical complexity of the system. Moreover, contrary to the Dunfield [2014] source semantics, the direct operational semantics studied in λ_{um} is type-sound.

5.3.1 Dunfield's calculus

Syntax. Syntax for Dunfield's system is shown in the upper part of Figure 5.6. Types consist of $\top$, $\bot$, Int, $A \to B$, $A \vee B$ and $A \wedge B$. Note that the type $\bot$ has not been studied in original Dunfield's calculus. Expressions consist of variables, literals, abstractions, applications, merge operator and the fix point operator. Note that lambda abstractions ($\lambda x.e$) do not carry types. Similarly, variables, literals, abstractions and a merge of values constitute values. The typing context is standard. Finally $\mathcal{E}$ represents the evaluation context.

Type system. The Dunfield's type system is shown in Figure 5.6. We suggest the reader to ignore the highlighted part in the rules for the sake of simplicity at this stage. We will explain the highlighted part later. Rules dtyp-int, dtyp-sub, dtyp-var, dtyp-app, and dtyp-abs are already explained modulo curly arrow and the highlighted part in each rule. Rules dtyp-mergA and dtyp-mergB type-check a merge operator. A notable difference in rule dtyp-abs is that the lambda expression no longer carries type annotations. Rule dtyp-and is an introduction rule for intersection types. It states that if an expression e has type A and type B then e has type $A \wedge B$. This rule is useful to type unannotated lambdas as a variant of function overloading, such as:

$$\textsc{dtyp-and}\ \frac{\overline{\lambda x.e : \mathsf{Int} \to \mathsf{Int}} \qquad \overline{\lambda x.e : \mathsf{Bool} \to \mathsf{Bool}}}{\lambda x.e : \mathsf{Int} \to \mathsf{Int} \wedge \mathsf{Bool} \to \mathsf{Bool}}$$

Rules dtyp-andl and dtyp-andr are for intersection elimination. They state that if an expression e has type $A \wedge B$, then the expression e can have either type A or type B. Similarly rules dtyp-orl and dtyp-orr are for union introduction. They state that an expression e can have type $A \vee B$ if the expression e has either type A or type B. Note that the rules dtyp-andl, dtyp-andr, dtyp-orl, and dtyp-orr can be subsumed by subsumption rule. Rule dtyp-switch is for the union elimination. Finally, rule dtyp-fix type-checks fix points. Interested readers may refer to the original paper [Dunfield 2014] for details.

Type	A, B, C	$::=$	$\top \mid \bot \mid \mathsf{Int} \mid A \to B \mid A \vee B \mid A \wedge B$
Expr	e	$::=$	$x \mid i \mid \lambda x.e \mid e_1\, e_2 \mid e_1 ,, e_2 \mid \mathsf{fix}\ x.e$
Value	v	$::=$	$x \mid i \mid \lambda x.e \mid v_1 ,, v_2$
Context	Γ	$::=$	$\cdot \mid \Gamma, x : A$
Evaluation Context	$\mathscr{E}$	$::=$	$[] \mid \mathscr{E}\, e \mid v\, \mathscr{E} \mid \mathscr{E} ,, e \mid e ,, \mathscr{E}$

$\boxed{\Gamma \vdash_d e : A \rightsquigarrow e'}$ *(Dunfield's type system)*

DTYP-INT
$$\frac{}{\Gamma \vdash_d i : \mathsf{Int} \rightsquigarrow i}$$

DTYP-SUB
$$\frac{\Gamma \vdash_d e : A \rightsquigarrow e' \qquad A <: B}{\Gamma \vdash_d e : B \rightsquigarrow e'}$$

DTYP-VAR
$$\frac{x : A \in \Gamma}{\Gamma \vdash_d x : A \rightsquigarrow x}$$

DTYP-APP
$$\frac{\Gamma \vdash_d e_1 : A \to B \rightsquigarrow e_1' \qquad \Gamma \vdash_d e_2 : A \rightsquigarrow e_2'}{\Gamma \vdash_d e_1\, e_2 : B \rightsquigarrow e_1'\, e_2'}$$

DTYP-ABS
$$\frac{\Gamma, x : A \vdash_d e : B \rightsquigarrow e'}{\Gamma \vdash_d \lambda x.e : A \to B \rightsquigarrow \lambda x.e' : A \to B}$$

DTYP-MERGA
$$\frac{\Gamma \vdash_d e_1 : A \rightsquigarrow e_1'}{\Gamma \vdash_d e_1 ,, e_2 : A \rightsquigarrow e_1'}$$

DTYP-MERGB
$$\frac{\Gamma \vdash_d e_2 : B \rightsquigarrow e_2'}{\Gamma \vdash_d e_1 ,, e_2 : B \rightsquigarrow e_2'}$$

DTYP-AND
$$\frac{\Gamma \vdash_d e : A \rightsquigarrow e_1 \qquad \Gamma \vdash_d e : B \rightsquigarrow e_2}{\Gamma \vdash_d e : A \wedge B \rightsquigarrow e_1 ,, e_2}$$

DTYP-ANDL
$$\frac{\Gamma \vdash_d e : A \wedge B \rightsquigarrow e'}{\Gamma \vdash_d e : A \rightsquigarrow e'}$$

DTYP-ANDR
$$\frac{\Gamma \vdash_d e : A \wedge B \rightsquigarrow e'}{\Gamma \vdash_d e : B \rightsquigarrow e'}$$

DTYP-ORL
$$\frac{\Gamma \vdash_d e : A \rightsquigarrow e'}{\Gamma \vdash_d e : A \vee B \rightsquigarrow e'}$$

DTYP-ORR
$$\frac{\Gamma \vdash_d e : B \rightsquigarrow e'}{\Gamma \vdash_d e : A \vee B \rightsquigarrow e'}$$

DTYP-SWITCH
$$\frac{\Gamma \vdash_d e : A \vee B \rightsquigarrow e' \qquad \Gamma, x : A \vdash_d \mathscr{E}[x] : C \rightsquigarrow e_1 \qquad \Gamma, y : B \vdash_d \mathscr{E}[y] : C \rightsquigarrow e_2}{\Gamma \vdash_d \mathscr{E}[e'] : C \rightsquigarrow \mathsf{switch}\ e'\ \{(x : A) \to e_1, (y : B) \to e_2\}}$$

DTYP-FIX
$$\frac{\Gamma \vdash_d e : A \rightsquigarrow e'}{\Gamma \vdash_d \mathsf{fix}\ x.e : A \rightsquigarrow \mathsf{fix}\ x.e' : A}$$

Figure 5.6: Source syntax and source typing of Dunfield's calculus.

A NOTE ON DUNFIELD'S TYPE SYSTEM. The Dunfield's type system allows certain (hidden) ill-typed parts of the programs. For example the program $1 ,, \mathsf{succ}\ \mathsf{true} : \mathsf{Int}$ type-checks following the rule DTYP-MERGA. It is important to note that a part of this program i.e. $\mathsf{succ}\ \mathsf{true}$ is ill-typed. A closer observation reveals that such ill-typed parts of the programs have no practical effects and are not essential. Therefore we do not account for such ill-typed parts

of the programs in our calculus. We prove completeness upto the point where all the parts of a program are well-typed.

5.3.2 Completeness with respect to Dunfield's calculus

The type system of λ_{um} is complete with respect to Dunfield's type system, meaning that all the programs that type-check in Dunfield's system, can also be encoded in λ_{um}. We prove completeness by elaborating programs that type-check in Dunfield's system to λ_{um}. The elaboration of the well-typed programs from the Dunfield's system to λ_{um} is explained next.

Elaboration to λ_{um}. The highlighted part of rules in Figure 5.6 shows the elaborated program in λ_{um}. Rules dtyp-int and dtyp-var state that an integer from the Dunfield's system elaborates to an integer, and a variable to a variable in λ_{um} respectively. Rule dtyp-abs is of significant interest. This rule elaborates an un-annotated lambda expression from Dunfield's calculus to a type annotated lambda expression in λ_{um} i.e. it elaborates $\lambda x.e$ to $\lambda x.e : A \rightarrow B$. Rule dtyp-and is also interesting. It elaborates an expression that checks against an intersection of two types into a merge of the same expressions. The case of unannotated lambda expressions is of particular interest with rule dtyp-and. The Dunfield's system has unannotated lambda expression and the typing rule dtyp-and is able to encode the following program:

$$\text{DTYP-AND}\ \frac{\overline{\lambda x.e : \mathsf{Int} \rightarrow \mathsf{Int}} \qquad \overline{\lambda x.e : \mathsf{Bool} \rightarrow \mathsf{Bool}}}{\lambda x.e : \mathsf{Int} \rightarrow \mathsf{Int} \wedge \mathsf{Bool} \rightarrow \mathsf{Bool}}$$

Whereas λ_{um} cannot encode the above program in its current form. Therefore we elaborate this program into an equivalent program in λ_{um} and preserve the completeness with respect to the Dunfield's system. The elaboration for such a program is shown next:

$$\overline{\lambda x.e : \mathsf{Int} \rightarrow \mathsf{Int} \wedge \mathsf{Bool} \rightarrow \mathsf{Bool} \rightsquigarrow \boxed{(\lambda x.e : \mathsf{Int} \rightarrow \mathsf{Int}),,(\lambda x.e : \mathsf{Bool} \rightarrow \mathsf{Bool})}}$$

The expression $\lambda x.e$ type-checks against $\mathsf{Int} \rightarrow \mathsf{Int} \wedge \mathsf{Bool} \rightarrow \mathsf{Bool}$ in the Dunfield's system and elaborates to $(\lambda x.e : \mathsf{Int} \rightarrow \mathsf{Int}),,(\lambda x.e : \mathsf{Bool} \rightarrow \mathsf{Bool})$ in λ_{um}. The elaborated program type-checks against $\mathsf{Int} \rightarrow \mathsf{Int} \wedge \mathsf{Bool} \rightarrow \mathsf{Bool}$ in λ_{um}.

$$\textsc{typ-mergea}\ \frac{\overline{\lambda x.e : \mathsf{Int} \to \mathsf{Int} : \mathsf{Int} \to \mathsf{Int}} \qquad \overline{\lambda x.e : \mathsf{Bool} \to \mathsf{Bool} : \mathsf{Bool} \to \mathsf{Bool}}}{(\lambda x.e : \mathsf{Int} \to \mathsf{Int}),,(\lambda x.e : \mathsf{Bool} \to \mathsf{Bool}) : \mathsf{Int} \to \mathsf{Int} \wedge \mathsf{Bool} \to \mathsf{Bool}}$$

Completeness. Lemma 5.11 states that if an expression e type-checks in the Dunfield's system against type A and elaborates to e' in λ_{um} then e' has type A in λ_{um}.

Lemma 5.11 (Completeness with respect to Dunfield's system). $\forall\ \Gamma\ e\ A\ e'$, *If* $\Gamma \vdash_d e : A \rightsquigarrow e'$ *then* $\Gamma \vdash e' : A$

Comparison of λ_{um} with Dunfield's calculus. The direct operational semantics proposed by Dunfield [2014] are not type preserving. Therefore Dunfield adopts an elaboration semantics in her calculus which significantly increases the theoretical complexity of the calculus. A reader has to understand both the source semantics and the target semantics to get the intuition of the programs. Moreover, elaboration itself is essential to understand. On the contrary, we propose a direct operational semantics for the source language. Direct operational semantics eliminates the need of elaboration and target language. This leaves only the source operational semantics without any intermediary steps.

5.4 Stumbling block: non-determinism

Determinism is an important property of a calculus in theory. But the calculus (λ_{um}) proposed in this chapter is non-deterministic. The non-determinism comes from two sources in λ_{um}. One source is the merge operator and the other is switch expression. We explain both of the sources next. A few proposals to deal with the non-determinism are discussed in Chapter 8.

5.4.1 Non-determinism in the presence of merge operator

Recall that λ_{um} drops the disjointness in merges and in switches. This allows to construct the merges of overlapping types such as 1,,2. When we construct a merge of 1,,2 and try to extract an integer out of it using type directed semantic, we may get either 1 or 2.

$$1,,2 \longrightarrow_{\mathsf{Int}} 1 \quad \text{rule } \textsc{cst-mergl}$$

or

$$1,,2 \longrightarrow_{\mathsf{Int}} 2 \quad \text{rule CST-MERGR}$$

The operational semantics depends upon the type casting. Therefore the evaluation of an expression to two different values results in a non-deterministic calculus.

Disjoint intersection types. Disjoint intersection types [Oliveira et al. 2016] address non-determinism in the context of intersection types and the merge operator. The disjoint intersection types forbid the construction of a merge with overlapping types. The typing rule for the merge operator with disjoint intersection types is:

$$\textsc{typ-mergb}\quad \frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B \qquad A * B}{\Gamma \vdash e_1,,e_2 : A \wedge B}$$

The third premise in the rule allows merges of only disjoint types or non-overlapping types. For example a merge of 1,,true is allowed. Whereas 1,,2 is not allowed.

5.4.2 Non-determinism in the presence of switch expression

The second source of non-determinism in λ_{um} is the type-based switch expression. For example a naive evaluation of the following program is not deterministic:

$$\mathsf{switch}\ (x : \mathsf{Int} \vee \top)\ \{(x : \mathsf{Int}) \rightarrow \mathsf{true}, (y : \top) \rightarrow \mathsf{false}\}$$

The above switch expression may evaluate either first or the second branch in a naive implementation of a switch expression. This is due to the fact that the type of scrutinee i.e 1 overlaps with both Int and $\top$. Therefore it may fall in either of the two branches. Such non-deterministic programs type-check by exploiting the following unconstrained typing rule for the switch expression:

$$\textsc{typ-switcha}\quad \frac{\Gamma \vdash e : A \vee B \qquad \Gamma, x : A \vdash e_1 : C \qquad \Gamma, y : B \vdash e_2 : C}{\Gamma \vdash \mathsf{switch}\ e\ \{(x : A) \rightarrow e_1, (y : B) \rightarrow e_2\} : C}$$

Disjoint switches. The λ_u (Chapter 3) proposes a deterministic solution for such type-based switch expressions. Similar to disjoint intersection types, λ_u allows two branches of a type-based switch expression to have only the non-overlapping types. The typing rule for a switch expression in λ_u is:

TYP-SWITCH

$$\frac{\Gamma \vdash e : A \vee B \qquad \Gamma, x : A \vdash e_1 : C \qquad \Gamma, y : B \vdash e_2 : C \qquad A * B}{\Gamma \vdash \mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} : C}$$

Note the last premise in rule TYP-SWITCH. It says that the branches of a switch expression must not have overlapping types. Therefore the following program does not type-check in λ_u because Int and $\top$ are overlapping types:

$$\mathsf{switch}\ (x : \mathsf{Int} \vee \top)\ \{(x : \mathsf{Int}) \to \mathsf{true}, (y : \top) \to \mathsf{false}\}$$

Whereas the following program type-checks because Int and Bool are non-overlapping or disjoint types:

$$\mathsf{switch}\ (x : \mathsf{Int} \vee \mathsf{Bool})\ \{(x : \mathsf{Int}) \to \mathsf{true}, (y : \mathsf{Bool}) \to \mathsf{false}\}$$

Note that the disjointness in disjoint intersection types and disjoint switches diverge. Two types are disjoint in disjoint intersection types when they do not share a common supertype. Whereas two types are disjoint in disjoint switches when they do not share a common subtype.

5.4.3 Non-determinism with merge operator and switch expression

While the non-determinism for merges and switches has been studied separately, combining disjoint intersection types and disjoint switches is non-trivial. The integration of disjoint intersection types and disjoint switches poses novel challenges.

$$(\lambda x.(\mathsf{switch}\ x\ \{(x : \mathsf{Int}) \to \mathsf{true}, (y : \mathsf{Bool}) \to \mathsf{false}\}) : \mathsf{Int} \vee \mathsf{Bool} \to \mathsf{Bool})\ (1,,\mathsf{true})$$

The above program can be encoded in λ_{um}. It is an application of a lambda expression i.e $(\lambda x.(\mathsf{switch}\ x\ \{(x : \mathsf{Int}) \to \mathsf{true}, (y : \mathsf{Bool}) \to \mathsf{false}\}) : \mathsf{Int} \vee \mathsf{Bool} \to \mathsf{Bool})$ to a merge i.e $(1,,\mathsf{true})$. The input type of lambda expression is Int $\vee$ Bool. It is safe to pass a term of type Int $\wedge$ Bool as Int $\vee$ Bool using the subsumption rule.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash 1 : \mathsf{Int}} \qquad \dfrac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}}}{\Gamma \vdash 1,,\mathsf{true} : \mathsf{Int} \wedge \mathsf{Bool}}\ \textsc{typ-mergeA} \qquad \dfrac{\dfrac{\dfrac{}{\mathsf{Int} <: \mathsf{Int}}}{\mathsf{Int} <: \mathsf{Int} \vee \mathsf{Bool}}\ \textsc{s-orB}}{\mathsf{Int} \wedge \mathsf{Bool} <: \mathsf{Int} \vee \mathsf{Bool}}\ \textsc{s-andB}}{\Gamma \vdash 1,,\mathsf{true} : \mathsf{Int} \vee \mathsf{Bool}}\ \textsc{typ-sub}$$

In a more human readable language the same program can be written as:

```
Bool isInt (x : Int | Bool) = switch (x)
                                (x:Int)  → true
                                (y:Bool) → false

isInt(1,,true)
```

Notice that the merge independently is deterministic i.e Int and Bool are disjoint types. The switch independently is also deterministic i.e Int and Bool are disjoint types. But the merge value 1,,true can fall either in first branch or in second branch. This is because the value of type $\mathsf{Int} \wedge \mathsf{Bool}$ can safely be used as value of type Int and Bool at the same time.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash 1 : \mathsf{Int}} \quad \dfrac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}}}{\Gamma \vdash 1,,\mathsf{true} : \mathsf{Int} \wedge \mathsf{Bool}}\ \textsc{typ-mergeA} \qquad \dfrac{\dfrac{}{\mathsf{Int} <: \mathsf{Int}}}{\mathsf{Int} \wedge \mathsf{Bool} <: \mathsf{Int}}\ \textsc{s-andB}}{\Gamma \vdash 1,,\mathsf{true} : \mathsf{Int}}\ \textsc{typ-sub}$$

and so as:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash 1 : \mathsf{Int}} \quad \dfrac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}}}{\Gamma \vdash 1,,\mathsf{true} : \mathsf{Int} \wedge \mathsf{Bool}}\ \textsc{typ-mergeA} \qquad \dfrac{\dfrac{}{\mathsf{Bool} <: \mathsf{Bool}}}{\mathsf{Int} \wedge \mathsf{Bool} <: \mathsf{Bool}}\ \textsc{s-andC}}{\Gamma \vdash 1,,\mathsf{true} : \mathsf{Bool}}\ \textsc{typ-sub}$$

Therefore a naive composition of disjoint intersection types and disjoint switches leads to non-determinism. We discuss a few proposals to deal with non-determinism in Chapter 8.

Part III

Duality of Intersection and Union Types

6 The Duality of Subtyping (Duotyping)

This chapter examines the duality of subtyping with intersection and union types. Duality is a common concept in logic and many features and their duals have been studied. For example, disjunctions are dual to conjunctions, universal quantifiers are dual to existential quantifiers. Surprisingly, duality has not formally been studied in the context of subtyping in programming languages. We start with an overview of duality in subtyping, then we discuss several calculi with duality and traditional subtyping. Finally, we conclude with the help of case studies that by employing duality in subtyping one can achieve significant benefits over traditional subtyping.

6.1 Overview

This section gives an overview of Duotyping. We show how to design subtyping relations employing Duotyping, and discuss the advantages of a design with Duotyping instead of a traditional subtyping formulation in more detail.

6.1.1 Subtyping with union and intersection types

To motivate the design of Duotyping relations we first consider a traditional subtyping relation with union and intersection types, as well as top and bottom types. We choose a system with union and intersection types because these features are nowadays common in various OOP languages, including Scala [Odersky et al. 2004], TypeScript [Bierman et al. 2014], Ceylon [King 2013] or Flow [Chaudhuri 2015]. Therefore union and intersection types are of practical interest. Furthermore union and intersection types are simple, intuitive and good for showing duality between concepts.

The types used for the subtyping relation include the top type $\top$, the bottom type $\bot$, integer type Int, function type $A \to B$, intersection type $A \wedge B$ and the union type $A \vee B$:

$$\text{Types} \quad A, B \quad ::= \quad \top \mid \bot \mid \mathsf{Int} \mid A \to B \mid A \wedge B \mid A \vee B$$

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

$$\frac{}{A <: \top}\ \text{S-TOP} \qquad \frac{}{\bot <: A}\ \text{S-BOT} \qquad \frac{}{\mathsf{Int} <: \mathsf{Int}}\ \text{S-INT} \qquad \frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2}\ \text{S-ARROW} \qquad \frac{A <: B \qquad A <: C}{A <: B \wedge C}\ \text{S-ANDA}$$

$$\frac{A <: C}{A \wedge B <: C}\ \text{S-ANDB} \qquad \frac{B <: C}{A \wedge B <: C}\ \text{S-ANDC} \qquad \frac{A <: C \qquad B <: C}{A \vee B <: C}\ \text{S-ORA} \qquad \frac{A <: B}{A <: B \vee C}\ \text{S-ORB} \qquad \frac{A <: C}{A <: B \vee C}\ \text{S-ORC}$$

Figure 6.1: Subtyping for union and intersection types.

TRADITIONAL SUBTYPING. A simple subtyping relation accounting for union and intersection types is given in Figure 6.1. Rule S-TOP defines that every type is a subtype of $\top$, and rule S-BOT states that every type is a supertype of $\bot$. Rule S-INT is for integers, and states that Int is a subtype of itself. Rule S-ARROW is the traditional subtyping rule for function types. Rules S-ANDA, S-ANDB, and S-ANDC are subtyping rules for intersection types. Rules S-ORA, S-ORB, and S-ORC are subtyping rules for union types. The rules that we employ here are quite common for systems with union and intersection types. For instance they are the same rules used in various DOT-calculi [Amin et al. 2012] (which model the essence of Scala). For simplicity we do not account for distributivity rules, which also appear in some type systems and calculi [Barendregt et al. 1983; Bessai et al. 2015; Zappa Nardelli et al. 2018].

6.1.2 Subtyping Specifications using Duotyping

In the subtyping relation presented in Figure 6.1, it is quite obvious that many rules look alike. Some rules are essentially a "mirror image" of other rules. The rules for top and bottom types are an example of this. Another example are the rules S-ANDB and S-ORB. Although informally humans can easily observe the similarity between many of the rules, this similarity/duality is not expressed directly in the formalism. For example, there is nothing preventing us from designing rules that are not duals. Duotyping aims at capturing duality in the rules themselves, and expressing duality as part of the formalism, rather than just leaving duality informally observable by humans. This can prevent, for instance, designing rules for dual concepts that do not really dualize. Therefore Duotyping can enforce consistency of dual rule designs.

To illustrate how Duotyping rules are designed and relate to the traditional subtyping rules, lets refactor the traditional rules in a few basic steps. Firstly, lets assume that we have a second relation $A :> B$ that captures the *supertyping* between a type A and B. Supertyping

is nothing but the subtyping relation with its arguments flipped. So, the rules of supertyping could be simply obtained by taking all the rules in Figure 6.1 and deriving corresponding rules where all the arguments are flipped around. We skip that boring definition here. With both supertyping and subtyping, the top and bottom rules can be presented as follows:

$$\frac{}{A <: \top}\ \textsc{s-top} \qquad \frac{}{A :> \bot}\ \textsc{sup-bot}$$

Similarly the rules rule s-andb and rule s-orb, can be presented as:

$$\frac{A <: C}{A \wedge B <: C}\ \textsc{s-andb} \qquad \frac{A :> C}{A \vee B :> C}\ \textsc{sup-orb}$$

This simple refactoring shows that the only difference between dual rules is the relation itself, and the (dual) language constructs. Apart from that everything else is the same.

Duotyping. With Duotyping we can provide a single unified rule, which captures the two distinct subtyping rules, instead. The Duotyping relation is parameterized by a mode $\Diamond$:

$$\text{Mode} \quad \Diamond \quad ::= \quad <: \mid :>$$

which can be subtyping ($<:$) or supertyping ($:>$). Thus the Duotyping relation is of the form:

$$A \Diamond B$$

The mode $\Diamond$ is a (third) *parameter* of the relation (besides A and B). With this mode in place, we can readily capture the two refactored rules for supertyping of bottom types and subtyping of top types as two Duotyping rules. However this still requires us to write two distinct rules. To unify those rules into a single one, we introduce a function $\rceil\Diamond\lceil$ that chooses the right bound depending on the mode being used:

$$\begin{aligned} \rceil<:\lceil &= \top \\ \rceil:>\lceil &= \bot \end{aligned}$$

If the mode is subtyping the upper bound of the relation is the top type, otherwise it is the bottom type. With $\rceil\Diamond\lceil$ we can then write a single unified rule that captures the upper bounds of subtyping and supertyping, and which generalizes both rule s-top and rule sup-bot:

$$\frac{}{A \Diamond \rceil\Diamond\lceil}\ \textsc{gds-topbot}$$

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

$$\frac{}{A \Diamond \rceil \Diamond \lceil}\ \text{GDS-TOPBOT} \qquad \frac{}{\mathsf{Int} \Diamond \mathsf{Int}}\ \text{GDS-INT} \qquad \frac{A_1 \overline{\Diamond} A_2 \qquad B_1 \Diamond B_2}{A_1 \to B_1 \Diamond A_2 \to B_2}\ \text{GDS-ARROW} \qquad \frac{A \Diamond C}{(A \Diamond_? B) \Diamond C}\ \text{GDS-LLEFT} \qquad \frac{B \Diamond C}{(A \Diamond_? B) \Diamond C}\ \text{GDS-RRIGHT}$$

$$\frac{A \Diamond B \qquad A \Diamond C}{A \Diamond (B \Diamond_? C)}\ \text{GDS-BOTH} \qquad \frac{B \overline{\Diamond} A}{A \Diamond B}\ \text{GDS-DUAL}$$

Figure 6.2: The Duotyping relation for a calculus with union and intersection types.

The Duality rule. The Duotyping rule above captures the 2 rules that were refactored above. However there are 4 rules in total for top and bottom types (two for subtyping and two for supertyping). The two missing rules are:

$$\frac{}{\bot <: A}\ \text{S-BOT} \qquad \frac{}{\top :> A}\ \text{SUP-TOP}$$

To capture these missing rules, the Duotyping relation includes a special duality rule:

$$\frac{B \overline{\Diamond} A}{A \Diamond B}\ \text{GDS-DUAL}$$

which simply inverts the mode and flips the arguments of the relation. The definition of $\overline{\Diamond}$ is, unsurprisingly:

$$\overline{<:} \; = \; :>$$
$$\overline{:>} \; = \; <:$$

With the duality rule it is clear that the two missing rules are now derivable from the Duotyping rule for bounds and the duality rule. In essence this is the overall idea of the design of Duotyping rules.

Complete set of rules. Figure 6.2 shows the complete version of declarative Duotyping rules for a system with union and intersection types. Rule GDS-TOPBOT defines the rule bounds (which generalizes the rules for top and bottom types). Rule GDS-INT is a simple rule for integers. Int is subtype and supertype of Int. Rule GDS-ARROW is an interesting case. In the first premise $A \overline{\Diamond} B$ we invert the mode instead of flipping the arguments of the relation,

as done in rule S-ARROW. One side-effect of this change is that it keeps the rule fully *covariant*, which contrasts with subtyping relations where for arrow types we need contravariance for subtyping of the inputs. This apparently innocent change has important consequences and plays a fundamental role to simplify transitivity proofs as we shall see in Section 6.1.5.

Rules GDS-LLEFT, GDS-RRIGHT, and GDS-BOTH each generalize two rules in the traditional formulation of subtyping. Rule GDS-LLEFT generalizes rules S-ANDB and S-ORB. Rule GDS-RRIGHT generalizes rules S-ANDC and S-ORC. Rule GDS-BOTH generalizes rules S-ANDA and S-ORA. In the three rules an operation $A \Diamond_? B$ is used:

$$\begin{aligned} A <:_? B &= A \wedge B \\ A :>_? B &= A \vee B \end{aligned}$$

This operation is used to choose between intersection or union types depending on the mode. If the Duotyping mode is subtyping then we get a rule for intersection types, otherwise we get a dual rule for union types.

UNIFORM AND DUAL RULES. In the context of Duotyping it is useful to distinguish between two different kinds of rules: uniform rules and dual rules.

Uniform rules are those that are essentially the same for supertyping and subtyping. Rules GDS-INT and GDS-ARROW are uniform rules. In those rules the arguments of the relation are exactly the same no matter which mode is being used (subtyping or supertyping).

Dual rules are those that employ dual constructs, like the rules for top and bottom or the rules for union and intersections. Rules GDS-TOPBOT, GDS-LLEFT, GDS-RRIGHT, and GDS-BOTH are dual rules. The interesting point in these rules is that they use different (dual) constructs depending on the mode. For example, when instantiated with subtyping and supertyping, respectively, the rule GDS-TOPBOT results in:

$$\frac{}{A <: \top} \qquad\qquad \frac{}{A :> \bot}$$

6.1.3 IMPLEMENTATIONS USING DUOTYPING

Fig. 6.2 showed the declarative Duotyping rules for a calculus with union and intersection types. All the rules are syntax directed, except for the duality rule (rule GDS-DUAL). This rule flips the mode and arguments to generate a formulation using the dual mode: i.e. it flips subtyping to provide the equivalent supertyping formulation and vice versa. A benefit of using a formulation with the duality rule is that it enables a short specification of Duotyping.

```
data Op = And | Or
data Typ = TInt | TArrow Typ Typ | TOp Op Typ Typ | TBot | TTop
data Mode = Sub | Sup

duo :: Bool -> Mode -> Typ -> Typ -> Bool
duo f m TInt TInt                        = True
duo f m _ b | b == mode_to_sub m         = True
duo f m (TArrow a b) (TArrow c d)        =
   duo True (flip m) a c && duo True m b d
duo f m (TOp op a b) c | choose m == op  = duo True m a c || duo True m b c
duo f m a (TOp op b c) | choose m == op  = duo True m a b && duo True m a c
duo True m a b                           = duo False (flip m) b a
duo _ _ _ _                              = False
```

Figure 6.3: Haskell code for implementing an algorithmic formulation of Duotyping rules.

Unfortunately the duality rule is *not algorithmic*, because the duality rule can always be applied indefinitely. In other words naively translating the rules into an program would easily result in a non-terminating procedure. Therefore to obtain an algorithmic formulation some additional work is needed.

Fortunately, for declarative formulations of Duotyping, there is a simple technique that can be used to obtain an algorithmic formulation. A key observation is that Duotyping only needs to be flipped (with the duality rule) *at most one time*. Flipping the relation two or more times simply gets us back to the starting point. To capture this idea we can use a (boolean) flag that keeps track of whether the procedure has already employed the duality rule or not.

To make such an idea concrete, Figure 6.3 shows Haskell code that implements a procedure `duo` for determining Duotyping for two types. The code is based on the rules in Fig. 6.2, but it uses a boolean flag to prevent the dual rule (the second to last case in `duo`) from being applied indefinitely. The boolean is true in the initial call or recursive calls to structurally smaller arguments. If the algorithm fails for the first five cases (which are basically a direct translation of the rules GDS-TOPBOT, GDS-INT, GDS-ARROW, GDS-LLEFT, and GDS-RRIGHT), then the algorithm simply flips the boolean flag, mode and arguments to run over a dual formulation. This is the second to last line of the algorithm.

For example, if the algorithm is called with the mode set to subtyping and it is not able to find any matching case with the first 5 rules, then it flips the boolean flag to `False`, subtyping to supertyping and the arguments to check the equivalent supertyping formulation. If again it fails to find a matching rule, `False` will be returned and the algorithm will terminate. This illustrates that it is enough to flip the boolean flag once to exploit Duotyping. In all our Coq formulations of Duotyping we have developed an alternative algorithmic formulation

of Duotyping which uses an extra boolean flag and is shown to be sound and complete to the declarative formulations with the duality rule. In short there is an easy, general and provably *sound* and *complete* way to implement algorithms based on the idea of Duotyping, while at the same time retaining the benefits of reuse of the logic for rules for dual constructs.

6.1.4 Discovering new features

Duotyping can provide interesting extra features essentially for free. For example, the hallmark feature of the well-known $F_{<:}$ calculus (a polymorphic calculus with subtyping) [Canning et al. 1989] is *bounded quantification*, which is a feature used in most modern OOP languages (such as Scala or Java). In $F_{<:}$, bounded quantification allows type variables to be defined with *upper bounds*. For example, the following Scala program illustrates the use of such upper bounds:

```
class Person {
  def name: String = "person"
}

class Student extends Person {
  override def name: String = "student"
  def id: String = "id"
}

class StudentsCollection[S <: Student](obj: S) {
  def student: S = obj
}
```

The Scala program shown above uses the upper bounds for the class *StudentsCollection* written as S <: Student. This upper bound restricts *StudentsCollection* to be instantiated with *Student* and its subtypes. Since the upper bound is *Student*, any class that is supertype of *Student* like *Person* cannot be instantiated in *StudentsCollection*.
However *lower bounds* are also useful, and indeed the Scala language allows them (though Java does not). One example of a program with lower bounds in Scala is:

```
class GraduateStudent extends  Student {
  def degree: String = "graduate degree"
}

class ResearchStudent extends GraduateStudent {
  override def degree: String = "research degree"
}

class CollectionExcludingResearchStudents[S >: GraduateStudent](obj: S) {
```

```
  def student: S = obj
}
```

In contrast to the upper bounds, the Scala program shown above uses the lower bounds for the class *CollectionExcludingResearchStudents* written as S >: GraduateStudent. This lower bound restricts *CollectionExcludingResearchStudents* to be instantiated with *GraduateStudent* and its supertypes. Since the lower bound is *GraduateStudent*. Any class that is a subtype of *GraduateStudent* (such as *ResearchStudent*) cannot be instantiated in *CollectionExcludingResearchStudents*. But any supertype of *GraduateStudent* like *Student* and *Person* (including *GraduateStudent*) can be instantiated in *CollectionExcludingResearchStudents*.

One can think of universal quantification with lower bounds as a dual to universal quantification with upper bounds. While there is no extension of $F_{<:}$ that we know of that presents universal quantification with lower bounds in the literature, applying a Duotyping design to $F_{<:}$ gives us, naturally, the two features at once (lower and upper bounded quantification).

Bounded quantification in $F_{<:}$. The traditional subtyping rule of System kernel $F_{<:}$ with upper bounded quantification is:

S-FORALLKFS

$$\frac{\Gamma, \alpha <: A \vdash B <: C}{\Gamma \vdash (\forall \alpha <: A.B) <: (\forall \alpha <: A.C)}$$

In the premise of this rule, we add the type variable α to the context with an upper bound A. If under the extended context the bodies of the universal quantifier (B and C) are in a subtyping relation then the universal quantifiers are also in a subtyping relation.

To add lower bounded quantification the obvious idea is to add a second rule:

S-FORALLKFSB

$$\frac{\Gamma, \alpha :> A \vdash B <: C}{\Gamma \vdash (\forall \alpha :> A.B) <: (\forall \alpha :> A.C)}$$

However this alone is not quite right because the environment is also extended with a lower bound ($X >: A$), which does not exist in $F_{<:}$ contexts. Therefore some additional care is also needed for the variable cases of $F_{<:}$ extended with lower bounded quantification. When an upper bounded constraint is found in the environment, the variable case needs to deal with the upper bound appropriately. Since there are two rules dealing with the variable case in $F_{<:}$, one possible approach is to add two more rules for dealing with upper bounds:

$$\frac{\alpha :> A \in \Gamma \qquad \Gamma \vdash B <: A}{\Gamma \vdash B <: \alpha}\ \textsc{s-TVarB} \qquad \frac{\alpha :> A \in \Gamma}{\Gamma \vdash \alpha <: \alpha}\ \textsc{s-ReflTvar}$$

However such a design feels a little unsatisfactory. We need a total of 6 rules to fully deal with lower and upper bounded quantification (instead of 3 rules in $F_{<:}$). At the same time the rules are nearly identical, differing only on the kind of bounds that is used. Furthermore the metatheory of $F_{<:}$ also needs to be significantly changed. In particular *narrowing* has to be adapted to account for the lower bounds and *transitivity* has to be extended with several new cases. Since both *narrowing* and *transitivity* proofs for $F_{<:}$ are non-trivial, this extension is also non-trivial and adds further complexity to already complex proofs.

A VARIANT OF KERNEL $F_{<:}$ WITH DUOTYPING. We now reconsider the design of Kernel $F_{<:}$ from scratch employing the Duotyping methodology. In the subtyping rule for universal quantification, it is important to note that the subtyping relation between two universal quantifiers in the conclusion is the same as the relation between types B and C in premise. Similarly, the (subtyping/supertyping) bounds of type variable α in the conclusion are the same as the bounds of type variable α in premise. In a design with Duotyping, we would like to generalize the two uses of subtyping. Therefore, we can design a single unified rule with the help of two modes:

$$\frac{\Gamma, \alpha \lozenge_1 A \vdash B \lozenge_2 C}{\Gamma \vdash (\forall \alpha \lozenge_1 A.B) \lozenge_2 (\forall \alpha \lozenge_1 A.C)}\ \textsc{gs-forallkfs}$$

Section 6.3.1 explains the Duotyping rules of our Duotyping kernel $F_{<:}$ variant with union and intersection types $F_{k\lozenge}^{\wedge\vee}$ in detail. Rule GS-FORALLKFS is the interesting case, capturing both upper and lower bounded quantification in an elegant way. This rule states that if in a well-formed context, type variable α has a $\lozenge_1$ relation with type A and if type B has $\lozenge_2$ relation with type C, then the universal quantification with body B has a $\lozenge_2$ relation with the universal quantification with body C. Correspondingly there are also two Duotyping rules for variables:

$$\frac{\alpha \lozenge A \in \Gamma \qquad \Gamma \vdash A \lozenge B}{\Gamma \vdash \alpha \lozenge B}\ \textsc{gs-TVarA} \qquad \frac{\alpha \lozenge_1 A \in \Gamma}{\Gamma \vdash \alpha \lozenge \alpha}\ \textsc{gs-ReflTvar}$$

In short, the design of a variant of $F_{<:}$ with Duotyping leads to a system that naturally accounts for both upper and lower bounded quantification. Moreover, the metatheory, and

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

$$\frac{}{A \Diamond \rceil \Diamond \lceil}\ \text{GS-TOPBTMA} \qquad \frac{}{\rceil \overline{\Diamond} \lceil \Diamond A}\ \text{GS-TOPBTMB} \qquad \frac{}{\mathsf{Int} \Diamond \mathsf{Int}}\ \text{GS-INT} \qquad \frac{A_1 \overline{\Diamond} A_2 \qquad B_1 \Diamond B_2}{A_1 \to B_1 \Diamond A_2 \to B_2}\ \text{GS-ARROW}$$

Figure 6.4: The Duotyping relation for simply typed lambda calculus calculus.

in particular the proofs of *narrowing* and *transitivity* are not more complex than the corresponding original $F_{<:}$ proofs. In fact the proof of transitivity is significantly simpler, because Duotyping enables novel proof techniques as we discuss next.

6.1.5 New proof techniques

Transitivity proofs are usually a challenge for systems with subtyping. This is partly because subtyping relations often need to deal with some *contravariance*. For instance, the rule S-ARROW (in Figure 6.1) is contravariant on the input types. Such contravariance causes problems in certain proofs, including transitivity. To illustrate the issue more concretely, let's distill the essence of the problem by considering a simple lambda calculus with subtyping called $\lambda_{<:}$, where the types are:

Types	A, B	$::=$	$\top \mid \mathsf{Int} \mid A \to B$

and the subtyping rules for those types are just the relevant subset of the rules in Figure 6.1. The transitivity proof for this simple calculus is:

Lemma 6.1 ($\lambda_{<:}$Transitivity)**.** *If $A <: B$ and $B <: C$ then $A <: C$.*

Proof. By induction on type B.

- Case $\top$ and case Int are trivial to prove by destructing the hypothesis in context.
- Case $B_1 \to B_2$ requires inversion of the two hypotheses to discover that A can only be a function type, while C is either a function type or $\top$.

□

In the arrow case, we need to invert both hypotheses to discover more information about A and C. For this very simply language this double inversion is not too problematic, but as the language of types grows and the subtyping relation becomes more complicated, such inversions become significantly harder to deal with.

At this point one may wonder if the transitivity proof could be done using a different inductive argument to start with, and thus avoid the double inversions. After all there are various other possible choices. Perhaps the most obvious choice is to try induction on the subtyping relation itself ($A <: B$), rather than on type B. However this does not work because of the contravariance for arrow types, which renders one of induction hypothesis in the arrow case useless (and thus do not allow the case to be proved). Other alternative choices for an inductive argument (such as type A or C) do not work for similar reasons.

Developing metatheory with Duotyping. In order to develop metatheory with Duotyping it is convenient to use an equivalent formulation of Duotyping that eliminates the duality rule (which is non-algorithmic and makes inversions more difficult). For $\lambda_{\Diamond}$, which is a Duotyping version of $\lambda_{<:}$, this would lead to the set of rules in Figure 6.4. This alternative algorithmic version eliminates the duality rule. Rules GS-TOPBTMA, GS-INT, and GS-ARROW are similar to the rules we discussed in Section 6.1.2. Rule GS-TOPBTMB is the dual rule of rule GS-TOPBTMA. With Rule GS-TOPBTMB, the duality rule is unnecessary.

Transitivity with Duotyping. Now we turn our attention to the proof of transitivity:

Lemma 6.2 ($\lambda_{\Diamond}$Transitivity)**.** *If $A \Diamond B$ and $B \Diamond C$ then $A \Diamond C$.*

Proof. By induction on $A \Diamond B$.

- All cases are trivial to prove by destructing $\Diamond$ and inversion of the second hypothesis ($B \Diamond C$).

□

Transitivity of systems with Duotyping can often be proved by induction on the subtyping relation itself. This has the nice advantage that all the cases essentially become trivial to prove (for $\lambda_{\Diamond}$) and only a single inversion is needed for arrow types. A key reason why such approach works in the formulation with Duotyping is that we can keep case for arrow types covariant. Instead we only flip the mode. Another important observation is that when we prove a transitivity lemma with Duotyping we are, in fact, proving two lemmas simultaneously: one lemma for transitivity of subtyping, and another one for transitivity of supertyping. When we use the induction hypothesis we have access to both lemmas (by choosing the appropriate mode).

The proof of the transitivity lemma by induction on the Duotyping relation can scale up to more complex subtyping/Duotyping relations. This includes subtyping relations with advanced features such as intersection types, union types, parametric polymorphism and

Types	A, B	$::=$	$\top \mid \bot \mid \mathsf{Int} \mid A \rightarrow B \mid A \wedge B \mid A \vee B$
Terms	e	$::=$	$x \mid i \mid \lambda x : A.e \mid e_1\, e_2$
Values	v	$::=$	$i \mid \lambda x : A.e$
Context	Γ	$::=$	$\cdot \mid \Gamma, x : A$
Mode	$\Diamond$	$::=$	$<: \mid :>$

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

$$\frac{}{A \Diamond \rceil \Diamond \lceil}\ \textsc{gs-TopBtmA} \qquad \frac{}{\rceil \overline{\Diamond} \lceil \Diamond A}\ \textsc{gs-TopBtmB} \qquad \frac{}{\mathsf{Int} \Diamond \mathsf{Int}}\ \textsc{gs-Int}$$

$$\frac{A_1 \overline{\Diamond} A_2 \qquad B_1 \Diamond B_2}{A_1 \rightarrow B_1 \Diamond A_2 \rightarrow B_2}\ \textsc{gs-Arrow} \qquad \frac{A \Diamond C}{(A \Diamond_? B) \Diamond C}\ \textsc{gs-LeftA}$$

$$\frac{A \Diamond B}{A \Diamond (B \overline{\Diamond}_? C)}\ \textsc{gs-LeftB} \qquad \frac{B \Diamond C}{(A \Diamond_? B) \Diamond C}\ \textsc{gs-RightA} \qquad \frac{A \Diamond C}{A \Diamond (B \overline{\Diamond}_? C)}\ \textsc{gs-RightB}$$

$$\frac{A \Diamond B \qquad A \Diamond C}{A \Diamond (B \Diamond_? C)}\ \textsc{gs-BothA} \qquad \frac{A \Diamond C \qquad B \Diamond C}{(A \overline{\Diamond}_? B) \Diamond C}\ \textsc{gs-BothB}$$

Figure 6.5: Syntax and Duotyping relation for union and intersection types.

bounded quantification. All of these can follow the same strategy (induction on the Duotyping relation) to simplify the transitivity proof, as we shall see in Section 6.4.

6.2 The $\lambda_{\Diamond}^{\wedge\vee}$ calculus

In Section 6.1 we gave an overview and discussed advantages of using the Duotyping relation. In this section we introduce a lambda calculus with union and intersection types that is based on Duotyping. We aim at showing that developing calculi and metatheory using Duotyping is simple, requiring only a few small adaptations compared with more traditional formulations based on subtyping. Our main aim is to show type soundness (subject-reduction and preservation) for $\lambda_{\Diamond}^{\wedge\vee}$.

6.2.1 Syntax and Duotyping

Syntax. Fig. 6.5 shows the syntax of the calculus. The types for $\lambda_{\Diamond}^{\wedge\vee}$ were already introduced in Section 6.1. Terms include all the constructs for the lambda calculus (variables x, functions $\lambda x : A.e$ and applications $e_1\, e_2$) and integers (i). Values are a subset of terms, consisting of abstractions and integers only. The mode $\Diamond$ is used to choose the mode of the relation: it can be either subtyping ($<:$) or supertyping ($:>$). Typing contexts Γ are standard and used to track the types of the variables in a program. Finally, a well-formedness relation $\Gamma \vdash ok$ ensures that typing contexts are well-formed.

DUOTYPING FOR $\lambda_{\Diamond}^{\wedge\vee}$. The Duotyping rules for $\lambda_{\Diamond}^{\wedge\vee}$ were already partly presented in Figure 6.2. In addition to the rules in the $\lambda_{\Diamond}$, we also need extra rules for union and intersection types. These extra rules are presented in Figure 6.5. Rules GS-LEFTA, GS-RIGHTA, and GS-BOTHA are also similar to the rules GDS-LLEFT, GDS-RRIGHT, and GDS-BOTH presented in Figure 6.2. Since we eliminate the *duality rule* in the algorithmic version, we add dual subtyping rules. Rules GS-LEFTB, GS-RIGHTB, and GS-BOTHB are the dual versions of rules GS-LEFTA, GS-RIGHTA, and GS-BOTHA respectively. This formulation is shown to be sound and complete with respect to the formulation with the duality rule in Figure 6.2. As explained in Section 6.1 this variant of the rules makes some proofs easier, thus we employ it here. The Duotyping relation is reflexive and transitive:

Theorem 6.3 (Reflexivity)**.** $A \Diamond A$.

Proof. By induction on type A. Reflexivity is trivial to prove by applying subtyping rules. □

Theorem 6.4 (Transitivity)**.** *If $A \Diamond B$ and $B \Diamond C$ then $A \Diamond C$.*

Proof. By induction on subtyping relation.

- Cases rule GS-TOPBTMA, rule GS-INT, rule GS-LEFTA, rule GS-RIGHTA and rule GS-BOTHA are trivial to prove.
- Case rule GS-TOPBTMB requires an additional Lemma 6.5.
- Case rule GS-ARROW requires induction on hypothesis and subtyping rules.
- Cases rule GS-LEFTB and rule GS-RIGHTB requires an additional Lemma 6.6 to be applied on hypothesis in context.
- Case rule GS-BOTHB requires induction on the hypothesis. This case also requires rule GS-LEFTB, rule GS-RIGHTB, and rule GS-BOTHA subtyping rules.

□

We used the following auxiliary lemmas to prove transitivity.

Lemma 6.5 (Bound Selection)**.** *If $\rceil \Diamond \lceil \Diamond B$ then $A \Diamond B$.*

This lemma captures the upper and lower bounds with respect to relation between two types. If the mode is subtyping, then it states that any type that is supertype of $\top$ is supertype of all the other types. If the mode is supertyping, then it states that any type that is subtype of $\bot$ is subtype of all the other types. In essence the lemma generalizes the following two lemmas (defined directly over subtyping and supertyping):

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

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsc{typ-var} \qquad \frac{}{\Gamma \vdash i : \mathsf{Int}}\ \textsc{typ-int} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B}\ \textsc{typ-abs} \qquad \frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B}\ \textsc{typ-app}$$

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

$\boxed{e_1 \longrightarrow e_2}$ *(Reduction)*

$$\frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \textsc{step-appl} \qquad \frac{e \longrightarrow e'}{v\, e \longrightarrow v\, e'}\ \textsc{step-appr} \qquad \frac{}{(\lambda x : A.e)\, v \longrightarrow e[x \rightsquigarrow v]}\ \textsc{step-dbeta}$$

Figure 6.6: Typing and reduction for $\lambda_{\Diamond}^{\wedge\vee}$.

- If $\top <: B$ then $A <: B$
- If $\bot :> B$ then $A :> B$

Lemma 6.6 (Inversion for rule GDS-Both)**.** *If $C \Diamond (A \Diamond_? B)$ then $(C \Diamond A)$ and $(C \Diamond B)$.*

This lemma captures the relation between types with respect to the duality of union and intersection types. It is the general form of two lemmas:

Lemma 6.7 (Inversion for Union types)**.** *If $(A \vee B) <: C$ then $(A <: C)$ and $(B <: C)$.*

Lemma 6.8 (Inversion for Intersection types)**.** *If $C <: (A \wedge B)$ then $(C <: A)$ and $(C <: B)$.*

Finally there is also a *duality lemma*, which complements reflexivity and transitivity:

Lemma 6.9 (Duality)**.** *$A \Diamond B = B \overline{\Diamond} A$.*

This lemma captures the essence of duality, and enables us to switch the mode of the relation by flipping the arguments as well. Furthermore, the duality lemma plays a crucial role when proving soundness and completeness with respect to the declarative version of Duotyping, which has duality as an axiom instead. All of these lemmas are used in later proofs for type soundness.

6.2.2 Semantics and type soundness

Typing. The first part of Fig. 6.6 presents the typing rules of $\lambda_{\Diamond}^{\wedge\vee}$. The rules are standard. Note that rule G-SUB is the subsumption rule: if an expression e has type B and B is a subtype

of A then e has type A. Noteworthy, $B <: A$ is the Duotyping relation being used with the subtyping mode.

Reduction. At the bottom of Fig. 6.6 we show the reduction rules of $\lambda_{\Diamond}^{\wedge\vee}$. Again, the reduction rules are standard. Rule step-dbeta is the usual beta-reduction rule, which substitutes a value v for x in the lambda body e. Rule step-appl and rule step-appr are the standard call-by-value rules for applications.

Type soundness. The proof for type soundness relies on the usual preservation and progress lemmas:

Lemma 6.10 (Type Preservation)**.** *If* $\Gamma \vdash e : A$ *and* $e \longrightarrow e'$ *then:* $\Gamma \vdash e' : A$.

Proof. By induction on the typing relation and with the help of Lemma 6.9. □

Lemma 6.11 (Progress)**.** *If* $\Gamma \vdash e : A$ *then:*

1. *either e is a value.*
2. *or e can take a step to* e'.

Proof. By induction on the typing relation. □

6.2.3 Summary and Comparison

Besides $\lambda_{\Diamond}^{\wedge\vee}$, which employs the Duotyping relation, we have also formalized a lambda calculus with union and intersection types using the traditional subtyping relation ($\lambda_{<:}^{\wedge\vee}$). Most of the metatheory is similar with a great deal of theorems being almost the same. The main differences are in the metatheory for subtyping which has to be generalized. For example both reflexivity and transitivity have to be generalized to operate in the Duotyping relation instead. The formalization with Duotyping only has two additional lemmas (the duality lemma and the bound selection lemma), which have no counterparts with subtyping. The number of lines of code for the formalization of $\lambda_{<:}^{\wedge\vee}$ is 596 whereas for $\lambda_{\Diamond}^{\wedge\vee}$ is 630. The total number of lemmas required for $\lambda_{<:}^{\wedge\vee}$ are 23 and 25 for $\lambda_{\Diamond}^{\wedge\vee}$. Following two lemmas in $\lambda_{<:}^{\wedge\vee}$ are captured as one lemma in $\lambda_{\Diamond}^{\wedge\vee}$ (Lemma 6.6):

INVERSION FOR UNION TYPES. This lemma is stated as Lemma 6.12: it is the inversion of the subtyping rule for the union types in the traditional subtyping relation. The lemma states that if the union of two types A and B is the subtype of a type C, then both types A and B are subtypes of type C.

Lemma 6.12 (Inversion for Union types)**.** *If $(A \vee B) <: C$ then $(A <: C)$ and $(B <: C)$.*

INVERSION FOR INTERSECTION TYPES. This lemma, which corresponds to Lemma 6.13, is the inversion of the subtyping rule for the intersection types with the traditional subtyping relation. It states that if a type C is the subtype of the intersection of two types A and B, then the type C is a subtype of both types A and B.

Lemma 6.13 (Inversion for Intersection types)**.** *If $C <: (A \wedge B)$ then $(C <: A)$ and $(C <: B)$.*

6.3 THE $F_{k\Diamond}^{\wedge\vee}$ CALCULUS

In Section 6.2 we introduced a simple calculus with union and intersection types using Duotyping. This section extends that calculus with bounded quantification based on kernel $F_{<:}$. This new variant also employs Duotyping and is called $F_{k\Diamond}^{\wedge\vee}$. The main aim of this section is to show that sometimes we can get interesting and novel dual features come for free. In addition to upper bounded quantification of $F_{<:}$, System $F_{k\Diamond}^{\wedge\vee}$ provides lower bounded quantification as well. Additionally, we also show the type soundness of $F_{k\Diamond}^{\wedge\vee}$.

6.3.1 SYNTAX AND DUOTYPING

SYNTAX. Fig. 6.7 shows the syntax of the calculus $F_{k\Diamond}^{\wedge\vee}$. Types $\top$, $\bot$, Int, $A \rightarrow B$, $A \wedge B$, $A \vee B$ are already introduced in Section 6.1. Type variable α and a universal quantifier on type variables $\forall(\alpha \Diamond A).B$ are the two additional types in $F_{k\Diamond}^{\wedge\vee}$. Terms x, i, $\lambda x : A.e$, $e_1\, e_2$ are already discussed in Section 6.2.1. Type abstraction $\Lambda(\alpha \Diamond A).e$ and type application $e\, A$ are two additional terms in $F_{k\Diamond}^{\wedge\vee}$. Values are a subset of terms, consisting of term abstraction, type abstraction and integers.

DUOTYPING FOR $F_{k\Diamond}^{\wedge\vee}$. Duotyping rules for a calculus with union and intersection types are presented in Fig. 6.4. $F_{k\Diamond}^{\wedge\vee}$ has two significant differences in its Duotyping rules in comparison to Fig. 6.4, which are presented in Figure 6.7. The first one is the addition of a typing context in the Duotyping rules. This is important to ensure that type variables are bound. Thus, Duotyping for $F_{k\Diamond}^{\wedge\vee}$ is now of the form $\Gamma \vdash A \Diamond B$. The second difference is that there are four more rules, three of them (rules GS-REFLTVARA, GS-TVARA, and GS-FORALLKFS)

Types	A, B	$::=$	$\top \mid \bot \mid \mathsf{Int} \mid A \to B \mid A \wedge B \mid A \vee B \mid \alpha \mid \forall(\alpha \Diamond A).B$
Terms	e	$::=$	$x \mid i \mid \lambda x : A.e \mid e_1\, e_2 \mid \Lambda(\alpha \Diamond A).e \mid e\, A$
Values	v	$::=$	$i \mid \lambda x : A.e \mid \Lambda(\alpha \Diamond A).e$
Context	Γ	$::=$	$\cdot \mid \Gamma, x : A \mid \Gamma,\ \alpha \Diamond A$
Mode	$\Diamond$	$::=$	$<: \mid :>$

$\boxed{\Gamma \vdash A \Diamond B}$ ($F_{k\Diamond}^{\wedge\vee}$ Duotyping)

GS-TOPBTMAP
$$\frac{}{\Gamma \vdash A \Diamond \rceil \Diamond \lceil}$$

GS-TOPBTMBP
$$\frac{}{\Gamma \vdash \rceil \overline{\Diamond} \lceil \Diamond A}$$

GS-INTP
$$\frac{}{\Gamma \vdash \mathsf{Int} \Diamond \mathsf{Int}}$$

GS-ARROWP
$$\frac{\Gamma \vdash A_1 \overline{\Diamond} A_2 \qquad \Gamma \vdash B_1 \Diamond B_2}{\Gamma \vdash A_1 \to B_1 \Diamond A_2 \to B_2}$$

GS-LEFTAP
$$\frac{\Gamma \vdash A \Diamond C}{\Gamma \vdash (A \Diamond_? B) \Diamond C}$$

GS-LEFTB
$$\frac{A \Diamond B}{A \Diamond (B \overline{\Diamond}_? C)}$$

GS-RIGHTAP
$$\frac{\Gamma \vdash B \Diamond C}{\Gamma \vdash (A \Diamond_? B) \Diamond C}$$

GS-RIGHTBP
$$\frac{\Gamma \vdash A \Diamond C}{\Gamma \vdash A \Diamond (B \overline{\Diamond}_? C)}$$

GS-BOTHAP
$$\frac{\Gamma \vdash A \Diamond B \qquad \Gamma \vdash A \Diamond C}{\Gamma \vdash A \Diamond (B \Diamond_? C)}$$

GS-BOTHBP
$$\frac{\Gamma \vdash A \Diamond C \qquad \Gamma \vdash B \Diamond C}{\Gamma \vdash (A \overline{\Diamond}_? B) \Diamond C}$$

GS-REFLTVARA
$$\frac{\Gamma \vdash ok \qquad \alpha \Diamond_1 A \in \Gamma}{\Gamma \vdash \alpha \Diamond_2 \alpha}$$

GS-TVARA
$$\frac{\alpha \Diamond A \in \Gamma \qquad \Gamma \vdash A \Diamond B}{\Gamma \vdash \alpha \Diamond B}$$

GS-TVARB
$$\frac{\alpha \Diamond A \in \Gamma \qquad \Gamma \vdash B \overline{\Diamond} A}{\Gamma \vdash B \overline{\Diamond} \alpha}$$

GS-FORALLKFS
$$\frac{\Gamma,\ \alpha \Diamond_1 A \vdash B \Diamond_2 C}{\Gamma \vdash (\forall \alpha \Diamond_1 A.B) \Diamond_2 (\forall \alpha \Diamond_1 A.C)}$$

Figure 6.7: Syntax and additional rules for Duotyping in $F_{k\Diamond}^{\wedge\vee}$.

were already explained in Section 6.1.4. Rule GS-TVARB is the dual of rule GS-TVARA. We introduce this rule to eliminate the *duality rule*.

The Duotyping relation for $F_{k\Diamond}^{\wedge\vee}$ is reflexive and transitive as well:

Theorem 6.14 (Reflexivity)**.** *$\Gamma \vdash A \Diamond A$.*

Proof. By induction on type A. □

Theorem 6.15 (Transitivity)**.** *If $\Gamma \vdash A \Diamond B$ and $\Gamma \vdash B \Diamond C$ then $\Gamma \vdash A \Diamond C$.*

Proof. By induction on $\Gamma \vdash A \Diamond B$.

- Cases rule GS-TOPBTMA, rule GS-TOPBTMB, rule GS-INT, rule GS-REFLTVAR, rule GS-TVARA, rule GS-LEFTA, rule GS-RIGHTA, rule GS-BOTHA are trivial to prove.
- Case rule GS-ARROW is proved using the induction hypotheses.
- Case rule GS-TVARB can be proved using Lemma 6.18.

- Case rule GS-FORALLKFS is proved using the induction hypotheses.
- Case rule GS-LEFTB can be proved using an additional Lemma 6.17.
- Case rule GS-RIGHTB also uses Lemma 6.17.
- Case rule GS-BOTHB is proved using the induction hypotheses.

□

The auxiliary lemmas for transitivity are described next and are essentially the same as in Section 6.2.1.

Lemma 6.16 (Bound Selection). *If* $\Gamma \vdash \rceil \Diamond \lceil \Diamond B$ *then* $\Gamma \vdash A \Diamond B$.

Lemma 6.17 (Inversion for rule GDS-Both). *If* $\Gamma \vdash C \Diamond (A \Diamond_? B)$ *then* $\Gamma \vdash (C \Diamond A)$ *and* $(C \Diamond B)$.

There is also a *duality lemma*:

Lemma 6.18 (Duality). $\Gamma \vdash A \Diamond B = \Gamma \vdash B \overline{\Diamond} A$.

Finally, We also proved weakening and the narrowing lemmas for Duotyping calculus. Here we briefly compare the narrowing lemma for $F_{k<:}^{\wedge\vee}$ and $F_{k\Diamond}^{\wedge\vee}$:

Lemma 6.19 ($F_{k<:}^{\wedge\vee}$ Narrowing Lemma). *If* $\Gamma \vdash A <: B$ *and* $\Gamma, \alpha <: B, \Gamma_1 \vdash C <: D$ *then* $\Gamma, \alpha <: A, \Gamma_1 \vdash C <: D$

Lemma 6.20 ($F_{k\Diamond}^{\wedge\vee}$ Narrowing Lemma). *If* $\Gamma \vdash A \Diamond_1 B$ *and* $\Gamma, X \Diamond_1 B, \Gamma_1 \vdash C \Diamond_2 D$ *then* $\Gamma, X \Diamond_1 A, \Gamma_1 \vdash C \Diamond_2 D$

Lemma 6.19 exploits only the subtyping relation while Lemma 6.20 exploits our Duotyping relation. Lemma 6.20 illustrates how lower and upper bounds are captured under a unified mode relation in narrowing. Like the transitivity statement using a Duotyping formulation, one can think of the Duotyping narrowing lemma as actually two distinct lemmas: one for narrowing of upper bounds and another for narrowing of lower bounds. Also, it is important to note that Lemma 6.20 is using two modes $\Diamond_1$ and $\Diamond_2$. $\Diamond_1$ is the relation between types A, B and the type variable X. Whereas, $\Diamond_2$ is the relation between type C and type D. Those two relations do not need to be the same.

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

$$\text{G-VAR}\quad \frac{\Gamma \vdash ok \qquad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \text{G-INT}\quad \frac{\Gamma \vdash ok}{\Gamma \vdash i : \mathsf{Int}} \qquad \text{G-ABS}\quad \frac{\Gamma, x : A_1 \vdash e_2 : A_2}{\Gamma \vdash \lambda x : A_1.e_2 : A_1 \to A_2}$$

$$\text{G-APP}\quad \frac{\Gamma \vdash e_1 : A_1 \to A_2 \qquad \Gamma \vdash e_2 : A_1}{\Gamma \vdash e_1\, e_2 : A_2} \qquad \text{G-SUBS}\quad \frac{\Gamma \vdash e : B \qquad \Gamma \vdash B <: A}{\Gamma \vdash e : A}$$

$$\text{G-TABS}\quad \frac{\Gamma, \alpha \Diamond A \vdash e : B}{\Gamma \vdash \Lambda(\alpha \Diamond A).e : \forall(\alpha \Diamond A).B} \qquad \text{G-TAPP}\quad \frac{\Gamma \vdash e : \forall(\alpha \Diamond A).B \qquad \Gamma \vdash C \Diamond A}{\Gamma \vdash e\, C : B[\alpha \rightsquigarrow C]}$$

$\boxed{e_1 \longrightarrow e_2}$ *(Reduction)*

$$\text{GRED-AppAbs}\quad \frac{}{(\lambda x : A_1.e_1)\, v_2 \longrightarrow e_1[x \rightsquigarrow v_2]} \qquad \text{GRED-Fun}\quad \frac{e_1 \longrightarrow e_1'}{e_1\, e \longrightarrow e_1'\, e} \qquad \text{GRED-Arg}\quad \frac{e_1 \longrightarrow e_1'}{v\, e_1 \longrightarrow v\, e_1'}$$

$$\text{GRED-TAppTAbs}\quad \frac{}{(\Lambda(\alpha \Diamond A).e_1)\, B \longrightarrow e_1[\alpha \rightsquigarrow B]} \qquad \text{GRED-TFun}\quad \frac{e_1 \longrightarrow e_1'}{e_1\, A \longrightarrow e_1'\, A}$$

Figure 6.8: Typing and reduction of the duotyped kernel $F_{<:}$.

6.3.2 Semantics and type soundness

Typing. The first part of Fig. 6.8 presents the typing rules of $F_{k\Diamond}^{\wedge\vee}$. The first five rules are standard and are already explained in Section 6.1.1. Rules G-TABS and G-TAPP are the two additional rules in $F_{k\Diamond}^{\wedge\vee}$. Rule G-TABS is similar to the standard rule for type abstractions in $F_{<:}$ except that it generalizes the subtyping bound to a $\Diamond$ bound, which could either be subtyping or supertyping. Rule G-APP again differs from the rule for type applications in $F_{<:}$ by using a $\Diamond$ bound instead of just a subtyping bound. These two rules rules are noteworthy because they also illustrate an advantage of using Duotyping in the typing relation. Without Duotyping we would need multiple typing rules to capture different variations of the bounds.

Reduction. The last part of Fig. 6.8 presents the reduction rules of our calculus. Again, reduction rules are standard except for the rule GRED-TAppTAbs. In rule GRED-TAppTAbs the duality relation captures both upper and the lower bounds. Rule GRED-TFun is the standard reduction rule for the type applications.

Type Soundness. We proved the type soundness for our calculus. All the proofs are formalized in Coq theorem prover.

Lemma 6.21 (Type Preservation)**.** *If* $\Gamma \vdash e : A$ *and* $e \longrightarrow e'$ *then:* $\Gamma \vdash e' : A$.

Proof. By induction on the typing relation.

- Case rules G-VAR, G-INT, G-ABS, G-TABS, and G-SUBS are trivial to solve.
- Case rule G-APP uses Theorem 6.14 and Lemma 6.18.
- Case rule G-TAPP uses Theorem 6.14.

□

Lemma 6.22 (Progress)**.** *If* $\Gamma \vdash e : A$ *then:*

1. *either e is value.*
2. *or e can take step to* e'.

Proof. By induction on the typing relation.

- Case rules G-VAR, G-INT, G-ABS, G-TABS, and G-SUBS are trivial to solve.
- Case rule G-APP requires canonical forms.
- Case rule G-TAPP requires canonical forms.

□

6.3.3 Summary and Comparison

Besides $F_{k\Diamond}^{\wedge\vee}$, which employs the Duotyping relation, we have also formalized a calculus $F_{k<:}^{\wedge\vee}$: an extension of kernel $F_{<:}$ (only with upper bounded quantification) with union and intersection types using the traditional subtyping relation. The essential differences are similar to what we already discussed in Section 6.2.3. The formalization with Duotyping only has two additional lemmas (the duality lemma and the bound selection lemma), besides a few minor auxiliary lemmas. The number of lines for proof for the formalization of $F_{k<:}^{\wedge\vee}$ is 1648 whereas for $F_{k\Diamond}^{\wedge\vee}$ is 1770. The total lemmas required for $F_{k<:}^{\wedge\vee}$ are 74 and 81 for $F_{k\Diamond}^{\wedge\vee}$. We emphasize that one significant difference between $F_{k<:}^{\wedge\vee}$ and $F_{k\Diamond}^{\wedge\vee}$ is the additional lower bounded quantification provided by $F_{k\Diamond}^{\wedge\vee}$. This is an extra feature which comes essentially for free with Duotyping.

6.4 A Case Study on Duotyping

In this section we present an empirical case study, which we conducted to validate some of the benefits of Duotyping. Overall, the results of our case study indicate that: Duotyping does allow for compact specifications; the complexity of developing formalization with Duotyping is comparable to similar developments using traditional subtyping relations; transitivity proofs are often significantly simpler; and Duotyping is a generally applicable technique.

6.4.1 Case Study

We formalized a number of different calculi using Duotyping. All the proofs and metatheory are mechanically checked by the Coq theorem prover. We also formalized a few traditional subtyping systems for comparison. Table 6.1 shows a brief overview of various systems that we formalized. $\lambda_{<:}$, $\lambda_{<:}^{\wedge\vee}$, $F_{k<:}$, $F_{k<:}^{\wedge\vee}$and $F_{F<:}$are the traditional subtyping systems. The Coq formalizations for the traditional subtyping systems are based on existing Coq formalizations from the locally nameless representation with cofinite quantification tutorial and repository (`https://www.chargueraud.org/softs/ln/`) by Charguéraud [Charguéraud 2011]. The formalizations of $\lambda_{\Diamond}$, $\lambda_{\Diamond}^{\wedge\vee}$, $F_{k\Diamond}$, $F_{k\Diamond}^{\wedge\vee}$and $F_{F\Diamond}$are their respective Duotyping formulations, and modify the original ones with traditional subtyping. Subscript $<:$ represents a calculus with traditional subtyping whereas $\Diamond$ represents a calculus with Duotyping. Superscript $\wedge\vee$ is the notation for a system with intersection and union types. Subscript k corresponds to the kernel version of a variant of $F_{<:}$, while subscript F corresponds to the corresponding full version. We also formalized a simple polymorphic system without bounded quantification using Duotyping. We have two Duotyping variants for this polymorphic type system without bounded quantification. One without union and intersection types ($F_{\Diamond}$) and another with union and intersection types ($F_{\Diamond}^{\wedge\vee}$).

In Table 6.1, the last column (Transitivity) summarizes the proof technique used in each system to prove transitivity. Recall the transitivity lemma (using the Duotyping formulation):

Theorem 6.23 (Transitivity)**.** *If $A \Diamond B$ and $B \Diamond C$ then $A \Diamond C$.*

Induction on the middle type means induction on type B (or well-formed type B for polymorphic systems), whereas induction on the Duotyping relation means induction on $A \Diamond B$.

Research Questions. Section 1.2 discussed benefits of using Duotyping. This section attempts to quantify some of these benefits. More concretely, we answer the following questions in this section:

- Does Duotyping provide shorter specifications?

Name	Description	SLOC	Transitivity
$\lambda_{<:}$	STLC with subtyping	537	By induction on the middle type.
$\lambda_{\Diamond}$	STLC with Duotyping	583	By induction on the Duotyping relation.
$\lambda_{<:}^{\wedge\vee}$	STLC with subtyping, union types and intersection types	595	By induction on the middle type.
$\lambda_{\Diamond}^{\wedge\vee}$	STLC with Duotyping, union types and intersection types	623	By induction on the Duotyping relation.
$F_{\Diamond}$	Simple polymorphic system with Duotyping and without bounded quantification	1466	By induction on the Duotyping relation.
$F_{\Diamond}^{\wedge\vee}$	Simple polymorphic system with Duotyping, union types and intersection types and without bounded quantification	1546	By induction on the Duotyping relation.
$F_{k<:}$	System $F_{<:}$ kernel	1542	By induction on the (well-formed) middle type.
$F_{k\Diamond}$	System $F_{<:}$ kernel with Duotyping	1579	By induction on the Duotyping relation.
$F_{k<:}^{\wedge\vee}$	System $F_{<:}$ kernel with subtyping, union types and intersection types	1648	By induction on the (well-formed) middle type.
$F_{k\Diamond}^{\wedge\vee}$	System $F_{<:}$ kernel with Duotyping, union types and intersection types	1770	By induction on the Duotyping relation.
$F_{F<:}$	System full $F_{<:}$	1518	By induction on the (well-formed) middle type.
$F_{F\Diamond}$	System full $F_{<:}$ with Duotyping	1786	By induction on the (well-formed) middle type.

Table 6.1: Description of all systems.

- Does Duotyping increase the complexity of the formalization and metatheory of the language?
- Does Duotyping make transitivity proofs simpler?
- Is Duotyping a generally applicable technique?

We follow an empirical approach to answer these questions and address each question in a separate (sub)section. Obviously a precise measure for complexity/simplicity is hard to

obtain. We use SLOC for the formalization and proofs as an approximation. All the formalizations are written in the same Coq style to ensure that the comparisons are fair.

6.4.2 Does Duotyping provide shorter specifications?

This section answers our first question. In short our case study seems to support this conclusion. The declarative Duotyping rules of all the systems that we formalized are shown in Table 6.2. Please note that the formulation also contains the *duality rule*. $\lambda_{\Diamond}$has the basic set of Duotyping rules. These rules are common in all of the systems. $\lambda_{\Diamond}^{\wedge\vee}$has the subtyping rules for intersection types and union types in addition to the rules from $\lambda_{\Diamond}$. $F_{\Diamond}$contains two more rules (rules GDS-ReflTvarP and GDS-forallfsp) in addition to the rules from $\lambda_{\Diamond}$. $F_{\Diamond}^{\wedge\vee}$has all the rules from $\lambda_{\Diamond}$, $\lambda_{\Diamond}^{\wedge\vee}$and $F_{\Diamond}$. $F_{k\Diamond}$has three additional subtyping rules GDS-ReflTvar, GDS-TVar, and GDS-forallkfs in addition to the rules from $\lambda_{\Diamond}$. $F_{k\Diamond}^{\wedge\vee}$has all the rules from $\lambda_{\Diamond}$, $\lambda_{\Diamond}^{\wedge\vee}$, and $F_{k\Diamond}$. $F_{F\Diamond}$has an additional subtyping rule GDS-forallffs.

Comparison with systems using traditional subtyping. Table 6.3 shows the number of rules and features for different calculi formulated with subtyping and Duotyping. In our formulation, $\lambda_{<:}$has 3 types $\top$, Int, *and* $A \rightarrow B$. This requires 3 subtyping rules to capture the subtyping relation of these 3 types. If we wanted to support the $\perp$ type in $\lambda_{<:}$we would need to add 1 more subtyping rule. In the table we express the extra rules required for extra features as (+n), where n is the number of extra rules. Duotyping supports $\perp$ for free by exploiting the dual nature of $\top$ with the help of *duality rule*. Systems with more rules follow the same approach for traditional systems i.e more types require more subtyping rules. If we wanted to support the $\perp$ type in $\lambda_{<:}^{\wedge\vee}$we also need 1 additional rule. To further extend our discussion to the polymorphic systems with bounded quantification, we would need 4 additional rules in $F_{k<:}$(1 for $\perp$ type and 3 for lower bounded quantification). Similarly we would need 4 additional rules to support lower bounds and lower bounded quantification in $F_{k<:}^{\wedge\vee}$.

In summary, in the systems that we compared Duotyping has a similar number of rules to systems with subtyping, but it comes with extra features. If we wanted to add those features to systems with traditional subtyping, then that would generally result in more rules for the traditional versions compared to Duotyping. This would also have an impact in the SLOC of the metatheory, increasing the metatheory for those systems considerably.

Name	Duotyping Rules
$\lambda_\Diamond$	$\boxed{A \Diamond B}$ *($\lambda_\Diamond$ Duotyping)* GDS-TOPBOT $\dfrac{}{A \Diamond \rceil \Diamond \lceil}$ GDS-INT $\dfrac{}{\mathsf{Int} \Diamond \mathsf{Int}}$ GDS-ARROW $\dfrac{A_1 \overline{\Diamond} A_2 \qquad B_1 \Diamond B_2}{A_1 \to B_1 \Diamond A_2 \to B_2}$ GDS-DUAL $\dfrac{B \overline{\Diamond} A}{A \Diamond B}$
$\lambda_\Diamond^{\wedge\vee}$	$\boxed{A \Diamond B}$ *($\lambda_\Diamond^{\wedge\vee}$ Duotyping plus all rules from $\lambda_\Diamond$)* GDS-LLEFT $\dfrac{A \Diamond C}{(A \Diamond_? B) \Diamond C}$ GDS-RRIGHT $\dfrac{B \Diamond C}{(A \Diamond_? B) \Diamond C}$ GDS-BOTH $\dfrac{A \Diamond B \qquad A \Diamond C}{A \Diamond (B \Diamond_? C)}$
$F_\Diamond$	$\boxed{A \Diamond B}$ *($F_\Diamond$ Duotyping plus all rules from $\lambda_\Diamond$)* GDS-REFLTVARP $\dfrac{}{\alpha \Diamond \alpha}$ GDS-FORALLFSP $\dfrac{A \Diamond B}{(\forall \alpha.A) \Diamond (\forall \alpha.B)}$
$F_\Diamond^{\wedge\vee}$	$\boxed{A \Diamond B}$ *($F_\Diamond^{\wedge\vee}$ Duotyping plus all rules from $\lambda_\Diamond$, $\lambda_\Diamond^{\wedge\vee}$ and $F_\Diamond$)*
$F_{k\Diamond}$	$\boxed{\Gamma \vdash A \Diamond B}$ *($F_{k\Diamond}$ Duotyping plus all rules from $\lambda_\Diamond$)* GDS-REFLTVAR $\dfrac{\Gamma \vdash ok \qquad \alpha \Diamond_1 A \in \Gamma}{\Gamma \vdash \alpha \Diamond_2 \alpha}$ GDS-TVAR $\dfrac{\alpha \Diamond A \in \Gamma \qquad \Gamma \vdash A \Diamond B}{\Gamma \vdash \alpha \Diamond B}$ GDS-FORALLKFS $\dfrac{\Gamma,\ \alpha \Diamond_1 A \vdash B \Diamond_2 C}{\Gamma \vdash (\forall \alpha \Diamond_1 A.B) \Diamond_2 (\forall \alpha \Diamond_1 A.C)}$
$F_{k\Diamond}^{\wedge\vee}$	$\boxed{\Gamma \vdash A \Diamond B}$ *($F_{k\Diamond}^{\wedge\vee}$ Duotyping plus all rules from $\lambda_\Diamond$, $\lambda_\Diamond^{\wedge\vee}$ and $F_{k\Diamond}$)*
$F_{F\Diamond}$	$\boxed{\Gamma \vdash A \Diamond B}$ *($F_{F\Diamond}$ Duotyping plus all rules from $F_{k\Diamond}$ excluding rule* GS-FORALLKFS *and union/intersection rules)* GDS-FORALLFFS $\dfrac{\Gamma \vdash A \ \lvert \Diamond_1 \rvert_{\Diamond_2} \ B \qquad \Gamma,\ \alpha \Diamond_1 (A \widetilde{\Diamond}_2 B) \vdash A_1 \Diamond_2 B_1}{\Gamma \vdash (\forall \alpha \Diamond_1 A.A_1) \Diamond_2 (\forall \alpha \Diamond_1 B.B_1)}$

Table 6.2: Declarative Duotyping rules of all systems.

System	Subtyping rules count	System	Duotyping rules count	Duotyping extra features
$\lambda_{<:}$	3 (+1)	$\lambda_{\Diamond}$	4	lower bounds in $\lambda_{\Diamond}$
$\lambda_{<:}^{\wedge\vee}$	9 (+1)	$\lambda_{\Diamond}^{\wedge\vee}$	7	lower bounds in $\lambda_{\Diamond}^{\wedge\vee}$
$F_{k<:}$	5 (+4)	$F_{k\Diamond}$	7	lower bounds and lower bounded quantification in $F_{k\Diamond}$
$F_{k<:}^{\wedge\vee}$	11 (+4)	$F_{k\Diamond}^{\wedge\vee}$	10	lower bounds and lower bounded quantification in $F_{k\Diamond}^{\wedge\vee}$

Table 6.3: Comparing the features and number of rules with subtyping and Duotyping.

Subtyping System	SLOC	Duotyping System	SLOC
$\lambda_{<:}$	537	$\lambda_{\Diamond}$	583
$\lambda_{<:}^{\wedge\vee}$	595	$\lambda_{\Diamond}^{\wedge\vee}$	623
$F_{k<:}$	1542	$F_{k\Diamond}$	1579
$F_{k<:}^{\wedge\vee}$	1648	$F_{k\Diamond}^{\wedge\vee}$	1770

Table 6.4: SLOC of traditional subtyping and Duotyping systems.

6.4.3 Does Duotyping increase the complexity of the formalization and metatheory of the language?

At first, one may think that Duotyping increases the complexity of formalization and metatheory of the language, since it provides interesting extra features and generalizations normally come at a cost. Interestingly, Duotyping does not add significant extra complexity in the formalization and metatheory of the language. Table 6.4 shows the SLOC for formalizations using traditional subtyping and Duotyping systems. The lines of code for $\lambda_{<:}^{\wedge\vee}$ are 595 and the lines of code for $\lambda_{\Diamond}^{\wedge\vee}$ are 623. Similarly, the lines of code for $F_{k<:}^{\wedge\vee}$ are 1648 and 1770 for $F_{k\Diamond}^{\wedge\vee}$. Although SLOC for Duotyping systems are slightly more than traditional subtyping systems, the Duotyping systems come with extra features. Nevertheless the mechanization effort is roughly the same for version with and without Duotyping. Also, as illustrated in Sections 6.2 and 6.3, the vast majority of the lemmas/metatheory for calculi with Duotyping are similar to traditional systems with subtyping.

Subtyping System	**Transitivity SLOC**	**Duotyping System**	**Transitivity SLOC**
$\lambda_{<:}$	7	$\lambda_{\Diamond}$	4
$\lambda_{<:}^{\wedge\vee}$	13	$\lambda_{\Diamond}^{\wedge\vee}$	11
$F_{k<:}$	26	$F_{k\Diamond}$	13
$F_{k<:}^{\wedge\vee}$	38	$F_{k\Diamond}^{\wedge\vee}$	18

Table 6.5: SLOC for transitivity proofs.

6.4.4 Does Duotyping make transitivity proofs simpler?

Transitivity is often the most difficult property to prove in the metatheory of a language with subtyping. Table 6.1 highlights a brief comparison between the techniques for the transitivity proof of various systems. Transitivity of systems with Duotyping is generally proved by induction on the Duotyping relation. One exception is $F_{F\Diamond}$ where induction on the Duotyping does not work. As discussed in Section 6.1.5 Duotyping allows us to simplify the transitivity proof by using a different inductive argument.

Table 6.5 shows the SLOC for transitivity proofs of various systems. The SLOC for $\lambda_{<:}$ transitivity proof are 7 and the SLOC for $\lambda_{\Diamond}$ transitivity proof are 4. Similarly, the SLOC for $F_{k<:}^{\wedge\vee}$ transitivity proof are 38 and 18 for the transitivity proof of $F_{k\Diamond}^{\wedge\vee}$. This evaluation shows that Duotyping always allows us to reduce the size of the transitivity proof. Again, it is important to note that Duotyping also provides extra features of lower bound and lower bounded quantification. Despite these additional features in Duotyping systems, their transitivity proofs are shorter than the traditional systems with subtyping.

However we could not employ this proof technique in our Duotyping version of full $F_{<:}$ ($F_{F\Diamond}$). The problem is related to *narrowing*, which in $F_{F\Diamond}$ is closely coupled with *transitivity*. Despite that we could still apply the technique to most systems with Duotyping, and even for $F_{F\Diamond}$ we can still prove transitivity using the same technique as in the traditional $F_{<:}$ (i.e. using the middle type as the inductive argument).

6.4.5 Is Duotyping a generally applicable technique?

Our case studies indicate that Duotyping is a generally applicable technique. In all the systems that we have tried to use Duotyping, we have managed to successfully apply it. Furthermore we believe that Duotyping can be essentially applied to any system with a traditional subtyping relation. The most complex system where we have employed Duotyping is $F_{F\Diamond}$. In $F_{F\Diamond}$ universal quantification allows Duotyping between the bounds, generalizing

the universal quantification presented in Section 6.3. Rule GDS-FORALLFFS in $F_{F\Diamond}$ employs two operations $|\Diamond_1|_{\Diamond_2}$ and $A \mathbin{\widetilde{\Diamond}} B$:

$$
\begin{aligned}
&|\Diamond_1|_{\Diamond_2} \\
&\qquad |{<:}|_{\Diamond_2} = \overline{\Diamond}_2 \\
&\qquad |{:>}|_{\Diamond_2} = \Diamond_2 \\
&A \mathbin{\widetilde{\Diamond}} B \\
&\qquad A \mathbin{\widetilde{<:}} B = B \\
&\qquad A \mathbin{\widetilde{:>}} B = A
\end{aligned}
$$

$|\Diamond_1|_{\Diamond_2}$ takes two modes $\Diamond_1$ and $\Diamond_2$ as input, and flips $\Diamond_2$ if $\Diamond_1$ is subtyping, otherwise it returns $\Diamond_2$. This operation chooses the mode to check the relationship between the bounds of the two universal quantifiers being compared for Duotyping. The second operation $A \mathbin{\widetilde{\Diamond}} B$ selects the bounds to use in the environment when checking the Duotyping of the bodies of the universal quantifiers. It takes a mode $\Diamond$ and two types A B as inputs, and returns the second type if the mode is subtyping, otherwise it returns the first type.

Part IV

Related Work

7 Related Work

7.1 Union Types

Union types were first introduced by MacQueen et al. [1984]. They proposed a typing rule that eliminates unions implicitly. The rule breaks type preservation under the conventional reduction strategy of the lambda calculus. Barbanera et al. [1995] solved the problem by reducing all copies of the same redex in parallel. Dunfield and Pfenning [2003] and Dunfield [2014] took another approach to support mutable references. They restricted the elimination typing rule to only allow a single occurrence of a subterm with a union type when typing an expression. Pierce [1991] proposed a novel single-branch case construct for unions. As pointed by Dunfield and Pfenning, compared to the single occurrence approach, the only effect of Pierce's approach is to make elimination explicit.

Union types and elimination constructs based on types are widely used in the context of XML processing languages [Benzaken et al. 2003; Hosoya and Pierce 2003], and have inspired proposals for object oriented languages [Igarashi and Nagira 2006]. Generally speaking, the elimination constructs in those languages offer a first-match semantics, where cases can overlap and reordering the cases may change the semantics of the program. This is in contrast to our approach. Union types have also been studied in the context of XDuce programming language [Hosoya and Pierce 2003]. XDuce employs regular expression types. Pattern matching can be on expressions and types in XDuce. Expressions are considered as special cases of types. CDuce [Benzaken et al. 2003] is an extension of XDuce. Work on the more foundational aspects of CDuce, and in particular on *semantic subtyping* [Frisch et al. 2002] and set-theoretic types, also employs a form of first-match semantics elimination construct, though in a different form. In particular, work by Castagna and Frisch [2005] and Castagna and Lanvin [2017] propose a conditional construct that can test whether a value matches a type. If it matches then the first branch is executed and the type for the value is refined. Otherwise, the second branch is executed and the type of the value is refined to be the negation of the type (expressing that the value does not have such type). Union types are also studied in the context of semantic subtyping and object-oriented calculi [Ancona and Corradi 2014, 2016; Dardha et al. 2013] which focus on designing subtyping algorithms

to employ semantic subtyping in OOP. In contrast, we study a deterministic and type-safe switch construct for union elimination.

Muehlboeck and Tate [2018] give a general framework for subtyping with intersection and union types. They illustrate the significance of their framework using the Ceylon programming language. The main objective of their work is to define a generic framework for deriving subtyping algorithms for intersection and union types in the presence of various distributive subtyping rules. For instance, their framework could be useful to derive an algorithmic formulation for the subtyping relation presented in Figure 3.5. They also briefly cover disjointness in their work. As part of their framework, they can also check disjointness given some disjointness axioms. For instance, for λ_u, such axioms could be similar to rule AD-BTMR or rule AD-INTL in Figure 3.2. However, they do not have a formal specification of disjointness. Instead they assume that some sound specification exists and that the axioms respect such specification. If some unsound axioms are given to their framework (say $\mathsf{Int} *_a \mathsf{Int}$) this would lead to a problematic algorithm for checking disjointness. In our work we provide specifications for disjointness together with sound and complete algorithmic formulations. In addition, unlike us, they do not study the semantics of disjoint switch expressions.

Occurrence Typing. Occurrence typing or flow typing [Tobin-Hochstadt and Felleisen 2008] specializes or refines the type of variable in a type test. An example of occurrence typing is:

```
Integer occurrence (Integer | String val) {
  if (val is Integer) { return val+1; }
  else                { return toInt(val)+2; }
}
```

In such code, `val` initially has type $\mathsf{Int} \vee \mathsf{String}$. The conditional checks if the `val` is of type Int. If the condition succeeds, it is safe to assume that `val` is of type Int, and the type of `val` is refined in the branch to be Int. Otherwise, it is safe to assume that `val` is of type String, in the other branch (and the type is refined accordingly). The motivation to study occurrence typing was to introduce typing in dynamically typed languages. Occurrence typing was further studied by Tobin-Hochstadt and Felleisen [2010], which resulted into the development of Typed Racket. Variants of occurrence typing are nowadays employed in mainstream languages such as TypeScript, Ceylon or Flow. Castagna et al. [2019] extended occurrence typing to refine the type of generic expressions, not just variables. They also studied the combination with gradual typing. Recently, Castagna et al. [2022] show that the classical union and intersection types along with a type-based switch construct encompasses occurrence typing. Occurrence

typing in a conditional construct, such as the above, provides an alternative means to eliminate union types using a first-match semantics. That is the order of the type tests determines the priority.

Nullable Types. Nullable types are types which may have the `null` value. Recently, Nieto et al. [2020] proposed an approach with explicit nulls in Scala using union types. The Ceylon language has implemented a similar approach for a few years now. However our's and Ceylon's approaches are based on disjoint switches to test for nullability, while Nieto et al. [2020]'s approach is based on a simplified form of occurrence typing.

Various approaches have been proposed to deal with nullability such as `T?` in Kotlin [Kotlin 2021], Swift [Apple 2021] and Flow [Chaudhuri et al. 2017]. The Checker Framework [Papi et al. 2008] is another line of related work to detect null pointer deferences in Java programs. Banerjee et al. [2019] proposes an approach to explicitly associate nullable and non-nullable properties with expressions in Java. However, differently from our work, in those approaches nullable types are not encoded with union types. Blanvillain et al. [2022] study a notion of match types for type level programming. They also employ a notion of disjointness in match types and can encode nullable types. However, they provide match types at the type level and do not use them for union elimination. Furthermore, they do not study intersection and union types formally. In contrast, we provide a term level switch construct for union elimination.

7.2 Disjoint Intersection Types

Disjoint intersection types were first studied by Oliveira et al. [2016] in the λ_i calculus to give a coherent calculus for intersection types with a merge operator. The notion of disjointness used in λ_u, discussed in Section 3.2, is inspired by the notion of disjointness of λ_i. In essence, disjointness in λ_u is the dual notion: while in λ_i two types are disjoint if they only have *top-like* supertypes, in λ_u two types are disjoint if they only have *bottom-like* subtypes. *Disjoint polymorphism* [Alpuim et al. 2017] has been studied for calculi with disjoint intersection types.

None of calculi with disjoint intersection types [Alpuim et al. 2017; Bi et al. 2018b, 2019; Oliveira et al. 2016] in the literature includes union types. One interesting discovery of our work is that the presence of both intersections and unions in a calculus can affect disjointness. In particular, as we have seen in Section 3.3, adding intersection types required us to change disjointness. The notion of disjointness that was derived from λ_i stops working in the presence of intersection types. Interestingly, a similar issue happens when union types

are added to a calculus with disjoint intersection types. If disjointness of two types A and B is defined to be that such types can only have top-like types, then adding union types immediately breaks such definition. For example, the types Int and Bool are disjoint but, with union types, Int $\vee$ Bool is a common supertype that is not top-like. We conjecture that, to add union types to disjoint intersection types, we can use the following definition of disjointness:

Definition 10. $A * B ::= \nexists\, C^{\circ}, A <: C^{\circ}$ and $B <: C^{\circ}$.

which is, in essence, the dual notion of the definition presented in Section 3.3. Under this definition Int and Bool would be disjoint since we cannot find a common ordinary supertype (and Int $\vee$ Bool is a supertype, but it is not ordinary). Furthermore, there should be a dual notion to LOS, capturing the greatest ordinary supertypes. Moreover, if a calculus includes both disjoint switches and a merge operator, then the two notions of disjointness must coexist in the calculus. This will be an interesting path of exploration for future work.

7.3 Overloading and Dynamic Dispatch

Overloading. Union and intersection types also provide a form of function overloading or ad-hoc polymorphism using the switch and type-based case analysis. A programmer may define the argument type to be a union type. By using type-based case analysis, it is possible to execute different code for each specific type of input. Intersection types have also been studied for function overloading. For example, a function with type Int $\rightarrow$ Int $\wedge$ Bool $\rightarrow$ Bool can take input values either of type Int or Bool. In such case, it returns either Int or Bool depending upon the input type. Function overloading [Cardelli and Wegner 1985; Castagna et al. 1995; Wadler and Blott 1989] has been studied in detail in the literature. Wadler and Blott [1989] studied type classes as an alternative way to provide overloading based on parametric polymorphism. Recently, Rioux et al. [2023] studied function overloading with intersection types, merge operator, and the union types in a calculus called $F_{\bowtie}$. Their calculus is type-safe and deterministic. However, the merge operator in their calculus is restricted to functions. Moreover, the $F_{\bowtie}$ calculus is proposed after Rehman et al. [2022].

Dynamic dispatch. A straightforward definition of dynamic dispatch [Bourdoncle and Merz 1997; Castagna et al. 1995, 2014b; Clifton et al. 2006] is the *runtime function overloading*. In contrast to the static overloading, a specific function implementation is selected based on the the dynamic or runtime type of the one (usually left) or more argument(s) in dynamic dispatch. This division categorizes the dynamic dispatch into single and multiple dispatch respectively. Single dispatch comes naturally with most of the OOP languages such as C++ [Stroustrup 1986] and Smalltalk [Goldberg and Robson 1983]. Multiple dispatch

[Bourdoncle and Merz 1997; Castagna et al. 1995, 2014b], on the other hand, requires special care. Normally, types overlap in multiple dispatch and the language implementation selects the best matching function implementation based on the argument type such as in Julia [Zappa Nardelli et al. 2018]. Recently, Park et al. [2019] study symmetric dynamic dispatch with multiple inheritance, parametric polymorphism, and variance. However, this line of work differs from ours in a way that we do not allow overlapping types in λ_u in the alternative branches of a switch expression. Thus λ_u provides a less expressive but deterministic variant of dynamic dispatch.

7.4 Duality in Logic and Programming Language Theory

Apart from informally observing duality of type system features, as far as we known, formally exploiting duality in subtyping relations has not been investigated in the past. However there is plenty of work on uses of duality in programming language theory. Furthermore there is related work on type systems that exploit various generalizations for added expressive power or economy in metatheory and implementation. We discuss these next.

In type theory [Andrews 1986] and/or category theory [Bird and de Moor 1996; Lane 1998] duality occurs in various forms. For instance, the duality between sum and product types is well-known in both type and category theory. Properties about such types often explicitly acknowledge duality. Many properties about sum types are presented as dual properties of corresponding properties on product types and vice-versa. Our Lemma 6.6 is an example of a property that applies to both union and/or intersection types. In this property duality is not only acknowledged, but directly exploited in the lemma itself to provide a generalized property that can be specialized to one construct and its dual. Various other dualities between constructs are known and exploited in various ways in type and/or category theory. For example, existential and universal quantification can be captured by an encoding by one through the other. The type $\exists\alpha.\ A$ can be encoded as $\forall\beta.(\forall\alpha.\ A \to \beta) \to \beta$, which requires a kind of CPS translation [Danvy and Filinski 1992] of the corresponding terms. Similar encodings exist for sums and products.

In the field of *proof-theoretic semantics* [Gentzen 1934] and in *natural deduction* the concept of *harmony* is used to describe introduction and elimination rules that are in some sense dual. For instance, the usual rules for introduction and elimination of conjunction are in perfect harmony. The inversion principles by Prawitz [1979] are a general procedure to associate to any arbitrary collection of introduction rules a specific collection of elimination rules. The elimination rules are in harmony with the given collection of introduction rules. Prawitz inversion principles attempt to capture harmony in a more precise way, directly expressing it

formally. Therefore inversion principles have similar considerations to Duotyping in terms of expressing some form of duality directly in a formalism. However inversion principles focus on introduction and/or elimination rules, while Duotyping is focused on subtyping. Nevertheless in future work we are interested in exploiting the use of duality in the typing relation more. We believe that the notion of harmony and inversion principles could be quite helpful in such work.

Double-line rules [Došen 1989] are deduction rules that can be read both from top to bottom (as usual) and also from bottom to top. In other words they express two standard (dual) deduction rules in a single double-line rule. Like Duotyping, double-line rules aim at expressing a form of duality in a single rule. Unlike Duotyping, double-line rules are concerned with (dual) rules where the premises and conclusions of one rule become the conclusions and premises of the other rule, respectively.

Bernardi et al. [2014] explain duality relations in the context in *session types*. Binary session types have two endpoints connected through one communication channel. In session types, connected endpoints should have a dual relation in their session types. The duality relation in session types is related to types and may have various interpretations. In contrast Duotyping is about subtyping (or supertyping).

The duality between data and codata is well-known in programming language theory [Bird and de Moor 1996]. Data types and codata types are duals in the sense that data types are defined in terms of constructors while codata types are defined in terms of destructors. More recently, such duality has been exploited in language design [Binder et al. 2019; Ostermann and Jabs 2018] to provide an automatic way to switch between programs defined on datatypes and equivalent programs defined on codata types. The use of duality in this line of work is quite different from ours.

7.5 Generalizations in Type Systems and Type Theory

Pure type systems (PTSs) [Adams 2006; Jutting 1993; McKinna and Pollack 1993; Severi and Poll 1994; van Benthem Jutting et al. 1993; Zwanenburg 1999] capture a generalization of various type systems (F, F^ω, λP). Typing rules of multiple type systems are expressed in pure type systems via parameterization. PTSs are parameterized by three sets: a set of sorts; a set of axioms; and a set of rules. Concrete type systems (such as System F), are recovered with concrete instantiations of those sets. Pure type systems with subtyping [Zwanenburg 1999] are a variant of pure type systems that captures a family of type systems with subtyping. This variant captures only the upper bounds. It does not provide subtyping generalization with both upper and the lower bounded quantification like our Duotyping generalizations

of $F_{<:}$. Pure subtype systems [Hutchins 2010] is a family of calculi based on subtyping only (and without a typing relation). This system eliminates the need of typing and presents an alternative to typing using subtyping only. Pure subtype systems support upper bounded quantification, but no lower bounded quantification.

Modal Type Theory. Modal type theory [Nanevski et al. 2008] is an extension of type theory which provides type rules using modalities. Modal type theory can represent a proposition as types which may be proved based upon the deduction rules in a given context. Modal type theory also employs modes, for instance *possibility* and *necessity* [Nanevski et al. 2008; Simpson 1994]. There are many type systems that use modes to generalize typing relations. One can view Duotyping as a simple instance of a relation with a mode. In Duotyping the mode is either subtyping or supertyping.

Bi-directional type checking. Bi-directional type checking [Dunfield and Krishnaswami 2019; Pierce and Turner 2000] also employs a mode, but in the typing relation instead. Bi-directional type checking is a common technique, used in implementations of programming languages, that can eliminate redundant type annotations. Bi-directional type-checking is also employed is several type systems, especially those where full type inference is undecidable [Dunfield and Krishnaswami 2013; Pierce and Turner 2000]. In such cases only partial inference methods are feasible in practice, which means that some type annotations are necessary. Bi-directional type checking is useful in such cases, allowing the type information to be easily propagated without requiring further (redundant) annotations. The modes in bi-directional type-checking are checking or synthesis. Checking checks a given term against a given type, whereas the synthesis infers the type based upon the available information in the context.

Unified Subtyping. Unified subtyping [Yang and Oliveira 2017] is a technique that can be used in dependently typed systems supporting unified syntax to model typing and subtyping in a single relation. The single unified subtyping relation generalizes both typing and subtyping. Like Duotyping, unified subtyping can also help reducing language metatheory and duplication. However unified subtyping is orthogonal to Duotyping and does not exploit duality of features. We believe that both techniques can complement each other.

Bounded quantification and generalizations. System $F_{<:}$ [Cardelli et al. 1994] is extensively studied due to its feature of bounded quantification. F-bounded quantification [Canning et al. 1989] is a generalization of bounded quantification to handle recursive types.

Although we are not aware of an extension of $F_{<:}$ with lower bounded quantification, such notion has appeared before in some calculi. For instance, Igarashi and Viroli [2002] have pointed out correspondence between use-site variance and existential types and, in order to capture contravariance, they introduced lower-bounded existential types.

One generalization of $F_{<:}$ is studied by Amin and Rompf [2017], which formalizes *type bounds* in Scala. Type bounds is an interesting feature in Scala as elaborated by the following code (code extended from Section 6.1.4):

```
class TypeBoundsCollection[S >: GraduateStudent <: Student](obj: S) {
  def student: S = obj
}
```

While in our variants of $F_{<:}$ we support either lower bounded quantification or upper bounded quantification (but not both at once), Scala's type bounds allow both upper and lower bounds at once. This is clearly more expressive than what we have, but it comes with its own problems. Formalisms with Scala-like type bounds often need to include a transitivity axiom (and thus are non-algorithmic) and they have to deal with the bad bounds problem. In contrast our simpler extension of type bounds is comparable in complexity to $F_{<:}$'s upper bounded quantification, and there is a set of algorithmic subtyping rules without a built-in transitivity axiom.

Duality in Subtyping of Intersection and Union Types. We exploit the duality of union and intersection types to illustrate Duotyping. Our Duotyping calculi manages to capture the six common rules for unions and intersections using three rules only (plus the duality rule), which provides a simple illustrative example of the use of duality. None of the calculi with intersection and union types discussed so far study the duality of subtyping formally as we do.

Part V

Epilogue

8 Conclusion and Future Work

8.1 Conclusion

The integration of the intersection and union types is known to be non-trivial in theory. This thesis examines the integration of intersection and union types in various settings. Chapter 3 (λ_u) discusses the deterministic elimination of union types with a type-based switch construct. The disjointness plays the essential role in making such a construct deterministic. The disjointness ensures that no two types in alternative branches overlap i.e. do not share a common subtype. Therefore, the type of the scrutinee matches with one branch at the most. Later sections in Chapter 3 further enrich λ_u with advance features including intersection types, nominal types, subtyping distributivity and disjoint polymorphism. Chapter 4 discusses an expressive version of the disjointness algorithm with disjoint polymorphism. All of the calculi discussed in Chapter 3 and Chapter 4 preserve the standard properties of type-safety and determinism.

Chapter 5 studies a calculus (λ_{um}) with an elimination construct for the union types and an introductory construct for the intersection types, the so called merge operator. λ_{um} is type-safe but lacks determinism. We also prove the completeness of λ_{um} with respect to Dunfield [2014]. Chapter 6 discusses Duotyping, a novel technique studied to unify the subtyping rules for the dual features such as intersection types and union types. Duotyping comes with certain benefits of reduced subtyping rules, easier proofs and metatheory, and extra features such as lower bounded quantification. We study Duotyping with various calculi to show that it is a practically applicable technique.

8.2 Future Work

8.2.1 Determinism for λ_{um}

Future work includes making the calculus discussed in Chapter 5 (λ_{um}) deterministic. We discuss a few proposals to make λ_{um} deterministic. Approaches include restricting subsumption rule to prohibit certain ambiguous upcasts, first-match semantics, and parallel application.

Proposal 1: restricting ambiguous upcasts

The meticulous observation concludes that the origin of non-determinism is primarily due to multiple upcast paths. For example, 1,,true can follow two paths to upcast to Int ∨ Bool. One is via Int and the other is via Bool. Our first proposal to deal with non-determinism is to reject the programs when there are multiple upcast paths involved. In this approach the following program will be rejected:

```
Bool isInt (x : Int | Bool) = switch (x)
                                (x:Int)  → true
                                (y:Bool) → false

isInt(1,,true) //rejected
```

We propose restricting subsumption rule with an extra condition so that it allows only unambiguous upcasts along with disjointness in merges and switches. Such a calculus will have three measures for determinism. One is the disjointness in switches, another is the disjointness in merges, and finally an extra restriction in subsumption rule to reject ambiguous upcasts. The revised subsumption rule is shown next:

$$\frac{\Gamma \vdash e : A \qquad A <: B \qquad A <_u B}{\Gamma \vdash e : B} \text{ TYP-SUB-RES}$$

Where, $A <_u B$ indicates type A is unambiguous to type B. Meaning that there is at most one path to upcast from A to B. Ideally, this relation allows upcasting $1 : \mathsf{Int}$ to $1 : \mathsf{Int} \vee \mathsf{Bool}$. But it does not allow upcasting $1,,\mathsf{true} : \mathsf{Int} \wedge \mathsf{Bool}$ to $1,,\mathsf{true} : \mathsf{Int} \vee \mathsf{Bool}$. This is because $1 : \mathsf{Int}$ upcasts to $1 : \mathsf{Int} \vee \mathsf{Bool}$ via only one path i.e.:

$$\frac{\Gamma \vdash 1 : \mathsf{Int} \qquad \mathsf{Int} <: \mathsf{Int} \vee \mathsf{Bool}}{\Gamma \vdash 1 : \mathsf{Int} \vee \mathsf{Bool}} \text{ TYP-SUB}$$

Whereas, 1,,true has two paths to upcast to Int ∨ Bool, one via Int and the other via Bool.

1) **First path:** Upcast of 1,,true to Int ∨ Bool via Int:

$$\dfrac{\textsc{typ-mergea}\ \dfrac{}{1,,\mathsf{true} : \mathsf{Int} \wedge \mathsf{Bool}} \qquad \dfrac{\dfrac{\mathsf{Int} <: \mathsf{Int}}{\mathsf{Int} \wedge \mathsf{Bool} <: \mathsf{Int}}\ \textsc{s-andb}}{\mathsf{Int} \wedge \mathsf{Bool} <: \mathsf{Int} \vee \mathsf{Bool}}\ \textsc{s-orb}}{\Gamma \vdash 1,,\mathsf{true} : \mathsf{Int} \vee \mathsf{Bool}}\ \textsc{typ-sub}$$

2) **Second path:** Upcast of 1,,true to Int $\vee$ Bool via Bool:

$$\dfrac{\textsc{typ-mergea}\ \dfrac{}{1,,\mathsf{true} : \mathsf{Int} \wedge \mathsf{Bool}} \qquad \dfrac{\dfrac{\mathsf{Bool} <: \mathsf{Bool}}{\mathsf{Int} \wedge \mathsf{Bool} <: \mathsf{Bool}}\ \textsc{s-andc}}{\mathsf{Int} \wedge \mathsf{Bool} <: \mathsf{Int} \vee \mathsf{Bool}}\ \textsc{s-orc}}{\Gamma \vdash 1,,\mathsf{true} : \mathsf{Int} \vee \mathsf{Bool}}\ \textsc{typ-sub}$$

Therefore the upcast of 1,,true to Int $\vee$ Bool is rejected with the restricted subsumption rule. The essence of restricted subsumption rule is to reject the programs that may be a cause of non-determinism in switch expression.

Challenges with unambiguous relation. Restricting multiple upcasts with unambiguous relation may assist to make the calculus deterministic but unambiguous relation itself is non-trivial in metatheory. Specifically, it becomes challenging with multiple type annotations with functions. For example, $\lambda x.e$: Int $\rightarrow$ Int $\wedge$ Bool : Int $\rightarrow$ Int : Int $\rightarrow$ Int $\vee$ Bool is deterministic as long as we carry the middle type i.e Int $\rightarrow$ Int. As soon as we drop the middle type, application of such lambdas may become non-deterministic. The contravariance of function input type adds further complexities. Possible remedies in such situations is to carry a list of type annotations with functions.

Proposal 2: first-match semantics

A natural approach to select a particular branch of a switch expression is to follow the first-match semantics, meaning that first branch that matches the type of scrutinee will be selected. In this remedy we propose to keep disjointness in merges but eliminate disjointness from switches. Merges will be deterministic as in disjoint intersection types Oliveira et al. [2016]. Whereas, the switches will follow the first-match semantics.

For example, with this approach, first branch will be selected in the following code. This is because 1,,true can be considered as a value of type Int and the type of scrutinee i.e Int ∧ Bool matches (is a subtype of) Int.

```
Bool isInt (x : Int | Bool) = switch (x)
                                (x:Int)   → true
                                (y:Bool)  → false

isInt (1,,true)
```

However, first-match semantics selects first branch in following code as well because of the same reason but with type Bool in first branch.

```
Bool isInt (x : Int | Bool) = switch (x)
                                (x:Bool)   → false
                                (y:Int)    → true

isInt (1,,true)
```

The semantics of the program may change by reordering the branches in this approach. The first code snippet returns true, whereas the second code snippet returns false.

Proposal 3: parallel application

Another proposal is to adopt the parallel application. In this approach, the code in all of the branches will be executed to which the type of scrutinee matches. Final result will be a merge of the values from all of the (match) branches. For example, the result of the following program will be (false,,true).

```
Bool isInt (x : Int | Bool) = switch (x)
                                (x:Bool)   → false
                                (y:Int)    → true

isInt (1,,true)
```

Challenge with parallel application. However, on of the challenges in naive implementation of this approach is that it may generate ill-typed programs as in the above example. The return value (false,,true) is an ill-typed program in disjoint intersection types. This is because (false,,true) is of type Bool ∧ Bool and Bool is not disjoint to Bool. If we allow type-checking such programs, the overall calculus will still be non-deterministic due to the ambiguous merge operator. We further propose two approaches to deal with such an issue.

Disjoint return type of branches. One approach to solve this problem is to employ disjointness in the return type of alternative branches. For example, the following problematic program will be rejected in this case:

```
Bool isInt (x : Int | Bool) = switch (x)
                               (x:Bool)  → false
                               (y:Int)   → true

isInt (1,,true)
```

This is because both of the branches return values of type Bool and Bool is not disjoint to Bool. Whereas, the following program will be accepted:

```
Int | Bool notSucc (x : Int | Bool) = switch (x)
                                       (x:Bool)  → not x
                                       (y:Int)   → succ x

notSucc (1,,True)
```

Note that the first branch returns Bool and the second branch returns Int. Since Int is disjoint to Bool, therefore, it is safe to accept such programs.

Record type for explicit disjointness. The second approach is to use record types with distinct labels as a return type of alternative branches to enforce disjointness. The return type of a switch in this case will always be sound. Therefore, the calculus will not generate ill-typed programs at runtime.

```
{l1:Bool, l2:Bool} isInt (x : Int | Bool) = switch (x)
                                             (x:Int)  → {l1:true}
                                             (y:Bool) → {l2:false}

isInt (1,,true)
```

8.2.2 Multiple Interface Inheritance

Multiple interface inheritance is a prominent feature available in many modern programming languages including Java and Scala. This feature is essential for the extensibility and scalability of software development. The calculi discussed in this thesis do not support multiple interface inheritance. Another line of future work is to allow multiple interface inheritance in λ_u.

Specifically, λ_u with nominal types can further be enriched to support multiple inheritance. The lack of multiple interface inheritance is primarily due to the fact that only the $\top$

type or a nominal type can be declared as a supertype of another nominal type in Δ. For example, in a nominal type environment with $\Delta = \mathsf{Person} <: \top, \mathsf{Robot} <: \top$, the following new declaration is allowed:

$$\Delta = \mathsf{Person} <: \top, \mathsf{Robot} <: \top, \boxed{\mathsf{Student} <: \mathsf{Person}}$$

We add a new nominal type named `Student` and extend it with `Person`. Notice that only the $\top$ type or a nominal type is declared as a parent type in Δ. An attempt to declare multiple parents of a nominal type is rejected. For example, the following extension of Δ is not allowed:

$$\Delta = \mathsf{Person} <: \top, \mathsf{Robot} <: \top, \boxed{\mathsf{Hybrid} <: \mathsf{Person}}, \boxed{\mathsf{Hybrid} <: \mathsf{Robot}}$$

This restricts multiple inheritance due to the fact that only one type can be declared as a parent type of another nominal type. It is essential for the multiple interface inheritance that a nominal type may define multiple parent types. Multiple inheritance can be achieved by allowing intersection types to be a supertype of nominal types in Δ. Specifically, allowing intersections as parent types in Δ will allow the following declaration, which naturally provides multiple interface inheritance:

$$\Delta = \mathsf{Person} <: \top, \mathsf{Robot} <: \top, \boxed{\mathsf{Hybrid} <: \mathsf{Person} \wedge \mathsf{Robot}}$$

8.2.3 Explicit Disjointness of Nominal Types

The Ceylon language employs an `of` construct to explicitly declare the disjointness of nominal types. We will elaborate this using the following Ceylon code:

```
abstract class Student() of PG | UG {}
class PG() extends Student() {}
class UG() extends Student() {}
```

The first line of the code creates a `Student` class and declares two subtypes of `Student`, `PG` and `UG` using the `of` construct. This declarations marks `PG` and `UG` disjoint. It is safe to use `PG` and `UG` in alternative branches of a switch expressions. On the other hand, it is prohibited to create a class that extends both `PG` and `UG` to retain determinism. The class `PG` and the class `UG` must later be defined in the code. The formalization of such a construct is also an interesting line of future work. In essence, the nominal type environment will be revised to handle the `of` construct. Each entry in the revised Δ contain three components. The name of the newly defined nominal type, its parent type(s), and its subtype(s).

$$\Delta = \mathsf{Student} <: \top \boxed{* [\mathsf{PG}, \mathsf{UG}]}, \mathsf{PG} <: \mathsf{Student} \boxed{* [\,]}, \mathsf{UG} <: \mathsf{Student} \boxed{* [\,]}$$

8.2.4 Gradual Typing

Gradual typing is another line of interest in practical programming languages with significant recent development. Union types naturally provide gradual typing in a restricted fashion in such a way that actual type is among the certain candidates from a union of types. In gradual typing, on the other hand, no information of the unknown type is statically available. Inclusion of gradual typing with λ_u is another line of future work with practical interest. The naive addition of gradual typing with disjoint switches may allow the following program (where * denotes unknown type):

```
Bool isInteger (x : Int | *) {
switch (x):
    Int  → true
    *    → false
}
```

The type-based switch construct in this code snippet is exhaustive, but how do we make sure that the two branches will not overlap? This question gives birth to a particular research question of redefining the disjointness in the presence of union types, type-based switches, and the unknown type (*). The unknown type makes the disjointness non-trivial because of the unavailability of the static type information.

Bibliography

[Citing pages are listed after each reference.]

Andreas Abel and Dulma Rodriguez. 2008. Syntactic metatheory of higher-order subtyping. In *International Workshop on Computer Science Logic*. Springer, 446–460. [cited on page 10]

Robin Adams. 2006. Pure type systems with judgemental equality. *Journal of Functional Programming* 16, 2 (2006), 219–246. [cited on page 138]

João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. 2017. Disjoint Polymorphism. In *European Symposium on Programming (ESOP)*. [cited on pages 20, 40, 54, 56, and 135]

Nada Amin, Adriaan Moors, and Martin Odersky. 2012. Dependent object types. In *19th International Workshop on Foundations of Object-Oriented Languages*. [cited on pages 10, 12, 13, and 104]

Nada Amin and Tiark Rompf. 2017. Type Soundness Proofs with Definitional Interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. 14 pages. [cited on page 140]

Davide Ancona and Andrea Corradi. 2014. Sound and complete subtyping between coinductive types for object-oriented languages. In *European Conference on Object-Oriented Programming*. Springer, 282–307. [cited on page 133]

Davide Ancona and Andrea Corradi. 2016. Semantic subtyping for imperative object-oriented languages. *ACM SIGPLAN Notices* 51, 10 (2016), 568–587. [cited on page 133]

Peter B. Andrews. 1986. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Academic Press, Inc. [cited on page 137]

Inc Apple. 2021. Swift language guide. `https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html` [cited on page 135]

Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. Nullaway: Practical type-based null safety for java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 740–750. [cited on page 135]

Franco Barbanera, Mariangiola Dezaniciancaglini, and Ugo Deliguoro. 1995. Intersection and union types: syntax and semantics. *Information and Computation* 119, 2 (1995), 202–230. [cited on pages 4, 10, 17, 33, and 133]

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, 4 (1983), 931–940. [cited on pages 6 and 104]

Jon Barwise and Robin Cooper. 1981. Generalized quantifiers and natural language. In *Philosophy, language, and artificial intelligence*. Springer, 241–301. [cited on page 10]

Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. 2003. CDuce: an XML-centric general-purpose language. *ACM SIGPLAN Notices* 38, 9 (2003), 51–63. [cited on pages 7, 8, 54, and 133]

Giovanni Bernardi, Ornela Dardha, Simon J Gay, and Dimitrios Kouzapas. 2014. On duality relations for session types. In *International Symposium on Trustworthy Global Computing*. Springer, 51–66. [cited on page 138]

Jan Bessai, Boris Düdder, Andrej Dudenhefner, Tzu-Chun Chen, and Ugo de'Liguoro. 2015. Typing classes and mixins with intersection types. *arXiv preprint arXiv:1503.04911* (2015). [cited on page 104]

Xuan Bi and Bruno C d S Oliveira. 2018. Typed first-class traits. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. [cited on page 21]

Xuan Bi, Bruno C d S Oliveira, and Tom Schrijvers. 2018a. The essence of nested composition. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. [cited on page 21]

Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2018b. The Essence of Nested Composition. In *European Conference on Object-Oriented Programming (ECOOP)*. [cited on pages 83 and 135]

Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. Distributive Disjoint Polymorphism for Compositional Programming. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 381–409. [cited on page 135]

Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding typescript. In *European Conference on Object-Oriented Programming*. Springer, 257–281. [cited on pages 7, 10, and 103]

David Binder, Julian Jabs, Ingo Skupin, and Klaus Ostermann. 2019. Decomposition diversity with symmetric data and codata. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 30. [cited on page 138]

Richard Bird and Oege de Moor. 1996. *The Algebra of Programming*. Prentice-Hall. `http://www.cs.ox.ac.uk/publications/books/algebra/` [cited on pages 10, 137, and 138]

Olivier Blanvillain, Jonathan Immanuel Brachthäuser, Maxime Kjaer, and Martin Odersky. 2022. Type-Level Programming with Match Types. *Proc. ACM Program. Lang.* 6, POPL, Article 37 (jan 2022), 24 pages. `https://doi.org/10.1145/3498698` [cited on page 135]

François Bourdoncle and Stephan Merz. 1997. Type checking higher-order polymorphic multi-methods. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 302–315. [cited on pages 136 and 137]

R. M. Burstall, D. B. MacQueen, and D. T. Sannella. 1981. *HOPE: An experimental applicative Language*. Technical Report CSR-62-80. Computer Science Dept, Univ. of Edinburgh. [cited on pages 7 and 21]

Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C Mitchell. 1989. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on functional programming languages and computer architecture*. 273–280. [cited on pages 10, 20, 28, 55, 109, and 139]

Luca Cardelli. 1996. Type systems. *ACM Computing Surveys (CSUR)* 28, 1 (1996), 263–264. [cited on page 3]

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. [cited on page 139]

Luca Cardelli and Peter Wegner. 1985. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)* 17, 4 (1985), 471–523. [cited on pages 4, 5, 12, 20, 28, 55, and 136]

Giuseppe Castagna and Alain Frisch. 2005. A gentle introduction to semantic subtyping. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*. 198–199. [cited on pages 33 and 133]

Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. 1995. A calculus for overloaded functions with subtyping. *Information and Computation* 117, 1 (1995), 115–135. [cited on pages 4, 5, 21, 28, 83, 136, and 137]

Giuseppe Castagna and Victor Lanvin. 2017. Gradual typing with union and intersection types. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–28. [cited on page 133]

Giuseppe Castagna, Victor Lanvin, Mickaël Laurent, and Kim Nguyen. 2019. Revisiting Occurrence Typing. *arXiv preprint arXiv:1907.05590* (2019). [cited on page 134]

Giuseppe Castagna, Mickaël Laurent, Kim Nguyen, and Matthew Lutze. 2022. On type-cases, union elimination, and occurrence typing. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 75. [cited on page 134]

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. 2014a. Polymorphic Functions with Set-Theoretic Types: Part 1: Syntax, Semantics, and Evaluation. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 5–17. `https://doi.org/10.1145/2535838.2535840` [cited on pages 7, 8, 23, 24, and 54]

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. 2014b. Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 5–17. [cited on pages 136 and 137]

Arthur Charguéraud. 2011. The Locally Nameless Representation. *Journal of Automated Reasoning* (2011). [cited on page 123]

Avik Chaudhuri. 2015. Flow: a static type checker for JavaScript. *SPLASH-I In Systems, Programming, Languages and Applications: Software for Humanity* (2015). [cited on pages 7, 10, and 103]

Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. 2017. Fast and precise type checking for JavaScript. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30. [cited on page 135]

Curtis Clifton, Todd Millstein, Gary T Leavens, and Craig Chambers. 2006. MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28, 3 (2006), 517–575. [cited on page 136]

Adriana B Compagnoni and Benjamin C Pierce. 1996. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science* 6, 5 (1996), 469–501. [cited on pages 5 and 18]

Irving M Copi, Carl Cohen, and Victor Rodych. 2018. *Introduction to logic*. Routledge. [cited on page 29]

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 1981. Functional characters of solvable terms. *Mathematical Logic Quarterly* 27, 2-6 (1981), 45–58. [cited on page 18]

Douglas Crockford. 2008. *JavaScript: The Good Parts: The Good Parts*. " O'Reilly Media, Inc.". [cited on page 3]

Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin. 1958. *Combinatory logic*. Vol. 1. North-Holland Amsterdam. [cited on pages 17 and 29]

Oliver Danvy and Andrzej Filinski. 1992. Representing Control: A Study of the CPS transformation. *Mathematical Structures in Computer Science* 2, 4 (December 1992), 361–391. [cited on page 137]

Ornela Dardha, Daniele Gorla, and Daniele Varacca. 2013. Semantic subtyping for objects and classes. In *Formal Techniques for Distributed Systems*. Springer, 66–82. [cited on page 133]

Rowan Davies and Frank Pfenning. 2000. Intersection types and computational effects. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 198–208. [cited on pages 35 and 45]

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, subtyping, and type inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 60–72. [cited on page 61]

Kosta Došen. 1989. Logical constants as punctuation marks. *Notre Dame J. Formal Logic* 30, 3 (06 1989), 362–381. `https://doi.org/10.1305/ndjfl/1093635154` [cited on page 138]

Joshua Dunfield. 2014. Elaborating intersection and union types. *Journal of Functional Programming* 24, 2-3 (2014), 133–165. [cited on pages 4, 7, 14, 15, 17, 18, 19, 20, 23, 33, 54, 83, 84, 93, 94, 97, 133, and 143]

Joshua Dunfield and Neel Krishnaswami. 2019. Bidirectional Typing. *arXiv preprint arXiv:1908.05839* (2019). [cited on page 139]

Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism. In *ICFP*. [cited on page 139]

Joshua Dunfield and Frank Pfenning. 2003. Type assignment for intersections and unions in call-by-value languages. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, 250–266. [cited on pages 33 and 133]

Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 268–277. [cited on page 18]

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2002. Semantic subtyping. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 137–146. [cited on page 133]

Jacques Garrigue. 1998. Programming with polymorphic variants. In *ML workshop*. [cited on pages 7 and 21]

Gerhard Gentzen. 1934. Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift* 39 (1934), 176–210, 405–431. [cited on page 137]

Adele Goldberg and David Robson. 1983. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc. [cited on page 136]

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. 2000. *The Java language specification*. Addison-Wesley Professional. [cited on page 3]

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. 2021. The Java language specification. `https://docs.oracle.com/javase/specs/jls/se14/html/index.html` [cited on page 24]

Eric Gunnerson. 2012. Nullable Types. In *A Programmer's Guide to C# 5.0*. Springer, 247–250. [cited on pages 7 and 26]

Haruo Hosoya and Benjamin C Pierce. 2003. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology (TOIT)* 3, 2 (2003), 117–148. [cited on page 133]

William A Howard. 1980. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), 479–490. [cited on pages 17 and 29]

Xuejing Huang and Bruno C. d. S. Oliveira. 2020. A Type-Directed Operational Semantics For a Calculus with a Merge Operator. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 26:1–26:32. `https://doi.org/10.4230/LIPIcs.ECOOP.2020.26` [cited on pages 19, 20, 83, 85, and 87]

Xuejing Huang and Bruno C d S Oliveira. 2021. Distributing intersection and union types with splits and duality (functional pearl). *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–24. [cited on pages 6, 14, 15, 36, 46, 47, and 61]

DeLesley S. Hutchins. 2010. Pure Subtype Systems. (2010). [cited on page 139]

Atsushi Igarashi and Hideshi Nagira. 2006. Union types for object-oriented programming. In *Proceedings of the 2006 ACM symposium on Applied computing*. 1435–1441. [cited on page 133]

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (2001), 396–450. [cited on page 46]

Atsushi Igarashi and Mirko Viroli. 2002. On Variance-Based Subtyping for Parametric Types. In *ecoop*. sv, Malaga, Spain, 441–469. To appear in ACM Transactions on Programming Languages and Systems.. [cited on page 140]

LSV Jutting. 1993. Typing in pure type systems. *Information and Computation* 105, 1 (1993), 30–41. [cited on page 138]

Gavin King. 2013. The Ceylon language specification, version 1.0. [cited on pages 4, 7, 8, 10, 25, and 103]

Gavin King. 2016. Disjointness in Ceylon. `http://web.mit.edu/ceylon_v1.3.3/ceylon-1.3.3/doc/en/spec/html_single` [cited on page 10]

Foundation Kotlin. 2021. Kotlin programming language. `https://kotlinlang.org/` [cited on page 135]

Saunders Mac Lane. 1998. *Categories for the Working Mathematician* (2 ed.). Number 5 in Graduate Texts in Mathematics. Springer. [cited on pages 11 and 137]

David MacQueen, Gordon Plotkin, and Ravi Sethi. 1984. An ideal model for recursive polymorphic types. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 165–174. [cited on pages 33 and 133]

James McKinna and Robert Pollack. 1993. Pure type systems formalized. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 289–305. [cited on page 138]

Corporation Microsoft. 2023. Union Types in TypeScript. `https://www.typescriptlang.org/docs/handbook/unions-and-intersections.html` [cited on page 26]

Fabian Muehlboeck and Ross Tate. 2018. Empowering union and intersection types with integrated subtyping. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29. [cited on pages 4, 9, 17, 23, 46, 50, and 134]

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)* 9, 3 (2008), 23. [cited on pages 28 and 139]

Abel Nieto, Yaoyu Zhao, Ondřej Lhoták, Angela Chang, and Justin Pu. 2020. Scala with Explicit Nulls. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs))*. 25:1–25:26. [cited on pages 7, 26, and 135]

Martin Odersky. 2021. Scala 3: A next generation compiler for Scala. `https://dotty.epfl.ch` [cited on pages 4 and 7]

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. [cited on pages 3, 10, and 103]

Bruno C. d. S. Oliveira, Zhiyuan Shi, and Joao Alpuim. 2016. Disjoint intersection types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 364–377. [cited on pages 6, 10, 14, 15, 19, 20, 34, 37, 39, 44, 50, 83, 84, 86, 98, 135, and 145]

Klaus Ostermann and Julian Jabs. 2018. Dualizing Generalized Algebraic Data Types by Matrix Transposition. In *European Symposium on Programming*. Springer, 60–85. [cited on page 138]

Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. 2008. Practical pluggable types for Java. In *Proceedings of the 2008 international symposium on Software testing and analysis*. 201–212. [cited on page 135]

Gyunghee Park, Jaemin Hong, Guy L Steele Jr, and Sukyoung Ryu. 2019. Polymorphic symmetric multiple dispatch with variance. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–28. [cited on page 137]

Benjamin C Pierce. 1991. *Programming with intersection types, union types*. Technical Report. and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University. [cited on pages 18, 23, 33, and 133]

Benjamin C Pierce. 2002a. Programming with intersection types, union types, and polymorphism. (2002). [cited on pages 4 and 17]

Benjamin C. Pierce. 2002b. *Types and Programming Languages* (1st ed.). The MIT Press. [cited on pages 3, 6, and 16]

Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2010. Software foundations. *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html* (2010). [cited on page 16]

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000). [cited on page 139]

Garrel Pottinger. 1980. A type assignment for the strongly normalizable λ-terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism* (1980), 561–577. [cited on page 18]

Dag Prawitz. 1979. Proofs and the Meaning and Completeness of the Logical Constants. In *Essays on Mathematical and Philosophical Logic. Synthese Library (Studies in Epistemology, Logic, Methodology, and Philosophy of Science)*. [cited on page 137]

Baber Rehman, Xuejing Huang, Ningning Xie, and Bruno C. d. S. Oliveira. 2022. Union Types with Disjoint Switches. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 25:1–25:31. `https://doi.org/10.4230/LIPIcs.ECOOP.2022.25` [cited on page 136]

John C Reynolds. 1988. Preliminary design of the programming language Forsythe. (1988). [cited on pages 10, 14, 15, 19, 20, 21, 50, 83, and 84]

John C Reynolds. 1997. Design of the Programming Language F orsythe. In *ALGOL-like languages*. Springer, 173–233. [cited on page 20]

Nick Rioux, Xuejing Huang, Bruno C d S Oliveira, and Steve Zdancewic. 2023. A Bowtie for a Beast: Overloading, Eta Expansion, and Extensible Data Types in F⍰. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 515–543. [cited on page 136]

Richard Routley and Robert K Meyer. 1972. The semantics of entailment—III. *Journal of philosophical logic* 1, 2 (1972), 192–208. [cited on page 46]

Paula Severi and Erik Poll. 1994. Pure type systems with definitions. In *International Symposium on Logical Foundations of Computer Science*. Springer, 316–328. [cited on page 138]

Alex K Simpson. 1994. The proof theory and semantics of intuitionistic modal logic. (1994). [cited on pages 28 and 139]

Martin Steffen and Benjamin Pierce. 1994. Higher-order subtyping. (1994). [cited on page 10]

Bjarne Stroustrup. 1986. An overview of C++. In *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*. 7–18. [cited on pages 3 and 136]

Ross Tate. 2011. Overloading in Ceylon. `https://github.com/ceylon/ceylon-spec/issues/73` [cited on page 8]

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of Typed Scheme. *ACM SIGPLAN Notices* 43, 1 (2008), 395–406. [cited on page 134]

Sam Tobin-Hochstadt and Matthias Felleisen. 2010. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. 117–128. [cited on page 134]

Tarmo Uustalu and Varmo Vene. 2008. Comonadic notions of computation. *Electronic Notes in Theoretical Computer Science* 203, 5 (2008), 263–284. [cited on page 11]

Steffen van Bakel, Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Yoko Motohoma. 2000. *The minimal relevant logic and the call-by-value lambda calculus*. Technical Report. Citeseer. [cited on page 46]

LS van Benthem Jutting, James McKinna, and Robert Pollack. 1993. Checking algorithms for pure type systems. In *International Workshop on Types for Proofs and Programs*. Springer, 19–61. [cited on page 138]

Guido Van Rossum, Fred L Drake, et al. 1995. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam. [cited on page 3]

Adriaan Van Wijngaarden, Barry J Mailloux, John EL Peck, Cornelius HA Koster, M Sintzoff, CH Lindsey, LGLT Meertens, and RG Fisker. 1969. Report on the algorithmic language ALGOL 68. *Numer. Math.* 14, 1 (1969), 79–218. [cited on page 6]

Adriaan van Wijngaarden, Barry James Mailloux, John Edward Lancelot Peck, Cornelis HA Koster, CH Lindsey, M Sintzoff, Lambert GLT Meertens, and RG Fisker. 2012. *Revised report on the algorithmic language Algol 68*. Springer Science & Business Media. [cited on page 6]

Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 60–76. [cited on pages 4, 5, 28, and 136]

Ningning Xie, Bruno C d S Oliveira, Xuan Bi, and Tom Schrijvers. 2020. Row and bounded polymorphism via disjoint polymorphism. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. [cited on page 21]

Xu Xue, Bruno C d S Oliveira, and Ningning Xie. 2022. Applicative Intersection Types. In *Programming Languages and Systems: 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings*. Springer, 155–174. [cited on page 90]

Yanpeng Yang and Bruno C. d. S. Oliveira. 2017. Unifying typing and subtyping. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 47. [cited on page 139]

Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. 2018. Julia subtyping: a rational reconstruction. *Proceedings of the*

ACM on Programming Languages 2, OOPSLA (2018), 1–27. [cited on pages 24, 104, and 137]

Jan Zwanenburg. 1999. Pure type systems with subtyping. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 381–396. [cited on page 138]

Part VI

Technical Appendix

A Alternative Disjointness

We discuss another variant of the disjointness algorithm that depends on so called Common Ordinary Subtypes (COST) in this chapter. Recall that the disjointness algorithm discussed in Section 3.3 is set-based (LOS). The COST defined in this chapter is not set-based. We also prove the equivalence of standard disjointness specifications and a variant based on COST in this chapter.

Syntax and subtyping. Syntax and the subtyping stays the same as in λ_u with intersection types and are shown in Figure A.1. Types, expressions, values, context and subtyping have already been explained. Subtyping relation preserves the standard properties of reflexivity and transitivity.

Lemma A.1 (Subtyping Reflexivity). $A <: A$

Lemma A.2 (Subtyping Transitivity). *If* $A <: B$ *and* $B <: C$ *then* $A <: C$

A.1 Common Ordinary Subtypes (COST)

COST play an integral role in the design of the disjointness algorithm presented in this chapter. This section explains the COST in detail by discussing the COST specifications as well as the corresponding algorithm that computes COST.

COST Specifications. The specifications for the COST are shown in Definition 11. It trivially states that two types A and B share a COST if there exist an ordinary type C such that C is subtype of A and B. The ordinary types are shown in the middle of Figure A.1. Int, $A \to B$, and Null constitute ordinary types.

Definition 11 (COST Specifications). $\mathrm{A} \sqcap_s \mathrm{B} ::= \exists\, \mathrm{C}, \mathsf{Ord}\ C$ and $C <: A$ and $C <: B$

$$
\begin{array}{rcl}
A, B, C & ::= & \top \mid \bot \mid \mathsf{Int} \mid A \to B \mid A \vee B \mid A \wedge B \mid \mathsf{Null} \\
e & ::= & x \mid i \mid \lambda x.e \mid e_1\, e_2 \mid \mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} \mid \mathsf{null} \\
v & ::= & i \mid \lambda x.e \mid \mathsf{null} \\
\Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

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

$$
\frac{}{A <: \top}\ \textsc{S-Top} \qquad \frac{}{\mathsf{Int} <: \mathsf{Int}}\ \textsc{S-Int} \qquad \frac{}{\bot <: A}\ \textsc{S-Bot} \qquad \frac{}{\mathsf{Null} <: \mathsf{Null}}\ \textsc{S-Null} \qquad \frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2}\ \textsc{S-Arrow}
$$

$$
\frac{A <: C \qquad B <: C}{A \vee B <: C}\ \textsc{S-OrA} \qquad \frac{A <: B}{A <: B \vee C}\ \textsc{S-OrB} \qquad \frac{A <: C}{A <: B \vee C}\ \textsc{S-OrC} \qquad \frac{A <: B \qquad A <: C}{A <: B \wedge C}\ \textsc{S-AndA}
$$

$$
\frac{A <: C}{A \wedge B <: C}\ \textsc{S-AndB} \qquad \frac{B <: C}{A \wedge B <: C}\ \textsc{S-AndC}
$$

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

$$
\frac{}{\mathsf{Ord}\ \mathsf{Int}}\ \textsc{Ord-Int} \qquad \frac{}{\mathsf{Ord}\ A \to B}\ \textsc{Ord-Arrow} \qquad \frac{}{\mathsf{Ord}\ \mathsf{Null}}\ \textsc{Ord-Null}
$$

$\boxed{A^{\circledcirc}}$ *(Union Ordinary Types)*

$$
\frac{}{\top^{\circledcirc}}\ \textsc{UO-Top} \qquad \frac{}{\mathsf{Int}^{\circledcirc}}\ \textsc{UO-Int} \qquad \frac{}{(A \to B)^{\circledcirc}}\ \textsc{UO-Arrow} \qquad \frac{}{\mathsf{Null}^{\circledcirc}}\ \textsc{UO-Unit} \qquad \frac{A^{\circledcirc} \qquad B^{\circledcirc}}{(A \wedge B)^{\circledcirc}}\ \textsc{UO-And}
$$

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

$$
\frac{}{A \lhd A \vee B \rhd B}\ \textsc{USp-Or} \qquad \frac{A_1 \lhd A \rhd A_2}{A_1 \wedge B \lhd A \wedge B \rhd A_2 \wedge B}\ \textsc{USp-OrAndL} \qquad \frac{B_1 \lhd B \rhd B_2}{A \wedge B_1 \lhd A \wedge B \rhd A \wedge B_2}\ \textsc{USp-OrAndR}
$$

Figure A.1: Syntax, subtyping, ordinary, union ordinary and union splittable types for λ_u with intersection types.

Union ordinary and union splittable types. Union ordinary and union splittable types are shown at the bottom in Figure A.1. These types have already been discussed in Chapter 4. Union ordinary types include $\top$, Int, $A \to B$, Null and an intersection of union ordinary types. Any type that is not union ordinary is union splittable.

$\boxed{A \sqcap B}$ *(Common Ordinary Subtypes)*

$$\frac{}{\top \sqcap \top}\ \text{COST-TOP} \qquad \frac{\mathsf{Ord}\ A}{A \sqcap \top}\ \text{COST-ORDL} \qquad \frac{\mathsf{Ord}\ A}{\top \sqcap A}\ \text{COST-ORDR} \qquad \frac{}{\mathsf{Int} \sqcap \mathsf{Int}}\ \text{COST-INT} \qquad \frac{}{\mathsf{Null} \sqcap \mathsf{Null}}\ \text{COST-NULL}$$

$$\frac{}{A_1 \to B_1 \sqcap A_2 \to B_2}\ \text{COST-ARROW} \qquad \frac{A_1 \lhd A \rhd A_2 \qquad A_1 \sqcap B}{A \sqcap B}\ \text{COST-ORLA} \qquad \frac{A_1 \lhd A \rhd A_2 \qquad A_2 \sqcap B}{A \sqcap B}\ \text{COST-ORLB}$$

$$\frac{B_1 \lhd B \rhd B_2 \qquad A \sqcap B_1}{A \sqcap B}\ \text{COST-ORRA} \qquad \frac{B_1 \lhd B \rhd B_2 \qquad A \sqcap B_2}{A \sqcap B}\ \text{COST-ORRB} \qquad \frac{B^{\circledcirc} \qquad A_1 \sqcap B \qquad A_2 \sqcap B \qquad A_1 \sqcap A_2}{(A_1 \land A_2) \sqcap B}\ \text{COST-ANDL}$$

$$\frac{A^{\circledcirc} \qquad A \sqcap B_1 \qquad A \sqcap B_2 \qquad B_1 \sqcap B_2}{A \sqcap (B_1 \land B_2)}\ \text{COST-ANDR}$$

Figure A.2: Common ordinary subtypes based on union splittable types for λ_u.

COST Algorithm. The algorithm that computes COST is shown in Figure A.2. In essence, the COST algorithm computes whether two given types potentially share an ordinary subtype or not. Rule COST-TOP states that $\top$ shares an ordinary subtype with $\top$. This is trivially true, such as Int is a subtype of $\top$. Rules COST-ORDL and COST-ORDR state that $\top$ shares an ordinary subtype with all the ordinary types. Rules COST-INT, COST-NULL, and COST-ARROW are the natural rules for Int, Null, and $A \to B$ respectively.

Rules COST-ORLA, COST-ORLB, COST-ORRA, and COST-ORRB deal with the union splittable types. These rules collectively state that if a type B shares an ordinary subtype with a part (A_1 or A_2) of union splittable type ($A_1 \lhd A \rhd A_2$), then B shares an ordinary subtype with A. This due to the fact that parts (A_1 and A_2) of a union splittable type are subtypes of the original type (A). Rules COST-ANDL and COST-ANDR deal with the intersection types. An intersection type $A_1 \land A_2$ shares a ordinary subtype with B if A_1 shares an ordinary subtype with B, A_2 shares an ordinary subtype with B, and A_1 shares an ordinary subtype with A_2. A side condition of union ordinary B ($B^{\circledcirc}$) must also hold. Note that the side condition of $B^{\circledcirc}$ is essential. The COST algorithm will not be sound without this condition.

Soundness and completeness of COST algorithm. We prove that the COST algorithm is sound and complete with respect to the COST specifications.

Lemma A.3 (COST Soundness)**.** *$A \sqcap B \to A \sqcap_s B$.*

Lemma A.4 (COST Completeness)**.** *$A \sqcap_s B \to A \sqcap B$.*

A.2 Disjointness

The disjointness for λ_u is the converse of COST. Two types are disjoint if they do not share any ordinary subtype. Whereas, the COST algorithm states the otherwise. Therefore, the disjointness is simply the negation of COST and is shown in Definition 12.

Definition 12 (Disjointness Algorithm). $\mathrm{A} *_a \mathrm{B} ::= \neg\,(A \sqcap B)$

Disjointness equivalence. We prove that the novel disjointness based on COST is sound and complete with respect to standard disjointness specifications. The disjointness specifications are shown again in Definition 13 for readability.

Definition 13 ($\wedge$-Disjointness). $A * B ::= \nexists\, \mathrm{C}$, $\mathsf{Ord}\ C$ and $C <: A$ and $C <: B$.

Lemma A.5 (Disjointness equivalence). *$A * B \leftrightarrow \neg\,(A \sqcap_s B)$*

A.3 Typing, Operational Semantics, and Type-safety

The typing and the operational semantics do not essentially require revision for this chapter and are shown in Figure A.3. Both of these relations are standard and have already been explained.

Type-safety and determinism. The standard properties of the type-safety consisting of type preservation and the progress hold in this system. Theorem A.6 states type preservation and the Theorem A.7 states progress. We also show that the reduction is deterministic (Theorem A.8).

Theorem A.6 (Type Preservation). *If $\Gamma \vdash e : A$ and $e \longrightarrow e'$ then $\Gamma \vdash e' : A$.*

Theorem A.7 (Progress). *If $\Gamma \vdash e : A$ then either e is a value; or e can take a step to e'.*

Theorem A.8 (Determinism). *If $\Gamma \vdash e : A$ and $e \longrightarrow e_1$ and $e \longrightarrow e_2$ then $e_1 = e_2$.*

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

$$\textsc{typ-int}\quad \frac{}{\Gamma \vdash i : \mathsf{Int}} \qquad \textsc{typ-null}\quad \frac{}{\Gamma \vdash \mathsf{null} : \mathsf{Null}} \qquad \textsc{typ-var}\quad \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{typ-app}\quad \frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B}$$

$$\textsc{typ-sub}\quad \frac{\Gamma \vdash e : A \qquad A <: B}{\Gamma \vdash e : B} \qquad \textsc{typ-abs}\quad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B} \qquad \textsc{typ-and}\quad \frac{\Gamma \vdash e : A \qquad \Gamma \vdash e : B}{\Gamma \vdash e : A \wedge B}$$

$$\textsc{typ-switch}\quad \frac{\Gamma \vdash e : A \vee B \qquad \Gamma, x : A \vdash e_1 : C \qquad \Gamma, y : B \vdash e_2 : C \qquad A * B}{\Gamma \vdash \mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} : C}$$

$\boxed{e \longrightarrow e'}$ *(Operational semantics)*

$$\textsc{step-appl}\quad \frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2} \qquad \textsc{step-appr}\quad \frac{e \longrightarrow e'}{v\, e \longrightarrow v\, e'} \qquad \textsc{step-beta}\quad \frac{}{(\lambda x.e)\, v \longrightarrow e[x \rightsquigarrow v]}$$

$$\textsc{step-switch}\quad \frac{e \longrightarrow e'}{\mathsf{switch}\ e\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow \mathsf{switch}\ e'\ \{(x : A) \to e_1, (y : B) \to e_2\}}$$

$$\textsc{step-switchl}\quad \frac{\lfloor v \rfloor <: A}{\mathsf{switch}\ v\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow e_1[x \rightsquigarrow v]}$$

$$\textsc{step-switchr}\quad \frac{\lfloor v \rfloor <: B}{\mathsf{switch}\ v\ \{(x : A) \to e_1, (y : B) \to e_2\} \longrightarrow e_2[y \rightsquigarrow v]}$$

Figure A.3: Typing for λ_u.