Union Types with Disjoint Switches

Baber Rehman
The University of Hong Kong

Xuejing Huang
The University of Hong Kong

Ningning Xie
University of Cambridge

Bruno C. d. S. Oliveira
The University of Hong Kong

Abstract

Union types are nowadays a common feature in many modern programming languages. This paper investigates a formulation of union types with an elimination construct that enables case analysis (or switches) on types. The interesting aspect of this construct is that each clause must operate on *disjoint* types. By using disjoint switches, it is possible to ensure *exhaustiveness* (i.e. all possible cases are handled), and that none of the cases overlap. In turn, this means that the order of the cases does not matter and that reordering the cases has no impact on the semantics, helping with program understanding and refactoring. While implemented in the Ceylon language, disjoint switches have not been formally studied in the research literature, although a related notion of disjointness has been studied in the context of *disjoint intersection types*.

We study union types with disjoint switches formally and in a language independent way. We first present a simplified calculus, called the *union calculus* (λ_u), which includes disjoint switches and prove several results, including *type soundness* and *determinism*. The notion of disjointness in λ_u is in essence the *dual* notion of disjointness for intersection types. We then present a more feature-rich formulation of λ_u, which includes intersection types, distributive subtyping and a simple form of nominal types. This extension reveals new challenges. Those challenges require us to depart from the dual notion of disjointness for intersection types, and use a more general formulation of disjointness instead. Among other applications, we show that disjoint switches provide an alternative to certain forms of *overloading*, and that they enable a simple approach to *nullable (or optional) types*. All the results about λ_u and its extensions have been formalized in the Coq theorem prover.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Union types, switch expression, disjointness, intersection types

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.25

Supplementary Material *ECOOP 2022 Artifact Evaluation approved artifact available at*: `https://github.com/baberrehman/disjoint-switches`

Acknowledgements We thank the anonymous reviewers for their helpful and constructive comments. This research was funded by the University of Hong Kong and Hong Kong Research Grants Council projects number 17209519, 17209520 and 17209821.

1 Introduction

Most programming languages support some mechanism to express terms with alternative types. Algol 68 [54, 55] 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 [48]), typically via either *algebraic datatypes* [15] or *variant types* [32]. 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 [44], Flow [24], TypeScript [13], and Ceylon [39], 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) [34, 43].

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 [10, 29, 22]. Typically, such constructs take the form of a type-based case analysis expression.

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 [10, 22] employ a first-match approach. All of these three options have been explored and studied in the literature.

The Ceylon language [39] 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 [2]. 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, programmers 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. 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[1]. 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.

While implemented in the Ceylon language, disjoint switches have not been studied formally. To our knowledge, the work by Muehlboeck and Tate [42] 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 [1]. 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 [45]. That line of work studies calculi with intersection types and a *merge operator* [49]. 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.

In this paper, 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 4) 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 the extended version of this paper. We prove that all the extensions retain the original properties of λ_u. Furthermore, for our subtyping relation in Section 4 we give a *sound*, *complete* and *decidable* algorithmic formulation by extending the algorithmic formulation employing *splittable types* by Huang and Oliveira [36].

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)*

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

types. All the results about λ_u and its extensions have been formalized in the Coq theorem prover. In summary, the contributions of this paper are:

- **The λ_u calculus:** We present a simple calculus with union types, nullable types and a disjoint switch construct. We then present a richer extension of λ_u with intersection types, distributive subtyping and nominal types. In addition, in the extended version of the paper, we study extensions with parametric polymorphism and a subtyping rule to detect empty types. All calculi and extensions are type sound and deterministic.
- **Sound, complete and decidable formulations of disjointness and subtyping:** We present two formulations of disjointness, which are general and language independent. The second formulation is novel and more general, and can be used in a calculus that includes intersection types as well. We also extend a previous subtyping relation [36] to include nominal types. For both disjointness and subtyping we show that the specifications are sound, complete and decidable and present the corresponding algorithmic formulations.
- **Mechanical formalization:** The results about λ_u and its extensions have been formalized in the Coq theorem prover and can be found in the supplementary materials, together with an extended version of the paper.

2 Overview

This section provides some background on union types and some common approaches to eliminate union types. Then it describes the Ceylon approach to union types, and discusses a few applications of union types. Finally, it presents the key ideas and challenges in our work.

2.1 Tagged Union Types

We start with a brief introduction to union types. An expression has a union type $A \vee B$, if it can be considered to have either type A *or* type B. 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 [15] or (polymorphic) variants [32]. 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 $A \vee B$. Using tagged union types, we can implement a safe integer division function, as[2]:

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

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 paper, 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.

[2] Throughout this paper, 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.

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

2.2 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 [29, 47, 42], explicit tags are replaced by implicit coercions represented by the two subtyping rules $A <: A \vee B$ and $B <: A \vee B$. In this paper 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.

Type-directed elimination. Some systems [22] 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
```

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 [29], 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 [22], 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 [33] and Julia [57]. 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 chosen 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.

2.3 Union Types and Disjoint Switches in Ceylon

The Ceylon language [39] 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 2.2, 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 [3], 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.

2.4 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*) [34] in a type-safe way. A similar approach to nullable types has also been recently proposed for Scala [43]. 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 1 presents a part of the subtyping lattice in Ceylon. `Anything`, the 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:

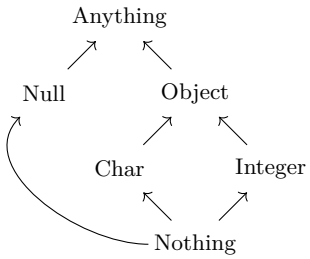

Figure 1 Ceylon's subtyping hierarchy. Note that `Null` only has `Nothing` as its subtype.

```
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 2.2 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.

2.5 Key Ideas in Our Work

We first introduce a simplified formulation of 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 [41, 9, 30, 29, 47, 18]. 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 2.3 and 2.4 can be encoded in λ_u.

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 λ_u in Section 3. Formally, we have:

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

Here we use $\lfloor C \rfloor$ 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 [45].

Disjointness in the presence of intersection types. The variant of λ_u in Section 3 does not include intersection types. Unfortunately, the disjointness definition 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 $\mathsf{Int} \wedge \mathsf{Bool}$, which are not bottom-like, and yet they have no inhabitants. That is, $\mathsf{Int} \wedge \mathsf{Bool}$ is not a subtype of $\bot$, but no value can have both type Int and type Bool. We address this issue by reformulating disjointness in terms of *ordinary types* [28], which are basically primitive types (such as integers or functions). 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 2** ($\wedge$-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.

Distributive Subtyping. In Section 4, 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 6). 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 [36]. 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
```

3 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 4 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* [45] to λ_u, and show that λ_u is type sound and deterministic.

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)*

$$\frac{}{A <: \top}\ \text{S-TOP} \qquad \frac{}{\mathsf{Null} <: \mathsf{Null}}\ \text{S-NULL} \qquad \frac{}{\mathsf{Int} <: \mathsf{Int}}\ \text{S-INT}$$

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

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

Figure 2 Syntax and subtyping for λ_u.

3.1 Syntax

Figure 2 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 Subtyping

The subtyping rules for λ_u are shown at the bottom of Figure 2. 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.

3.3 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 2.5:

```
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, 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** (Bottom-Like Soundness and Completeness). $\lfloor A \rfloor$ *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* $\lfloor C \rfloor$

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:

1. $\boldsymbol{A = \mathsf{Int},\ B = \mathsf{Int} \to \mathsf{Bool}}$ **:** Int and $\mathsf{Int} \to \mathsf{Bool}$ are disjoint types. All common subtypes of Int and $\mathsf{Int} \to \mathsf{Bool}$ are *bottom-like* types, including $\bot$ and unions of $\bot$ types.
2. $\boldsymbol{A = \mathsf{Int} \vee \mathsf{Bool},\ B = \bot}$ **:** $\mathsf{Int} \vee \mathsf{Bool}$ 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. $\boldsymbol{A = \mathsf{Int},\ B = \top}$ **:** 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*.
4. $\boldsymbol{A = \mathsf{Int} \to \mathsf{Bool},\ B = \mathsf{String} \to \mathsf{Char}}$ **:** The types $\mathsf{Int} \to \mathsf{Bool}$ and $\mathsf{String} \to \mathsf{Char}$ are not disjoint, since we can find non-bottom-like types that are subtypes of both types. For instance $\top \to \bot$ 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 [45], 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

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B}\ \text{TYP-APP} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B}\ \text{TYP-ABS}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \text{TYP-VAR}$$

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

Figure 3 Bottom-like types, algorithmic disjointness and typing for λ_u.

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. 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 5** (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 [4].

▶ **Lemma 6** (Disjointness contravariance). *If* $A * B$ *and* $C <: A$ *and* $D <: B$ *then* $C * D$.

3.4 Typing

The typing rules are shown at the bottom of Figure 3. 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 *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 Int ∨ String.

3.5 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 4 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*.

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

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

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

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

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

Approximate Type $\lfloor v \rfloor$

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

Figure 4 Operational semantics and approximate type definitions for λ_u.

3.6 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 7) states that types are preserved during reduction. Progress (Theorem 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 7** (Type Preservation). *If* $\Gamma \vdash e : A$ *and* $e \longrightarrow e'$ *then* $\Gamma \vdash e' : A$.

▶ **Theorem 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 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 9).

▶ **Lemma 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 10** (Determinism). *If* $\Gamma \vdash e : A$ *and* $e \longrightarrow e_1$ *and* $e \longrightarrow e_2$ *then* $e_1 = e_2$.

3.7 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 11** ($\wedge$-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 5 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 12** (Disjointness Equivalence). *Definition 11 ($\wedge$-Disjointness) is sound and complete to Definition 4 ($\bot$-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.

4 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 [45] 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.

4.1 Syntax, Well-formedness and Ordinary Types

The syntax for this section mostly follows from Section 3, with the additional syntax given in Figure 5. 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. 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 [28]. 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 5.

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 [38]. 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.

$$
\begin{array}{lcl}
A, B, C & ::= & \ldots \mid A \wedge B \mid \mathsf{P} \\
A^\circ, B^\circ, C^\circ & ::= & \mathsf{Int} \mid \mathsf{Null} \mid A \to B \mid \mathsf{P} \\
e & ::= & \ldots \mid \mathsf{new}\ \mathsf{P} \\
v & ::= & \ldots \mid \mathsf{new}\ \mathsf{P} \\
\Gamma & ::= & \cdot \mid \Gamma, x : A \\
\Delta & ::= & \cdot \mid \Delta, \mathsf{P}_1 \leq \mathsf{P}_2 \mid \Delta, \mathsf{P} \leq \top
\end{array}
$$

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

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

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

$$\frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \wedge B}\ \text{WFT-AND}$$

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

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

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

Figure 5 Additional syntax and well-formedness.

4.2 Distributive Subtyping

Another interesting feature of this section is the addition of distributive subtyping to λ_u. Ceylon employs an enriched distributive subtyping relation [42] that is based on the B+ logic [50, 53]. To obtain an equivalent algorithmic formulation of subtyping, we employ the idea of *splittable types* [36], but extend that algorithm with the `Null` type and nominal types.

Distributive subtyping relation. Figure 6 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. 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

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

$$\frac{ok\ \Delta \qquad \Delta \vdash A}{\Delta \vdash A \leq A}\ \text{DS-REFL} \qquad \frac{\Delta \vdash A \leq B \qquad \Delta \vdash B \leq C}{\Delta \vdash A \leq C}\ \text{DS-TRANS}$$

$$\frac{\Delta \vdash B_1 \leq A_1 \qquad \Delta \vdash A_2 \leq B_2}{\Delta \vdash A_1 \to A_2 \leq B_1 \to B_2}\ \text{DS-ARROW} \qquad \frac{ok\ \Delta \qquad \mathsf{P} \leq A \in \Delta}{\Delta \vdash \mathsf{P} \leq A}\ \text{DS-PRIM}$$

$$\frac{\Delta \vdash A_1 \leq B \qquad \Delta \vdash A_2 \leq B}{\Delta \vdash A_1 \vee A_2 \leq B}\ \text{DS-ORA} \qquad \frac{ok\ \Delta \qquad \Delta \vdash A_1 \qquad \Delta \vdash A_2}{\Delta \vdash A_1 \leq A_1 \vee A_2}\ \text{DS-ORB}$$

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

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

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

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

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

Figure 6 Distributive subtyping for λ_u with intersection types and nominal types.

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.

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 [36]. 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 $A \cong B \wedge C$ 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. $(A_1 \vee A_2) \wedge B \cong A_1 \wedge B \vee A_2 \wedge B$) 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.

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.

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

$$\frac{\Delta \vdash B_1 <: A_1 \qquad \Delta \vdash A_2 <: B_2}{\Delta \vdash A_1 \to A_2 <: B_1 \to B_2}\ \text{AS-ARROW} \qquad \frac{ok\ (\Delta, \mathsf{P}_1 <: \mathsf{P}_2) \qquad \Delta \vdash \mathsf{P}_2 <: \mathsf{P}_3}{\Delta, \mathsf{P}_1 <: \mathsf{P}_2 \vdash \mathsf{P}_1 <: \mathsf{P}_3}\ \text{AS-PRIMEQ}$$

$$\frac{ok\ \Delta \qquad \Delta \vdash A}{\Delta \vdash A <: A}\ \text{AS-REFL} \qquad \frac{ok\ (\Delta, \mathsf{P}_2 <: A) \qquad \mathsf{P}_1 \neq \mathsf{P}_2 \qquad \Delta \vdash \mathsf{P}_1 <: \mathsf{P}_3}{\Delta, \mathsf{P}_2 <: A \vdash \mathsf{P}_1 <: \mathsf{P}_3}\ \text{AS-PRIMNEQ}$$

$$\frac{A \approxeq A_1 \vee A_2 \qquad \Delta \vdash A_1 <: B \qquad \Delta \vdash A_2 <: B}{\Delta \vdash A <: B}\ \text{AS-ORA} \qquad \frac{ok\ \Delta \qquad \Delta \vdash A}{\Delta \vdash A <: \top}\ \text{AS-TOP}$$

$$\frac{A \approxeq A_1 \vee A_2 \qquad \Delta \vdash B <: A_1}{\Delta \vdash B <: A}\ \text{AS-ORB} \qquad \frac{A \approxeq A_1 \vee A_2 \qquad \Delta \vdash B <: A_2}{\Delta \vdash B <: A}\ \text{AS-ORC}$$

$$\frac{A \approxeq A_1 \wedge A_2 \qquad \Delta \vdash B <: A_1 \qquad \Delta \vdash B <: A_2}{\Delta \vdash B <: A}\ \text{AS-ANDA} \qquad \frac{ok\ \Delta \qquad \Delta \vdash A}{\Delta \vdash \bot <: A}\ \text{AS-BOT}$$

$$\frac{A \approxeq A_1 \wedge A_2 \qquad \Delta \vdash A_1 <: B}{\Delta \vdash A <: B}\ \text{AS-ANDB} \qquad \frac{A \approxeq A_1 \wedge A_2 \qquad \Delta \vdash A_2 <: B}{\Delta \vdash A <: B}\ \text{AS-ANDC}$$

Figure 7 Algorithmic subtyping for λ_u with distributivity, intersection and nominal types.

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.

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 13** (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 14** (Equivalence of subtyping). $\Delta \vdash A \leq B$ *if and only if* $\Delta \vdash A <: B$.

▶ **Lemma 15** (Decidability of subtyping). $\Delta \vdash A \leq B$ *is decidable.*

4.3 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 (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 $\bot$.

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

This rule is adopted by the Ceylon language [42]. 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 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 [49] is allowed in the calculus, intersection types can be inhabited with values (for example, in λ_i [45], 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 the extended version of the paper 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 3.7. Note that now the disjointness definition also contains Δ as an argument to account for nominal types.

▶ **Definition 16** ($\wedge$-Disjointness). $\Delta \vdash A * B ::= \nexists\, C^\circ,\ \Delta \vdash C^\circ <: A \text{ and } \Delta \vdash C^\circ <: B.$

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 $\boxed{\Delta(A)}$

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

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

Figure 8 Lowest ordinary subtypes function and additional typing rule for λ_u with intersection types and nominal types.

Interestingly, while in Section 3 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 16. We further illustrate Definition 16 with a few concrete examples:

1. $A = \mathsf{Int} \vee \mathsf{Bool},\ B = \bot$ **:** Since there is no ordinary type that is a subtype of both $\mathsf{Int} \vee \mathsf{Bool}$ and $\bot$, $\mathsf{Int} \vee \mathsf{Bool}$ and $\bot$ are disjoint types. In general, the $\bot$ type is disjoint to all types because $\bot$ does not have any ordinary subtype.
2. $A = \mathsf{Int} \wedge \mathsf{Bool},\ B = \mathsf{Int} \vee \mathsf{Bool}$ **:** There is no ordinary type that is a subtype of both $\mathsf{Int} \wedge \mathsf{Bool}$ and $\mathsf{Int} \vee \mathsf{Bool}$. Therefore, $\mathsf{Int} \wedge \mathsf{Bool}$ and $\mathsf{Int} \vee \mathsf{Bool}$ are disjoint types. In general, an intersection of two disjoint types ($\mathsf{Int} \wedge \mathsf{Bool}$ in this case) is always disjoint to all types.

4.4 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. 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 8 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 paper, the case for function types $A \to B$ returns $\top \to \bot$. This type is the least ordinary function type, which is a subtype of all function types. Lowest ordinary 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 17.** $\Delta \vdash A *_a B ::= |A|_\Delta \cap |B|_\Delta = \{\}$.

The algorithmic formulation of disjointness in Definition 17 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 16.

Soundness and completeness of algorithmic disjointness. Next, we show that disjointness algorithm is sound and complete with respect to disjointness specifications (Theorem 18). Soundness and completeness of LOS are essential to prove Theorem 18. Both of these properties are shown in Lemma 19 and Lemma 20 respectively.

▶ **Theorem 18** (Disjointness Equivalence). *$\Delta \vdash A *_a B$ if and only if $\Delta \vdash A * B$.*

▶ **Lemma 19** (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 20** (Completeness of $|A|_\Delta$). *$\forall$ A B°, if $\Delta \vdash B^\circ <: A$, then $B^\circ \in |A|_\Delta$, or B° is an arrow type and $\top \to \bot \in |A|_\Delta$.*

4.5 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 at the right side in Figure 8. The typing rule PTYP-PRIM states that under a well-formed context Δ and well-formed type P, $\mathsf{new\ P}$ has type P. No additional reduction rule is required because $\mathsf{new\ P}$ 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:

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

Rule STEP-SWITCHR is updated similarly. All the other rules are essentially the same as in Section 3, 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 21** (Type Preservation). *If $\Delta; \Gamma \vdash e : A$ and $e \longrightarrow e'$ then $\Delta; \Gamma \vdash e' : A$.*

▶ **Theorem 22** (Progress). *If $\Delta; \cdot \vdash e : A$ then either e is a value; or e can take a step to e'.*

▶ **Theorem 23** (Determinism). *If $\Delta; \Gamma \vdash e : A$ and $e \longrightarrow e_1$ and $e \longrightarrow e_2$ then $e_1 = e_2$.*

5 Further Extensions and Discussion

The calculus introduced in Section 3 is a simple foundational lambda calculus with union types, similar to prior work on union types and their elimination forms [10, 29, 22]. In Section 4 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* [4], 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.

5.1 Polymorphism

Polymorphism is an essential feature of almost all the modern programming languages. 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: $\Gamma, \alpha * \mathsf{Int} \vdash \mathsf{switch}\ e\ \{(x : \mathsf{Int}) \to \mathsf{true}, (y : \alpha) \to \mathsf{false}\}$ 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 $A \to B$ is allowed. Whereas, an instantiation of α with Int is rejected by the type system.

Syntax. Figure 9 shows the extension in the syntax of λ_u with polymorphism. Types are extended with type variables α and disjoint quantifiers $\forall(\alpha * G).B$. $\forall(\alpha * G).B$ is also an ordinary type. The reader can think of this extension in the context of bounded quantification [17, 16] 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

$$
\begin{array}{rcl}
A, B, C & ::= & \ldots \mid \alpha \mid \forall(\alpha * G).B \\
e & ::= & \ldots \mid e\,A \mid \Lambda(\alpha * G).e \\
v & ::= & \ldots \mid \Lambda(\alpha * G).e \\
\Gamma & ::= & \ldots \mid \Gamma,\ \alpha * G \\
G & ::= & \top \mid \bot \mid \mathsf{Int} \mid \mathsf{Null} \mid A \to B \\
 & & \mid G_1 \vee G_2 \mid G_1 \wedge G_2 \mid \forall(\alpha * G).B
\end{array}
$$

Lowest Ordinary Subtypes (LOS) $\lvert A\rvert_{\Delta;\Gamma}$		
...	=	...
$\lvert\forall(\alpha * G).B\rvert_{\Delta;\Gamma}$	=	$\{\forall(\alpha * \bot).\bot\}$
$\lvert\alpha\rvert_{\Delta;\Gamma}$	=	$(\lvert\top\rvert_{\Delta;\Gamma})$ - $(\lvert G\rvert_{\Delta;\Gamma})$ where $\alpha * G \in \Gamma$

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

$$\frac{ok\ \Delta \qquad \Delta;\Gamma \vdash \alpha}{\Delta;\Gamma \vdash \alpha <: \alpha}\ \textsc{polys-tvar} \qquad \frac{\Delta;\Gamma \vdash G_1 <: G_2 \qquad \Delta;\Gamma,\ \alpha * G_2 \vdash B_1 <: B_2}{\Delta;\Gamma \vdash \forall(\alpha * G_1).B_1 <: \forall(\alpha * G_2).B_2}\ \textsc{polys-all}$$

$\boxed{\Delta;\Gamma \vdash e : A}$ *(Additional typing rules)*

$$\frac{\Delta;\Gamma \vdash e : \forall(\alpha * G).C \qquad \Delta;\Gamma \vdash G_1 * G}{\Delta;\Gamma \vdash e\,G_1 : C[\alpha \rightsquigarrow G_1]}\ \textsc{ptyp-tap}$$

$$\frac{\Delta;\Gamma,\ \alpha * G \vdash e : B}{\Delta;\Gamma \vdash \Lambda(\alpha * G).e : \forall(\alpha * G).B}\ \textsc{ptyp-tabs}$$

$\boxed{\Delta;\Gamma \vdash e \longrightarrow e'}$ *(Additional reduction rules)*

$$\frac{\Delta;\Gamma \vdash e \longrightarrow e'}{\Delta;\Gamma \vdash e\,B \longrightarrow e'\,B}\ \textsc{polystep-tappl} \qquad \frac{}{\Delta;\Gamma \vdash (\Lambda(\alpha * G).e)\,B \longrightarrow e[\alpha \rightsquigarrow B]}\ \textsc{polystep-tapp}$$

Figure 9 Syntax, additional typing, subtyping, and reduction rules for λ_u with polymorphism.

restricted to ground types (G), which includes all the types except type variables. Ground types are shown at the top left of Figure 9.

Subtyping, Typing and Operational Semantics. Figure 9 shows additional rules in the formalization of λ_u with polymorphism. 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 [4], 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.

Similarly, typing is extended to assign the type to two newly added expressions. Rule PTYP-TAP is for type applications and rule PTYP-TABS is for type abstractions. 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

Section 4, 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 9. 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 24** (Disjointness). $\Delta; \Gamma \vdash A * B ::= |A|_{\Delta;\Gamma} \cap |B|_{\Delta;\Gamma} = \{\}$.

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. 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 25. 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 26.

▶ **Lemma 25** (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 26** (Disjointness Narrowing). *If* $\Delta; \Gamma, \alpha * G_1 \vdash B * C$ *and* $\Delta; \Gamma \vdash G_1 <: G_2$ *then* $\Delta; \Gamma, \alpha * G_2 \vdash B * C$

5.2 A More General Subtyping Rule for Bottom Types

As discussed in Section 4.3, 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 4 and prove Lemma 27 to show this property instead:

▶ **Lemma 27** (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 28** (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.

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.

5.3 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 ∨ 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.

6 Related Work

Union types. Union types were first introduced by MacQueen et al. [41]. 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. [9] solved the problem by reducing all copies of the same redex in parallel. Dunfield and Pfenning [30, 29] 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 [47] 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 [35, 10], and have inspired proposals for object oriented languages [37]. 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 [35]. XDuce employs regular expression types. Pattern matching can be on expressions and types in XDuce. Expressions are considered as special cases of types. CDuce [10] is an extension of XDuce. Work on the more foundational aspects of CDuce, and in particular on *semantic subtyping* [31] and set-theoretic types, also employs a form of first-match semantics elimination construct, though in a different form. In particular, work by Castagna et al. [18, 20] proposes 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 [6, 5, 27] 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 [42] 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 6. 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. 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 [51] 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 [52], 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. [21] extended occurrence typing to refine the type of generic expressions, not just variables. They also studied the combination with gradual 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. [43] 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.'s [43] approach is based on a simplified form of occurrence typing.

Various approaches have been proposed to deal with nullability such as `T?` in Kotlin [40], Swift [7] and Flow [25]. The Checker Framework [46] is another line of related work to detect null pointer deferences in Java programs. Banerjee et al. [8] 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. [14] 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.

Disjoint Intersection Types. Disjoint intersection types were first studied by Oliveira et al. [45] 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, 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* [4] has been studied for calculi with disjoint intersection types.

None of calculi with disjoint intersection types [45, 11, 4, 12] 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 4, 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, $\mathsf{Int} \vee \mathsf{Bool}$ 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 29.** $A * B ::= \nexists\, C^\circ,\ A <: C^\circ \ \textit{and}\ B <: C^\circ.$

which is, in essence, the dual notion of the definition presented in Section 4. Under this definition Int and Bool would be disjoint since we cannot find a common ordinary supertype (and $\mathsf{Int} \vee \mathsf{Bool}$ 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.

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 $\mathsf{Int} \to \mathsf{Int} \wedge \mathsf{Bool} \to \mathsf{Bool}$ 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 [19, 17, 56] has been studied in detail in the literature. Wadler and Blott [56] studied type classes as an alternative way to provide overloading based on parametric polymorphism.

7 Conclusion and Future Work

This work develops the union calculus (λ_u) with union types and a type-based union elimination construct based on disjointness. We presented the operational semantics of the calculus, and showed type-soundness and determinism. Disjointness plays a crucial role for the determinism result, as it ensures that only one branch in the switch elimination construct can apply for any given value. A nice aspect of the work was that we were able to adapt the notion of disjointness used in disjoint intersection types to our variant of λ_u with union types.

We believe that this reinforces fundamental connections between union and intersection types via duality. The addition of intersection types to λ_u lead to some interesting discoveries. In particular, it showed that the notion of disjointness that we were able to formulate, inspired by the work on disjoint intersection types, breaks. This is not showing that the duality stops working. Instead, it shows that the combination of intersections and unions in the same system affects disjointness. As discussed in Section 6, adding union types to calculi with disjoint intersection types leads to a similar problem, and the solution in λ_u can inspire solutions for adding union types to disjoint intersection types.

We plan to extend λ_u for practical programming languages with more advanced features. An interesting line of research for λ_u is to study the addition of the merge operator, which calculi with disjoint intersection types include. The main challenge is that types such as $\mathsf{Int} \wedge \mathsf{Bool}$ become inhabited. It could also be interesting to study a variant of λ_u that uses a best-match approach based on the dynamic type. This would relate to the extensive line of research on *multi-methods* [23] and *multiple dispatching* [26]. Finally, a current limitation of our approach is that it relies on a global context for nominal types. This enables some simplifications, since we can search the global nominal environment for subtypes. However this assumption breaks in a setting where new nominal types can be added. Ceylon solves this issue in a modular way using `of` clauses that enumerate all the possible subtypes that a class can have. It would be interesting to adopt this approach to enable the addition of new nominal types.

References

1 Disjointness in ceylon. URL: `http://web.mit.edu/ceylon_v1.3.3/ceylon-1.3.3/doc/en/spec/html_single`.

2 Overloading in ceylon. URL: `https://github.com/ceylon/ceylon-spec/issues/73`.

3 Union types in typescript. URL: `https://www.typescriptlang.org/docs/handbook/unions-and-intersections.html`.

4 João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. Disjoint polymorphism. In *European Symposium on Programming (ESOP)*, 2017.

5 Davide Ancona and Andrea Corradi. Sound and complete subtyping between coinductive types for object-oriented languages. In *European Conference on Object-Oriented Programming*, pages 282–307. Springer, 2014.

6 Davide Ancona and Andrea Corradi. Semantic subtyping for imperative object-oriented languages. *ACM SIGPLAN Notices*, 51(10):568–587, 2016.

7 Inc Apple. Swift language guide, 2021. URL: `https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html`.

8 Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 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*, pages 740–750, 2019.

9 Franco Barbanera, Mariangiola Dezaniciancaglini, and Ugo Deliguoro. Intersection and union types: syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

10 Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. Cduce: an xml-centric general-purpose language. *ACM SIGPLAN Notices*, 38(9):51–63, 2003.

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

12 Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. Distributive disjoint polymorphism for compositional programming. In Luís Caires, editor, *Programming Languages and Systems*, pages 381–409, Cham, 2019. Springer International Publishing.

13 Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

14 Olivier Blanvillain, Jonathan Immanuel Brachthäuser, Maxime Kjaer, and Martin Odersky. Type-level programming with match types. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022. doi:10.1145/3498698.

15 R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: An experimental applicative language. Technical Report CSR-62-80, Computer Science Dept, Univ. of Edinburgh, 1981.

16 Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on functional programming languages and computer architecture*, pages 273–280, 1989.

17 Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

18 Giuseppe Castagna and Alain Frisch. A gentle introduction to semantic subtyping. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 198–199, 2005.

19 Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.

20 Giuseppe Castagna and Victor Lanvin. Gradual typing with union and intersection types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–28, 2017.

21 Giuseppe Castagna, Victor Lanvin, Mickaël Laurent, and Kim Nguyen. Revisiting occurrence typing. *arXiv preprint arXiv:1907.05590*, 2019.

22 Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. 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*, POPL '14, page 5–17, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2535838.2535840.

23 Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen, editor, *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615, pages 33–56. Springer-Verlag, 1992.

24 Avik Chaudhuri. Flow: a static type checker for javascript. *SPLASH-I In Systems, Programming, Languages and Applications: Software for Humanity*, 2015.

25 Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and precise type checking for javascript. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–30, 2017.

26 Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: Modular open classes and symmetric multiple dispatch for java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, page 130–145, 2000.

27 Ornela Dardha, Daniele Gorla, and Daniele Varacca. Semantic subtyping for objects and classes. In *Formal Techniques for Distributed Systems*, pages 66–82. Springer, 2013.

28 Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 198–208, 2000.

29 Joshua Dunfield. Elaborating intersection and union types. *Journal of Functional Programming*, 24(2-3):133–165, 2014.

30 Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In *International Conference on Foundations of Software Science and Computation Structures*, pages 250–266. Springer, 2003.

31 Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE, 2002.

32 Jacques Garrigue. Programming with polymorphic variants. In *ML workshop*, 1998.

33 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. The java language specification, 2021. URL: `https://docs.oracle.com/javase/specs/jls/se14/html/index.html`.

34 Eric Gunnerson. Nullable types. In *A Programmer's Guide to C# 5.0*, pages 247–250. Springer, 2012.

35 Haruo Hosoya and Benjamin C Pierce. Xduce: A statically typed xml processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.

36 Xuejing Huang and Bruno C d S Oliveira. Distributing intersection and union types with splits and duality (functional pearl). *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–24, 2021.

37 Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1435–1441, 2006.

38 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

39 Gavin King. The ceylon language specification, version 1.0, 2013.

40 Foundation Kotlin. Kotlin programming language, 2021. URL: `https://kotlinlang.org/`.

41 David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 165–174, 1984.

42 Fabian Muehlboeck and Ross Tate. Empowering union and intersection types with integrated subtyping. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.

43 Abel Nieto, Yaoyu Zhao, Ondřej Lhoták, Angela Chang, and Justin Pu. Scala with Explicit Nulls. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 25:1–25:26, 2020.

44 Martin Odersky. Scala 3: A next generation compiler for scala, 2021. URL: `https://dotty.epfl.ch`.

45 Bruno C. d. S. Oliveira, Zhiyuan Shi, and Joao Alpuim. Disjoint intersection types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pages 364–377, 2016.

46 Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical pluggable types for java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212, 2008.

47 Benjamin C Pierce. Programming with intersection types, union types. Technical report, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.

48 Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

49 John C Reynolds. Preliminary design of the programming language forsythe. 1988.

50 Richard Routley and Robert K Meyer. The semantics of entailment—iii. *Journal of philosophical logic*, 1(2):192–208, 1972.

51 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. *ACM SIGPLAN Notices*, 43(1):395–406, 2008.

52 Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 117–128, 2010.

53 Steffen van Bakel, Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Yoko Motohoma. The minimal relevant logic and the call-by-value lambda calculus. Technical report, Citeseer, 2000.

54 Adriaan Van Wijngaarden, Barry J Mailloux, John EL Peck, Cornelius HA Koster, M Sintzoff, CH Lindsey, LGLT Meertens, and RG Fisker. Report on the algorithmic language algol 68. *Numerische Mathematik*, 14(1):79–218, 1969.

55 Adriaan van Wijngaarden, Barry James Mailloux, John Edward Lancelot Peck, Cornelis HA Koster, CH Lindsey, M Sintzoff, Lambert GLT Meertens, and RG Fisker. *Revised report on the algorithmic language Algol 68*. Springer Science & Business Media, 2012.

56 Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, 1989.

57 Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. Julia subtyping: a rational reconstruction. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018.