Resolution as Intersection Subtyping via Modus Ponens

KOAR MARNTIROSIAN and TOM SCHRIJVERS, KU Leuven, Belgium
BRUNO C. D. S. OLIVEIRA, The University of Hong Kong, Hong Kong
GEORGIOS KARACHALIAS, Tweag, France

Resolution and subtyping are two common mechanisms in programming languages. Resolution is used by features such as type classes or Scala-style implicits to synthesize values automatically from contextual type information. Subtyping is commonly used to automatically convert the type of a value into another compatible type. So far the two mechanisms have been considered independently of each other. This paper shows that, with a small extension, subtyping with intersection types can subsume resolution. This has three main consequences. Firstly, resolution does not need to be implemented as a separate mechanism. Secondly, the interaction between resolution and subtyping becomes apparent. Finally, the integration of resolution into subtyping enables *first-class (implicit) environments*. The extension that recovers the power of resolution via subtyping is the *modus ponens* rule of propositional logic. While it is easily added to declarative subtyping, significant care needs to be taken to retain desirable properties, such as *transitivity* and *decidability* of algorithmic subtyping, and *coherence*. To materialize these ideas we develop λ_i^{MP}, a calculus that extends a previous calculus with disjoint intersection types, and develop its metatheory in the Coq theorem prover.

CCS Concepts: • **Software and its engineering → Functional languages**; **Object oriented languages**; **Semantics**.

Additional Key Words and Phrases: resolution, nested composition, family polymorphism, intersection types, coherence, modus ponens

ACM Reference Format:
Koar Marntirosian, Tom Schrijvers, Bruno C. d. S. Oliveira, and Georgios Karachalias. 2020. Resolution as Intersection Subtyping via Modus Ponens. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 206 (November 2020), 30 pages. https://doi.org/10.1145/3428274

1 INTRODUCTION

Subtyping is a well-known programming language feature. It is most notably employed in object-oriented languages, but there are many other uses of subtyping in the literature [Barendregt et al. 1983; Chen 2003; Gay and Hole 2005; Pierce and Sangiorgi 1996]. Subtyping automatically converts the type of a value into another compatible type. One particular form of subtyping arises from *intersection types* [Barendregt et al. 1983; Reynolds 1991]. Intersection types support—among others—a form of *finitary overloading* [Castagna et al. 1992]. For example, in various systems with intersection types, it is possible to build overloaded functions such as:

```
f : (Int -> Int) & (Char -> Char) = succ ,, toUpper
```


Authors' addresses: Koar Marntirosian, mardikoh15@gmail.com; Tom Schrijvers, KU Leuven, Leuven, Belgium; Bruno C. d. S. Oliveira, bruno@cs.hku.hk, The University of Hong Kong, Hong Kong; Georgios Karachalias, georgios.karachalias@tweag.io, Tweag, Paris, France.

2475-1421/2020/11-ART206
https://doi.org/10.1145/3428274

The *merge operator* `(,,)` allows the function `f` to behave as both the `succ` and the `toUpper` functions. Thus, `f` can be applied both to an integer and to a character. When applied to an integer, `f` behaves like the successor function, whereas when applied to a character `f` capitalizes its input. The automatic selection of the appropriate behavior (from the overloaded definition `f`) is based on the type of the input, which triggers an upcast induced by the subtyping relation. For instance, in the case of the application `f 3`, the input of the function must be `Int` and therefore, the following upcast is triggered in such an application:

```
(Int -> Int) & (Char -> Char) <: Int -> Int
```

Resolution is another programming language mechanism. Haskell type classes [Wadler and Blott 1989] employ resolution to automatically *search* for and *compose* instances of type classes. Resolution is also used by Scala's implicits [Oliveira et al. 2010]. Many other programming languages have mechanisms similar to type classes and implicits [Devriese and Piessens 2011; Sozeau and Oury 2008a; White et al. 2015], and consequently employ a form of resolution. Type classes and implicits can also be used to define overloaded functions. For example, in Haskell, we may define:

```
class Over a where  f :: a -> a
instance Over Int where f = succ
instance Over Char where f = toUpper
```

This achieves an overloaded behavior similar to the intersection types version of `f`. Indeed, the Haskell version of `f` can also be applied both to an integer and to a character with similar results. However, in Haskell, the selection of the right function in the application `f 3` uses resolution. In Haskell (and languages with implicits) there is an implicit environment that stores information about type class instances. When class methods are used, resolution searches the implicit environment to find the right instance or to combine existing instances to find a suitable method implementation.

The function `f` illustrates the similarities and overlapping use cases for subtyping and resolution. However there are also various use cases that are not in common. On the one hand, with subtyping we can, for instance, have `Int -> Int <: Int -> Top`. Such a conversion is typically not allowed by resolution mechanisms documented in the literature [Hall et al. 1996; Odersky et al. 2017; Oliveira et al. 2012; Schrijvers et al. 2019; Wadler and Blott 1989]. On the other hand, a key feature of resolution is its ability to compose instances (or implicits). For instance, suppose that we want an overloaded version of `f` for lists of integers. With type classes we can define the instance:

```
instance Over Int => Over [Int] where
  f []       = []
  f (x : xs) = f x : f xs
```

Note that here the context `Over Int` is not necessary in Haskell, but for illustration purposes, we add the redundant context. With this instance, if we call `f [1,2,3]`, then the result is `[2,3,4]`. For this to work, resolution needs to find an instance for `Over [Int]`. While no instance directly matches the required type, resolution composes the instance for lists with the instance for integers to build an instance of the appropriate type. The ability to perform such compositions is lacking in traditional mechanisms with subtyping and intersection types.

This paper shows that, with a small extension, subtyping with intersection types can in fact subsume forms of resolution. The extension that is necessary to recover the power of resolution via subtyping is essentially the logical rule for *modus ponens*:

$$\frac{A \vdash B \to C \qquad A \vdash B}{A \vdash C}\ \textsc{Modus Ponens}$$

Modus ponens states that if from A one can conclude both $B \to C$ and B then one can also conclude C. The key idea in this paper is that we can adapt modus ponens to subtyping, and thus enable its

power (and the essential power of resolution). To accomplish this, we simply reinterpret entailment ($A \vdash B$) as subtyping ($A <: B$). Modus ponens captures the essence of composition that is necessary to enable the following derivation:

```
(Int -> Int) & ((Int -> Int) -> [Int] -> [Int]) <: [Int] -> [Int]
```

which could be used in a language with intersection types and subtyping, to find an overloaded implementation of `f` in a call such as `f [1,2,3]` (just like the version with type classes).

While modus ponens is easy to add in a declarative version of subtyping, significant care needs to be taken to retain desirable properties, such as transitivity and decidability of subtyping. Our work builds and extends upon the NeColus calculus [Bi et al. 2018]—or λ_i^+—which has both transitivity and decidability of subtyping, as well as coherence of the elaboration semantics. However, NeColus does not support modus ponens. Extending the proofs of transitivity, decidability and coherence of NeColus to support modus ponens is highly non-trivial. The key difficulty is that modus ponens makes the context in which a type appears relevant for resolution (which is not the case in NeColus).

Adding modus ponens to subtyping, and viewing resolution as a special case of subtyping, comes with important advantages. Firstly, resolution does not need to be implemented as a completely separate mechanism. This can make implementations as well as metatheory more compact, because the two mechanisms are generalized into a single relation. Secondly, the interaction between resolution and subtyping becomes apparent, which contrasts with the current state-of-the-art where it is ignored.[1] Indeed, various foundational calculi related to Scala formalize either subtyping [Amin et al. 2016, 2012] or implicits [Odersky et al. 2017; Oliveira et al. 2012; Schrijvers et al. 2019], but usually not both. The only exception is the recent work by Jeffery [2019], which studies a calculus with both features (but where resolution and subtyping are modeled independently). Finally, the integration of resolution into subtyping enables *first-class (implicit) environments*. That is, we can create (implicit) environments (encoded as intersection types), use them as arguments, or return them from functions. This is impossible in existing approaches for type classes and calculi with implicits, as there is no explicit notion of a type for environments.

To materialize these ideas we develop λ_i^{MP}: a calculus that extends the NeColus calculus (a calculus with disjoint intersection types [Oliveira et al. 2016]) and develop its metatheory in the Coq theorem prover. In summary, the contributions of this work are:

- **Resolution as subtyping:** We extend subtyping of intersection types with *modus ponens*. This enables composing components from intersections to solve subtyping problems, and (together with other standard intersection rules) subsumes forms of resolution.
- **The λ_i^{MP} calculus:** We develop a concrete calculus that illustrates the idea of subtyping as resolution. The calculus is an extension of NeColus [Bi et al. 2018]. We prove several basic properties, including type-safety, which is proved via an elaboration into the target language λ_c^{MP}, a simply typed lambda calculus extended with coercions.
- **Algorithmic subtyping for λ_i^{MP}:** We incorporate modus ponens in the NeColus subtyping algorithm by (1) restructuring it in two mutually recursive judgments—one of which builds coercions by accumulating them in a continuation-passing form—and (2) adding a loop detector for termination. Our Coq proofs show that our algorithm subsumes that of NeColus, and is *sound*, *complete* and *terminating*; this establishes that the subtyping of λ_i^{MP} is decidable. The proofs of termination and transitivity, which is used to show completeness, are both highly non-trivial. The former uses different arguments for three different kinds of loops, including reasoning about the effectiveness of loop detection. The latter consists of 7 mutually

[1] For example, GHC issue ticket 17295 is a closely related problem, caused by an unexpected interaction between resolution and overlapping "intersections".

recursive lemmas with complex interdependencies. To establish the well-foundness of their proof we employ the AProVE tool [Giesl et al. 2017].

- **Conditional Coherence for λ_i^{MP}:** Establishing coherence of λ_i^{MP} is also a challenging technical goal. The addition of modus ponens makes it a lot harder due to the ability to compose information. We formulate a non-trivial extension of the canonicity relation of NeColus, based on which we build a mechanized partial coherence proof in the Coq theorem prover.

The supplementary material contains a Coq mechanization of the definitions (4k lines) and theorems (5.6k lines) in this paper, as well as a prototype implementation in Haskell (~820 lines). It is available online at https://zenodo.org/record/4043646#.X4gBAHVfhjs.

An extended version of this paper, containing appendices, can be found online, by following this link: https://arxiv.org/abs/2010.06216.

2 BACKGROUND

This section introduces background on resolution, coercive subtyping, intersection types and their use in programming languages.

2.1 Coercive Subtyping, Resolution and their Computational Interpretation

This paper brings together two different programming language features: (implicit) *resolution* as used in Haskell type classes and Scala implicits on the one hand, and *type-directed coercive subtyping* on the other hand. While these two features are very different on the surface, they are in fact both based on similar computational interpretations of logical inference. The different rules of logical inference they use are compatible. In this paper we show how to extend this compatibility to the language design level and how to ensure properties that, while typically of no concern for logical inference, are key for programming languages, like coherence and decidable type checking.

In the sequent notation, the judgment $A \vdash B$ asserts that proposition B can be inferred from proposition A. An inference rule provides a way to construct a valid (or true) judgment. For instance, the inference rule

$$\frac{}{A \vdash A}$$

states that the judgment $A \vdash A$ (from A follows A) is valid for any proposition A.

The $A \vdash B$ judgment can be given a computational interpretation following the "propositions as types" approach [Wadler 2015]. This means that we take the types of a programming language as the possible propositions. The judgment $A \vdash B$ then means that there exists a computable function that turns values of type A into values of type B. We call this function the (computational) evidence for the judgment. The evidence for a judgment follows from the inference rules that have been used to establish that judgment. For instance, the evidence for the trivial $A \vdash A$ judgment above is the identity function $\lambda x.x$. A more advanced inference rule is that for transitivity

$$\frac{A \vdash B \qquad B \vdash C}{A \vdash C}$$

which allows to conclude C from A provided that B can be concluded from A and C from B. Given evidence f for $B \vdash C$ and evidence g for $A \vdash B$, the evidence for $A \vdash C$ is $\lambda x.g\,(f\,x)$.

Coercive Subtyping. A first programming language application of this mechanism is coercive subtyping [Luo 1999]. In this case we write $A <: B$ instead of $A \vdash B$ and call A and B subtype and supertype respectively, rather than premise and conclusion. Also we call the associated evidence of the judgment a coercion. Unlike other forms of subtyping, coercive subtyping changes the runtime representation of a value of type A when upcasting to type B. This change is accomplished by

```
object Test {
  class P                                          // proposition P
  class Q                                          // proposition Q
  class U                                          // proposition U

  implicit def f(implicit o : P) : Q = new Q()     // P -> Q
  implicit def g(implicit o : Q) : U = new U()     // Q -> U
  implicit val p : P = new P()                     // P

  val u : U = implicitly[U]                        // triggers resolution from
                                                   // the implicit environment
}
```

Fig. 1. Encoding of the judgment $P \to Q, Q \to U, P \vdash U$ as a Scala program.

the coercion. Even though the coercion is executed at runtime, it is already statically known at what point in the program it needs to be performed, namely at the point in the static type system where an appeal is made to subtyping. In order to introduce the coercion, coercive subtyping can therefore make use of a static program transformation approach called (type-directed) elaboration.

Elaboration uses the information of static typing to transform a program with implicit appeals to subtyping into a program with explicit applications of coercions. For example, consider a function f that expects a value of type B and a value v of type A where $A <: B$. Hence, in the function call $f\,v$ there is a type mismatch that the type system bridges by appealing to subtyping. Elaboration then transforms this function call into $f\,(c\,v)$ where c is the coercion for $A <: B$. The explicit coercion application replaces the implicit appeal to subtyping.

Resolution. A rather different application of logic inference in programming languages are implicits in Scala. In their basic form, they involve an environment of premises that the programmer can populate and extend with particular types and values of those types as evidence. Functions can take implicit parameters which the type system fills in by searching values of the appropriate type in the implicit environment. Elaboration turns those implicit parameters into explicit ones. This behavior is extended with the modus ponens inference rule, to make it more powerful.

The computational evidence associated to this rule, shown in Section 1, is $\lambda x.(f\,x)\;(g\,x)$ where f is the evidence for $A \vdash B \to C$ and g for $A \vdash B$. The modus ponens rule can be applied recursively, e.g. to establish B from A by modus ponens via $D \to B$ and D, and we call the general reasoning principle *recursive resolution*.

Resolution in Scala. As a concrete example of resolution in programming languages, consider the judgment

$$P \to Q, Q \to U, P \vdash U$$

In this statement, P implies Q, Q implies U and P holds as a fact. Thus, we can conclude that U holds by straightforward logical reasoning.

Figure 1 illustrates how to encode the judgment in Scala. In this Scala program, we encode propositions as classes, and we define a few implicit rules that are added to the "implicit environment" of Scala. The definition of `f` encodes $P \to Q$, the definition of `g` encodes $Q \to U$ and the definition of `p` encodes that P holds. In the definition of the value `u`, we trigger resolution (via the `implicitly` keyword), which essentially searches the implicit environment and attempts to compose the rules

and facts in the implicit environment, to find a value of type U. This value will be given eventually by the synthesized expression `g(f(p))`.

Essentially, in logic, resolution can be seen as a mechanism for proof search. In programming languages, resolution is a mechanism that searches for values of a given type by using the rules and facts of the implicit environment. It can be viewed as a type-directed form of program synthesis.

The connection to our work is that we interpret the resolution relation as subtyping, and the conjunction of propositions in the environment as intersections. Thus, the above judgment corresponds to:

$$(P \to Q)\&(Q \to U)\&P <: U$$

using subtyping and intersection types.

Haskell type classes also employ resolution, but there are several superficial differences to Scala implicits. The name spaces of implicit values—called type classes—is separate from that of other types and has global scope. Moreover type classes are used as a mechanism for overloading function names. Yet, in essence, just like Scala implicits, type classes are based on the modus ponens inference rule for constructing wanted evidence for types in terms of given evidence of other types. We will look at type classes in more detail in Section 3.

2.2 Different Interpretations for Intersection Types in the Literature

There are several different interpretations of intersection types in the literature. Intersection types date back to work from Coppo et al. [1981] and Pottinger [1980] and were first used to characterize the strongly normalizable terms of the λ-calculus.

Following Forsythe [Reynolds 1988], intersection types were used in various other programming languages, including CDuce [Benzaken et al. 2003] and Object Oriented languages like TypeScript[2], Flow [Raskin 1974], Ceylon[3] and Scala [Odersky 2010], enabling multiple inheritance [Pierce 1991]. They were also studied as part of the syntax of several calculi, to enable various forms of polymorphism such as finitary function overloading [Castagna et al. 1992], bounded polymorphism [Pierce 1997] and evaluation-order polymorphism [Dunfield 2015]. Refinement types is yet another feature where intersection types has been studied [Davies and Pfenning 2000; Dunfield 2007; Freeman and Pfenning 1991]. However, in those systems the type Int & Bool is invalid, since Int and Bool are not refinements of each other.

Note that intersection types alone only increase the expressive power of types. While languages with intersection types do not necessarily have a merge operator, the latter adds expressive power to the terms. This term construct has been used in various, but usually limited, forms like in Forsythe or in the work of Castagna et al. [1992], discussed in Section 7.

The use of intersection types in this paper follows the interpretation of Dunfield [2012] who has proposed a particular flavor of intersection types with an unrestricted merge operator, allowing the construction of values that inhabit intersection types such as Int & Bool. In her work, Dunfield uses coercive subtyping, based on the inference rules for the logical conjunction operator. In particular, conjunction elimination is expressed in two rules:

$$\frac{}{A \wedge B \vdash A} \qquad \frac{}{A \wedge B \vdash B}$$

Here, we write $A\&B$, instead of $A \wedge B$, to denote an intersection type. The runtime representation of values of type $A\&B$ is a tuple (v, w) where v has type A and w has type B. The coercions associated with the two inference rules are respectively π_1 and π_2, the projection on respectively the first and second component of the tuple. Dunfield observes that her formulation of the calculus does not

[2] https://www.typescriptlang.org/
[3] https://ceylon-lang.org/

eliminate ambiguous expressions. However, recent work by Oliveira et al. [2016] remedies this issue by introducing a notion of disjoint intersection types.

Our proposed design fits well with this interpretation, i.e. intersection types as a mechanism to implicitly provide coercions on pairs. Our source language provides a layer of convenience on top of an ordinary target language with pairs, by generating various types of coercions involving pairs. In particular, the modus ponens rule is just one of the possible coercions involving pairs.

3 OVERVIEW

This section gives an overview of our work. We are mainly interested in the *type-based overloading* and *resolution* capability at the core of type classes or implicits. In what follows, we use Haskell type classes for illustration, but we could have used Scala implicits (or other similar mechanisms) just as well. We start by establishing the relationship between type-based overloading based on type classes and/or implicits and disjoint intersection types. Then we explain the key idea of our work to add modus ponens to the subtyping relation for a calculus with (disjoint) intersection types to recover resolution. Finally, we discuss several technical challenges that need to be addressed for this addition to work.

3.1 Type-based Overloading with Type Classes

Type classes in Haskell were initially meant to provide a less ad-hoc mechanism to overloading. In Section 1 we have already seen a simple example of an overloaded function `f` defined via a type class. Overloading with type classes is based on *parametric polymorphism* [Reynolds 1974]. All type classes require at least one type argument. This type argument is used to specify the places in the signatures of overloaded methods where different types can occur. For the type class `Over` the method `f :: a -> a` can be overloaded both on the input type and the output type and those two types need to be the same for a particular overloading.

Conventionally, type class instances cannot overlap. That is, two instances for a type class must use distinct types. This property is satisfied by the three instances in Section 1:

```
instance Over Int where ...
instance Over Char where ...
instance Over Int => Over [Int] where ...
```

Adding a fourth instance for a type that is distinct from those in the previous instances (i.e. `Int`, or `Char` or `[Int]`) is allowed. For example, the following `Bool` instance is allowed:

```
instance Over Bool where f = not
```

However adding another instance for integers is not allowed:

```
instance Over Int where f x = x + 2  -- rejected!
```

The compiler complains that there is already another instance in scope for `Over Int`.

Overlapping instances and ambiguity. The restriction of non-overlapping instances is put into place to ensure that there is no ambiguity in a program. If the second instance for integers were accepted then the definition:

```
p = f 1         -- ambiguous: could be 2 or 3
```

would be ambiguous as it could choose either of the two integer instances.

3.2 The Merge Operator and Disjoint Intersection Types

The merge operator, originally proposed by Reynolds [1988], adds expressive power to intersection types. In most calculi with intersection types, a type such as `Int&Char` is not inhabited, since these

two types are *disjoint* (i.e. have no common values), so their intersection is empty. With the merge operator, however, it is possible to build values that inhabit such an intersection type. For example:

```
v : Int & Char = 1 ,, 'c'
```

In the definition of `v`, the merge operator `(,,)`, is used to build a value of type `Int&Char` from an integer and a character. In calculi with a merge operator, intersection types are similar to conventional pair types. The merge operator is used to build values of the intersection types, similarly to how we build values for pairs. The main difference to pairs and pair types is that while for pairs the eliminations (i.e. projections such as `fst` and `snd`) are explicit, for intersection types they are implicit and driven by the type system.

Disjointness and ambiguity. The merge operator is powerful, but without any restrictions it is easy to have ambiguous programs. For instance, consider the following value:

```
n : Int & Int = 1 ,, 2
```

If we use `n` in an expression such as `n + 1` there could be two possible results (`2` and `3`) depending on what integer gets selected from the merge.

Disjoint intersection types [Oliveira et al. 2016] were proposed to solve ambiguity problems in calculi with intersection types and a merge operator. The key idea is to introduce a restriction on merges: a valid merge can only be built from two values with disjoint types. Two types are disjoint if the only supertypes that they have in common are top-like types [Oliveira et al. 2016]. This restriction prevents the merged value `n` from type-checking, while accepting `v`. Furthermore, for previous calculi with disjoint intersection types it has been proven that the disjointness restriction is sufficient to guarantee *coherence* [Reynolds 1991], a correctness property that rules out any ambiguity. This property is important for calculi with non-deterministic semantics, where well-formed expressions can take on multiple meanings, and ensures that these meanings are equivalent, in the sense that the program which the initial expression models will always behave the same.

3.3 Disjoint Merges, Non-Overlapping Instances and Resolution

Disjointness checks for merges and non-overlap checks for type class instances are closely related. The non-overlap check for type classes can be viewed as a simple form of disjointness checking. If we ignore polymorphic types, then the non-overlap check is simply testing whether two types are distinct[4], which is subsumed by a disjointness check for merges.

Type class instances as merges. The fact that disjointness checking relates/subsumes to non-overlapping checking is crucial for expressing code analogous to type class instances as merges. Consider a variant of `f` in Section 1:

```
f : (Int -> Int) & (Char -> Char) & ((Int -> Int) -> [Int] -> [Int])
  = succ ,, toUpper ,, listInstance
```

where `listInstance` is analogous to the type class instance `Over [Int]` from Section 1, with the type `((Int -> Int) -> [Int] -> [Int])`. Here, the merge creates an overloaded function and checks that the three implementations have disjoint types. This is similar to checking whether `Over Int`, `Over Char` and `Over [Int]` overlap. Therefore, we can essentially express type class instances through merges, whose components define the methods of the instances for overloading.

Resolution via Modus Ponens. While we can express some type class instances with disjoint intersection types, an important "class" of instances cannot be expressed because previous work on disjoint intersection types lacks resolution. That is, in previous work a program such as:

[4]With polymorphism we also need to account for instantiation. For example the type `[Int]` is not equal to `[a]`, but it does instantiate `[a]`. So two instances `C [a]` and `C [Int]` (for some type class `C`) overlap.

```
p : [Int] = f [1,2,3]
```

would not type check because `[Int] -> [Int]`, which is the type that `f` needs to have, *is not* a supertype of the type of `f`. However the addition of modus ponens changes this and `[Int] -> [Int]` becomes a supertype of the type of `f`, thus enabling the above use of `f`.

3.4 First-Class Environments

Viewing instances/implicits as merges opens up the possibility of new and interesting features for programming with implicit values and resolution. In particular, unlike type class instances (which are second class), merges are *first-class*. In other words, we can pass merged values as arguments or return them as results. This is not possible with type class instances. An instance declaration is top-level, but we cannot create instances on the fly.[5]

To illustrate the use of first-class environments, consider a program that processes data from a sensor measuring temperatures. The temperatures are collected at fixed time intervals, and the sensor may fail to read temperatures (for example due to temperatures being too high, low or to some unknown error). The sensor data is collected in a text file using a sequence of numbers or strings (for errors) separated by commas. For instance:

```
32,-9,"LOW",10,"ERROR"
```

With Haskell type classes one may be tempted to use the `Read` type class:

```
class Read a where
   read :: String -> a
```

to process the String using the existing instances of the `Read` class:

```
instance Read Int where ...
instance Read String where ...
instance Read a => Read [a] where ...
instance (Read a, Read b) => Read (Either a b) where ...
```

The programmer's hope here would be that we could interpret the type of sensor values as `[Either String Int]`, and then simply employ resolution to compose the above instances:

```
read "32,-9,\"LOW\",10,\"ERROR\"" :: [Either String Int]
```

However this does not work because the input string is not in the correct format to be parsed by the `Read` instances. In Haskell the `Read` instances for types such as `Either a b` and `[a]` use the Haskell syntax[6]. The `read` method provided by `Read [Either String Int]` would require the string:

```
"[Right 32, Right -9, Left \"LOW\", Right 10, Left \"ERROR\"]"
```

Because Haskell does not allow more than one instance of a type class for the same type, it is not possible to use alternative instances for `Either a b` and `[a]`. There are various workarounds for this issue, but none of these is ideal.

A solution using first-class environments. An alternative approach to solve this problem is to use merges to encode first-class environments for instances of `Read`. We present such a solution using the λ_i^{MP} calculus (assuming a few convenience source language features). For the purposes of illustration and readability we assume the following definition of `Read`:

```
type Read a = String -> a
```

[5] Though recent work by Winant and Devriese [2018] aims to rectify this.

[6] The Haskell format requires that strings to be read as `Either a b` values start with a `Left` or `Right` substring. For list values, the string should start and end with opening and closing brackets.

and we also assume built-in types for lists and sum types (i.e. modeling, respectively, Haskell's `List a` and `Either a b`). Given those, we can model four Haskell instances for `Read`:

```
readInt : Read Int = ...
readString : Read String = ...
readEitherSI : Read String -> Read Int -> Read (Either String Int) = ...
readSensor : Read (Either String Int) -> Read [Either String Int] = ...
```

Here, `readInt` and `readString` behave just like the corresponding Haskell instances, while `readEitherSI` and `readSensor` are custom instances that read values in the sensor format. With intersection types and the merge operator we can define both the type of environments for instances that read sensor data, and the environment that collects the above instances:

```
type SensorEnv = Read Int & Read String
               & (Read String -> Read Int -> Read (Either String Int))
               & (Read (Either String Int) -> Read [(Either String Int)])

env : SensorEnv = readInt ,, readString ,, readEitherSI ,, readSensor
```

Finally, the `parseSensor` function parametrizes over a suitable sensor environment:

```
parseSensor (read : SensorEnv) (s: String) : [Either String Int] = read s
```

Calling `parseSensor` with `env` can parse strings in the expected format for sensor data. Moreover, an additional advantage of the `parseSensor` function is that it is *decoupled* from concrete implementations of `Read`. For instance, if later we want to express sensor data using Haskell's own format for `Read` instances, we just have to provide an alternative environment:

```
haskEnv : SensorEnv = readInt ,, readString ,, readEither ,, readList
```

where `readEither` and `readList` behave just like the Haskell instances for `Read (Either a b)` and `Read [a]`. By calling `parseSensor` with `haskEnv` as its first argument, the system will infer an instance of the same type, but taken from the newly provided environment encoding, `haskEnv`.

Without the modus ponens rule in subtyping, the same functionality would require a more verbose program, where both the provided environment and the parser function should be parametrized over the alternative instances for `Either String Int` and `[Either String Int]`, and where application of instances should be explicit:

```
type PSensor =  (Read Int -> Read String -> Read (Either String Int)))
             -> (Read (Either String Int) -> Read [Either String Int])
             -> Read Int & Read String
                & Read (Either String Int) & Read [Either String Int]

env' (rEitherSI : Read Int -> Read String -> Read (Either String Int))
     (rList     : Read (Either String Int) -> Read [Either String Int])
 : Read Int & Read String & Read (Either String Int) & Read [Either String Int]
 = readInt ,, readString ,, (rEitherSI readInt readString)
   ,, (rList (rEitherSI readInt readString))

parseSensor' (rEither : Read Int -> Read String -> Read (Either String Int )))
             (rList   : Read (Either String Int) -> Read [Either String Int])
             (readArg : PSensor) (s : String)
  : [Either String Int]
  = readArg rEither rList s
```

In summary, modus ponens captures the essence of resolution and brings convenience to programmers by automatically composing instances. This is also the reason why resolution is heavily used by Haskell and Scala programmers. In addition, the integration of modus ponens into a system with intersection subtyping adds expressive power that is not available in Haskell or Scala. In contrast to Haskell, implicit environments are first-class. With respect to Scala, viewing resolution as subtyping makes obvious how the two features interact, which we discuss in more detail next.

3.5 Interaction Between Subtyping and Resolution

In languages like Scala, which supports both subtyping and resolution, we expect that the two mechanisms interact. For example, here is a simple Scala program that requires both mechanisms:

```
class A {}
class B extends A {}

implicit val v : B = new B() {}; // adds v to the implicit environment
val u : A = implicitly  // resolves to v
```

This program defines two classes `A` and `B`. The value `v` is marked with an `implicit` keyword, which has the effect of adding `v` to the implicit environment, so that it can be used by resolution later. In the definition of `u`, `implicitly` triggers resolution to search for a value of type `A`. The search succeeds and resolves to `v` which has type `B`, a subtype of `A`. Clearly this only works if resolution accounts for subtyping.

The Scala compiler accepts this program and its implementation of resolution does indeed account for subtyping. However there is very little work formalizing the (non-trivial) interaction between resolution and subtyping. Most foundational calculi for Scala formalize subtyping, but not resolution [Amin et al. 2016, 2012]. Similarly, works that formalize resolution typically do not account for subtyping [Odersky et al. 2017; Oliveira et al. 2012; Schrijvers et al. 2019].

One advantage of making resolution a part of subtyping is that the interaction between the two mechanisms then becomes trivial. For instance, the following program:

```
a : Int                  = 1
b : Int & Char           = a,,'c'
u (env : Int&Char) : Int = env
```

is roughly analogous to the previous Scala program, except that we use a first-class environment (the argument `env` of function `u`) instead of an implicit environment. In this case resolution triggers a search for an integer in the body of `u`, which is successful because `Int&Char <: Int`. Other more complex interactions between subtyping and resolution, such as in:

```
(Int -> Top -> Bool) & Int  <: String -> Bool
```

work as well. In this case the two types in the intersection need to be composed with modus ponens to obtain `Top -> Bool` and `Top -> Bool <: String -> Bool` via subtyping.

3.6 Technical challenges

To formalize our integration of resolution and subtyping, we create the λ_i^{MP} calculus by extending the NeColus calculus with the modus ponens rule. While the extension is seemingly a small change, it has far-reaching consequences with respect to the metatheory. Conceptually, this is due to the fact that modus ponens invalidates much of the reasoning about types in the NeColus metatheory. In NeColus it is possible to consider the syntactic components of types in isolation. This is no longer possible in λ_i^{MP} because one component may interact with another through modus ponens.

$$
\begin{array}{lllll}
\text{Types} & A, B, C & ::= & \mathsf{Nat} \mid \top \mid A \to B \mid A \,\&\, B \\
\text{Terms} & E & ::= & x \mid \top \mid i \mid \lambda x.\, E \mid E_1\, E_2 \mid E_1 \,,, E_2 \mid E : A \\
\text{Typing Contexts} & \Gamma & ::= & \bullet \mid \Gamma, x : A
\end{array}
$$

Fig. 2. λ_i^{MP} Syntax

$\boxed{\Gamma \vdash E \Rightarrow A \rightsquigarrow e}$ Type Synthesis

$$\frac{\vdash \Gamma}{\Gamma \vdash \top \Rightarrow \top \rightsquigarrow \langle\rangle}\ \textsc{T-top} \qquad \frac{\vdash \Gamma}{\Gamma \vdash i \Rightarrow \mathsf{Nat} \rightsquigarrow i}\ \textsc{T-nat} \qquad \frac{x : A \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash x \Rightarrow A \rightsquigarrow x}\ \textsc{T-var}$$

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

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

$\boxed{\Gamma \vdash E \Leftarrow A \rightsquigarrow e}$ Type Checking

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

Fig. 3. Bidirectional Typing

This affects coherence: the NeColus disjointness constraint is insufficient for λ_i^{MP} and needs to be tightened. Also the coherence proof requires a considerably more complex canonicity relation that features a form of context dependency to capture the possible interaction between type components.

Algorithmic subtyping is also affected by this. The algorithm no longer terminates naturally and requires a loop detector. This makes the termination proof a lot harder. Moreover, the size-based induction used to prove transitivity for the algorithmic subtyping of NeColus breaks. Instead, our transitivity proof involves 7 mutually recursive lemmas and relies on the AProVE automatic termination checker to establish their well-foundedness.

The following sections set up λ_i^{MP} and explain these aspects in detail.

4 THE λ_i^{MP} CALCULUS

Syntax. Figure 2 shows the syntax of λ_i^{MP}. It is the same as that of NeColus, but—for the sake of conciseness—without records. Types A, B, C include naturals Nat, the top type $\top$, function types $A \to B$ and intersection types $A \,\&\, B$. Terms E include variables x, natural numbers i, the top value $\top$, function abstractions $\lambda x.\, E$, function applications $E_1\, E_2$, merges $E_1 \,,, E_2$ and (type-)annotated terms $E : A$.

Bidirectional Typing. The type system of λ_i^{MP}, shown in Figure 3, is bidirectional and thus has two modes: the *synthesis* mode ($\Rightarrow$) and the *checking* mode ($\Leftarrow$). The synthesis judgment $\Gamma \vdash E \Rightarrow A \rightsquigarrow e$ means that we can synthesize type A from expression E in context Γ, while the checking judgment $\Gamma \vdash E \Leftarrow A \rightsquigarrow e$ checks whether E has given type A. As usual, type annotations switch from synthesis to checking mode (rule T-ANNO). We reserve further discussion over the parts highlighted in gray for Section 4.2.

The rules are identical to those of NeColus with one notable difference. Rule T-MERGE not only requires the types A_1 and A_2 of the two subterms to be disjoint, $A_1 * A_2$, but also requires them

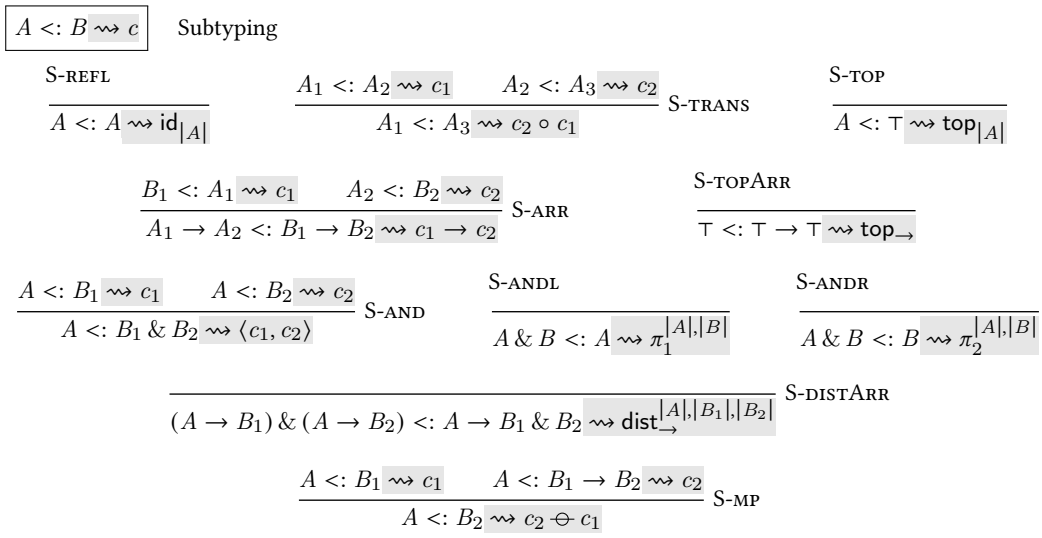

Fig. 4. Declarative Specification of Subtyping

to be *internally* disjoint, $\vdash_d A_1$ and $\vdash_d A_2$, all summarised in the internal disjointness premise $\vdash_d A_1 \,\&\, A_2$ of the rule. The two disjointness conditions are discussed in Section 4.4.

The judgment $\vdash \Gamma$ used in the premise of rules T-TOP, T-NAT and T-VAR expresses that the typing context Γ is well-formed, i.e. it holds no duplicate variables. Its definition can be found in Appendix B.

4.1 Declarative Subtyping

Figure 4 declaratively specifies the subtyping relation of λ_i^{MP}. The judgment $A <: B \rightsquigarrow c$ states that A is a subtype of B and produces coercion c. Intuitively, c captures the way to convert values of type A into values of type B.

BCD-Style Subtyping. All but the last subtyping rule are the same as those of NeColus and originate from the BCD type system [Barendregt et al. 1983]. Rules S-REFL, S-TRANS and S-TOP ensure that subtyping is a partial order with $\top$ as greatest element. Rule S-ARR is the standard subtyping rule for function types, while rules S-ANDL, S-ANDR and S-AND are standard for intersection types. Rules S-DISTARR and S-TOPARR are standard BCD subtyping rules; the former enables *nested composition* [Bi et al. 2018], while the latter provides a more unified treatment of top-like types in the meta-theory of the language: without that rule, we cannot have $A \to B <: \top \to \top$, if A is not top-like itself.

Modus Ponens. Our calculus extends the subtyping relation of NeColus with one new rule, S-MP, which corresponds to the Modus Ponens inference rule of propositional logic. It states that if A is a subtype of both $B_1 \to B_2$ and of B_1, then it is also a subtype of B_2. While this extension of subtyping is conceptually small, it has far-reaching consequences for the meta-theory of the language.

4.2 Elaboration Semantics

Following NeColus we assign a semantics to λ_i^{MP} by means of an elaboration. The target of the elaboration is λ_c^{MP}, which is the simply typed lambda-calculus extended with products and coercions.

$$
\begin{array}{lrcl}
\text{Types} & \tau & ::= & \mathsf{Nat} \mid \langle\rangle \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \\
\text{Terms} & e & ::= & x \mid \langle\rangle \mid i \mid \lambda x : \tau.\, e \mid e_1\, e_2 \mid \langle e_1, e_2 \rangle \mid c\, e \\
\text{Coercions} & c & ::= & \mathsf{id}_\tau \mid c_1 \circ c_2 \mid \mathsf{top}_\tau \mid c_1 \to c_2 \mid \pi_1^{\tau_1,\tau_2} \mid \pi_2^{\tau_1,\tau_2} \mid \langle c_1, c_2 \rangle \mid \mathsf{top}_\to \mid \mathsf{dist}_\to^{\tau_1,\tau_2,\tau_3} \mid \\
 & & & \quad c_1 \multimap c_2 \\
\text{Typing contexts} & \Delta & ::= & \bullet \mid \Delta, x : \tau \\
\text{Values} & v & ::= & \langle\rangle \mid i \mid \lambda x : \tau.\, e \mid \langle v_1, v_2 \rangle \mid (c_1 \to c_2)\, v \mid \mathsf{top}_\to\, v \mid (\mathsf{dist}_\to^{\tau_1,\tau_2,\tau_3})\, v
\end{array}
$$

Fig. 5. λ_c^{MP} Syntax

$\boxed{c \vdash \tau_1 \triangleright \tau_2}$ Coercion Typing

$$\frac{c_1 \vdash \tau_1 \triangleright \tau_2 \qquad c_2 \vdash \tau_1 \triangleright (\tau_2 \to \tau_3)}{c_2 \multimap c_1 \vdash \tau_1 \triangleright \tau_3}\ \text{CT-MP}$$

$\boxed{e_1 \longrightarrow e_2}$ Small-Step Operational Semantics

$$\frac{}{(c_2 \multimap c_1)\, v \longrightarrow (c_2\, v)\, (c_1\, v)}\ \text{STEP-MP}$$

Fig. 6. λ_c^{MP} Coercion Typing and Operational Semantics (new rules)

Pairs are the elaboration target of merges (rule T-MERGE), while coercions explicitly witness the implicit use of subtyping in λ_i^{MP} (rule T-SUB).

The λ_c^{MP} Calculus. Figure 5 shows the syntax of λ_c^{MP}. This syntax is nearly identical to that of NeColus' target language. There are only two differences, both situated in the coercions c. These coercions express the conversion of a term from one type to another. The reason why we use a separate syntactic sort of coercions for this—rather than encoding the conversions as regular function terms—is that it facilitates the coherence proof. Each coercion form corresponds to a particular subtyping rule (Figure 4). The first and main difference to NeColus' target calculus is that, because we have a new modus ponens subtyping rule in λ_i^{MP}, there is also a new corresponding coercion form $c_1 \multimap c_2$ in λ_c^{MP} to witness it. The second and minor difference is that we have annotated several existing coercion forms with type information in order to facilitate type checking.

Figure 6 lists the new inference rules in the type system and small-step operational semantics of λ_c^{MP} to support the new coercion form. The full definitions of these relations can be found in Appendix A. The coercion typing rule CT-MP essentially replicates the subtyping rule S-MP of Figure 4, but now with λ_c^{MP} types. The new small-step rule, STEP-MP, splits up an application of a modus ponens coercion $c_2 \multimap c_1$ to value v into separate coercion applications $c_2\, v$ and $c_1\, v$ where the former should yield a function that is applied to the latter.

How the features of λ_i^{MP} are mapped to λ_c^{MP} is given in the grey parts of Figures 4 and 3. In those figures the meta-function $|\cdot|$ transforms λ_i^{MP} types to λ_c^{MP} types, and extends naturally to typing contexts. Its definition is in Appendix B.

4.3 Target and Elaboration Metatheory

We have formally investigated the metatheory of our calculus, λ_i^{MP}, and of its target language, λ_c^{MP}, and mechanised it in Coq.

Target Language. Firstly, we have proven that the key metatheoretical properties of λ_c^{MP} are not invalidated by the new coercion form. Indeed, λ_c^{MP} is type safe:

THEOREM 4.1 (PRESERVATION). *If $\bullet \vdash e : \tau$ and $e \longrightarrow e'$, then $\bullet \vdash e' : \tau$.*

THEOREM 4.2 (PROGRESS). *If $\bullet \vdash e : \tau$, then either e is a value or there is an e' such that $e \longrightarrow e'$.*

Moreover, every well-typed λ_c^{MP} term is normalising.

$\boxed{A * B}$ Disjointness

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

$$\frac{A_2 * B}{(A_1 \to A_2) * B}\ \text{D-mpL} \qquad \frac{A * B_2}{A * (B_1 \to B_2)}\ \text{D-mpR}$$

$\boxed{\vdash_d A}$ Internal Disjointness

$$\frac{}{\vdash_d \mathsf{Nat}}\ \text{ID-nat} \qquad \frac{}{\vdash_d \top}\ \text{ID-top} \qquad \frac{\vdash_d B}{\vdash_d A \to B}\ \text{ID-arr} \qquad \frac{A * B \qquad \vdash_d A \qquad \vdash_d B}{\vdash_d A \,\&\, B}\ \text{ID-and}$$

Fig. 7. Disjointness conditions

Theorem 4.3 (Normalisation). *If* $\bullet \vdash e : \tau$ *then there exists a value* v *such that* $e \longrightarrow^* v$.

Here $\longrightarrow^*$, is the reflexive transitive closure of $\longrightarrow$.

Elaboration. Also, the elaboration of λ_i^{MP} to λ_c^{MP} preserves well-typing.

Theorem 4.4 (Coercions Preserve Types). *If* $A <: B \rightsquigarrow c$, *then* $c \vdash |A| \triangleright |B|$.

Theorem 4.5 (Elaboration Preserves Types).

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

The meta-function $|\Gamma|$ translates a source typing context Γ to a target one. The full definition can be found in Appendix B.

4.4 Disjointness Conditions

Figure 7 shows the rules for disjointness (top) and internal disjointness (bottom).

Disjointness. The purpose of the disjointness condition is to avoid ambiguous expressions like $1,,2$. Judgment $A * B$ conservatively captures the notion that the common supertypes of A and B are all top-like. The first 4 rules of Figure 7, for the top type and intersection types, are the same as those of NeColus. We turn our focus to the new rules D-mpL and D-mpR, added to accommodate the new modus ponens subtyping rule. While they can be combined to recover NeColus' D-arr rule[7],

$$\frac{B_1 * B_2}{(A_1 \to B_1) * (A_2 \to B_2)}\ \text{D-arr}$$

they cannot be used to recover the NeColus axioms $\mathsf{Nat} * (A \to B)$ and $(A \to B) * \mathsf{Nat}$. On the contrary, λ_i^{MP} does not consider the types Nat and Bool $\to$ Nat to be disjoint. For the sake of conciseness and compositionality, the disjointness definition is conservative. Indeed, merging any expressions of those two types does not lead to ambiguity. Yet, if we merge two expressions whose types contain the above two types, ambiguity can in fact arise. Consider the example $((\mathsf{true},, (\lambda x.1) : \mathsf{Bool} \to \mathsf{Nat}),, 2) : \mathsf{Nat}$. The elaborated form of the merge expression on the left of the outer type annotation is the tuple $\langle\langle \mathsf{true}, \lambda x : \mathsf{Bool}.\, 1\rangle, 2\rangle$. The outer type annotation will trigger the subtyping mechanism, first via rule T-anno and then through rule T-sub, in order to produce a value of type Nat. This example is ambiguous, because it can produce—through subtyping—two different values of this type: 2 by projecting on the right component of the outer tuple, and 1 by projecting on the left component of the outer tuple and using modus ponens to

[7]This rule states that two arrow types are disjoint if their codomains are disjoint.

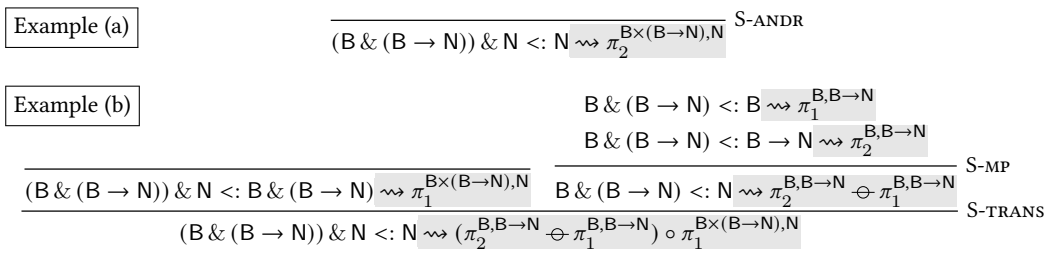

Fig. 8. Example subtyping derivations. Types Bool and Nat are abbreviated here as B and N, respectively.

apply $(\lambda x : \mathsf{Bool}.\, 1)$ to true. The respective subtyping derivations and the coercions they generate are shown in Figure 8. For similar reasons, expressions $(\lambda x.\, 1) : \mathsf{Bool} \to \mathsf{Nat}\,,,\, (2,,\, \mathsf{true})$ and $(2,,\, (\lambda x.\, 1) : \mathsf{Bool} \to \mathsf{Nat}),,\, \mathsf{true}$ are also ambiguous. In all three cases, the component types of a merge will not be disjoint.

The two new rules D-MPL and D-MPR bring the notion of disjointness between two types closer to the notion of no-overlaps between two (or more) instances. Types B and $A \to B$ are not disjoint, similarly to how two Haskell instances `C Int` and `D Int => C Int` overlap.

Harmless overlaps and (non-)ambiguity. Note that not all forms of overlap introduce semantic ambiguity. In particular, we identify two such *harmless* cases. Firstly, in the presence of an overlap, it can still be the case that there is only one way to resolve a value of a given (overlapping) type. For example, in the term $2,,\, (\lambda x.\, 1) : \mathsf{Bool} \to \mathsf{Nat}$ of type $\mathsf{Nat} \,\&\, (\mathsf{Bool} \to \mathsf{Nat})$, the types Nat and $\mathsf{Bool} \to \mathsf{Nat}$ overlap, but there is only one applicable way to resolve a value of type Nat, because there is no Bool value to feed to the function $(\lambda x.\, 1) : \mathsf{Bool} \to \mathsf{Nat}$. Secondly, expressions of an overlapping type are harmless when all possible resolution paths for a wanted type result in the same inferred value. For example, the term $(1,,\, 1)$, while containing an overlap, is unambiguous.

Because of the disjointness condition imposed over merges, the source calculus conservatively rejects expressions that fall under either of the two harmless forms of overlap, when they are explicitly written by the programmer. However, they are still possible implicitly, through a type annotation, which invokes subtyping. For example, we can encode the rejected term $(1,,\, 1)$ as $1 : \mathsf{Nat} \,\&\, \mathsf{Nat}$, which is accepted and semantically same. Its full typing derivation is in Appendix E.

Internal Disjointness. The internal disjointness predicate, $\vdash_d A$, requires that the two components of any intersection type that appear in A are disjoint. Intuitively, $\vdash_d A$ ensures that if $A <: B$, then B can only contain harmless overlaps.

The internal disointness condition is imposed on the subterms of a merge in rule T-MERGE. Again this is a conservative measure to avoid ambiguity, this time of a more convoluted nature that involves nested merges interleaved with subtyping. Here is a minimal ambiguous example to illustrate the problem:

$$((\mathsf{true},,\, \mathsf{b2n}) : \mathsf{Nat} \,\&\, (\mathsf{Bool} \to \mathsf{Nat}),,\, \mathsf{false}) : \mathsf{Nat}$$

This example features b2n, which we assume to be a built-in function of type $\mathsf{Bool} \to \mathsf{Nat}$ that maps true to 1 and false to 0. The inner merge has type $\mathsf{Bool} \,\&\, (\mathsf{Bool} \to \mathsf{Nat})$, but is converted to type $\mathsf{Nat} \,\&\, (\mathsf{Bool} \to \mathsf{Nat})$ by means of the type annotation, which triggers the subtyping mechanism. The latter produces a coercion that is then applied on (the elaborated form of) the subterm $\mathsf{true},,\, \mathsf{b2n}$. The corresponding value is $1,,\, \mathsf{b2n}$, which could not be written explicitly because Nat and $\mathsf{Bool} \to \mathsf{Nat}$ are not considered disjoint. Yet, because this value is not explicitly written by the programmer, but implicitly produced through subtyping, we know that the overlap

is harmless. Indeed, while the expression $1\,,\,$ b2n potentially produces two different values of type Nat, one of those requires a Bool, which is not available at this point.

However, the outer merge with false provides the missing Bool value, making the outer Nat type annotation ambiguously yield both 0 and 1 as possible results. The disjointness check rejects none of the two merges in the example. Here, the internal disjointness constraint comes in. It catches the ambiguity problem at the outer merge where it identifies $1\,,\,$ b2n as unfit to merge with.

Internal disjointness inherits the conservative behavior of the (binary) disjointness condition, as witnessed in rule ID-ARR. This rule regards the case of a function type $A \to B$ that potentially is a supertype of some internally disjoint type, say A' (formally, $A' <: A \to B$). If A' is not a subtype of A (thus, $A' <: A$ does not hold), then rule S-MP does not apply. In fact, it becomes impossible to implicitly produce B from A' and, therefore, unnecessary to demand internal disjointness of B.

The indifference of rule ID-ARR around domains of arrow types is justified by the intuition that if a supertype of an internally disjoint type contains overlaps, then those are harmless. A function of type $A \to B$ is unambiguous if it outputs unambiguous values of type B, given unambiguous input values of type A. This implies the requirement that A is also internally disjoint. However, at this point, only values produced implicitly from A' are considered, and these can only contain harmless ambiguity. Indeed, explicit values can be added only through the merge operator, thus falling under rule ID-AND. Thus, unambiguity is ensured for A.

5 ALGORITHMIC SUBTYPING

In this section, we present an algorithm that decides the subtyping relation of Figure 4.

In the declarative specification of the subtyping relation (Figure 4), rules S-TRANS and S-MP hinder the decidability of subtyping: they both require guessing an appropriate intermediate type (A_2 in the former, and B_1 in the latter). We address this by employing a technique originating from proof search known as *focusing* [Andreoli 1992]. The general strategy for checking a constraint of the form $A <: B$ is to structurally analyze (focus on) B, and when it is stripped down to a base type do the same for A. This task is performed by two mutually-recursive judgments, the first focusing on the *right* type B, and the second focusing on the *left* type A. This technique has already been employed by Bi et al. [2018]—and, before that, by Pierce [1989]—though with a single all-in-one judgment. Our algorithm separates the two phases for clarity (and ease of reasoning) and also deals with the additional complexity introduced by rule S-MP. Unfortunately, the support for modus ponens greatly complicates the algorithm's proofs of transitivity and termination. For the former, we had to come up with an approach external to Coq, involving the AProVE termination checker [Giesl et al. 2017], to establish well-foundedness. In the remainder of the section we describe the subtyping algorithm in detail (Section 5.1), as well as its most important meta-theoretical properties (Section 5.2). We follow the earlier convention to highlight the parts of the rules that concern coercion generation.

5.1 The Subtyping Algorithm

In Figure 9, we present the algorithmic subtyping judgment $A <: B \rightsquigarrow c$; the algorithmic counterpart of the declarative judgment $A <: B \rightsquigarrow c$ of Section 4.1. The main judgment is defined in terms of two auxiliary, mutually-recursive judgments $\mathcal{L} \vdash_{\mathrm{R}} A <: B \rightsquigarrow c$ and $\mathcal{L}; \mathcal{M}; A_0; X \vdash_{\mathrm{L}} A <: B \rightsquigarrow X'$, each focusing on the right or the left component of a type inequality, respectively. We explain the former in Section 5.1.1 and the latter in Section 5.1.2.

Preliminaries. Meta-variables $\mathcal{L}$ and $\mathcal{M}$ stand for queues of types. The algorithm uses them to capture the left-hand sides of arrow types. Operation $\mathcal{L} \twoheadrightarrow B$ converts a queue $\mathcal{L}$ back into a type, given the right-hand side type B. It is inductively defined as follows:

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

$\boxed{A <: B \rightsquigarrow c}$ Algorithmic Subtyping

$$\frac{[] \vdash_{\mathrm{R}} A <: B \rightsquigarrow c}{A <: B \rightsquigarrow c}\ \textsc{A-main}$$

$\boxed{\mathcal{L} \vdash_{\mathrm{R}} A <: B \rightsquigarrow c}$ Right Focusing

$$\frac{\mathcal{L} \vdash_{\mathrm{R}} A <: B_1 \rightsquigarrow c_1 \qquad \mathcal{L} \vdash_{\mathrm{R}} A <: B_2 \rightsquigarrow c_2}{\mathcal{L} \vdash_{\mathrm{R}} A <: B_1 \,\&\, B_2 \rightsquigarrow [\![\mathcal{L}]\!]_{\&}^{B_1,B_2} \circ \langle c_1, c_2 \rangle}\ \textsc{AR-and} \qquad \frac{\mathcal{L}, B_1 \vdash_{\mathrm{R}} A <: B_2 \rightsquigarrow c}{\mathcal{L} \vdash_{\mathrm{R}} A <: B_1 \to B_2 \rightsquigarrow c}\ \textsc{AR-arr}$$

$$\frac{}{\mathcal{L} \vdash_{\mathrm{R}} A <: \top \rightsquigarrow [\![\mathcal{L}]\!]_{\top} \circ \mathsf{top}_{|A|}}\ \textsc{AR-top} \qquad \frac{\mathcal{L}; []; A; \ (\lambda c.c) \vdash_{\mathrm{L}} A <: \mathsf{Nat} \rightsquigarrow \mathcal{X}}{\mathcal{L} \vdash_{\mathrm{R}} A <: \mathsf{Nat} \rightsquigarrow \mathcal{X}[\mathsf{id}_{|\mathsf{Nat}|}]}\ \textsc{AR-nat}$$

$\boxed{\mathcal{L}; \mathcal{M}; A_0; \ \mathcal{X} \vdash_{\mathrm{L}} A <: B \rightsquigarrow \mathcal{X}'}$ Left Focusing

$$\frac{}{[]; \mathcal{M}; A_0; \ \mathcal{X} \vdash_{\mathrm{L}} \mathsf{Nat} <: \mathsf{Nat} \rightsquigarrow \mathcal{X}}\ \textsc{AL-nat} \qquad \frac{\mathcal{L}; \mathcal{M}; A_0; \ (\lambda c.\mathcal{X}[c \circ \pi_1^{|A_1|,|A_2|}]) \vdash_{\mathrm{L}} A_1 <: \mathsf{Nat} \rightsquigarrow \mathcal{X}'}{\mathcal{L}; \mathcal{M}; A_0; \ \mathcal{X} \vdash_{\mathrm{L}} A_1 \,\&\, A_2 <: \mathsf{Nat} \rightsquigarrow \mathcal{X}'}\ \textsc{AL-and1}$$

$$\frac{\mathcal{L}; \mathcal{M}; A_0; \ (\lambda c.\mathcal{X}[c \circ \pi_2^{|A_1|,|A_2|}]) \vdash_{\mathrm{L}} A_2 <: \mathsf{Nat} \rightsquigarrow \mathcal{X}'}{\mathcal{L}; \mathcal{M}; A_0; \ \mathcal{X} \vdash_{\mathrm{L}} A_1 \,\&\, A_2 <: \mathsf{Nat} \rightsquigarrow \mathcal{X}'}\ \textsc{AL-and2}$$

$$\frac{[] \vdash_{\mathrm{R}} B_1 <: A_1 \rightsquigarrow c_1 \qquad \mathcal{L}; \mathcal{M}, B_1; A_0; \ (\lambda c.\mathcal{X}[c_1 \to c]) \vdash_{\mathrm{L}} A_2 <: \mathsf{Nat} \rightsquigarrow \mathcal{X}'}{B_1, \mathcal{L}; \mathcal{M}; A_0; \ \mathcal{X} \vdash_{\mathrm{L}} A_1 \to A_2 <: \mathsf{Nat} \rightsquigarrow \mathcal{X}'}\ \textsc{AL-arr}$$

$$\frac{[] \vdash_{\mathrm{R}} A_0 <: (\mathcal{M} \twoheadrightarrow A_1) \rightsquigarrow c_2 \qquad \mathcal{L}; \mathcal{M}; A_0; \ \mathcal{X}' \vdash_{\mathrm{L}} A_2 <: \mathsf{Nat} \rightsquigarrow \mathcal{X}'' \qquad \mathcal{X}' = (\lambda c.[\![\mathcal{M}]\!]_{\&}^{(c \circ (\pi_1^{|(A_1 \to A_2)|,|A_1|} \ominus \pi_2^{|(A_1 \to A_2)|,|A_1|})),(A_1 \to A_2),A_1} \circ \langle \mathcal{X}[\mathsf{id}_{|A_1 \to A_2|}], c_2 \rangle)}{\mathcal{L}; \mathcal{M}; A_0; \ \mathcal{X} \vdash_{\mathrm{L}} A_1 \to A_2 <: \mathsf{Nat} \rightsquigarrow \mathcal{X}''}\ \textsc{AL-mp}$$

Fig. 9. Algorithmic Subtyping

Meta-variable $\mathcal{X}$ denotes *algorithmic contexts*. An algorithmic context is merely the Cayley representation of a coercion: a function from coercions to coercions. Given a coercion c, we can convert an algorithmic context $\mathcal{X}$ into a coercion by function application, denoted as $\mathcal{X}[c]$. The role of algorithmic contexts is explained in Section 5.1.2 where we discuss the left-focusing rules.[8]

5.1.1 Right Focusing. As mentioned earlier, the first judgment ($\mathcal{L} \vdash_{\mathrm{R}} A <: B \rightsquigarrow c$) structurally analyzes B, until it is stripped down to a base type (Nat or $\top$). The accumulating parameter $\mathcal{L}$ captures the domains of arrows encountered in B, such that the following holds:

Lemma 5.1 (Soundness of Right Focusing). *If* $\mathcal{L} \vdash_R A <: B \rightsquigarrow c$ *then* $A <: (\mathcal{L} \twoheadrightarrow B) \rightsquigarrow c$.

The first judgment is mostly borrowed from NeColus; we explain it only briefly here.

Subtyping. Ignoring coercion generation for now, the judgment works as follows: Rule AR-and decomposes an intersection type, similar to its declarative counterpart (rule S-and). Rule AR-arr deals with arrow types: the left sides of arrow types are inserted in the rear of type queue $\mathcal{L}$

[8] In both our mechanization and implementation of the algorithm we use a defunctionalized [Reynolds 1972] representation of algorithmic contexts, which we then interpret as actual functions between coercions.

$$
\begin{array}{lcccl}
\textsc{ar-nat} \Longrightarrow & \mathcal{L}_0; \mathcal{M}_0; A_0; \mathcal{X}_0 \vdash_L A_0 <: \mathsf{Nat} & \rightsquigarrow & \mathcal{X}_n & (\mathcal{M}_0 = [], \mathcal{X}_0 = \square) \\
& \downarrow & & \uparrow & \\
& \mathcal{L}_1; \mathcal{M}_1; A_0; \mathcal{X}_1 \vdash_L A_1 <: \mathsf{Nat} & \rightsquigarrow & \mathcal{X}_n & \\
& \downarrow & & \uparrow & \\
& \cdots & & \cdots & \\
& \downarrow & & \uparrow & \\
\textsc{al-nat} \Longrightarrow & \mathcal{L}_n; \mathcal{M}_n; A_0; \mathcal{X}_n \vdash_L A_n <: \mathsf{Nat} & \rightsquigarrow & \mathcal{X}_n & (A_n = \mathsf{Nat}, \mathcal{L}_n = [])
\end{array}
$$

Fig. 10. Visual Representation of the Execution of $(\mathcal{L}; \mathcal{M}; A_0; \mathcal{X} \vdash_{\mathsf{L}} A <: B \rightsquigarrow \mathcal{X}')$

and removed again later, left-to-right, as arrows are encountered in A (see rule AL-ARR below). There are two base cases: either $B = \top$ or $B = \mathsf{Nat}$. Rule AR-TOP handles the first case, where subtyping directly produces a coercion with the use of the meta-function $[\![\mathcal{L}]\!]_\top$, discussed in the next paragraph. If $B = \mathsf{Nat}$, then we perform structural analysis on A, using judgment $\mathcal{L}; \mathcal{M}; A_0; \mathcal{X} \vdash_{\mathsf{L}} A <: B \rightsquigarrow \mathcal{X}'$, which we elaborate on in Section 5.1.2.

Coercion Generation. The elaboration side is a bit more involved. Coercion generation can be understood via Lemma 5.1 and works as follows. Rule AR-AND deals with intersection types. By induction we have that

$$
\left.\begin{array}{l} A <: (\mathcal{L} \twoheadrightarrow B_1) \rightsquigarrow c_1 \\ A <: (\mathcal{L} \twoheadrightarrow B_2) \rightsquigarrow c_2 \end{array}\right\} \Longrightarrow A <: (\mathcal{L} \twoheadrightarrow B_1) \,\&\, (\mathcal{L} \twoheadrightarrow B_2) \rightsquigarrow \langle c_1, c_2 \rangle
$$

We finish the transition to $\mathcal{L} \twoheadrightarrow (B_1 \,\&\, B_2)$ using meta-function $[\![\mathcal{L}]\!]_{\&}^{B_1,B_2}$, whose meaning is given by the following lemma:[9]

Lemma 5.2. $\forall \mathcal{L}, B_1, B_2, (\mathcal{L} \twoheadrightarrow B_1) \,\&\, (\mathcal{L} \twoheadrightarrow B_2) <: \mathcal{L} \twoheadrightarrow (B_1 \,\&\, B_2) \rightsquigarrow [\![\mathcal{L}]\!]_{\&}^{B_1,B_2}$.

Rule AR-TOP deals with the same problem that rule AR-AND does: according to Lemma 5.1, the coercion needs to witness $A <: (\mathcal{L} \twoheadrightarrow \top)$, and not simply $A <: \top$. As $(\mathcal{L} \twoheadrightarrow \top)$ is *top-like* [Alpuim et al. 2017] we can always cast $\top$ to it, which is what $[\![\mathcal{L}]\!]_\top$ witnesses:[9]

Lemma 5.3. $\forall \mathcal{L}, \top <: (\mathcal{L} \twoheadrightarrow \top) \rightsquigarrow [\![\mathcal{L}]\!]_\top$.

Rule AR-ARR is straightforward. Finally, rule AR-NAT handles the base case where B is Nat. In this case we resort to the auxiliary judgment $\mathcal{L}; \mathcal{M}; A_0; \mathcal{X} \vdash_{\mathsf{L}} A <: B \rightsquigarrow \mathcal{X}'$, which we discuss next.

5.1.2 Left Focusing. The intuition behind $\mathcal{L}; \mathcal{M}; A_0; \mathcal{X} \vdash_{\mathsf{L}} A <: B \rightsquigarrow \mathcal{X}'$ is that the (input) algorithmic context $\mathcal{X}$ *witnesses* the transition from A_0 to $(\mathcal{M} \twoheadrightarrow A)$, the component of A_0 that we are currently focusing on. The (output) algorithmic context $\mathcal{X}'$ is meant to capture the complete transition from A_0 to $(\mathcal{M} \twoheadrightarrow (\mathcal{L} \twoheadrightarrow B))$. Essentially, starting with rule AR-NAT as the entry point, the judgment continuously decomposes A (via rules AL-AND1, AL-AND2, AL-ARR, and AL-MP) until it equals B (rule AL-NAT); then the accumulated context $\mathcal{X}$ witnesses the complete transition from A_0 to $(\mathcal{M} \twoheadrightarrow (\mathcal{L} \twoheadrightarrow B))$ and the job is done. Figure 10 visualizes this process.

Subtyping. Ignoring coercion generation, the judgment works as follows: Rule AL-NAT constitutes the base case where subtyping holds trivially. Rules AL-AND1 and AL-AND2 non-deterministically focus on the left or the right component of an intersection, similarly to their declarative counterparts (rules S-ANDL and S-ANDR, respectively). Rule AL-ARR is the algorithmic version of rule S-ARR: we dequeue B_1 from $\mathcal{L}$ in a first-in-first-out manner, in order to match the order in which arrows are encountered in A. After we prove subtyping for the domains of the functions, we record that we

[9] Its definition is straightforward and thus omitted; it can be found in Appendix D.

are going under a binder of type B_1 in the queue $\mathcal{M}$ and proceed recursively with the function images. Queue $\mathcal{M}$ thus captures all the arrow domains in B that have been removed in lockstep with arrow domains in A using rule S-ARR.[10] This context becomes useful when we need to use Modus Ponens under a binder, e.g. to prove

$$\mathsf{Nat} \to (\mathsf{Bool} \,\&\, (\mathsf{Bool} \to \mathsf{String})) <: \mathsf{Nat} \to \mathsf{String}$$

(or even $\top \to (\mathsf{Bool} \,\&\, (\mathsf{Bool} \to \mathsf{String})) <: \mathsf{Nat} \to \mathsf{String}$). Lastly, rule AL-MP is the algorithmic version of the S-MP rule. The first premise reveals the synergistic relationship between this rule and rule AL-ARR: when we need to synthesize an argument for Modus Ponens, we restart from the original type A_0—thus capturing the complete implicit context of $A_1 \to A_2$—and use the contextualized version of A_1, $(\mathcal{M} \twoheadrightarrow A_1)$. For the above example, this amounts to proving

$$\mathsf{Nat} \to (\mathsf{Bool} \,\&\, (\mathsf{Bool} \to \mathsf{String})) <: \mathsf{Nat} \to \mathsf{Bool}$$

The second premise is straightforward.

Note that rules AL-ARR and AL-MP overlap and, to ensure completeness with respect to the declarative specification of subtyping, backtracking is required. However, it is never the case that both rules can apply for resolving a subtyping judgment. That is, backtracking will always fail in at least one of the two choices.

Coercion Generation. The elaboration side, again, requires a bit more explanation. The complication arises in rule AL-MP, where the first premise restarts from the original type A_0. This is essential for recovering the unfocused context needed for Modus Ponens, but it goes against the inductive, top-down approach to coercion generation that we follow in the right-focusing rules. Coercions produced by the recursive calls can no longer be directly combined. This is why the left focusing rules use algorithmic contexts instead of coercions. The essence of the algorithmic contexts is captured in the following invariant, which is preserved by each step of the left focusing algorithm:

INVARIANT 1. $\forall c\ A', A_i <: A' \rightsquigarrow c \implies A_0 <: (\mathcal{M}_i \twoheadrightarrow A') \rightsquigarrow \mathcal{X}_i[c]$

Metavariables A_i, A_0, $\mathcal{M}_i$, and $\mathcal{X}_i$ refer to the corresponding parameters at the i-th step in Figure 10. Informally, $\mathcal{X}$ captures the transition from A_0 to A_i: $\mathcal{X}_i[c]$ witnesses the complete transition from A_0 to *any* type A', if we have a way to perform the remaining transition from A_i to A'. Hence, every rule of the judgment strips a layer off A and extends the "input" algorithmic context accordingly, so that Invariant 1 is preserved.

Rule AL-MP is by far the most interesting rule. It commits to using the S-MP rule: the left premise produces a coercion c_2 that provides the argument (of type $\mathcal{M} \twoheadrightarrow A_1$), while the appropriate instantiation of the current context $\mathcal{X}$ gives us the function (of type $\mathcal{M} \twoheadrightarrow (A_1 \to A_2)$):

$$\begin{array}{l} A_0 <: \mathcal{M} \twoheadrightarrow A_1 \rightsquigarrow c_2 \\ A_0 <: \mathcal{M} \twoheadrightarrow (A_1 \to A_2) \rightsquigarrow \mathcal{X}[\mathsf{id}_{\lfloor A_1 \to A_2 \rfloor}] \end{array}$$

Yet, the above coercions cannot be directly combined, due to the type queue $\mathcal{M}$. We remedy this using meta-function $[\![\mathcal{M}]\!]_{\&}^{c,B_1,B_2}$, whose meaning is given by the following lemma:[11]

LEMMA 5.4. $\forall \mathcal{L}\ B_1\ B_2\ B\ c, B_1 \,\&\, B_2 <: B \rightsquigarrow c \implies (\mathcal{L} \twoheadrightarrow B_1) \,\&\, (\mathcal{L} \twoheadrightarrow B_2) <: \mathcal{L} \twoheadrightarrow B \rightsquigarrow [\![\mathcal{L}]\!]_{\&}^{c,B_1,B_2}$.

The general idea is that if $(A_0 <: \mathcal{M} \twoheadrightarrow A_1 \rightsquigarrow c_2)$ and $(A_0 <: \mathcal{M} \twoheadrightarrow (A_1 \to A_2) \rightsquigarrow c_1)$ then

$$A_0 <: \mathcal{M} \twoheadrightarrow A_2 \rightsquigarrow (\lambda c.[\![\mathcal{M}]\!]_{\&}^{(c \circ (\pi_1^{\lfloor (A\to B) \rfloor, \lfloor A \rfloor} \mathbin{\ominus} \pi_2^{\lfloor (A\to B) \rfloor, \lfloor A \rfloor})), (A \to B), A} \circ \langle c_1, c_2 \rangle)\ [\mathsf{id}_{\lfloor A_2 \rfloor}]$$

[10] In Figure 10, it is easy to see that $\mathcal{M}_i +\!\!+ \mathcal{L}_i = \mathcal{L}_0$, where $+\!\!+$ denotes list concatenation. At the end, we have $\mathcal{M}_n = \mathcal{L}_0$.
[11] Its definition is straightforward and thus omitted; it can be found in Appendix D.

Returning to rule AL-MP, the second premise uses the above algorithmic context (named $\mathcal{X}'$ for convenience) to further focus on A_2.

5.2 Algorithm Metatheory

Subsumption of NeColus' Algorithmic Subtyping. First, we have proven that the subtyping algorithm of Figure 9 subsumes the subtyping algorithm of NeColus [Bi et al. 2018, Fig. 12]:

Theorem 5.5 (Subsumption of NeColus' Algorithm). *If* $[] \vdash A \ll B \rightsquigarrow c$ *then* $A <: B \rightsquigarrow c$.

where $\mathcal{L} \vdash A \ll B \rightsquigarrow c$ denotes the subtyping algorithm of NeColus.

Transivity and Modus Ponens. A major difference between algorithmic and declarative subtyping is that the latter has an explicit transitivity rule while the former does not. Hence, transitivity of algorithmic subtyping requires a separate proof. Unfortunately, this proof is considerably more involved than in NeColus. Instead of one inductive core lemma, it consists of 7 mutually recursive lemmas. The well-foundedness of the induction used in the proofs of those 7 lemmas is no longer a straightforward size-based induction that can be easily understood by Coq. To overcome this problem we have used a new approach, external to Coq, to establish well-foundedness. We have encoded the proof—which is essentially an algorithm that transforms derivation trees—as a term rewriting system. Then we have used the AProVE tool [Giesl et al. 2017] to verify its termination automatically. AProVE reports success after 40 steps of transformation, decomposition and basic termination arguments.

Theorem 5.6 (Transitivity of Algorithmic Subtyping). *If* $A_1 <: A_2 \rightsquigarrow c_1$ *and* $A_2 <: A_3 \rightsquigarrow c_2$, *then* $A_1 <: A_3 \rightsquigarrow c_3$ *for some* c_3.

The modus ponens property for algorithmic subtyping is closely tied to transitivity proof and is another consequence of the 7 mutually recursive lemmas.

Corollary 5.7 (Modus Ponens of Algorithmic Subtyping). *If* $A <: B_1 \to B_2 \rightsquigarrow c_1$ *and* $A <: B_1 \rightsquigarrow c_2$, *then* $A <: B_2 \rightsquigarrow c_3$ *for some* c_3.

Soundness and Completeness. We have proven that the subtyping algorithm (Figure 9) is sound and complete with respect to its declarative specification (Figure 4). The soundness of the overall algorithm is a direct corollary of the soundness of right focusing (Lemma 5.1):

Theorem 5.8 (Soundness). *If* $A <: B \rightsquigarrow c$ *then* $A <: B \rightsquigarrow c$.

The completeness proof largely follows the same strategy as for NeColus. The main challenge was the new transitivity proof discussed above.

Theorem 5.9 (Completeness). *If* $A <: B \rightsquigarrow c$ *then there exists* c' *such that* $A <: B \rightsquigarrow c'$.

Termination. The final key property of the algorithm is termination. Actually, the algorithm presented in Figure 9 is not terminating. Consider for example the trace of $\mathsf{Nat} \to \mathsf{Nat} <: \mathsf{Nat}$:

$$([] \vdash_{\mathrm{R}} \mathsf{Nat} \to \mathsf{Nat} <: \mathsf{Nat}) \xrightarrow{\text{AR-NAT}} ([]; []; \mathsf{Nat} \to \mathsf{Nat} \vdash_{\mathrm{L}} \mathsf{Nat} \to \mathsf{Nat} <: \mathsf{Nat}) \xrightarrow{\text{AL-MP}}$$
$$([] \vdash_{\mathrm{R}} \mathsf{Nat} \to \mathsf{Nat} <: \mathsf{Nat}) \xrightarrow{\text{AR-NAT}} ([]; []; \mathsf{Nat} \to \mathsf{Nat} \vdash_{\mathrm{L}} \mathsf{Nat} \to \mathsf{Nat} <: \mathsf{Nat}) \xrightarrow{\text{AL-MP}} \ldots$$

Such a subtyping judgment can result from an erroneous type annotation by the user. Nonetheless, the language does not statically detect annotations that may cause non-termination in the subtyping algorithm. Another example, perhaps more plausible as a programming error, is an annotation that leads to the judgment $(\mathsf{Nat} \to \mathsf{Bool}) \,\&\, (\mathsf{Bool} \to \mathsf{Nat}) <: \mathsf{Nat}$. One simple ingredient needs to be

added to the algorithm to make it terminating: a loop detection on the $[] \vdash_R A <: B$ call in rule AL-MP. This loop detection should reject any calls that repeat one of their ancestor calls.

We can show that (1) any infinite derivation involves an infinite number of these nested calls and that (2) detecting repeated calls is effective at stopping the non-termination.

A priori there are multiple possible ways in which the algorithm can diverge. It might involve an infinite sequence of recursive calls within one of the two judgments. However, following the focusing approach, the two judgments structurally decompose their focal argument. Hence, on their own they are clearly terminating. Any divergence must thus involve an infinite number of mutually recursive calls between the two judgments. There are two "gatekeepers" of the mutual recursion, calling back from the left-focusing into the right-focusing judgment. These two are the problematic one in rule AL-MP and the one in rule AL-ARR. Yet, we can show that an infinite number of recursion steps through rule AL-ARR is impossible and thus we can eliminate it from consideration. Indeed, the *left-arrow depth* $||\cdot||$ defined below decreases strictly in every AL-ARR loop step, and non-strictly in every other step. Moreover, a value of 0 inhibits rule AL-ARR. Thus, after a finite number of AL-ARR steps, the rule can no longer be used.

$$\begin{array}{rclcrcl} ||\mathsf{Nat}|| & = & 0 & & ||A \to B|| & = & \max(||A|| + 1, ||B||) \\ ||\top|| & = & 0 & & ||A \,\&\, B|| & = & \max(||A||, ||B||) \end{array}$$
$$||[] \vdash_R A <: B|| = \max(||A||, ||B||)$$

That leaves only the problematic mutually recursive call in rule AL-MP, which we have already demonstrated to be a source of non-termination. Fortunately, we can show that the number of distinct nested calls is finite. Hence, on any infinite derivation path, a repeated call has to occur after a finite number of steps. This makes the loop detection effective at curtailing divergence.

A more extensive account can be found in Appendix G.

Theorem 5.10 (Termination). *The algorithm extended with loop detection terminates.*

Decidability. In summary, if we take the soundness, completeness and termination of the subtyping together, we can conclude that the declarative subtyping is decidable.

Corollary 5.11 (Decidable Subtyping). *In λ_i^{MP}, $A <: B$ is decidable for any types A and B.*

6 COHERENCE

A desired property for calculi with a merge operator is *coherence*. This ensures that no semantic ambiguity arises in programs, i.e. programs and their subparts always have a single, unambiguous meaning. Recall that the semantics of λ_i^{MP} is given indirectly, through an elaboration to λ_c^{MP} and its operational semantics. Hence, coherence for λ_i^{MP} means that all possible elaborations of a well-typed source expression for a given type are contextually equivalent in the target language.

Two target expressions, that may be syntactically different, are *contextually equivalent* iff, when used in the context of any program, there is no difference in that program's behavior. In our purely functional and strongly normalising setting, we can reduce a program's behavior to the value it returns. Moreover, as usual, we can restrict ourselves without loss of generality to programs of type Nat to facilitate the comparison of the results.

Program contexts express the above notion of using an expression in the context of a program and are defined as follows:

$$\text{Program contexts } C ::= [\cdot] \mid \lambda x.C \mid C\,E \mid E\,C \mid C\,,\,E \mid E\,,\,C \mid C : A$$

Intuitively, a program context is an expression with a missing subexpression. This hole can be filled in with an appropriate term to form an actual expression. Which terms are eligible to be used in a program context is captured by the context typing judgment that consists of judgments of the form

$C : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Leftarrow B) \rightsquigarrow \mathcal{D}$.[12] Essentially, the judgment expresses that program context C expects a term of type A under typing context Γ to form an expression of type B under context Γ'. Moreover, the source context C elaborates straightforwardly to a target context $\mathcal{D}$. Indeed, for the coherence of λ_i^{MP} we do not account for all possible target contexts, but only for those that are the image of a source context C. This restriction is important, as Bi et al. [2018] have demonstrated that there are other target contexts that can distinguish between alternative elaborations.

We can now formally state coherence[13] .

Theorem 6.1 (Coherence of λ_i^{MP}). *For any* $C : (\Gamma \Rightarrow A) \mapsto (\bullet \Rightarrow \mathsf{Nat}) \rightsquigarrow \mathcal{D}$, *if* $\Gamma \vdash E \Rightarrow A \rightsquigarrow e_1$ *and* $\Gamma \vdash E \Rightarrow A \rightsquigarrow e_2$, *then there is a value* $\bullet \vdash v : \mathsf{Nat}$ *such that* $\mathcal{D}\{e_1\} \longrightarrow^* v$ *and* $\mathcal{D}\{e_2\} \longrightarrow^* v$.

Here, $\mathcal{D}\{e\}$ denotes the term that results from filling in the hole of $\mathcal{D}$ with expression e.

The remainder of this section summarizes our proof approach for coherence and discusses the issues for our proof's partiality. A more extensive account can be found in Appendix F.

6.1 A Context Dependent Logical Relation

We follow the general approach for showing coherence as NeColus. This approach is based on a rather atypical heterogeneous binary logical relation, called *canonicity*, by Bi et al. [2019].

Canonicity. The canonicity relation is in fact two families of relations, one for values $\mathcal{V}$ and one for expressions $\mathcal{E}$, indexed by types. Two values v_1 and v_2 of types τ_1 and τ_2 are related by canonicity, written $(v_1, v_2) \in \mathcal{V}(\tau_1, \tau_2)$, iff any parts of the same type have the same meaning. By parts we mean values v_1' and v_2' derivable from respectively v_1 and v_2 by means of subtyping coercions. If two values have non-trivial (i.e., non-top-like) parts with the same type, we say that they overlap. For instance, values $\langle 1, \mathsf{true}\rangle$ and 1 overlap since both have a part of type Nat; because their overlapping part is the same, namely 1, these values are related by canonicity. In contrast, $\langle 1, \mathsf{true}\rangle$ and 2 are not related since their overlapping Nat parts are different.

While the λ_i^{MP} canonicity relation follows the above principles just like that of NeColus, there are considerable differences in the actual definition because of the addition of modus ponens.

Context Dependency. A major complicating factor in the design of the new logical relation is that the modus ponens subtyping rule introduces a form of context dependency. Indeed, when reasoning about overlapping parts, the NeColus approach of decomposing terms and their types, and looking at their syntactic components in isolation no longer works. Instead, in λ_i^{MP} the components have to be considered in the context of the whole value they appear in. For example, the target term $\lambda x : \mathsf{Nat}.\, \mathsf{true}$ does not overlap with false, since one is an arrow type and the other a boolean. Yet, when $\lambda x : \mathsf{Nat}.\, \mathsf{true}$ is part of the larger term $\langle \lambda x : \mathsf{Nat}.\, \mathsf{true}, 1\rangle$, a problematic overlap arises: via modus ponens, the function applied to 1 to yields the boolean value true that differs from false.

To account for that dependency on the whole value, or contextual dependency for short, we equip the canonicity relation of λ_i^{MP} with two additional indices: the whole values and their types. Thus, we write

$$(v_1, v_2) \in \mathcal{V}^{(v_1' : \tau_1'; v_2' : \tau_2')}[\![\tau_1; \tau_2]\!]$$

to express that v_1 and v_2 are related values of types τ_1 and τ_2 that are components of respectively the values v_1' of type τ_1' and v_2' of type τ_2'.

We refer to Appendix F for the full definition of this context-dependent canonicity relation $\mathcal{V}$ for values and its companion $\mathcal{E}$ for expressions.

[12] There are also judgments for the three other combinations of the two typing directions that appear in the judgment. The full specification can be found in Appendix C.

[13] There is a second statement for the type checking direction of the two hypotheses.

Properties. A key property of the canonicity relation is *coercion compatibility*. This states that any coercions of related values are related as well.

Lemma 6.2 (Coercion Compatibility).
If $(v_1, v_2) \in \mathcal{V}^{(v_1:\tau_1;v_2:\tau_2)}[\![\tau_1;\tau_2]\!]$, *then for all coercions* c_1, c_2 *and types* τ_1', τ_2' *such that* $c_1 \vdash \tau_1 \triangleright \tau_1'$ *and* $c_2 \vdash \tau_2 \triangleright \tau_2'$ *it holds that* $(c_1\ v_1, c_2\ v_2) \in \mathcal{E}^{(c_1\ v_1:\tau_1';c_2\ v_2:\tau_2')}[\![\tau_1';\tau_2']\!]$.

The following lemma asserts another key property of the value relation, which establishes its connection to disjointness. This follows from the design principle that any overlap in types should concern indistinguishable values. If there is no overlap, then this is trivially satisfied.

Lemma 6.3 (Disjointly-Typed Values are Related). *If* $\bullet \vdash v_1 : |A_1|$ *and* $\bullet \vdash v_1 : |A_2|$ *where* $A_1 * A_2$, *then* $(v_1, v_2) \in \mathcal{V}^{(v_1:|A_1|;v_2:|A_2|)}[\![|A_1|;|A_2|]\!]$.

6.2 Coherence Proof

With the canonicity relation in place, we can turn to the actual coherence proof. This proof is split in two parts by the canonicity relation. The first and hardest part is to show that any two elaborations of the same source term are related by canonicity; this is the so-called *fundamental property*, . The second part is to show that related programs of type Nat return the same value; it follows almost trivially from the definition of the canonicity relation.

Logical Equivalence. To express the fundamental property, we need to introduce a notion of logical equivalence for open expressions on top of the canonicity relations for closed values ($\mathcal{V}$) and closed terms ($\mathcal{E}$). Open expressions are considered in some typing context Δ. A *closing substitution* γ of a typing context Δ maps variables to values of the type specified in Δ. Two open expressions are related iff applying any two related closing substitutions to e_1 and e_2, respectively, results in $\mathcal{E}$-related closed terms:

$$\Delta_1;\Delta_2 \vdash e_1 \simeq_{\log} e_2 : \tau_1;\tau_2 \triangleq \forall\,(\gamma_1;\gamma_2) \in \mathcal{G}[\![\Delta_1;\Delta_2]\!],\ (e_1[\gamma_1], e_2[\gamma_2]) \in \mathcal{V}^{(e_1[\gamma_1]:\tau_1;e_2[\gamma_2]:\tau_2)}[\![\tau_1;\tau_2]\!]$$

Two closing substitutions γ_1 and γ_2 of Δ_1 and Δ_2 are related iff the two typing contexts differ only in the types of the variables (otherwise, they contain the same variables and in the same order) and the values that γ_1 and γ_2 contain for each variable are $\mathcal{V}$-related.

Fundamental Property. Now we can state the fundamental property:[14]

Theorem 6.4 (Fundamental property). *If* $\Gamma \vdash E \Rightarrow A \rightsquigarrow e_1$ *and* $\Gamma \vdash E \Rightarrow A \rightsquigarrow e_2$,
then $|\Gamma|;|\Gamma| \vdash e_1 \simeq_{log} e_2 : |A|;|A|$.

Due to the complexity of the canonicity relation, we have proven this theorem under Conjectures 1 and 2. The proof proceeds by induction on the first hypothesis and calls coercion compatibility for the T-sub case. The conjectures are used in the inductive cases of T-abs and T-merge.

Conjecture 1 (Compatibility of abstractions). *If* $\Delta, x : \tau_1';\Delta, x : \tau_2' \vdash e_1 \simeq_{log} e_2 : \tau_1;\tau_2$,
then $\Delta;\Delta \vdash \lambda x : \tau_1'.\ e_1 \simeq_{log} \lambda x : \tau_2'.\ e_2 : \tau_1' \to \tau_1;\tau_2' \to \tau_2$.

Conjecture 2 (Compatibility of pairs). *If* $\Delta;\Delta \vdash e_1 \simeq_{log} e_2 : \tau_1;\tau_2$
and $\Delta;\Delta \vdash e_1' \simeq_{log} e_2' : \tau_1';\tau_2'$ *and* $\tau_1 * \tau_2'$ *and* $\tau_2 * \tau_1'$, *then* $\Delta;\Delta \vdash \langle e_1, e_1'\rangle \simeq_{log} \langle e_2, e_2'\rangle : \tau_1 \times \tau_1';\tau_2 \times \tau_2'$.

Regarding compatibility of pairs, a tuple value can result either from an explicit merge or from a coercion applied on some value. In the first case, the subcomponents of the tuple are disjoint with each other and thus related (by Lemma 6.3). In the second case, given that the original expression

[14]There is a second statement for the type checking direction of the two hypotheses.

is unambiguous and since coercions do not introduce ambiguity, the coercion application is also unambiguous. However, because of the context dependency it is not easy to recover the original expression and inspect it. For a similar reason, compatibility of abstraction is also hard to prove.

Logical Equivalence Entails Contextual Equivalence. Lastly, we need to show that any logically related target expressions are contextually equivalent. The proof is fairly straightforward.

Theorem 6.5 (Logical Equivalence Entails Contextual Equivalence).
If $|\Gamma|; |\Gamma| \vdash e_1 \simeq_{log} e_2 : |A|; |A|$, *then for any* $C : (\Gamma \Rightarrow A) \mapsto (\bullet \Rightarrow \mathsf{Nat}) \rightsquigarrow \mathcal{D}$, *there is a value* $\bullet \vdash v : \mathsf{Nat}$ *such that* $\mathcal{D}\{e_1\} \longrightarrow^* v$ *and* $\mathcal{D}\{e_2\} \longrightarrow^* v$.

Taken together, the (conditionally proven) fundamental property and the entailment of contextual equivalence establish Theorem 6.1, the coherence of λ_i^{MP}.

7 RELATED WORK

Resolution and Subtyping. The main contribution of our work is to unify resolution and subtyping into a single mechanism. Resolution is a common technique for automatically constructing implicit values, such as Haskell's *type class dictionaries* [Wadler and Blott 1989] or Scala's *implicits* [Odersky 2010]. Subtyping is widely used in Object-Oriented programming and various other type systems [Barendregt et al. 1983; Chen 2003; Gay and Hole 2005; Pierce and Sangiorgi 1996]. In our work we interpret resolution as a special case of coercive subtyping [Chen 2003], where coercions are generated automatically in a type-directed fashion and then "inserted" into the program. This fits well with the common type-directed elaboration style usually employed by resolution mechanisms, which work in a similar way.

As far as we are aware, the unification of subtyping and resolution is novel. Despite the fact that languages like Scala integrate both mechanisms, only Jeffery [2019] combines both mechanisms within his DIF calculus. Unfortunately—in contrast to λ_i^{MP}—DIF's subtyping is known to be undecidable. Furthermore, Jeffery does not study the coherence of DIF. Scala implicits [Odersky 2010] have motivated other researchers to look into the foundations of the mechanism. However, all previous work on implicit calculi, including the *implicit calculus* [Oliveira et al. 2012], *Cochis* [Schrijvers et al. 2019] and the *SI calculus* [Odersky et al. 2017], do not account for subtyping. Instead, they all focus on a System F [Reynolds 1974] based language (without subtyping), and model resolution for such a language. One aspect that these implicit calculi preserve from Scala is a biased form of resolution where innermost implicits are preferred. In contrast, our disjointness approach is more reminiscent of Haskell's non-overlapping type class instances, which are unbiased.

The most common technique to algorithmically construct values by resolution, attributed to Kowalski [1974], is commonly known as *SLD resolution*, or *backwards chaining*. Another technique, mostly used as a proof search method, is *focusing* [Andreoli 1992; Liang and Miller 2009; Miller et al. 1989] and has been used to great effect in similar calculi (e.g. Cochis [Schrijvers et al. 2019] and Haskell with quantified class constraints [Bottu et al. 2017]) to create coherent resolution mechanisms. Our subtyping algorithm of Section 5 closely resembles the focusing technique.

Coherence. Since Reynolds [1991] proved coherence for a calculus with intersection types, based on denotational semantics, many researchers have studied it in a variety of typed calculi. Below we summarize two commonly-found approaches in the literature: normal forms, and logical relations.

Breazu-Tannen et al. [1991] proved the coherence of a coercion translation from Fun [Cardelli and Wegner 1985] extended with recursive types to System F, by showing that any two typing derivations of the same judgment are normalizable to the same form. Curien and Ghelli [1994] presented a translation of System $F_{\leq}$ into a calculus with explicit coercions and showed that any derivations of the same judgment are translated to terms that are normalizable to a unique normal

form. Following the same approach, Schwinghammer [2009] proved the coherence of coercion translation from Moggi's computational lambda calculus [Moggi 1991] with subtyping.

Central to the first approach is to find a unique normal form for derivations. However, this approach cannot be directly applied to Curry-style calculi, i.e., where the lambda abstractions are not type-annotated. Also, it does not work when the calculus has general recursion. Biernacki and Polesiuk [2015] considered the coherence problem of coercion semantics. Their criterion for coherence of the translation is *contextual equivalence* in the target calculus. They used a logical relation for establishing coherence for coercion semantics, applicable in a variety of calculi, including delimited continuations and control-effect subtyping.

As far as we know, the first work to use logical relations to show the coherence for intersection types and the merge operator was that of Bi et al. [2018] for NeColus, and later for F_i^+ [Bi et al. 2019]. The BCD subtyping in both calculi (as well as our own) poses a non-trivial complication over Bernacki and Polesiuk's simple structural subtyping. In their system any two coercions between given types are behaviorally equivalent in the target language, so their coherence reasoning can all take place in the target language. In NeColus, F_i^+, and λ_i^{MP} this is not the case; coercions can be distinguished by arbitrary target programs but not those that are elaborations of source programs. Hence, reasoning is restricted to the latter class, which is reflected in a more complicated notion of contextual equivalence and the logical relation's non-trivial treatment of pairs. In addition to these complications, our logical relation (and consequently the coherence proof) is significantly more convoluted than that of NeColus and F_i^+, due to the non-compositionality of Modus Ponens.

Intersection Types and the Merge Operator. Forsythe [Reynolds 1988] is one of the first languages with intersection types and a merge-like operator. However, merges in Forsythe cannot contain more than one function; this restriction ensures that the language is coherent. Similarly, Castagna et al.'s $\lambda\&$ calculus [Castagna et al. 1992] includes a special merge operator that works on functions only. More recently, Dunfield [2012] showed significant expressiveness of type systems with intersection types and a merge operator, at the cost of coherence. The limitation was addressed by Oliveira et al. [2016], who introduced disjointness to ensure coherence. The combination of intersection types, a merge operator and parametric polymorphism—while achieving coherence—was first studied in the F_i calculus [Alpuim et al. 2017]. Its successor, F_i^+ [Bi et al. 2019] additionally incorporates BCD subtyping [Barendregt et al. 1983], while remaining coherent. NeColus [Bi et al. 2018] has been the starting point for our design of λ_i^{MP} and incorporates intersection types, a merge operator, and BCD-style distributivity. Though neither NeColus nor λ_i^{MP} accounts for parametric polymorphism, this is a direction we would like to explore in the future.

BCD Type System and Decidability. The BCD type system was first introduced by Barendregt et al. [1983]. It is derived from a filter lambda model in order to characterize exactly the strongly normalizing terms. The BCD type system features a powerful subtyping relation, which serves as a base for ours. Bessai et al. [2014] showed how to type classes and mixins in a BCD-style record calculus with the merge operator of Bracha and Cook [1990]. Their merge only operates on records, and they only study a type assignment system. The decidability of BCD subtyping has been shown in several works [Kurata and Takahashi 1995; Pierce 1989; Rehof and Urzyczyn 2011; Statman 2014]. Laurent [2012] has formalized the relation in Coq in order to eliminate transitivity cuts from it, but he does not deliver an algorithm. Based on the work of Statman [2014], Bessai et al. [2016] show a formally verified subtyping algorithm in Coq. Our algorithm follows the decision procedure of Bi et al. [2018] and Pierce [1989]. However the addition of modus ponens requires significant changes to the algorithm, and complicates the metatheory.

Coq Type Classes. Coq employs a flexible type class system [Sozeau and Oury 2008b] that permits overlapping instances and provides the user with the means to control resolution: Coq instances are named and can be explicitly passed as arguments. The system poses no restrictions over the implicit environment, like non-overlapping instances in Haskell or disjointness in λ_i^{MP}. Therefore, coherence becomes a choice and responsibility of the user.

In contrast, λ_i^{MP} employs a stricter system that provides less control over subtyping. First, implicit environments are not possible if not internally disjoint. Second, going against the intuition which relates classes to types and instances to appropriately typed terms, expressions in λ_i^{MP} can be rather viewed as typed implicit environments, which can be composed with the use of the merge operator. Passing a value to a function in λ_i^{MP} can be seen as bringing an implicit environment in scope within the body of the function. The function, then, can use that environment to infer values from it, but it does not have access to the names of the instances contained in it. This is conceptually different from the named instances of Coq that can be optionally passed explicitly by the user. Last, Coq uses a global environment to which instances are inserted at their point of declaration.

8 CONCLUSION AND FUTURE WORK

This paper has shown that, with a small extension, subtyping with intersection types can be used to model resolution. We have materialized our ideas in the design of λ_i^{MP}, a type-safe extension of NeColus with the logical rule for Modus Ponens. This small extension significantly increases the expressive power of λ_i^{MP}'s predecessor, allowing us to model features such as type class resolution and first-class environments.

Future Work. A natural avenue for future work is the extension of λ_i^{MP} with parametric polymorphism. When done naively, it can make the algorithm diverge and, like in Haskell, additional conditions will have to be imposed to recover termination. Moreover, we believe that there are various applications for our work that we wish to explore in the future.

In a Haskell-like setting, our design would enable multiple implicit environments, instead of a single global one. Such environments could be first-class, could be explicitly passed as arguments or returned as a result, or could be composed to form larger implicit environments. The programmer would then be able to choose which of the implicit environments to use for resolution.

In a Scala-like setting, where subtyping is present, our design helps clarify the interaction between resolution and subtyping. This is important, from the practical point of view, since the interaction between the two mechanisms has been a continuous source of compiler bugs[15]. Furthermore, our work could also be interesting to offer a unified view of subtyping and resolution, which could significantly reduce the implementation effort for the two mechanisms.

Finally, languages like TypeScript can support a merge operator for intersection types via an encoding [Alpuim et al. 2017]. Since Typescript has subtyping, intersection types and the merge operator (via an encoding), it would be quite interesting to explore the potential to extend the current formulation of intersection types in TypeScript with the ideas of our work.

ACKNOWLEDGMENTS

We are grateful to anonymous reviewers that helped improving the presentation of our work. This work has been sponsored by Hong Kong Research Grant Council projects number 17210617 and 17209519, Flemish Fund for Scientific Research project number G073816N and KU Leuven project number C14/20/079.

[15] See for example https://github.com/scala/bug/issues/2509 or https://github.com/scala/bug/issues/9764.

REFERENCES

João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. 2017. Disjoint Polymorphism. In *ESOP (Lecture Notes in Computer Science, Vol. 10201)*. Springer, 1–28.

Nada Amin, Karl Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday* (2016), 249–272. http://infoscience.epfl.ch/record/215280

Nada Amin, Adriaan Moors, and Martin Odersky. 2012. Dependent Object Types. (2012). http://infoscience.epfl.ch/record/183030

Jean-marc Andreoli. 1992. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation* 2 (1992), 297–347.

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A Filter Lambda Model and the Completeness of Type Assignment. *J. Symb. Log.* 48, 4 (1983), 931–940.

Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. 2003. CDuce: An XML-Centric General-Purpose Language. *SIGPLAN Not.* 38, 9 (Aug. 2003), 51–63. https://doi.org/10.1145/944746.944711

Jan Bessai, Boris Düdder, Andrej Dudenhefner, Tzu-Chun Chen, and Ugo de'Liguoro. 2014. Typing Classes and Mixins with Intersection Types. In *ITRS (EPTCS, Vol. 177)*. 79–93.

Jan Bessai, Andrej Dudenhefner, Boris Düdder, and Jakob Rehof. 2016. Extracting a Formally Verified Subtyping Algorithm for Intersection Types from Ideals and Filters *(TYPES)*.

Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2018. The Essence of Nested Composition. In *ECOOP (LIPIcs, Vol. 109)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 22:1–22:33.

Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. Distributive Disjoint Polymorphism for Compositional Programming. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 381–409.

Dariusz Biernacki and Piotr Polesiuk. 2015. Logical Relations for Coherence of Effect Subtyping. In *TLCA (LIPIcs, Vol. 38)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 107–122.

Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. Quantified Class Constraints. *SIGPLAN Not.* 52, 10 (Sept. 2017), 148–161.

Gilad Bracha and William R. Cook. 1990. Mixin-based Inheritance. In *OOPSLA/ECOOP*. ACM, 303–311.

Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. 1991. Inheritance As Implicit Coercion. *Inf. Comput.* 93, 1 (July 1991), 172–221.

Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.* 17, 4 (Dec. 1985), 471–523.

Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. 1992. A Calculus for Overloaded Functions with Subtyping. *SIGPLAN Lisp Pointers* V, 1 (Jan. 1992), 182–192.

Gang Chen. 2003. Coercive subtyping for the Calculus of Constructions.. In *POPL*. 150–159.

M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. 1981. Functional Characters of Solvable Terms. *Math. Log. Q.* 27 (1981), 45–58.

Pierre-Louis Curien and Giorgio Ghelli. 1994. Theoretical Aspects of Object-oriented Programming. MIT Press, Cambridge, MA, USA, Chapter Coherence of Subsumption, Minimum Typing and Type-checking in F&Le;, 247–292.

Rowan Davies and Frank Pfenning. 2000. Intersection Types and Computational Effects. *SIGPLAN Not.* 35, 9 (Sept. 2000), 198–208. https://doi.org/10.1145/357766.351259

Dominique Devriese and Frank Piessens. 2011. On the bright side of type classes: instance arguments in Agda. In *ICFP '11: Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*. ACM, 143–155.

Joshua Dunfield. 2007. Refined Typechecking with Stardust. In *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification* (Freiburg, Germany) *(PLPV '07)*. Association for Computing Machinery, New York, NY, USA, 21–32. https://doi.org/10.1145/1292597.1292602

Jana Dunfield. 2012. Elaborating Intersection and Union Types. *SIGPLAN Not.* 47, 9 (Sept. 2012), 17–28.

Joshua Dunfield. 2015. Elaborating Evaluation-Order Polymorphism. *SIGPLAN Not.* 50, 9 (Aug. 2015), 256–268. https://doi.org/10.1145/2858949.2784744

Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 268–277.

Simon J. Gay and Malcolm J. Hole. 2005. Subtyping for Session Types in the Pi Calculus. *Acta Informatica* 42, 2/3 (2005), 191–225.

J. Giesl, C. Aschermann, M. Brockschmidt, et al. 2017. Analyzing Program Termination and Complexity Automatically with AProVE. *J Autom Reasoning* 58 (2017), 3–31.

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems* 18, 2 (1996), 109–138.

Alex Jeffery. 2019. Dependent Object Types with Implicit Functions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala (Scala '19)*. Association for Computing Machinery, New York, NY, USA, 1–11.

Robert A. Kowalski. 1974. Predicate Logic as Programming Language. In *IFIP Congress*. North-Holland, 569–574.

Toshihiko Kurata and Masako Takahashi. 1995. Decidable Properties of Intersection Type Systems. In *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications (TLCA '95)*. Springer-Verlag, Berlin, Heidelberg, 297–311.

Olivier Laurent. 2012. Intersection Types with Subtyping by Means of Cut Elimination. *Fundam. Inf.* 121, 1-4 (Jan. 2012), 203–226.

Chuck Liang and Dale Miller. 2009. Focusing and Polarization in Linear, Intuitionistic, and Classical Logics. *Theor. Comput. Sci.* 410, 46 (2009), 4747–4768.

Zhaohui Luo. 1999. Coercive subtyping. *Journal of Logic and Computation* 9, 1 (1999), 105–130.

Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. 1989. *Uniform Proofs As a Foundation for Logic Programming*. Technical Report.

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (July 1991), 55–92.

Martin Odersky. 2010. The Scala language specification, version 2.8.

Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. 2017. Simplicitly: Foundations and Applications of Implicit Function Types. *Proc. ACM Program. Lang.* 2, POPL (2017).

Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type Classes As Objects and Implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, 341–360.

Bruno C.d.S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. 2012. The Implicit Calculus: A New Foundation for Generic Programming. 47, 6 (2012).

Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint Intersection Types. *SIGPLAN Not.* 51, 9 (Sept. 2016), 364–377.

Benjamin C. Pierce. 1989. *A Decision Procedure for the Subtype Relation on Intersection Types with Bounded Variables*. Technical Report. Carnegie Mellon University.

Benjamin C. Pierce. 1991. *Programming with Intersection Types and Bounded Polymorphism*. Ph.D. Dissertation. Carnegie Mellon University. Available as School of Computer Science technical report CMU-CS-91-205.

Benjamin C. Pierce. 1997. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science* 7, 2 (1997), 129–193. https://doi.org/10.1017/S096012959600223X

B. C. Pierce and D. Sangiorgi. 1996. Typing and Subtyping for Mobile Processes. *Math. Structures in Comput. Sci.* 6, 5 (1996), 409–453.

Garrel Pottinger. 1980. A type assignment for the strongly normalizable λ-terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism* (1980), 561–577.

Jef Raskin. 1974. FLOW: a teaching language for computer programming in the humanities. *Computers and the Humanities* 8, 4 (1974), 231–237.

Jakob Rehof and PawełUrzyczyn. 2011. Finite Combinatory Logic with Intersection Types. In *Proceedings of the 10th International Conference on Typed Lambda Calculi and Applications* (Novi Sad, Serbia) *(TLCA'11)*. Springer-Verlag, Berlin, Heidelberg, 169–183.

John C. Reynolds. 1972. Definitional Interpreters for Higher-order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2* (Boston, Massachusetts, USA) *(ACM '72)*. ACM, New York, NY, USA, 717–740.

John C. Reynolds. 1974. Towards a Theory of Type Structure. Lecture Notes in Computer Science, Vol. 19. 408–425.

John C. Reynolds. 1988. *Preliminary Design of the Programming Language Forsythe*. Technical Report. Carnegie Mellon University.

John C. Reynolds. 1991. The Coherence of Languages with Intersection Types. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS '91)*. Springer-Verlag, Berlin, Heidelberg, 675–700.

Tom Schrijvers, Bruno C. d. S. Oliveira, Philip Wadler, and Koar Marntirosian. 2019. COCHIS: Stable and coherent implicits. *Journal of Functional Programming* 29 (2019), e3.

Jan Schwinghammer. 2009. Coherence of Subsumption for Monadic Types. *J. Funct. Program.* 19, 2 (March 2009), 157–172.

Matthieu Sozeau and Nicolas Oury. 2008a. First-Class Type Classes. In *TPHOLs '08: Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*.

M. Sozeau and N. Oury. 2008b. First-Class Type Classes. In *TPHOLs*.

Rick Statman. 2014. A Finite Model Property for Intersection Types. In *ITRS (EPTCS, Vol. 177)*. 1–9.

Philip Wadler. 2015. Propositions as Types. *Commun. ACM* 58, 12 (Nov. 2015), 75–84.

P. Wadler and S. Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. ACM, New York, NY, USA, 60–76.

Leo White, Frédéric Bour, and Jeremy Yallop. 2015. Modular implicits. *Electronic Proceedings in Theoretical Computer Science* 198 (12 2015).

Thomas Winant and Dominique Devriese. 2018. Coherent Explicit Dictionary Application for Haskell. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (St. Louis, MO, USA) *(Haskell 2018)*. Association for Computing Machinery, New York, NY, USA, 81–93.