The Essence of Nested Composition

John Q. Double[1] and Joan R. Blind[2]

1 **Dummy University Computing Laboratory, Address/City, Country**
`open@dummyuniversity.org`
2 **Department of Informatics, Dummy College, Address/City, Country**
`access@dummycollege.org`

Abstract

Calculi with *disjoint intersection types* support an introduction form for intersections called the *merge operator*, while retaining a *coherent* semantics. Disjoint intersections types have great potential to serve as a foundation for powerful, flexible and yet type-safe and easy to reason OO languages. This paper shows how to significantly increase the expressive power of disjoint intersection types by adding support for *nested subtyping and composition*, which enables simple forms of *family polymorphism* to be expressed in the calculus. The extension with nested subtyping and composition is challenging, for two different reasons. Firstly, the subtyping relation that supports these features is non-trivial, especially when it comes to obtaining an algorithmic version. Secondly, the syntactic method used to prove coherence for previous calculi with disjoint intersection types is too inflexible, making it hard to extend those calculi with new features (such as nested subtyping). We show how to address the first problem by adapting and extending the Barendregt, Coppo and Dezani (BCD) subtyping rules for intersections with records and coercions. A sound and complete algorithmic system is obtained by using an approach inspired by Pierce's work. To address the second problem we replace the syntactic method to prove coherence, by a semantic proof method based on *logical relations*. Our work has been fully formalized in Coq, and we have an implementation of our calculus.

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Intersection types [44, 16] have a long history in programming languages. They were originally introduced to characterize exactly all strongly normalizing lambda terms. Since then, starting with Reynolds's work on Forsythe [48], they have also been employed to express useful programming language constructs, such as key aspects of *multiple inheritance* [15] in Object-Oriented Programming (OOP). One notable example is the Scala language [39] and its DOT calculus [2], which make fundamental use of intersection types to express a class/trait that extends multiple other classes/traits. Other modern languages, such as TypeScript [34], Flow [26] and Ceylon [45], also adopt some form of intersection types.

Intersection types come in different varieties in the literature. Some calculi provide an *explicit* introduction form for intersections, called the *merge operator*. This operator was introduced by Reynolds in Forsythe [48] and adopted by a few other calculi [13, 21, 41, 1]. Unfortunately, while the merge operator is powerful, it also makes it hard to get a *coherent* (or unambiguous) semantics. Unrestricted uses of the merge operator can be ambiguous, leading to an incoherent semantics where the same program can evaluate to different values. A far more common form of intersection types are the so-called *refinement types* [28, 19, 22]. Refinement types restrict the formation of intersection types so that the two types in an intersection are refinements of the same simple (unrefined) type. Refinement intersection increases only the expressiveness of types and not of terms. For this reason, Dunfield [21]

argues that refinement intersection is unsuited for encoding various useful language features that require the merge operator (or an equivalent term-level operator).

Disjoint Intersection Types. λ_i is a recently proposed calculus with a variant of intersection types called *disjoint intersection types* [41]. Calculi with disjoint intersection types feature the merge operator, with restrictions that all expressions in a merge operator must have disjoint types and all well-formed intersections are also disjoint. A bidirectional type system and the disjointness restrictions are sufficient to ensure that the semantics of the resulting calculi remains coherent.

Disjoint intersection types have great potential to serve as a foundation for powerful, flexible and yet type-safe OO languages that are easy to reason about. As shown by Alpuim et al. [1], calculi with disjoint intersection types are very expressive and can be used to statically type-check JavaScript-style programs using mixins. Yet they retain both type safety and coherence. While coherence may seem at first of mostly theoretical relevance, it turns out to be very relevant for OOP. Multiple inheritance is renowned for being tricky to get right, largely because of the possible *ambiguity* issues caused by the same field/method names inherited from different parents [7, 52]. Disjoint intersection types enforce that the types of parents are disjoint and thus that no conflicts exist. Any violations are statically detected and can be manually resolved by the programmer. This is very similar to existing trait models [27, 20]. Therefore in an OO language modelled on top of disjoint intersection types, coherence implies that no ambiguity arises from multiple inheritance. This makes reasoning a lot simpler.

Family Polymorphism. One powerful and long-standing idea in OOP is *family polymorphism* [23]. In family polymorphism inheritance is extended to work on a *whole family of classes*, rather than just a single class. This enables high degrees of modularity and reuse, including simple solutions to hard programming language problems, like the Expression Problem [57]. An essential feature of family polymorphism is *nested composition* [17, 25, 37], which allows the automatic inheritance/composition of nested (or inner) classes when the top-level classes containing them are composed. Designing a sound type system that fully supports family polymorphism and nested composition is notoriously hard; there are only a few, quite sophisticated, languages that manage this [25, 37, 14, 51].

NeColus. This paper presents NeColus[1]: a simply-typed calculus with records and disjoint intersection types that supports *nested composition*. Nested composition enables encoding simple forms of family polymorphism. More complex forms of family polymorphism, involving binary methods [9] and mutable state are not yet supported, but are interesting avenues for future work. Nevertheless, in NeColus, it is possible, for example, to encode Ernst's elegant family-polymorphism solution [23] to the Expression Problem. Compared to λ_i the essential novelty of NeColus are distributivity rules between function/record types and intersection types. These rules are the delta that enable extending the simple forms of multiple inheritance/composition supported by λ_i into a more powerful form supporting nested composition. The distributivity rule between function types and intersections is common in calculi with intersection types aimed at capturing the set of all strongly normalizable terms, and was first proposed by Barendregt et al. [3] (BCD). However the distributivity rule is not common in calculi or languages with intersection types aimed at programming. For example the rules employed in languages that support intersection types (such as Scala, TypeScript, Flow or Ceylon) lack distributivity rules. Moreover distributivity is also missing from several

[1] NeColus stands for **Ne**sted **Co**mposition calcu**lus**

calculi with a merge operator. This includes all calculi with disjoint intersection types and Dunfield's work on elaborating intersection types, which was the original foundation for λ_i. A possible reason for this omission in the past is that distributivity adds substantial complexity (both algorithmically and metatheoretically), without having any obvious practical applications. This paper shows how to deal with the complications of BCD subtyping, while identifying a major reason to include it in a programming language: BCD enables nested composition and subtyping, which is of significant practical interest.

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

In this work we remove the brittleness of the previous syntactic method to prove coherence, by employing a more semantic proof method based on *logical relations* [56, 43, 54]. This new proof method has several advantages. Firstly, with the new proof method, several restrictions that were enforced by λ_i to enable the syntactic proof method are removed. For example the work on λ_i has to carefully distinguish between so-called *top-like types* and other types. In NeColus this distinction is not necessary; top-like types are handled like all other types. Secondly, the method based on logical relations is more powerful because it is based on semantic rather than syntactic equality. Finally, the removal of the ad-hoc side-conditions makes adding new extensions, such as support for BCD-style subtyping, easier. In order to deal with the complexity of the elaboration semantics of NeColus, we employ binary logical relations that are heterogeneous, parameterized by two types; the fundamental property is also reformulated to account for bidirectional type-checking.

In summary the contributions of this paper are:

- NeColus: a calculus with (disjoint) intersection types that features both *BCD-style subtyping* and *the merge operator*. This calculus is both type-safe and coherent, and supports *nested composition*.
- A more flexible notion of disjoint intersection types where only merges need to be checked for disjointness. This removes the need for enforcing disjointness for all well-formed types, making the calculus more easily extensible.
- An extension of BCD subtyping with both records and elaboration into coercions, and algorithmic subtyping rules with coercions, inspired by Pierce's decision procedure [42].
- A more powerful proof strategy for coherence of disjoint intersection types based on logical relations.
- Illustrations of how the calculus can encode essential features of *family polymorphism* through nested composition.
- A comprehensive Coq mechanization of all meta-theory. This has notably revealed several missing lemmas and oversights in Pierce's manual proof [42] of BCD's algorithmic subtyping. We also have an implementation of a language built on top of NeColus; it runs and type-checks all examples shown in the paper[2].

[2] The Coq formalization, implementation and appendix are submitted as supplementary material.

2 Overview

This section illustrates NeColus with an encoding of a family polymorphism solution to the Expression Problem, and informally presents its salient features.

2.1 Motivation: Family Polymorphism

In OOP *family polymorphism* is the ability to simultaneously refine a family of related classes through inheritance. This is motivated by a need to not only refine individual classes, but also to preserve and refine their mutual relationships. Nystrom et al. [37] call this *scalable extensibility*: "the ability to extend a body of code while writing new code proportional to the differences in functionality". A well-studied mechanism to achieve family inheritance is *nested inheritance* [37]. Nested inheritance combines two aspects. Firstly, a class can have nested class members; the outer class is then a family of (inner) classes. Secondly, when one family extends another, it inherits (and can override) all the class members, as well as the relationships within the family (including subtyping) between the class members. However, the members of the new family do not become subtypes of those in the parent family.

The Expression Problem, Scandinavian Style. Ernst [23] illustrates the benefits of nested inheritance for modularity and extensibility with one of the most elegant and concise solutions to the *Expression Problem* [57]. The objective of the Expression Problem is to extend a datatype, consisting of several cases, together with several associated operations in two dimensions: by adding more cases to the datatype and by adding new operations for the datatype. Ernst solves the Expression Problem in the gbeta language, which he adorns with a Java-like syntax for presentation purposes, for a small abstract syntax tree (AST) example. His starting point is the code shown in Fig. 1a. The outer class `Lang` contains a family of related AST classes: the common superclass `Exp` and two cases, `Lit` for literals and `Add` for addition. The AST comes equipped with one operation, `toString`, which is implemented by both cases.

Adding a New Operation. One way to extend the family, is to add an additional evaluation operation, as shown in the top half of Fig. 1b. This is done by subclassing the `Lang` class and refining all the contained classes by implementing the additional `eval` method. Note that the inheritance between, e.g., `Lang.Exp` and `Lang.Lit` is transferred to `LangEval.Exp` and `LangEval.Lit`. Similarly, the `Lang.Exp` type of the `left` and `right` fields in `Lang.Add` is automatically refined to `LangEval.Exp` in `LangEval.Add`.

Adding a New Case. A second dimension to extend the family is to add a case for negation, shown in the bottom half of Fig. 1b. This is similarly achieved by subclassing `Lang`, and now adding a new contained class `Neg`, for negation, that implements the `toString` operation.

Finally, the two extensions are naturally combined by means of multiple inheritance, closing the diamond.

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

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

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

(a) Base family: the language `Lang`

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

(b) Extending in two dimensions

Figure 1 The Expression Problem, Scandinavian Style

2.2 The Expression Problem, NeColus Style

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

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

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

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

Here is the implementation of `Lang`.

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

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

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

The type for `LangEval` is then:

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

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

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

Adding Negation. Adding negation to `Lang` works similarly.

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

implNegPrint : NegPrint = {
  neg (exp : IPrint) = { print = "-" ++ exp.print }
};
implLangNeg : LangNeg = implLang ,, implNegPrint;
```

Putting Everything Together. Finally, we can combine the two extensions and provide the missing implementation of evaluation for the negation case.

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

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

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

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

Multi-Field Records. One relevant remark is that NeColus does not have multi-field record types built in. They are merely syntactic sugar for intersections of single-field record types. Hence, the following is an equivalent definition of `Lang`:

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

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

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

Subtyping. A big difference with gbeta is that many more NeColus types are related through subtyping. Indeed, gbeta is unnecessarily conservative [24]: none of the families is related through subtyping, nor is any of the class members of one family related to any of the class members in another family. For instance, `LangEval` is not a subtype of `Lang`, nor is `LangNeg.Lit` a subtype of `Lang.Lit`.

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

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

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

Nevertheless, the definition of `implLangEval` is valid because `Lang & EvalExt` is a subtype of `LangEval`. Indeed, if we consider for the sake of simplicity only the `lit` field, we have that `(Int → IPrint) & (Int → IEval)` is a subtype of `Int → IPrint & IEval`. This follows from a standard subtyping axiom for distributivity of functions and intersections in the BCD system inherited by NeColus. In conclusion, `Lang & EvalExt` is a subtype of both `Lang` and of `LangEval`. However, neither of the latter two types is a subtype of the other. Indeed, `LangEval` is not a subtype of `Lang` as the type of `add` is not covariantly refined and thus admitting the subtyping is unsound. For the same reason `Lang` is not a subtype of `LangEval`. Figure 2 shows the various relationships between the language components.

Stand-Alone Extensions. Unlike in gbeta and other class-based inheritance systems, in NeColus the extension `implEval` is not tied to `LangEval`. In that sense, it resembles trait and mixin systems that can apply the same extension to different classes. However, unlike those systems, `implEval` can also exist as a value on its own, i.e., it is not an extension per se.

2.3 Disjoint Intersection Types and Ambiguity

The above example shows that intersection types and the merge operator are closely related to multiple inheritance. Indeed, they share a major concern with multiple inheritance, namely ambiguity. When a subclass inherits an implementation of the same method from two different parent classes, it is unclear which of the two methods is to be adopted by the subclass. In the case where the two parent classes have a common superclass, this is known as the *diamond problem*. The ambiguity problem also appears in NeColus, e.g., if we merge two numbers to obtain $1\,,\,2$ of type $\mathsf{Nat}\,\&\,\mathsf{Nat}$. Is the result of $1\,,\,2+3$ either 4 or 5?

Disjoint intersection types offer to statically detect potential ambiguity and to ask the programmer to explicitly resolve the ambiguity by rejecting the program in its ambiguous form. In the previous work on λ_i, ambiguity is avoided by dictating that all intersection types have to be disjoint, i.e., $\mathsf{Nat}\,\&\,\mathsf{Nat}$ is ill-formed because the first component has the same type as the second.

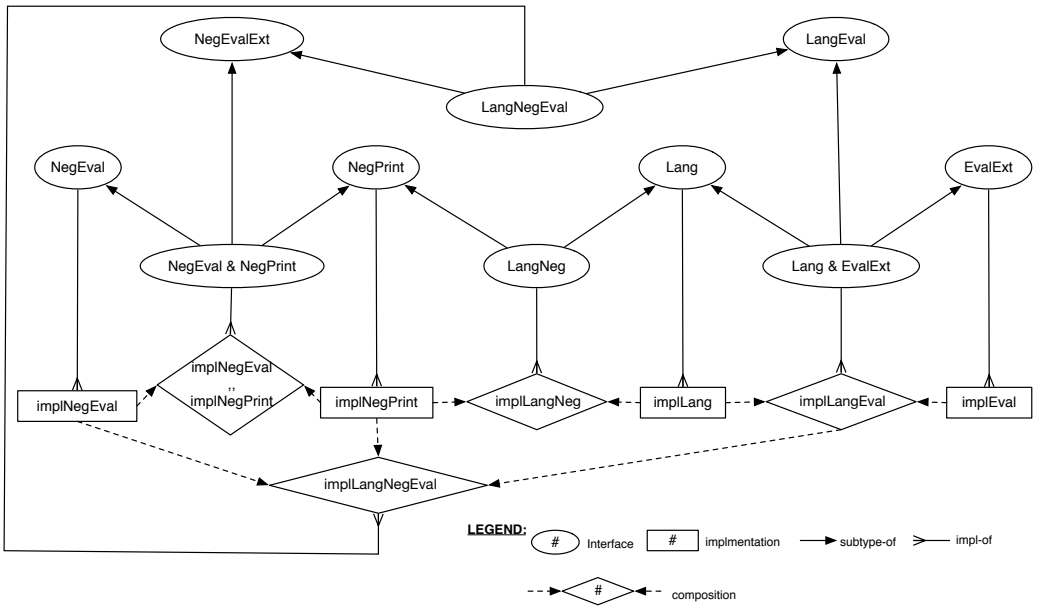

Figure 2 Summary diagram of the relationships between language components

Duplication is Harmless. While requiring that all intersections are disjoint is sufficient to guarantee coherence, it is not necessary. In fact, such requirement unnecessarily encumbers the subtyping definition with disjointness constraints and an ad-hoc treatment of "top-like" types. Indeed, the value $1\,,\,1$ of the non-disjoint type $\mathsf{Nat}\,\&\,\mathsf{Nat}$ is entirely unambiguous, and $(1\,,\,1)+3$ can obviously only result in 4. More generally, when the overlapping components of an intersection type have the same value, there is no ambiguity problem. NeColus uses this idea to relax λ_i's enforcement of disjointness. In the case of a merge, it is hard to statically decide whether the two arguments have the same value, and thus NeColus still requires disjointness. This is why in Fig. 2 we cannot define `implLangNegEval` by directly composing the two existing `implLangEval` and `implLangNeg`, even though the latter two both contain the same `implLang`. Yet, disjointness is no longer required for the well-formedness of types and overlapping intersections can be created implicitly through subtyping, which results in duplicating values at runtime. For instance, while $1\,,\,1$ is not expressible $1 : \mathsf{Nat}\,\&\,\mathsf{Nat}$ creates the equivalent value implicitly. In short, duplication is harmless and subtyping only generates duplicated values for non-disjoint types.

2.4 Logical Relations for Coherence

Coherence is easy to establish for λ_i as its rigid rules mean that there is at most one possible subtyping derivation between any two types. As a consequence there is only one possible elaboration and thus one possible behavior for any program.

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

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

Figure 3 Syntax of NeColus

become a much more subtle property that requires sophisticated reasoning. For instance, in the example, we can see that coherence holds because at runtime any value of type Nat & Nat has identical components, and thus both projections yield the same result.

For NeColus in general, we show coherence by capturing the non-ambiguity invariant in a logical relation and showing that it is preserved by the operational semantics. A complicating factor is that not one, but two languages are involved: the source language NeColus and the target language, essentially the simply-typed lambda calculus extended with coercions and records. The logical relation does not hold for target programs and program contexts in general, but only for those that are the image of a corresponding source program or program context. Thus we must view everything through the lens of elaboration.

3 NeColus: Syntax and Semantics

In this section we formally present the syntax and semantics of NeColus. We focus on the differences and improvements over λ_i. Notably, NeColus has a much more powerful subtyping relation compared to λ_i. The new subtyping relation is inspired by BCD-style subtyping, but with two noteworthy differences: subtyping is coercive (in contrast to traditional formulations of BCD); and it is extended with records. Furthermore NeColus removes the restrictions on ordinary types and top-like types, and does not require a well-formedness judgement restricting intersections to be disjoint. A final difference is a new target language with explicit coercions inspired by the work of Biernacki and Polesiuk [6].

3.1 Syntax

Figure 3 shows the syntax of NeColus, with the differences from λ_i highlighted. The NeColus calculus is essentially a simply-typed λ-calculus (STLC) extended with intersection types, the merge operator and singleton records. For brevity of the meta-theoretic study, we do not consider primitive operations on natural numbers, or other primitive types. They can be easily added to the language, and our prototype implementation is indeed equipped with common primitive types and their operations.

Metavariables A, B, C range over types. Apart from STLC types, NeColus includes a top type $\top$; intersection types $A \,\&\, B$; and singleton record types $\{l : A\}$. Metavariable E ranges over expressions. Apart from STLC expressions, NeColus includes a canonical top value $\top$; the merge expression $E_1 \,,\, E_2$; singleton records $\{l = E\}$; and record selections $E.l$.

3.2 Declarative Subtyping

The main difference between NeColus and λ_i lies in its subtyping relation, given in Fig. 4. We ignore the highlighted parts, and explain them later in Section 3.4.

BCD-Style Subtyping. The subtyping rules are essentially those of the BCD type system [3], extended with subtyping for singleton records. Rules S-TOP and S-RCD for top types and

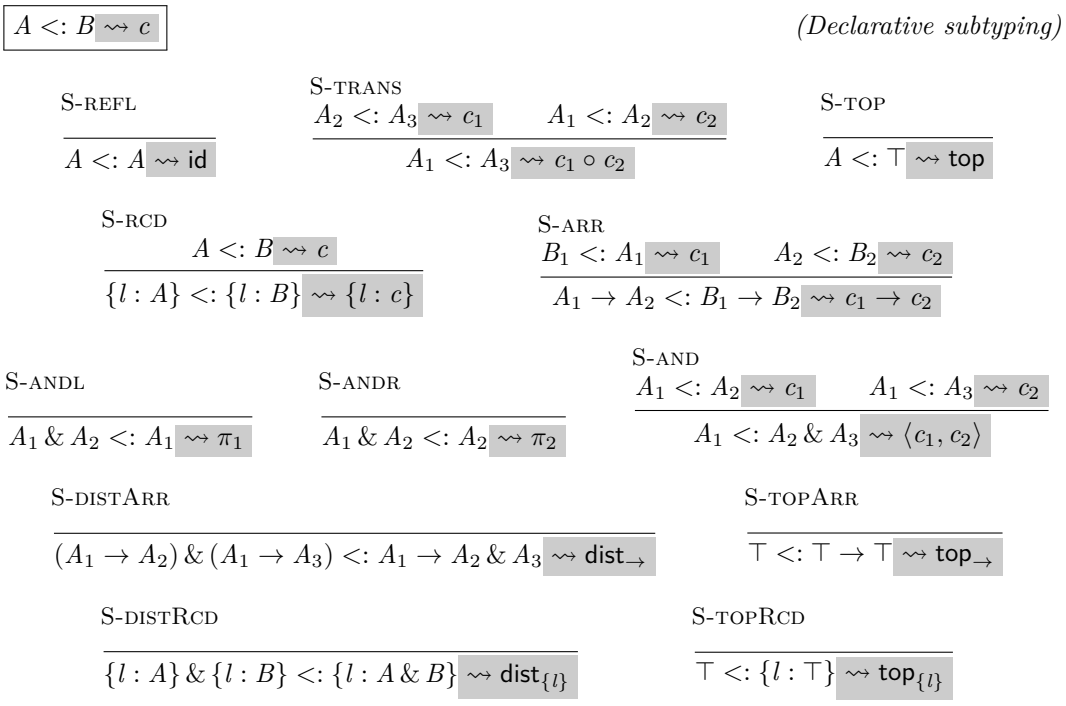

Figure 4 Declarative specification of subtyping

record types are straightforward. Rule S-ARR for function subtyping is standard. Rules S-ANDL, S-ANDR, and S-AND for intersection types axiomatize that $A \,\&\, B$ is the greatest lower bound of A and B. Rule S-DISTARR is perhaps the most interesting rule. This, so-called "distributivity" rule, describes the interaction between the subtyping relations for function types and those for intersection types. Note that the other direction (i.e., $A_1 \to A_2 \,\&\, A_3 <: (A_1 \to A_2) \,\&\, (A_1 \to A_3)$) is admissible relative to the existing subtyping rules. In the same vein, Rule S-DISTRCD, which is not found in the original BCD system, prescribes the distribution of records over intersection types. The two distributivity rules are the key to enable nested composition. The rule S-TOPARR is standard in BCD subtyping, and the new rule S-TOPRCD plays a similar role for record types.

Non-Algorithmic. The subtyping relation in Fig. 4 is clearly no more than a specification due to the two subtyping axioms: rules S-REFL and S-TRANS. A sound and complete algorithmic version is discussed in Section 5. Nevertheless, for the sake of establishing coherence, the declarative subtyping relation is sufficient.

No Ordinary Types. Apart from the extra subtyping rules, there is an important difference from the λ_i subtyping relations. The subtyping relations of λ_i employ an auxiliary unary relation called ordinary, which plays a fundamental role in both calculi for ensuring coherence and obtaining an algorithm [19]. For reasons that will become clear in Section 4, NeColus improves over λ_i in that it discards the notion of ordinary types completely; this yields a clean and elegant formulation of the subtyping relation. Another minor difference is that due to the addition of the transitivity axiom (rule S-TRANS), rules S-ANDL and S-ANDR are simplified: an intersection type $A \,\&\, B$ is a subtype of both A and B, instead of the more general form $A \,\&\, B <: C$.

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

T-TOP
$$\frac{}{\Gamma \vdash \top \Rightarrow \top \rightsquigarrow \langle\rangle}$$

T-LIT
$$\frac{}{\Gamma \vdash i \Rightarrow \mathsf{Nat} \rightsquigarrow i}$$

T-VAR
$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A \rightsquigarrow x}$$

T-APP
$$\frac{\Gamma \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e_1 \quad \Gamma \vdash E_2 \Leftarrow A_1 \rightsquigarrow e_2}{\Gamma \vdash E_1\ E_2 \Rightarrow A_2 \rightsquigarrow e_1\ e_2}$$

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

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

T-MERGE
$$\frac{\Gamma \vdash E_1 \Rightarrow A_1 \rightsquigarrow e_1 \quad \Gamma \vdash E_2 \Rightarrow A_2 \rightsquigarrow e_2 \quad A_1 * A_2}{\Gamma \vdash E_1\ ,,\ E_2 \Rightarrow A_1 \,\&\, A_2 \rightsquigarrow \langle e_1, e_2\rangle}$$

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

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

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

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

Figure 5 Bidirectional type system of NeColus

3.3 Typing of NeColus

The bidirectional type system for NeColus is shown in Fig. 5. Again we ignore the highlighted parts for now. The main difference to λ_i is the absence of a well-formedness judgement. Unlike λ_i, which requires a well-formedness judgement to ensure that all intersection types are disjoint, NeColus only requires a disjointness check at the merge operator. Non-disjoint types such as $\mathsf{Nat} \,\&\, \mathsf{Nat}$ are allowed in other parts of the program.

Typing Rules and Disjointness. As with traditional bidirectional type systems, we employ two modes: the inference mode ($\Rightarrow$) and the checking mode ($\Leftarrow$). The inference judgement $\Gamma \vdash E \Rightarrow A$ says that we can synthesize a type A for expression E in the context Γ. The checking judgement $\Gamma \vdash E \Leftarrow A$ checks E against A in the context Γ. The disjointness judgement $A * B$ used in rule T-MERGE is shown in Fig. 6, which states that the types A and B are disjoint. The disjointness judgement is important in order to rule out ambiguous expressions such as $1\ ,,\ 2$. Most of the typing and disjointness rules are standard and are explained in detail in previous work [41, 1].

No Well-Formedness Judgement. A key difference from the type system of λ_i is the complete omission of the well-formedness judgement. In λ_i, the well-formedness judgement $\Gamma \vdash A$ appears in both rules T-ABS and T-SUB. The sole purpose of this judgement is to enforce the invariant that all intersection types are disjoint. However, as Section 4 will explain, the syntactic restriction is unnecessary for coherence, and merely complicates the type system. The NeColus calculus discards this well-formedness judgement altogether in favour of a simpler design that is still coherent. So an expression such as $1 : \mathsf{Nat} \,\&\, \mathsf{Nat}$

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

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

$$\text{D-andR}\ \frac{A * B_1 \qquad A * B_2}{A * B_1 \,\&\, B_2} \qquad \text{D-rcdEq}\ \frac{A * B}{\{l : A\} * \{l : B\}} \qquad \text{D-rcdNeq}\ \frac{l_1 \neq l_2}{\{l_1 : A\} * \{l_2 : B\}} \qquad \text{D-axNatArr}\ \frac{}{\mathsf{Nat} * A_1 \to A_2}$$

$$\text{D-axArrNat}\ \frac{}{A_1 \to A_2 * \mathsf{Nat}} \qquad \text{D-axNatRcd}\ \frac{}{\mathsf{Nat} * \{l : A\}} \qquad \text{D-axRcdNat}\ \frac{}{\{l : A\} * \mathsf{Nat}} \qquad \text{D-axArrRcd}\ \frac{}{A_1 \to A_2 * \{l : A\}}$$

$$\text{D-axRcdArr}\ \frac{}{\{l : A\} * A_1 \to A_2}$$

Figure 6 Disjointness

Target types	τ	::=	$\mathsf{Nat} \mid \langle\rangle \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \{l : \tau\}$
Typing contexts	Δ	::=	$\bullet \mid \Delta, x : \tau$
Target terms	e	::=	$x \mid i \mid \langle\rangle \mid \lambda x.\, e \mid e_1\ e_2 \mid \langle e_1, e_2\rangle \mid \{l = e\} \mid e.l \mid c\ e$
Coercions	c	::=	$\mathsf{id} \mid c_1 \circ c_2 \mid \mathsf{top} \mid \mathsf{top}_{\to} \mid \mathsf{top}_{\{l\}} \mid c_1 \to c_2 \mid \langle c_1, c_2\rangle$
		$\mid$	$\pi_1 \mid \pi_2 \mid \{l : c\} \mid \mathsf{dist}_{\{l\}} \mid \mathsf{dist}_{\to}$
Target values	v	::=	$\lambda x.\, e \mid \langle\rangle \mid i \mid \langle v_1, v_2\rangle \mid (c_1 \to c_2)\ v \mid \mathsf{dist}_{\to}\ v \mid \mathsf{top}_{\to}\ v$

Figure 7 λ_c types, terms and coercions

is accepted in NeColus but rejected in λ_i. This simplification is based on an important observation: incoherence can only originate in merges. Therefore disjointness checking is only necessary in rule T-merge.

3.4 Elaboration Semantics

The operational semantics of NeColus is given by elaborating source expressions E into target terms e. Our target language λ_c is the standard simply-typed call-by-value λ-calculus extended with singleton records, products and coercions. The syntax of λ_c is shown in Fig. 7. The meta-function $|\cdot|$ transforms NeColus types to λ_c types, and extends naturally to typing contexts. Its definition is in the appendix.

Explicit Coercions and Coercive Subtyping. The separate syntactic category for explicit coercions c is a distinct difference from the target language of λ_i. Coercions express the conversion of a term from one type to another. Because of the addition of coercions, the grammar contains explicit coercion applications $c\ e$ as a term, and various unsaturated coercion applications as values. The use of coercions is inspired by the work of Biernacki and Polesiuk [6], and useful for the new semantic proof of coherence based on logical relations. Since our subtyping relation is more complex than that of theirs, we have more forms of

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

$$\textsc{cotyp-refl}\ \frac{}{\mathsf{id} \vdash \tau \triangleright \tau} \qquad \textsc{cotyp-trans}\ \frac{c_1 \vdash \tau_2 \triangleright \tau_3 \qquad c_2 \vdash \tau_1 \triangleright \tau_2}{c_1 \circ c_2 \vdash \tau_1 \triangleright \tau_3} \qquad \textsc{cotyp-top}\ \frac{}{\mathsf{top} \vdash \tau \triangleright \langle\rangle} \qquad \textsc{cotyp-topArr}\ \frac{}{\mathsf{top}_{\to} \vdash \langle\rangle \triangleright \langle\rangle \to \langle\rangle}$$

$$\textsc{cotyp-topRcd}\ \frac{}{\mathsf{top}_{\{l\}} \vdash \langle\rangle \triangleright \{l : \langle\rangle\}} \qquad \textsc{cotyp-arr}\ \frac{c_1 \vdash \tau_1' \triangleright \tau_1 \qquad c_2 \vdash \tau_2 \triangleright \tau_2'}{c_1 \to c_2 \vdash \tau_1 \to \tau_2 \triangleright \tau_1' \to \tau_2'} \qquad \textsc{cotyp-pair}\ \frac{c_1 \vdash \tau_1 \triangleright \tau_2 \qquad c_2 \vdash \tau_1 \triangleright \tau_3}{\langle c_1, c_2\rangle \vdash \tau_1 \triangleright \tau_2 \times \tau_3}$$

$$\textsc{cotyp-projL}\ \frac{}{\pi_1 \vdash \tau_1 \times \tau_2 \triangleright \tau_1} \qquad \textsc{cotyp-projR}\ \frac{}{\pi_2 \vdash \tau_1 \times \tau_2 \triangleright \tau_2} \qquad \textsc{cotyp-rcd}\ \frac{c \vdash \tau_1 \triangleright \tau_2}{\{l : c\} \vdash \{l : \tau_1\} \triangleright \{l : \tau_2\}}$$

$$\textsc{cotyp-distRcd}\ \frac{}{\mathsf{dist}_{\{l\}} \vdash \{l : \tau_1\} \times \{l : \tau_2\} \triangleright \{l : \tau_1 \times \tau_2\}} \qquad \textsc{cotyp-distArr}\ \frac{}{\mathsf{dist}_{\to} \vdash (\tau_1 \to \tau_2) \times (\tau_1 \to \tau_3) \triangleright \tau_1 \to \tau_2 \times \tau_3}$$

Figure 8 Coercion typing

coercions.[3] The subtyping judgement in Fig. 4 has the form $A <: B \rightsquigarrow c$, which says that the subtyping derivation of $A <: B$ produces a coercion c that converts terms of type $|A|$ to type $|B|$. Each subtyping rule has its own specific form of coercion.

No Top-like Types. Apart from the syntactic category of explicit coercions, there is a notable difference from the coercive subtyping of λ_i, where *top-like types* are introduced and used in rules S-ANDL and S-ANDR. Because of their syntactic proof method, λ_i requires special treatment of top-like types in order to retain coherence. However in NeColus, as with ordinary types, we do not need this kind of ad-hoc treatment, thanks to the adoption of a more powerful proof method (c.f. Section 4).

Target Typing. The typing of λ_c has the form $\Delta \vdash e : \tau$, which is entirely standard. Only the typing of coercion applications, shown below, deserves attention:

$$\textsc{typ-capp}\ \frac{\Delta \vdash e : \tau \qquad c \vdash \tau \triangleright \tau'}{\Delta \vdash c\ e : \tau'}$$

Here the judgement $c \vdash \tau_1 \triangleright \tau_2$ expresses the typing of coercions, which are essentially functions from τ_1 to τ_2. Their typing rules correspond exactly to the subtyping rules of the source language NeColus, as shown in Fig. 8.

Target Operational Semantics and Type Safety. The operational semantics of λ_c is mostly unremarkable. What may be interesting is the operational semantics of coercions, whose single-step ($\longrightarrow$) reduction rules are shown in Fig. 9. As standard, $\longrightarrow^*$ is the reflexive, transitive closure of $\longrightarrow$. We show that λ_c is type safe:

▶ **Theorem 1** (Preservation). *If* $\bullet \vdash e : \tau$ *and* $e \longrightarrow e'$, *then* $\bullet \vdash e' : \tau$.

▶ **Theorem 2** (Progress). *If* $\bullet \vdash e : \tau$, *then either* e *is a value, or there exists* e' *such that* $e \longrightarrow e'$.

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

$\boxed{e \longrightarrow e'}$ *(Coercion reduction)*

STEP-ID

$$\frac{}{\mathsf{id}\ v \longrightarrow v}$$

STEP-TRANS

$$\frac{}{(c_1 \circ c_2)\ v \longrightarrow c_1\ (c_2\ v)}$$

STEP-TOP

$$\frac{}{\mathsf{top}\ v \longrightarrow \langle\rangle}$$

STEP-TOPARR

$$\frac{}{(\mathsf{top}_{\to}\ \langle\rangle)\ \langle\rangle \longrightarrow \langle\rangle}$$

STEP-TOPRCD

$$\frac{}{\mathsf{top}_{\{l\}}\ \langle\rangle \longrightarrow \{l = \langle\rangle\}}$$

STEP-PAIR

$$\frac{}{\langle c_1, c_2\rangle\ v \longrightarrow \langle c_1\ v, c_2\ v\rangle}$$

STEP-ARR

$$\frac{}{((c_1 \to c_2)\ v_1)\ v_2 \longrightarrow c_2\ (v_1\ (c_1\ v_2))}$$

STEP-DISTARR

$$\frac{}{(\mathsf{dist}_{\to}\ \langle v_1, v_2\rangle)\ v_3 \longrightarrow \langle v_1\ v_3, v_2\ v_3\rangle}$$

STEP-PROJL

$$\frac{}{\pi_1\ \langle v_1, v_2\rangle \longrightarrow v_1}$$

STEP-PROJR

$$\frac{}{\pi_2\ \langle v_1, v_2\rangle \longrightarrow v_2}$$

STEP-CRCD

$$\frac{}{\{l : c\}\ \{l = v\} \longrightarrow \{l = c\ v\}}$$

STEP-DISTRCD

$$\frac{}{\mathsf{dist}_{\{l\}}\ \langle\{l = v_1\}, \{l = v_2\}\rangle \longrightarrow \{l = \langle v_1, v_2\rangle\}}$$

Figure 9 Coercion reduction

Elaboration. We are now in a position to explain the elaboration judgements $\Gamma \vdash E \Rightarrow A \rightsquigarrow e$ and $\Gamma \vdash E \Leftarrow A \rightsquigarrow e$ in Fig. 5. The only interesting rule is rule T-SUB, which applies the coercion c produced by subtyping to the target term e to form a coercion application $c\ e$. All the other rules do straightforward translations between source and target expressions.

To conclude, we show two lemmas that relate NeColus expressions to λ_c terms.

▶ **Lemma 1** (Coercions preserve types). *If* $A <: B \rightsquigarrow c$, *then* $c \vdash |A| \triangleright |B|$.

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

▶ **Lemma 2** (Elaboration soundness). *Let* $\Leftrightarrow \in \{\Rightarrow, \Leftarrow\}$, *if* $\Gamma \vdash E \Leftrightarrow A \rightsquigarrow e$, *then* $|\Gamma| \vdash e : |A|$.

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

4 Coherence

This section constructs logical relations to establish the coherence of NeColus. It turns out that finding a suitable definition of coherence for NeColus is already challenging in its own right. In what follows we reproduce the steps of finding a definition for coherence that is both intuitive and applicable. Then we present the construction of logical (equivalence) relations tailored to this definition, and the connection between logical equivalence and coherence.

4.1 In Search of Coherence

In λ_i the definition of coherence is based on α-equivalence. More specifically, their coherence property states that any two target terms that a source expression elaborates into must be exactly the same (up to α-equivalence). Unfortunately this syntactic notion of coherence is very fragile with respect to extensions. For example, it is not obvious how to retain this notion of coherence when adding more subtyping rules such as those in Fig. 4.

λ_c contexts	$\mathcal{D}$	::=	$[\cdot] \mid \lambda x.\, \mathcal{D} \mid \mathcal{D}\, e_2 \mid e_1\, \mathcal{D} \mid \langle \mathcal{D}, e_2 \rangle \mid \langle e_1, \mathcal{D} \rangle \mid c\, \mathcal{D} \mid \{l = \mathcal{D}\} \mid \mathcal{D}.l$
NeColus contexts	$\mathcal{C}$	::=	$[\cdot] \mid \lambda x.\, \mathcal{C} \mid \mathcal{C}\, E_2 \mid E_1\, \mathcal{C} \mid E_1 \,,\, \mathcal{C} \mid \mathcal{C} \,,\, E_2 \mid \mathcal{C} : A \mid \{l = \mathcal{C}\} \mid \mathcal{C}.l$

Figure 10 Expression contexts of NeColus and λ_c

If we permit ourselves to consider only the syntactic aspects of expressions, then very few expressions can be considered equal. The syntactic view is also in conflict with the intuition that the significance of an expression lies in its contribution to the *outcome* of a computation [29]. Drawing inspiration from a wide range of literature on contextual equivalence [36], we want a context-based notion of coherence. It is helpful to consider several examples before presenting the formal definition of our new semantically founded notion of coherence.

▶ **Example 1.** The same NeColus expression 3 can be typed Nat in many ways: for instance, by rule T-LIT; by rules T-SUB and S-REFL; or by rules T-SUB, S-TRANS, and S-REFL, resulting in λ_c terms 3, id 3 and (id ∘ id) 3, respectively. It is apparent that these three λ_c terms are "equal" in the sense that all reduce to the same numeral 3.

Expression Contexts and Contextual Equivalence. To formalize the intuition, we introduce the notion of *expression contexts*. An expression context $\mathcal{D}$ is a term with a single hole $[\cdot]$ (possibly under some binders) in it. The syntax of λ_c expression contexts can be found in Fig. 10. The typing judgement for expression contexts has the form $\mathcal{D} : (\Delta \vdash \tau) \rightsquigarrow (\Delta' \vdash \tau')$ where $(\Delta \vdash \tau)$ indicates the type of the hole. This judgement essentially says that plugging a well-typed term $\Delta \vdash e : \tau$ into the context $\mathcal{D}$ gives another well-typed term $\Delta' \vdash \mathcal{D}\{e\} : \tau'$. We define a *complete program* to mean any closed term of type Nat. The following two definitions capture the notion of *contextual equivalence*.

▶ **Definition 1** (Kleene Equality)**.** Two complete programs, e and e', are Kleene equal, written $e \backsimeq e'$, iff there exists i such that $e \longrightarrow^* i$ and $e' \longrightarrow^* i$.

▶ **Definition 2** (λ_c Contextual Equivalence)**.** If $\Delta \vdash e_1 : \tau$ and $\Delta \vdash e_2 : \tau$, we write $\Delta \vdash e_1 \backsimeq_{ctx} e_2 : \tau$ to mean

$$\forall \mathcal{D}.\ \mathcal{D} : (\Delta \vdash \tau) \rightsquigarrow (\bullet \vdash \mathsf{Nat}) \implies \mathcal{D}\{e_1\} \backsimeq \mathcal{D}\{e_2\}$$

Regarding Example 1, it seems adequate to say that 3 and id 3 are contextually equivalent. Does this imply that coherence can be based on Definition 2? Unfortunately it cannot, as demonstrated by the following example.

▶ **Example 2.** It may be counter-intuitive that two λ_c terms $\lambda x.\, \pi_1\, x$ and $\lambda x.\, \pi_2\, x$ should also be considered equal. To see why, first note that they are both translations of the same NeColus expression: $(\lambda x.\, x) : \mathsf{Nat} \,\&\, \mathsf{Nat} \to \mathsf{Nat}$. What can we do with this lambda abstraction? We can apply it to $1 : \mathsf{Nat} \,\&\, \mathsf{Nat}$ for example. In that case, we get two translations $(\lambda x.\, \pi_1\, x)\ \langle 1, 1 \rangle$ and $(\lambda x.\, \pi_2\, x)\ \langle 1, 1 \rangle$, which both reduce to the same numeral 1. However, $\lambda x.\, \pi_1\, x$ and $\lambda x.\, \pi_2\, x$ are definitely not equal according to Definition 2, as one can find a context $[\cdot]\ \langle 1, 2 \rangle$ where the two terms reduce to two different numerals. The problem is that not every well-typed λ_c term can be obtained from a well-typed NeColus expression through the elaboration semantics. For example, $[\cdot]\ \langle 1, 2 \rangle$ should not be considered because the (non-disjoint) source expression $1 \,,\, 2$ is rejected by the type system of the source calculus NeColus and thus never gets elaborated into $\langle 1, 2 \rangle$.

NeColus Contexts and Refined Contextual Equivalence. Example 2 hints at a shift from λ_c contexts to NeColus contexts $\mathcal{C}$, whose syntax is shown in Fig. 10. Due to the bidirectional nature of the type system, the typing judgement of $\mathcal{C}$ features 4 different forms:

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

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

We write $\mathcal{C} : (\Gamma \Leftrightarrow A) \mapsto (\Gamma' \Leftrightarrow' A') \rightsquigarrow \mathcal{D}$ to abbreviate the above 4 different forms. Take $\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A') \rightsquigarrow \mathcal{D}$ for example, it reads given a well-typed NeColus expression $\Gamma \vdash E \Rightarrow A$, we have $\Gamma' \vdash \mathcal{C}\{E\} \Rightarrow A'$. The judgement also generates a λ_c context $\mathcal{D}$ such that $\mathcal{D} : (|\Gamma| \vdash |A|) \rightsquigarrow (|\Gamma'| \vdash |A'|)$ by construction. The typing rules appear in the appendix. Now we are ready to refine Definition 2's contextual equivalence to take into consideration both NeColus and λ_c contexts.

▶ **Definition 3** (NeColus Contextual Equivalence)**.** Let $\Leftrightarrow \in \{\Rightarrow, \Leftarrow\}$, if $\Gamma \vdash E_1 \Leftrightarrow A \rightsquigarrow e_1$ and $\Gamma \vdash E_2 \Leftrightarrow A \rightsquigarrow e_2$, we write $\Gamma \vdash E_1 \precsim_{ctx} E_2 : A$ to mean

$$\forall \mathcal{C}, \Leftrightarrow' . \mathcal{C} : (\Gamma \Leftrightarrow A) \mapsto (\bullet \Leftrightarrow' \mathsf{Nat}) \rightsquigarrow \mathcal{D} \Longrightarrow \mathcal{D}\{e_1\} \precsim \mathcal{D}\{e_2\}$$

Regarding Example 2, a possible NeColus context is $[\cdot]\ (1 : \mathsf{Nat} \,\&\, \mathsf{Nat})$. According to Definition 3, a corresponding λ_c context is $[\cdot]\ \langle 1, 1\rangle$. Now both $\lambda x.\, \pi_1\, x$ and $\lambda x.\, \pi_2\, x$ produce the same numeral 1 in this context. Of course we should consider all possible contexts to be certain they are truly equal. From now on, we use the symbol $\precsim_{ctx}$ to refer to contextual equivalence in Definition 3. With Definition 3 we can formally state that NeColus is coherent in the following theorem:

▶ **Theorem 3** (Coherence)**.** *Let* $\Leftrightarrow \in \{\Rightarrow, \Leftarrow\}$*, if* $\Gamma \vdash E \Leftrightarrow A \rightsquigarrow e_1$ *and* $\Gamma \vdash E \Leftrightarrow A \rightsquigarrow e_2$*, then* $\Gamma \vdash E \precsim_{ctx} E : A$*.*

The rest of the section is devoted to proving that Theorem 3 holds.

4.2 Logical Relations

Intuitive as Definition 3 may seem, it is generally very hard to prove contextual equivalence directly, since it involves quantification over *all* possible contexts. Worse still, two kinds of contexts are involved in Theorem 3, which makes reasoning even more tedious. The key to simplifying the reasoning is to exploit types by using logical relations [56, 54, 43].

In Search of a Logical Relation. It is worth pausing to ponder what kind of relation we are looking for. The high-level intuition behind the relation is to capture the notion of "coherent" values. These values are unambiguous in every context. A moment of thought leads us to the following important observations:

▶ Observation 1 (Disjoint values are unambiguous). The relation should contain values originating from disjoint intersection types. Those values are essentially translated from merges, and since rule T-MERGE ensures disjointness, they are unambiguous. For example, $\langle 1, \{l = 1\}\rangle$ corresponds to the type $\mathsf{Nat} \,\&\, \{l : \mathsf{Nat}\}$. It is always clear which one to choose (1 or $\{l = 1\}$) no matter how this pair is used in certain contexts.

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

$$
\begin{aligned}
(v_1, v_2) \in \mathcal{V}[\![\mathsf{Nat}; \mathsf{Nat}]\!] &\triangleq \exists i, v_1 = v_2 = i \\
(v_1, v_2) \in \mathcal{V}[\![\tau_1 \to \tau_2; \tau_1' \to \tau_2']\!] &\triangleq \forall (v, v') \in \mathcal{V}[\![\tau_1; \tau_1']\!], (v_1\ v, v_2\ v') \in \mathcal{E}[\![\tau_2; \tau_2']\!] \\
(\{l = v_1\}, \{l = v_2\}) \in \mathcal{V}[\![\{l : \tau_1\}; \{l : \tau_2\}]\!] &\triangleq (v_1, v_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!] \\
(\langle v_1, v_2\rangle, v_3) \in \mathcal{V}[\![\tau_1 \times \tau_2; \tau_3]\!] &\triangleq (v_1, v_3) \in \mathcal{V}[\![\tau_1; \tau_3]\!] \wedge (v_2, v_3) \in \mathcal{V}[\![\tau_2; \tau_3]\!] \\
(v_3, \langle v_1, v_2\rangle) \in \mathcal{V}[\![\tau_3; \tau_1 \times \tau_2]\!] &\triangleq (v_3, v_1) \in \mathcal{V}[\![\tau_3; \tau_1]\!] \wedge (v_3, v_2) \in \mathcal{V}[\![\tau_3; \tau_2]\!] \\
(v_1, v_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!] &\triangleq \mathsf{true} \quad \text{otherwise}
\end{aligned}
$$

$$
(e_1, e_2) \in \mathcal{E}[\![\tau_1; \tau_2]\!] \triangleq \exists v_1\, v_2, e_1 \longrightarrow^* v_1 \wedge e_2 \longrightarrow^* v_2 \wedge (v_1, v_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!]
$$

Figure 11 Logical relations for λ_c

To formalize values being "coherent" based on the above observations, Figure 11 defines two (binary) logical relations for λ_c, one for values ($\mathcal{V}[\![\tau_1; \tau_2]\!]$) and one for terms ($\mathcal{E}[\![\tau_1; \tau_2]\!]$). We require that any two values $(v_1, v_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!]$ are closed and well-typed. For succinctness, we write $\mathcal{V}[\![\tau]\!]$ to mean $\mathcal{V}[\![\tau; \tau]\!]$, and similarly for $\mathcal{E}[\![\tau]\!]$.

▶ Remark. The logical relations are heterogeneous, parameterized by two types, one for each argument. This is intended to relate values of different types.

▶ Remark. The logical relations resemble those given by Biernacki and Polesiuk [6], as both are heterogeneous. However, two important differences are worth pointing out. Firstly, our value relation for product types ($\mathcal{V}[\![\tau_1 \times \tau_2; \tau_3]\!]$ and $\mathcal{V}[\![\tau_3; \tau_1 \times \tau_2]\!]$) is unusual. Secondly, their value relation disallows relating functions with natural numbers, while ours does not. As we explain shortly, both are related to disjointness.

First let us consider $\mathcal{V}[\![\tau_1; \tau_2]\!]$. The first three cases are standard: Two natural numbers are related iff they are the same numeral. Two functions are related iff they map related arguments to related results. Two singleton records are related iff they have the same label and their fields are related. These cases reflect Observation 2 in that the same type corresponds to the same value.

In the next two cases one of the parameterized types is a product type. In those cases, the relation distributes over the product constructor $\times$. This may look strange at first, since the traditional way of relating pairs is by relating their components pairwise. That is, $\langle v_1, v_2\rangle$ and $\langle v_1', v_2'\rangle$ are related iff (1) v_1 and v_1' are related and (2) v_2 and v_2' are related. According to our definition, we also require that (3) v_1 and v_2' are related and (4) v_2 and v_1' are related. The design of these two cases is influenced by the disjointness of intersection types. Below are two rules dealing with intersection types:

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

Notice the structural similarity between these two rules and the two cases. Now it is clear that the cases for products manifest disjointness of intersection types, reflecting Observation 1. Together with the last case, we can show that disjointness and the value relation are connected by the following lemma:

▶ **Lemma 3** (Disjoint values are in a value relation). *If $A_1 * A_2$ and $v_1 : |A_1|$ and $v_2 : |A_2|$, then $(v_1, v_2) \in \mathcal{V}[\![|A_1|; |A_2|]\!]$.*

Proof. By induction on the derivation of disjointness. ◀

Next we consider $\mathcal{E}[\![\tau_1;\tau_2]\!]$, which is standard. Informally it expresses that two closed terms e_1 and e_2 are related iff they evaluate to two values v_1 and v_2 that are related.

Logical Equivalence. The logical relations can be lifted to open terms in the usual way. First we give the semantic interpretation of typing contexts:

▶ **Definition 4** (Interpretation of Typing Contexts). $(\gamma_1,\gamma_2) \in \mathcal{G}[\![\Delta_1;\Delta_2]\!]$ is inductively defined:

$$\frac{}{(\emptyset,\emptyset) \in \mathcal{G}[\![\bullet;\bullet]\!]} \qquad \frac{(v_1,v_2) \in \mathcal{V}[\![\tau_1;\tau_2]\!] \quad (\gamma_1,\gamma_2) \in \mathcal{G}[\![\Delta_1;\Delta_2]\!] \quad \mathsf{fresh}\, x}{(\gamma_1[x \mapsto v_1],\gamma_2[x \mapsto v_2]) \in \mathcal{G}[\![\Delta_1, x:\tau_1;\Delta_2, x:\tau_2]\!]}$$

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

▶ **Definition 5** (Logical equivalence). Let $\Delta_1 \vdash e_1 : \tau_1$ and $\Delta_2 \vdash e_2 : \tau_2$.

$$\Delta_1;\Delta_2 \vdash e_1 \backsimeq_{log} e_2 : \tau_1;\tau_2 \triangleq \forall \gamma_1,\gamma_2.\ (\gamma_1,\gamma_2) \in \mathcal{G}[\![\Delta_1;\Delta_2]\!] \Longrightarrow (\gamma_1\ e_1, \gamma_2\ e_2) \in \mathcal{E}[\![\tau_1;\tau_2]\!]$$

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

4.3 Establishing Coherence

With all the machinery in place, we now show several auxiliary lemmas that are crucial in the subsequent proofs.

First we show compatibility lemmas, which state that logical equivalence is preserved by every language construct. Most are standard and thus are omitted. We show only one compatibility lemma that is specific to our relations:

▶ **Lemma 4** (Coercion Compatibility). *Suppose that* $c \vdash \tau_1 \triangleright \tau_2$,

- *If* $\Delta_1;\Delta_2 \vdash e_1 \backsimeq_{log} e_2 : \tau_1;\tau_0$ *then* $\Delta_1;\Delta_2 \vdash c\ e_1 \backsimeq_{log} e_2 : \tau_2;\tau_0$.
- *If* $\Delta_1;\Delta_2 \vdash e_1 \backsimeq_{log} e_2 : \tau_0;\tau_1$ *then* $\Delta_1;\Delta_2 \vdash e_1 \backsimeq_{log} c\ e_2 : \tau_0;\tau_2$.

Proof. By induction on the typing derivation of the coercion c. ◀

Next we show that logical equivalence is preserved by NeColus contexts:

▶ **Lemma 5** (Congruence). *If* $\mathcal{C} : (\Gamma \Leftrightarrow A) \mapsto (\Gamma' \Leftrightarrow' A') \rightsquigarrow \mathcal{D}$, $\Gamma \vdash E_1 \Leftrightarrow A \rightsquigarrow e_1$, $\Gamma \vdash E_2 \Leftrightarrow A \rightsquigarrow e_2$ *and* $|\Gamma| \vdash e_1 \backsimeq_{log} e_2 : |A|$, *then* $|\Gamma'| \vdash \mathcal{D}\{e_1\} \backsimeq_{log} \mathcal{D}\{e_2\} : |A'|$.

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

We now are in a position to establish coherence. We require Theorem 4, which is also true in λ_i.

▶ **Theorem 4** (Type-Inference Uniqueness). *If* $\Gamma \vdash E \Rightarrow A_1$ *and* $\Gamma \vdash E \Rightarrow A_2$, *then* $A_1 \equiv A_2$.

Finally we show that NeColus is coherent with respect to logical equivalence:

▶ **Lemma 6** (Coherence wrt Logical Equivalence). *Let* $\Leftrightarrow \in \{\Rightarrow, \Leftarrow\}$, *if* $\Gamma \vdash E \Leftrightarrow A \rightsquigarrow e$ *and* $\Gamma \vdash E \Leftrightarrow A \rightsquigarrow e'$, *then* $|\Gamma| \vdash e \backsimeq_{log} e' : |A|$.

Proof. The proof follows by induction on the first derivation. The most interesting case is rule T-SUB where we need Theorem 4 to be able to apply the induction hypothesis. Then we apply Lemma 4 to say that the coercion generated preserves the relation between terms. For the other cases we use the appropriate compatibility lemmas. ◀

▶ Remark. Usually, with logical relations, we need to prove the *fundamental property*, which states that if $\Delta \vdash e : \tau$ then $\Delta \vdash e \backsimeq_{log} e : \tau$. Clearly this does not hold for every well-typed λ_c term. However, as we have emphasized, we do not need to consider every λ_c term. Instead, Lemma 6 acts as a kind of fundamental property for well-typed NeColus expressions.

We restate the central theorem below:

▶ **Theorem 3** (Coherence). *Let* $\Leftrightarrow \in \{\Rightarrow, \Leftarrow\}$*, if* $\Gamma \vdash E \Leftrightarrow A \rightsquigarrow e_1$ *and* $\Gamma \vdash E \Leftrightarrow A \rightsquigarrow e_2$*, then* $\Gamma \vdash E \backsimeq_{ctx} E : A$*.*

Proof. By Lemmas 5 and 6 and the definition of logical equivalence. ◀

4.4 Some Interesting Corollaries

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

Corollary 1 says that merging a term of some type with something else does not affect its semantics. Corollaries 2 and 3 express that merges are commutative and associative, respectively. Corollary 4 states that coercions from the same types are "coherent".

▶ **Corollary 1** (Merge is Neutral). *Let* $\Leftrightarrow \in \{\Rightarrow, \Leftarrow\}$*, if* $\Gamma \vdash E_1 \Leftrightarrow A \rightsquigarrow e_1$ *and* $\Gamma \vdash E_1 \,,\, E_2 \Leftrightarrow A \rightsquigarrow e_2$*, then* $\Gamma \vdash E_1 \backsimeq_{ctx} E_1 \,,\, E_2 : A$

▶ **Corollary 2** (Merge is Commutative). *Let* $\Leftrightarrow \in \{\Rightarrow, \Leftarrow\}$*, if* $\Gamma \vdash E_1 \,,\, E_2 \Leftrightarrow A \rightsquigarrow e_1$ *and* $\Gamma \vdash E_2 \,,\, E_1 \Leftrightarrow A \rightsquigarrow e_2$*, then* $\Gamma \vdash E_1 \,,\, E_2 \backsimeq_{ctx} E_2 \,,\, E_1 : A$*.*

▶ **Corollary 3** (Merge is Associative). *Let* $\Leftrightarrow \in \{\Rightarrow, \Leftarrow\}$*, if* $\Gamma \vdash (E_1 \,,\, E_2) \,,\, E_3 \Leftrightarrow A \rightsquigarrow e_1$ *and* $\Gamma \vdash E_1 \,,\, (E_2 \,,\, E_3) \Leftrightarrow A \rightsquigarrow e_2$*, then* $\Gamma \vdash (E_1 \,,\, E_2) \,,\, E_3 \backsimeq_{ctx} E_1 \,,\, (E_2 \,,\, E_3) : A$*.*

▶ **Corollary 4** (Coercions Preserve Semantics). *If* $A <: B \rightsquigarrow c_1$ *and* $A <: B \rightsquigarrow c_2$*, then* $\lambda x.\, c_1\, x$ *and* $\lambda x.\, c_2\, x$ *can be used interchangeably.*

5 Algorithmic Subtyping

This section presents an algorithm that implements the subtyping relation in Fig. 4. While BCD subtyping is well-known, the presence of a transitivity axiom in the rules means that the system is not algorithmic. This raises an obvious question: how to obtain an algorithm for this subtyping relation? Laurent [32] has shown that simply dropping the transitivity rule from the BCD system is not possible without losing expressivity. Hence, this avenue for obtaining an algorithm is not available. Instead, we adapt Pierce's decision procedure [42] for a subtyping system (closely related to BCD) to obtain a sound and complete algorithm for our BCD extension. Our algorithm extends Pierce's decision procedure with subtyping of singleton records and coercion generation. We prove in Coq that the algorithm is sound and complete with respect to the declarative version. At the same time we find some errors and missing lemmas in Pierce's original manual proofs.

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

$$\text{A-AND}\ \frac{\mathcal{L} \vdash A \prec: B_1 \rightsquigarrow c_1 \qquad \mathcal{L} \vdash A \prec: B_2 \rightsquigarrow c_2}{\mathcal{L} \vdash A \prec: B_1 \,\&\, B_2 \rightsquigarrow [\![\mathcal{L}]\!]_{\&} \circ \langle c_1, c_2 \rangle} \qquad \text{A-ARR}\ \frac{\mathcal{L}, B_1 \vdash A \prec: B_2 \rightsquigarrow c}{\mathcal{L} \vdash A \prec: B_1 \to B_2 \rightsquigarrow c}$$

$$\text{A-RCD}\ \frac{\mathcal{L}, \{l\} \vdash A \prec: B \rightsquigarrow c}{\mathcal{L} \vdash A \prec: \{l : B\} \rightsquigarrow c} \qquad \text{A-TOP}\ \frac{}{\mathcal{L} \vdash A \prec: \top \rightsquigarrow [\![\mathcal{L}]\!]_{\top} \circ \mathsf{top}}$$

$$\text{A-ARRNAT}\ \frac{[] \vdash A \prec: A_1 \rightsquigarrow c_1 \qquad \mathcal{L} \vdash A_2 \prec: \mathsf{Nat} \rightsquigarrow c_2}{A, \mathcal{L} \vdash A_1 \to A_2 \prec: \mathsf{Nat} \rightsquigarrow c_1 \to c_2} \qquad \text{A-RCDNAT}\ \frac{\mathcal{L} \vdash A \prec: \mathsf{Nat} \rightsquigarrow c}{\{l\}, \mathcal{L} \vdash \{l : A\} \prec: \mathsf{Nat} \rightsquigarrow \{l : c\}}$$

$$\text{A-ANDN1}\ \frac{\mathcal{L} \vdash A_1 \prec: \mathsf{Nat} \rightsquigarrow c}{\mathcal{L} \vdash A_1 \,\&\, A_2 \prec: \mathsf{Nat} \rightsquigarrow c \circ \pi_1} \qquad \text{A-ANDN2}\ \frac{\mathcal{L} \vdash A_2 \prec: \mathsf{Nat} \rightsquigarrow c}{\mathcal{L} \vdash A_1 \,\&\, A_2 \prec: \mathsf{Nat} \rightsquigarrow c \circ \pi_2} \qquad \text{A-NAT}\ \frac{}{[] \vdash \mathsf{Nat} \prec: \mathsf{Nat} \rightsquigarrow \mathsf{id}}$$

Figure 12 Algorithmic subtyping of NeColus

5.1 The Subtyping Algorithm

Figure 12 shows the algorithmic subtyping judgement $\mathcal{L} \vdash A \prec: B \rightsquigarrow c$. This judgement is the algorithmic counterpart of the declarative judgement $A <: \mathcal{L} \to B \rightsquigarrow c$, where the symbol $\mathcal{L}$ stands for a queue of types and labels. Definition 6 converts a queue to a type:

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

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

For instance, if $\mathcal{L} = A, B, \{l\}$, then $\mathcal{L} \to C$ abbreviates $A \to B \to \{l : C\}$.

The basic idea of $\mathcal{L} \vdash A \prec: B \rightsquigarrow c$ is to first perform a structural analysis of B, which descends into both sides of $\&$'s (rule A-AND), into the right side of $\to$'s (rule A-ARR), and into the fields of records (rule A-RCD) until it reaches one of the two base cases, Nat or $\top$. If the base case is $\top$, then the subtyping holds trivially (rule A-TOP). If the base case is Nat, the algorithm performs a structural analysis of A, in which $\mathcal{L}$ plays an important role. The left sides of $\to$'s are pushed onto $\mathcal{L}$ as they are encountered in B and popped off again later, left to right, as $\to$'s are encountered in A (rule A-ARRNAT). Similarly, the labels are pushed onto $\mathcal{L}$ as they are encountered in B and popped off again later, left to right, as records are encountered in A (rule A-RCDNAT). The remaining rules are similar to their declarative counterparts. To get the intuition of how the algorithm works, below we give an example derivation (for space reasons we use N and S to denote Nat and String respectively), which is essentially the one used by the `add` field in Section 2. The readers can try to give a corresponding derivation using the declarative subtyping and see how rule S-TRANS plays an essential role there.

$$\dfrac{\dfrac{\dfrac{D \qquad D'}{\{l\}, \mathsf{N}\,\&\,\mathsf{S}, \mathsf{N}\,\&\,\mathsf{S} \vdash \{l : \mathsf{N} \to \mathsf{N} \to \mathsf{N}\}\,\&\,\{l : \mathsf{S} \to \mathsf{S} \to \mathsf{S}\} \prec: \mathsf{N}\,\&\,\mathsf{S}}\ \text{A-AND}}{\{l\} \vdash \{l : \mathsf{N} \to \mathsf{N} \to \mathsf{N}\}\,\&\,\{l : \mathsf{S} \to \mathsf{S} \to \mathsf{S}\} \prec: \mathsf{N}\,\&\,\mathsf{S} \to \mathsf{N}\,\&\,\mathsf{S} \to \mathsf{N}\,\&\,\mathsf{S}}\ \text{A-ARR}(\textit{twice})}{\{l : \mathsf{N} \to \mathsf{N} \to \mathsf{N}\}\,\&\,\{l : \mathsf{S} \to \mathsf{S} \to \mathsf{S}\} \prec: \{l : \mathsf{N}\,\&\,\mathsf{S} \to \mathsf{N}\,\&\,\mathsf{S} \to \mathsf{N}\,\&\,\mathsf{S}\}}\ \text{A-RCD}$$

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

Figure 13 Meta-functions of coercions

where the sub-derivation D is shown below (D' is similar):

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\mathsf{N} \,\&\, \mathsf{S} \prec: \mathsf{N}} \quad \dfrac{\dots}{\mathsf{N} \,\&\, \mathsf{S} \vdash \mathsf{N} \to \mathsf{N} \prec: \mathsf{N}}}{\mathsf{N} \,\&\, \mathsf{S}, \mathsf{N} \,\&\, \mathsf{S} \vdash \mathsf{N} \to \mathsf{N} \to \mathsf{N} \prec: \mathsf{N}}\ \text{A-ARRNAT}}{\{l\}, \mathsf{N} \,\&\, \mathsf{S}, \mathsf{N} \,\&\, \mathsf{S} \vdash \{l : \mathsf{N} \to \mathsf{N} \to \mathsf{N}\} \prec: \mathsf{N}}\ \text{A-RCDNAT}}{\{l\}, \mathsf{N} \,\&\, \mathsf{S}, \mathsf{N} \,\&\, \mathsf{S} \vdash \{l : \mathsf{N} \to \mathsf{N} \to \mathsf{N}\} \,\&\, \{l : \mathsf{S} \to \mathsf{S} \to \mathsf{S}\} \prec: \mathsf{N}}\ \text{A-ANDN1}
$$

Now consider the coercions. Algorithmic subtyping uses the same set of coercions as declarative subtyping. However, because algorithmic subtyping has a different structure, the rules generate slightly more complicated coercions. Two meta-functions $[\![\cdot]\!]_\top$ and $[\![\cdot]\!]_\&$ used in rules A-TOP and A-AND respectively, are meant to generate correct forms of coercions. They are defined recursively on $\mathcal{L}$ and are shown in Fig. 13.

5.2 Correctness of the Algorithm

To establish the correctness of the algorithm, we must show that the algorithm is both sound and complete with respect to the declarative specification. While soundness follows quite easily, completeness is much harder. The proof of completeness essentially follows that of Pierce [42] in that we need to show the algorithmic subtyping is reflexive and transitive.

5.2.1 Soundness of the Algorithm.

The following two lemmas connect the declarative subtyping with the meta-functions.

▶ **Lemma 7.** $\top <: \mathcal{L} \to \top \rightsquigarrow [\![\mathcal{L}]\!]_\top$

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

▶ **Lemma 8.** $(\mathcal{L} \to A) \,\&\, (\mathcal{L} \to B) <: \mathcal{L} \to (A \,\&\, B) \rightsquigarrow [\![\mathcal{L}]\!]_\&$

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

The proof of soundness is straightforward.

▶ **Theorem 5** (Soundness). *If* $\mathcal{L} \vdash A \prec: B \rightsquigarrow c$ *then* $A <: \mathcal{L} \to B \rightsquigarrow c$.

Proof. By induction on the derivation of the algorithmic subtyping and applying Lemmas 7 and 8 where appropriate. ◀

5.2.2 Completeness of the Algorithm.

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

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

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

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

▶ **Lemma 9.** $A \in \mathcal{U}(A)$

Proof. By induction on the structure of A. ◀

▶ **Lemma 10.** *If $A \in \mathcal{U}(B)$ and $B \in \mathcal{U}(C)$, then $A \in \mathcal{U}(C)$.*

Proof. By induction on the structure of B. ◀

▶ Remark. Lemma 10 is not found in [42], which is crucial in Lemma 11, from which reflexivity (Lemma 12) follows immediately.

▶ **Lemma 11.** *If $\mathcal{L} \to B \in \mathcal{U}(A)$ then there exists c such that $\mathcal{L} \vdash A \prec: B \rightsquigarrow c$.*

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

Now it immediately follows that the algorithmic subtyping is reflexive.

▶ **Lemma 12** (Reflexivity)**.** *For every A there exists c such that $[] \vdash A \prec: A \rightsquigarrow c$.*

Proof. Immediate from Lemmas 9 and 11. ◀

We omit the details of the proof of transitivity.

▶ **Lemma 13** (Transitivity)**.** *If $[] \vdash A_1 \prec: A_2 \rightsquigarrow c_1$ and $[] \vdash A_2 \prec: A_3 \rightsquigarrow c_2$, then there exists c such that $[] \vdash A_1 \prec: A_3 \rightsquigarrow c$.*

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

▶ **Theorem 6** (Completeness)**.** *If $A <: B \rightsquigarrow c$ then there exists c' such that $[] \vdash A \prec: B \rightsquigarrow c'$.*

Proof. By induction on the derivation of the declarative subtyping and applying Lemmas 12 and 13 where appropriate. ◀

▶ Remark. Pierce's proof is wrong [42, pp. 20, Case F] in the case

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

where he directly concludes from the inductive hypothesis $[] \vdash B_1 \prec: A_1$ and $[] \vdash A_2 \prec: B_2$ that $[] \vdash A_1 \to A_2 \prec: B_1 \to B_2$ holds, which is clearly wrong since no algorithmic rules apply here. It turns out we need a few technical lemmas to support the argument, which are omitted for reasons of space.

▶ Remark. It is worth pointing out that the two coercions c and c' in Theorem 6 are contextually equivalent, which follows from Theorem 5 and Corollary 4.

6 Related Work

Coherence. In calculi that feature coercive subtyping, a semantics that interprets the subtyping judgement by introducing explicit coercions is typically defined on typing derivations rather than on typing judgements. A natural question that arises for such systems is whether the semantics is *coherent*, i.e., distinct typing derivations of the same typing judgement possess the same meaning. Since Reynolds [49] proved the coherence of a calculus with intersection types, based on the denotational semantics for intersection types, many researchers have studied the problem of coherence in a variety of typed calculi. Below we summarize two commonly-found approaches in the literature.

The first approach is based on normalization. Breazu-Tannen et al. [8] proved the coherence of the language Fun [11] extended with recursive types, by translating Fun into System F and showing that any two typing derivations of the same typing judgement are normalizable to a unique normal derivation. Ghelli [18] presented a translation of System $F_{\leq}$ into a calculus with explicit coercions and showed that any derivations of the same judgement are translated to terms that are normalizable to a unique normal form. Following the same approach, Schwinghammer [53] proved the coherence of coercion translation from Moggi's computational lambda calculus [35] with subtyping.

Central to the first approach is to find a normal form for a representation of the derivation and show that normal forms are unique for a given typing judgement. However, this approach cannot be directly applied to Curry-style calculi, i.e, where the lambda abstractions are not type annotated. Also this line of reasoning cannot be used when the source language has general recursion. Instead, Biernacki and Polesiuk [6] considered the coherence problem based on contextual equivalence, where they presented a construction of logical relations for establishing coherence. They showed that this approach is applicable in a variety of calculi, including delimited continuations and control-effect subtyping.

As far as we know, our work is the first to use logical relations to show the coherence for intersection types and the merge operator. The BCD subtyping in our setting poses a non-trivial complication over Biernacki and Polesiuk's simple structural subtyping. Indeed, because any two coercions between given types are behaviorally equivalent in the target language, their coherence reasoning can all take place in the target language. This is not true in our setting, where coercions can be distinguished by arbitrary target programs, but not those that are elaborations of source programs. Hence, we have to restrict our reasoning to the latter class, which is reflected in a more complicated notion of contextual equivalence and our logical relation's non-trivial treatment of pairs.

Intersection Types and the Merge Operator. Forsythe has intersection types and a merge operator, and was proven to be coherent [49]. However to ensure coherence, various restrictions were added to limit the use of merges. For example, in Forsythe merges cannot contain more than one function. Castagna et al. [13] proposed a coherent calculus with a special merge operator that works on functions only. More recently, Dunfield [21] shows significant expressiveness of type systems with intersection types and a merge construct. However his calculus lacks coherence. The limitation was addressed by Oliveira et al. [41], where they introduced disjointness to ensure coherence. The combination of intersection types, a merge construct and parametric polymorphism, while achieving coherence was first studied in the F_i calculus [1], where they proposed the notion of disjoint polymorphism. Compared to prior work, NeColus simplifies type systems with disjoint intersection types by removing several restrictions. Furthermore, NeColus adopts a much more powerful subtyping relation based on BCD subtyping, which in turn requires the use of a more powerful logical relations based

method for proving coherence.

BCD Type System and Decidability. The BCD type system was first introduced by Barendregt et al. [3]. It is derived from a filter lambda model in order to characterize exactly the strongly normalizing terms. The BCD type system features a powerful subtyping relation, which serves as a base for our subtyping relation. Bessai el at. [4] showed how to type classes and mixins in a BCD-style record calculus with Bracha-Cook's merge operator [7]. Their merge can only operate on records, and they only study a type assignment system. The decidability of BCD subtyping has been shown in several works [42, 30, 46, 55]. Laurent [31] has formalized the relation in Coq in order to eliminate transitivity cuts from it, but his formalization does not deliver an algorithm. Based on Statman's work [55], Bessai et al. [5] show a formally verified subtyping algorithm in Coq. Our Coq formalization follows a different idea based on Pierce's decision procedure [42], which is shown to be easily extensible to coercions and records. In the course of our mechanization we identified several mistakes in Pierce's proofs, as well as some important missing lemmas.

Family Polymorphism. There has been much work on family polymorphism since Ernst's original proposal [23]. Family polymorphism provides an elegant solution to the Expression Problem. Although a simple Scala solution does exist without requiring family polymorphism (e.g., see Wang and Oliveira [58]), Scala does not support nested composition: programmers need to manually compose all the classes from multiple extensions. Broadly speaking, systems that support family polymorphism can be divided into two categories: those that support *object families* and those that support *class families*. In object families, classes are nested in objects, whereas in class families, classes are nested in other classes. The former choice is considered more expressive [25], but requires a complex type system usually involving dependent types.

Virtual classes [33] are one means for achieving family polymorphism. Virtual classes are attributes of an object, and virtual classes from different instances are not compatible. The original design of virtual classes in BETA is not statically safe. To ensure type safety, virtual classes were formalized using path-dependent types in the calculus *vc* [25]. Due to the introduction of dependent types, their system is rather complex. Subtyping in *vc* is more restrictive, compared to NeColus, in that there is no subtyping relationship between classes in the base family and classes in the derived family, nor is there between the base family and the derived family. As for conflicts, *vc* follows the mixin-style by allowing the rightmost class to take precedence. This is in contrast to NeColus where conflicts are detected statically and resolved explicitly. Tribe [14] is another language that provides a variant of virtual classes, with support for both class and object families.

Jx [37] supports a notion of *nested inheritance*, which is a limited form of family polymorphism. Nested inheritance works much like class families with support for nesting of arbitrary depth. Unlike virtual classes, subclass and subtype relationships are preserved by inheritance: the overriding class is also a subtype of the class it overrides. J& [38] is a language that supports *nested intersection*, building on top of Jx. Similar to NeColus, intersection types play an important role in J&, which are used to compose packages/classes. Unlike NeColus, J& does not have a merge-like operator, and requires special treatment (e.g., prefix types) to deal with conflicts. Saito et al. [51] identified a minimal, lightweight set of language features to enable family polymorphism, though inheritance relations between nested classes are not preserved by extension. Corradi et al. [17] present a language design that integrates modular composition and nesting of Java-like classes. It features a set of composition operators that allow to manipulate nested classes at any depth level. Compared with those systems, which usually focus on getting a relatively complex Java-like language

with family polymorphism, NeColus focuses on a minimal calculus that supports nested composition. NeColus shows that a calculus with the merge operator and a variant of BCD captures the essence of nested composition. Moreover NeColus enables new insights on the subtyping relations of families. NeColus's goal is not to support full family polymorphism which, besides nested composition, also requires dealing with other features such as self types [10, 50] and mutable state. Supporting these features is not the focus of this paper, but we expect to investigate those features in the future.

7 Conclusions and Future Work

We have proposed NeColus, a type-safe and coherent calculus with disjoint intersection types, and support for nested composition/subtyping. NeColus improves upon earlier work with a more flexible notion of disjoint intersection types, which leads to a clean and elegant formulation of the type system. Due to the added flexibility we have had to employ a more powerful proof method based on logical relations to rigorously prove coherence. We also show how NeColus supports essential features of family polymorphism, such as nested composition. We believe NeColus provides insights into family polymorphism, and has potential for practical applications for extensible software designs.

A natural direction for future work is to enrich NeColus with parametric polymorphism. There is abundant literature on logical relations for parametric polymorphism [47] and we foresee no fundamental difficulties in extending our proof method.[4] The resulting calculus will be more expressive than F_i. An interesting application that we intend to investigate is native support for *object algebras* [40] (or the finally tagless approach [12]). For example, we can define the object algebra interfaces for the Expression Problem example in Section 2 as follows:

```
type ExpAlg[E] = { lit : Int → E, add : E → E → E };
type ExpAlgExt[E] = ExpAlg[E] & { sub : E → E → E };
```

By instantiating `E` with `IPrint`, i.e., `ExpAlg[IPrint]`, we get the interface of the `Lang` family. In that sense, object algebra interfaces can be viewed as family interfaces. Moreover, combing algebras implementing `ExpAlg[IPrint]` and `ExpAlg[IEval]` to form `ExpAlg[IPrint & IEval]` is trivial with nested composition. Polymorphism also improves code reuse across expressions in the base and extended languages. For example, the following creates two expressions, one in the base language, the other in the extended language:

```
e1 E (f : ExpAlg[E]) : E = f.add (f.lit 2) (f.lit 3);     -- 2 + 3
e2 E (f : ExpAlgExt[E]) : E = f.sub (f.lit 5) (e1 E f);   -- 5 - (2 + 3)
```

Notice how we can reuse `e1` of the base language in the definition of `e2`.

References

1 João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. Disjoint polymorphism. In *ESOP*, 2017.

2 Nada Amin, Adriaan Moors, and Martin Odersky. Dependent object types. In *FOOL*, 2012.

[4] Our prototype implementation already supports polymorphism, but we are still in the process of extending our Coq development with polymorphism.

3 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *JSL*, 48(04):931–940, 1983.
4 Jan Bessai, Boris Düdder, Andrej Dudenhefner, Tzu-Chun Chen, and Ugo de'Liguoro. Typing classes and mixins with intersection types. In *Workshop on Intersection Types and Related Systems (ITRS)*, 2014.
5 Jan Bessai, Andrej Dudenhefner, Boris Düdder, and Jakob Rehof. Extracting a formally verified subtyping algorithm for intersection types from ideals and filters. In *TYPES*, 2016.
6 Dariusz Biernacki and Piotr Polesiuk. Logical relations for coherence of effect subtyping. In *LIPIcs*, 2015.
7 Gilad Bracha and William R. Cook. Mixin-based inheritance. In *OOPSLA*, 1990.
8 Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, 1991.
9 Kim Bruce, Luca Cardelli, Giuseppe Castagna, Gary T Leavens, Benjamin Pierce, et al. On binary methods. In *TAPOS*, 1996.
10 Kim B. Bruce, Angela Schuett, and Robert van Gent. Polytoil: A type-safe polymorphic object-oriented language. In *ECOOP*, 1995.
11 Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, 1985.
12 Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *JFP*, 19(05):509, 2009.
13 Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. In *LFP*, 1992.
14 David Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: More Types for Virtual Classes. In *AOSD*, 2007.
15 Adriana B Compagnoni and Benjamin C Pierce. Higher-order intersection types and multiple inheritance. *MSCS*, 6(5):469–501, 1996.
16 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 1981.
17 Andrea Corradi, Marco Servetto, and Elena Zucca. DeepFJig — modular composition of nested classes. *The Journal of Object Technology*, 11(2):1:1, 2012.
18 Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in f $\leq$. *MSCS*, 2(01):55, 1992.
19 Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *ICFP*, 2000.
20 Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P Black. Traits: A mechanism for fine-grained reuse. *TOPLAS*, 28(2):331–388, 2006.
21 Joshua Dunfield. Elaborating intersection and union types. *JFP*, 24:133–165, 2014.
22 Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In *FoSSaCS*, 2003.
23 Erik Ernst. Family polymorphism. In *ECOOP*, 2001.
24 Erik Ernst. Higher-order hierarchies. In *ECOOP*, 2003.
25 Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *POPL*, 2006.
26 Facebook. Flow. `https://flow.org/`, 2014.
27 Kathleen Fisher and John Reppy. A typed calculus of traits. In *FOOL*, 2004.
28 Tim Freeman and Frank Pfenning. Refinement types for ML. In *PLDI*, 1991.
29 Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016.
30 Toshihiko Kurata and Masako Takahashi. Decidable properties of intersection type systems. *Typed Lambda Calculi and Applications*, pages 297–311, 1995.

31 Olivier Laurent. Intersection types with subtyping by means of cut elimination. *Fundamenta Informaticae*, 121(1-4):203–226, 2012.

32 Olivier Laurent. A syntactic introduction to intersection types. Unpublished note, 2012.

33 O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA*, 1989.

34 Microsoft. Typescript. `https://www.typescriptlang.org/`, 2012.

35 Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

36 James Hiram Morris Jr. *Lambda-calculus models of programming languages*. PhD thesis, Massachusetts Institute of Technology, 1969.

37 Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA*, 2004.

38 Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested Intersection for Scalable Software Composition. In *OOPSLA*, 2006.

39 Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, EPFL, 2004.

40 Bruno C. d. S. Oliveira and William R. Cook. Extensibility for the masses. In *ECOOP*, 2012.

41 Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. Disjoint intersection types. In *ICFP*, 2016.

42 Benjamin C Pierce. A decision procedure for the subtype relation on intersection types with bounded variables. Technical report, Carnegie Mellon University, 1989.

43 Gordon Plotkin. *Lambda-definability and logical relations*. Edinburgh University, 1973.

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

45 Redhat. Ceylon. `https://ceylon-lang.org/`, 2011.

46 Jakob Rehof and Paweł Urzyczyn. Finite combinatory logic with intersection types. In *International Conference on Typed Lambda Calculi and Applications*, 2011.

47 John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP*, 1983.

48 John C Reynolds. Preliminary design of the programming language forsythe. Technical report, Carnegie Mellon University, 1988.

49 John C. Reynolds. The coherence of languages with intersection types. In *LNCS*, pages 675–700. Springer Berlin Heidelberg, 1991.

50 Chieri Saito and Atsushi Igarashi. Matching *ThisType* to subtyping. In *Symposium on Applied Computing (SAC)*, 2009.

51 Chieri Saito, Atsushi Igarashi, and Mirko Viroli. Lightweight family polymorphism. *JFP*, 18(03), 2007.

52 Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. Traits: Composable units of behaviour. In *ECOOP*, 2003.

53 Jan Schwinghammer. Coherence of subsumption for monadic types. *JFP*, 19(02):157, 2008.

54 Richard Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65(2-3):85–97, 1985.

55 Rick Statman. A finite model property for intersection types. *Electronic Proceedings in Theoretical Computer Science*, 177:1–9, 2015.

56 W. W. Tait. Intensional interpretations of functionals of finite type i. *JSL*, 32(2):198–212, 1967.

57 Philip Wadler. The expression problem. *Java-genericity mailing list*, 1998.

58 Yanlin Wang and Bruno C d S Oliveira. The expression problem, trivially! In *Proceedings of the 15th International Conference on Modularity*, 2016.