The Essence of Nested Composition

Xuan Bi[1]
The University of Hong Kong, Hong Kong, China
xbi@cs.hku.hk

Bruno C. d. S. Oliveira[1]
The University of Hong Kong, Hong Kong, China
bruno@cs.hku.hk

Tom Schrijvers[2]
KU Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

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.

2012 ACM Subject Classification Software and its engineering → Object oriented languages

Keywords and phrases nested composition, family polymorphism, intersection types, coherence

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.22

Acknowledgements We thank the anonymous reviewers for their helpful comments.

1 Introduction

Intersection types [49, 18] 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 [54], they have also been employed to express useful programming language constructs, such as key aspects of *multiple inheritance* [17] in Object-Oriented Programming (OOP). One notable example is the Scala language [44] and its DOT calculus [3], which make fundamental use of intersection types to express a class/trait

[1] Funded by Hong Kong Research Grant Council projects number 17210617 and 17258816
[2] Funded by The Research Foundation - Flanders

32nd European Conference on Object-Oriented Programming (ECOOP 2018).
Editor: Todd Millstein; Article No. 22; pp. 22:1–22:33
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

that extends multiple other traits. Other modern languages, such as TypeScript [39], Flow [28] and Ceylon [51], 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 [54] and adopted by a few other calculi [15, 23, 46, 2]. 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* [30, 21, 24]. 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 [23] 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 recent calculus with a variant of intersection types called *disjoint intersection types* [46]. 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 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. [2], 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 [9, 58]. 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 [29, 22]. 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* [25]. 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 [64]. An essential feature of family polymorphism is *nested composition* [19, 27, 42], 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 [27, 42, 16, 57].

NeColus. This paper presents an improved variant of λ_i called NeColus[3] (or λ_i^+): a simple calculus with records and disjoint intersection types that supports *nested composition*. Nested

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

composition enables encoding simple forms of family polymorphism. More complex forms of family polymorphism, involving binary methods [11] 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 [25] 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. [4] (BCD). However the distributivity rule is not common in calculi or languages with intersection types aimed at programming. For example the rules employed in languages that support intersection types (such as Scala, TypeScript, Flow or Ceylon) lack distributivity rules. Moreover distributivity is also missing from several calculi with a merge operator. This includes all calculi with disjoint intersection types 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* [63, 48, 61]. 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 [47].
- 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 [47] 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.[4]

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. [42] 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* [42]. 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 [25] illustrates the benefits of nested inheritance for modularity and extensibility with one of the most elegant and concise solutions to the *Expression Problem* [64]. 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`.

[4] The Coq formalization and implementation are available at `https://goo.gl/R5hUAp`.

```
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

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.

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 compared to gbeta is that many more NeColus types are related through subtyping. Indeed, gbeta is unnecessarily conservative [26]: 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. Admittedly, the figure looks quite complex because our calculus features a structural type system (as often more foundational calculi do), whereas mainstream OO languages have nominal type systems. This is part of the reason why we have so many subtyping relations in Fig. 2.

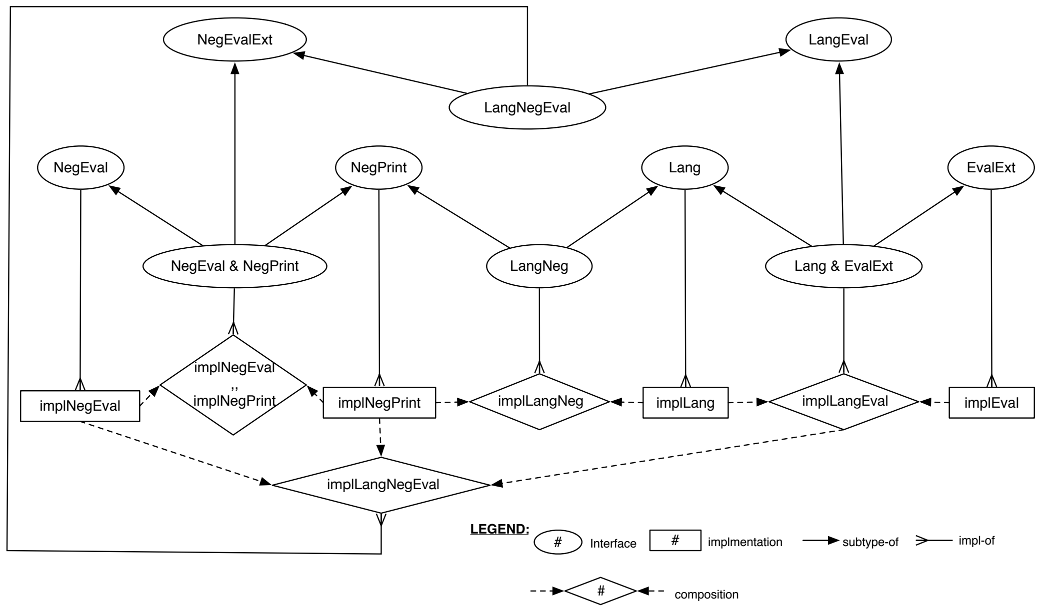

Figure 2 Summary diagram of the relationships between language components

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.

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 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 $\mathsf{Nat} \,\&\, \mathsf{Nat}$ 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. Compared to prior work [2, 46], NeColus has a more powerful subtyping relation. The new subtyping relation is inspired by BCD-style subtyping, but with two noteworthy differences: subtyping is coercive (in contrast to traditional formulations of BCD); and it is extended with records. We also have a new target language with explicit coercions inspired by the coercion calculus of Henglein [32]. A full technical comparison between λ_i^+ and λ_i can be found in Section 3.5.

3.1 Syntax

Figure 3 shows the syntax of NeColus. 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. Types include naturals Nat, a top type $\top$, function types $A \to B$, intersection types $A \,\&\, B$, and singleton record types $\{l : A\}$. Metavariable E ranges over expressions. Expressions include variables x, natural numbers i, a canonical top value $\top$, lambda abstractions $\lambda x.\, E$,

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

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

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

$$\text{S-RCD}\quad \frac{A <: B \rightsquigarrow c}{\{l : A\} <: \{l : B\} \rightsquigarrow \{l : c\}} \qquad \text{S-ARR}\quad \frac{B_1 <: A_1 \rightsquigarrow c_1 \qquad A_2 <: B_2 \rightsquigarrow c_2}{A_1 \to A_2 <: B_1 \to B_2 \rightsquigarrow c_1 \to c_2} \qquad \text{S-ANDL}\quad \frac{}{A_1 \,\&\, A_2 <: A_1 \rightsquigarrow \pi_1}$$

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

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

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

Figure 4 Declarative specification of subtyping

applications $E_1\, E_2$, merges $E_1 \,,,\, E_2$, annotated terms $E : A$, singleton records $\{l = E\}$, and record selections $E.l$.

3.2 Declarative Subtyping

Figure 4 presents the subtyping relation. We ignore the highlighted parts, and explain them later in Section 3.4.

BCD-Style Subtyping. The subtyping rules are essentially those of the BCD type system [4], extended with subtyping for singleton records. Rules S-TOP and S-RCD for top types and record types are straightforward. Rule S-ARR for function subtyping is standard. Rules S-ANDL, S-ANDR, and S-AND for intersection types axiomatize that $A \,\&\, B$ is the greatest lower bound of A and B. Rule S-DISTARR is perhaps the most interesting rule. This, so-called "distributivity" rule, describes the interaction between the subtyping relations for function types and those for intersection types. It can be shown[5] that the other direction $(A_1 \to A_2 \,\&\, A_3 <: (A_1 \to A_2) \,\&\, (A_1 \to A_3))$ and the contravariant distribution $((A_1 \to A_2) \,\&\, (A_3 \to A_2) <: A_1 \,\&\, A_3 \to A_2)$ are both derivable from the existing subtyping rules.

[5] The full derivations are found in the appendix.

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

$$\text{T-TOP}\ \frac{}{\Gamma \vdash \top \Rightarrow \top \rightsquigarrow \langle\rangle} \qquad \text{T-LIT}\ \frac{}{\Gamma \vdash i \Rightarrow \mathsf{Nat} \rightsquigarrow i} \qquad \text{T-VAR}\ \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A \rightsquigarrow x}$$

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

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

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

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

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

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

Figure 5 Bidirectional type system of NeColus

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.

3.3 Typing of NeColus

The bidirectional type system for NeColus is shown in Fig. 5. Again we ignore the highlighted parts for now.

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 intuition of two types being disjoint is that their least upper bound is (isomorphic to) $\top$. 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 [46, 2].

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

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

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

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

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

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 is a distinct difference from the prior works (in which they are regular terms). Our coercions are based on those of Henglein [32], and we add more forms due to our extra subtyping rules. Metavariable c ranges over coercions.[6] Coercions express the conversion of a term from one type to another. Because of the addition of coercions, the grammar contains explicit coercion applications $c\, e$ as a term, and various unsaturated coercion applications as values. The use of explicit coercions is useful for the new semantic proof of coherence based on logical relations. The subtyping 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.

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

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

$$\frac{}{\mathsf{id} \vdash \tau \rhd \tau}\ \textsc{CoTyp-Refl} \qquad \frac{c_1 \vdash \tau_2 \rhd \tau_3 \qquad c_2 \vdash \tau_1 \rhd \tau_2}{c_1 \circ c_2 \vdash \tau_1 \rhd \tau_3}\ \textsc{CoTyp-Trans}$$

$$\frac{}{\mathsf{top} \vdash \tau \rhd \langle\rangle}\ \textsc{CoTyp-Top} \qquad \frac{}{\mathsf{top}_{\rightarrow} \vdash \langle\rangle \rhd \langle\rangle \rightarrow \langle\rangle}\ \textsc{CoTyp-TopArr}$$

$$\frac{}{\mathsf{top}_{\{l\}} \vdash \langle\rangle \rhd \{l : \langle\rangle\}}\ \textsc{CoTyp-TopRcd} \qquad \frac{c_1 \vdash \tau_1' \rhd \tau_1 \qquad c_2 \vdash \tau_2 \rhd \tau_2'}{c_1 \rightarrow c_2 \vdash \tau_1 \rightarrow \tau_2 \rhd \tau_1' \rightarrow \tau_2'}\ \textsc{CoTyp-Arr}$$

$$\frac{c_1 \vdash \tau_1 \rhd \tau_2 \qquad c_2 \vdash \tau_1 \rhd \tau_3}{\langle c_1, c_2\rangle \vdash \tau_1 \rhd \tau_2 \times \tau_3}\ \textsc{CoTyp-Pair}$$

$$\frac{}{\pi_1 \vdash \tau_1 \times \tau_2 \rhd \tau_1}\ \textsc{CoTyp-ProjL} \qquad \frac{}{\pi_2 \vdash \tau_1 \times \tau_2 \rhd \tau_2}\ \textsc{CoTyp-ProjR} \qquad \frac{c \vdash \tau_1 \rhd \tau_2}{\{l : c\} \vdash \{l : \tau_1\} \rhd \{l : \tau_2\}}\ \textsc{CoTyp-Rcd}$$

$$\frac{}{\mathsf{dist}_{\{l\}} \vdash \{l : \tau_1\} \times \{l : \tau_2\} \rhd \{l : \tau_1 \times \tau_2\}}\ \textsc{CoTyp-DistRcd} \qquad \frac{}{\mathsf{dist}_{\rightarrow} \vdash (\tau_1 \rightarrow \tau_2) \times (\tau_1 \rightarrow \tau_3) \rhd \tau_1 \rightarrow \tau_2 \times \tau_3}\ \textsc{CoTyp-DistArr}$$

Figure 8 Coercion typing

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:

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

Here the judgement $c \vdash \tau_1 \rhd \tau_2$ expresses the typing of coercions, which are essentially functions from τ_1 to τ_2. Their typing rules correspond exactly to the subtyping rules of 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. Figure 9 shows the single-step ($\longrightarrow$) reduction rules for coercions. Our coercion reduction rules are quite standard but not efficient in terms of space. Nevertheless, there is existing work on space-efficient coercions [60, 33], which should be applicable to our work as well. 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 $\exists e'$ such that $e \longrightarrow e'$.*

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

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

▶ **Lemma 4** (Elaboration soundness). *We have that:*

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

$$\frac{}{\mathsf{id}\; v \longrightarrow v}\;\text{STEP-ID} \qquad \frac{}{(c_1 \circ c_2)\; v \longrightarrow c_1\,(c_2\; v)}\;\text{STEP-TRANS} \qquad \frac{}{\mathsf{top}\; v \longrightarrow \langle\rangle}\;\text{STEP-TOP} \qquad \frac{}{(\mathsf{top}_{\rightarrow}\,\langle\rangle)\,\langle\rangle \longrightarrow \langle\rangle}\;\text{STEP-TOPARR}$$

$$\frac{}{\mathsf{top}_{\{l\}}\,\langle\rangle \longrightarrow \{l = \langle\rangle\}}\;\text{STEP-TOPRCD} \qquad \frac{}{\langle c_1, c_2\rangle\; v \longrightarrow \langle c_1\; v, c_2\; v\rangle}\;\text{STEP-PAIR} \qquad \frac{}{((c_1 \rightarrow c_2)\; v_1)\; v_2 \longrightarrow c_2\,(v_1\,(c_1\; v_2))}\;\text{STEP-ARR}$$

$$\frac{}{(\mathsf{dist}_{\rightarrow}\,\langle v_1, v_2\rangle)\; v_3 \longrightarrow \langle v_1\; v_3, v_2\; v_3\rangle}\;\text{STEP-DISTARR} \qquad \frac{}{\pi_1\,\langle v_1, v_2\rangle \longrightarrow v_1}\;\text{STEP-PROJL} \qquad \frac{}{\pi_2\,\langle v_1, v_2\rangle \longrightarrow v_2}\;\text{STEP-PROJR}$$

$$\frac{}{\{l : c\}\,\{l = v\} \longrightarrow \{l = c\; v\}}\;\text{STEP-CRCD} \qquad \frac{}{\mathsf{dist}_{\{l\}}\,\langle\{l = v_1\}, \{l = v_2\}\rangle \longrightarrow \{l = \langle v_1, v_2\rangle\}}\;\text{STEP-DISTRCD}$$

Figure 9 Coercion reduction

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

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

3.5 Comparison with λ_i

Below we identify major differences between λ_i^+ and λ_i, which, when taken together, yield a simpler and more elegant system. The differences may seem superficial, but they have far-reaching impacts on the semantics, especially on coherence, our major topic in Section 4.

No Ordinary Types. Apart from the extra subtyping rules, there is an important difference from the λ_i subtyping relation. The subtyping relation of λ_i employs an auxiliary unary relation called ordinary, which plays a fundamental role for ensuring coherence and obtaining an algorithm [21]. The NeColus calculus discards the notion of ordinary types completely; this yields a clean and elegant formulation of the subtyping relation. Another minor difference is that due to the addition of the transitivity axiom (rule S-TRANS), rules S-ANDL and S-ANDR are simplified: an intersection type $A \,\&\, B$ is a subtype of both A and B, instead of the more general form $A \,\&\, B <: C$.

No Top-Like Types. There is a notable difference from the coercive subtyping of λ_i. Because of their syntactic proof method, they have special treatment for coercions of *top-like types* in order to retain coherence. For 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 (cf. Section 4).

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

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

type system. The NeColus calculus discards this well-formedness judgement altogether in favour of a simpler design that is still coherent. An important implication is that even without adding BCD subtyping, NeColus is already more expressive than λ_i: an expression such as $1 : \mathsf{Nat} \,\&\, \mathsf{Nat}$ 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.

4 Coherence

This section constructs logical relations to establish the coherence of NeColus. 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.

If we permit ourselves to consider only the syntactic aspects of expressions, then very few expressions can be considered equal. The syntactic view also conflicts with the intuition that the significance of an expression lies in its contribution to the *outcome* of a computation [31]. Drawing inspiration from a wide range of literature on contextual equivalence [41], 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 5.** 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 6** (Kleene Equality). Two complete programs, e and e', are Kleene equal, written $e \simeq e'$, iff there exists i such that $e \longrightarrow^* i$ and $e' \longrightarrow^* i$.

▶ **Definition 7** (λ_c Contextual Equivalence).

$$\Delta \vdash e_1 \backsimeq_{ctx} e_2 : \tau \quad \triangleq \quad \Delta \vdash e_1 : \tau \wedge \Delta \vdash e_2 : \tau \wedge$$
$$\forall \mathcal{D}.\ \mathcal{D} : (\Delta \vdash \tau) \rightsquigarrow (\bullet \vdash \mathsf{Nat}) \Longrightarrow \mathcal{D}\{e_1\} \backsimeq \mathcal{D}\{e_2\}$$

Regarding Example 5, it seems adequate to say that 3 and $\mathsf{id}\ 3$ are contextually equivalent. Does this imply that coherence can be based on Definition 7? Unfortunately it cannot, as demonstrated by the following example.

▶ **Example 8.** 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 7, 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 8 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 $\mathcal{D} : (|\Gamma| \vdash |A|) \rightsquigarrow (|\Gamma'| \vdash |A'|)$ holds by construction. The typing rules appear in the appendix. Now we are ready to refine Definition 7's contextual equivalence to take into consideration both NeColus and λ_c contexts.

▶ **Definition 9** (NeColus Contextual Equivalence).

$$\Gamma \vdash E_1 \backsimeq_{ctx} E_2 : A \quad \triangleq \quad \forall e_1, e_2, \mathcal{C}, \mathcal{D}.\ \Gamma \vdash E_1 \Rightarrow A \rightsquigarrow e_1 \wedge \Gamma \vdash E_2 \Rightarrow A \rightsquigarrow e_2 \wedge$$
$$\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\bullet \Rightarrow \mathsf{Nat}) \rightsquigarrow \mathcal{D} \Longrightarrow \mathcal{D}\{e_1\} \backsimeq \mathcal{D}\{e_2\}$$

▶ Remark. For brevity we only consider expressions in the inference mode. Our Coq formalization is complete with two modes.

Regarding Example 8, a possible NeColus context is $[\cdot]\, 1 : (\bullet \Rightarrow \mathsf{Nat} \,\&\, \mathsf{Nat} \to \mathsf{Nat}) \mapsto (\bullet \Rightarrow \mathsf{Nat}) \rightsquigarrow [\cdot]\, \langle 1, 1\rangle$. We can verify that both $\lambda x.\, \pi_1\, x$ and $\lambda x.\, \pi_2\, x$ produce 1 in the context $[\cdot]\, \langle 1, 1\rangle$. Of course we should consider all possible contexts to be certain they are truly equal. From now on, we use the symbol $\backsimeq_{ctx}$ to refer to contextual equivalence in Definition 9. With Definition 9 we can formally state that NeColus is coherent in the following theorem:

▶ **Theorem 10** (Coherence). *If* $\Gamma \vdash E \Rightarrow A$ *then* $\Gamma \vdash E \backsimeq_{ctx} E : A$.

For the same reason as in Definition 9, we only consider expressions in the inference mode. The rest of the section is devoted to proving that Theorem 10 holds.

$$
\begin{aligned}
(v_1, v_2) \in \mathcal{V}[\![\mathsf{Nat}; \mathsf{Nat}]\!] &\triangleq \exists i, v_1 = v_2 = i \\
(v_1, v_2) \in \mathcal{V}[\![\tau_1 \to \tau_2; \tau_1' \to \tau_2']\!] &\triangleq \forall (v, v') \in \mathcal{V}[\![\tau_1; \tau_1']\!], (v_1\ v, v_2\ v') \in \mathcal{E}[\![\tau_2; \tau_2']\!] \\
(\{l = v_1\}, \{l = v_2\}) \in \mathcal{V}[\![\{l : \tau_1\}; \{l : \tau_2\}]\!] &\triangleq (v_1, v_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!] \\
(\langle v_1, v_2 \rangle, v_3) \in \mathcal{V}[\![\tau_1 \times \tau_2; \tau_3]\!] &\triangleq (v_1, v_3) \in \mathcal{V}[\![\tau_1; \tau_3]\!] \land (v_2, v_3) \in \mathcal{V}[\![\tau_2; \tau_3]\!] \\
(v_3, \langle v_1, v_2 \rangle) \in \mathcal{V}[\![\tau_3; \tau_1 \times \tau_2]\!] &\triangleq (v_3, v_1) \in \mathcal{V}[\![\tau_3; \tau_1]\!] \land (v_3, v_2) \in \mathcal{V}[\![\tau_3; \tau_2]\!] \\
(v_1, v_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!] &\triangleq \mathsf{true} \quad \text{otherwise} \\
(e_1, e_2) \in \mathcal{E}[\![\tau_1; \tau_2]\!] &\triangleq \exists v_1 v_2, e_1 \longrightarrow^* v_1 \land e_2 \longrightarrow^* v_2 \land (v_1, v_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!]
\end{aligned}
$$

Figure 11 Logical relations for λ_c

4.2 Logical Relations

Intuitive as Definition 9 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 10, which makes reasoning even more tedious. The key to simplifying the reasoning is to exploit types by using logical relations [63, 61, 48].

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 relate values originating from disjoint intersection types. Those values are essentially translated from merges, and since rule T-MERGE ensures disjointness, they are unambiguous. For example, $\langle 1, \{l = 1\} \rangle$ 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 relate values originating from non-disjoint intersection types, only if the values are duplicates. This may sound baffling since the whole point of disjointness is to rule out (ambiguous) expressions such as $1\,,,\,2$. However, $1\,,,\,2$ never gets elaborated, and the only values corresponding to $\mathsf{Nat} \,\&\, \mathsf{Nat}$ 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.

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 [8], 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 points 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: 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:

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

Notice the structural similarity between these two rules and the two cases. Now it is clear that the cases for products manifests 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 11** (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 12** (Interpretation of Typing Contexts). $(\gamma_1,\gamma_2) \in \mathcal{G}[\![\Delta_1;\Delta_2]\!]$ is defined as follows:

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

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

▶ **Definition 13** (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 are now ready to prove Theorem 10. But we need several lemmas to set the stage.

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 14** (Coercion Compatibility). *Suppose that* $c \vdash \tau_1 \rhd \tau_2$*,*

- *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. ◀

The "Fundamental Property" states that any well-typed expression is related to itself by the logical relation. In our elaboration setting, we rephrase it so that any two λ_c terms elaborated from the *same* NeColus expression are related by the logical relation. To prove it, we require Theorem 15.

▶ **Theorem 15** (Inference Uniqueness). *If* $\Gamma \vdash E \Rightarrow A_1$ *and* $\Gamma \vdash E \Rightarrow A_2$*, then* $A_1 \equiv A_2$.

▶ **Theorem 16** (Fundamental Property). *We have that:*

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

Proof. The proof follows by induction on the first derivation. The most interesting case is rule T-SUB where we need Theorem 15 to be able to apply the induction hypothesis. Then we apply Lemma 14 to say that the coercion generated preserves the relation between terms. For the other cases we use the appropriate compatibility lemmas. ◀

▶ Remark. It is interesting to ask whether the Fundamental Property holds in the target language. That is, 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. Our logical relation is carefully formulated so that the Fundamental Property holds in the source language.

We show that logical equivalence is preserved by NeColus contexts:

▶ **Lemma 17** (Congruence). *If* $\mathcal{C} : (\Gamma \Leftrightarrow A) \mapsto (\Gamma' \Leftrightarrow' A') \rightsquigarrow \mathcal{D}$*,* $\Gamma \vdash E_1 \Leftrightarrow A \rightsquigarrow e_1$*,* $\Gamma \vdash E_2 \Leftrightarrow A \rightsquigarrow e_2$ *and* $|\Gamma| \vdash e_1 \backsimeq_{log} e_2 : |A|$*, then* $|\Gamma'| \vdash \mathcal{D}\{e_1\} \backsimeq_{log} \mathcal{D}\{e_2\} : |A'|$.

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

▶ **Lemma 18** (Adequacy). *If* $\bullet \vdash e_1 \backsimeq_{log} e_2 : \mathsf{Nat}$ *then* $e_1 \backsimeq e_2$.

Proof. Adequacy follows easily from the definition of the logical relation. ◀

Next up is the proof that logical relation is sound with respect to contextual equivalence:

▶ **Theorem 19** (Soundness w.r.t. Contextual Equivalence). *If* $\Gamma \vdash E_1 \Rightarrow A \rightsquigarrow e_1$ *and* $\Gamma \vdash E_2 \Rightarrow A \rightsquigarrow e_2$ *and* $|\Gamma| \vdash e_1 \backsimeq_{log} e_2 : |A|$ *then* $\Gamma \vdash E_1 \backsimeq_{ctx} E_2 : A$.

Proof. From Definition 9, we are given a context $\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\bullet \Rightarrow \mathsf{Nat}) \rightsquigarrow \mathcal{D}$. By Lemma 17 we have $\bullet \vdash \mathcal{D}\{e_1\} \backsimeq_{log} \mathcal{D}\{e_2\} : \mathsf{Nat}$, thus $\mathcal{D}\{e_1\} \backsimeq \mathcal{D}\{e_2\}$ by Lemma 18. ◀

Armed with Theorem 16 and Theorem 19, coherence follows directly.

▶ **Theorem 10** (Coherence). *If* $\Gamma \vdash E \Rightarrow A$ *then* $\Gamma \vdash E \backsimeq_{ctx} E : A$.

Proof. Immediate from Theorem 16 and Theorem 19. ◀

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 20 says that merging a term of some type with something else does not affect its semantics. Corollary 21 and Corollary 22 express that merges are commutative and associative, respectively. Corollary 23 states that coercions from the same types are "coherent".

▶ **Corollary 20** (Merge is Neutral). *If* $\Gamma \vdash E_1 \Rightarrow A$ *and* $\Gamma \vdash E_1 \,,\, E_2 \Rightarrow A$*, then* $\Gamma \vdash E_1 \backsimeq_{ctx} E_1 \,,\, E_2 : A$

▶ **Corollary 21** (Merge is Commutative). *If* $\Gamma \vdash E_1 \,,\, E_2 \Rightarrow A$ *and* $\Gamma \vdash E_2 \,,\, E_1 \Rightarrow A$*, then* $\Gamma \vdash E_1 \,,\, E_2 \backsimeq_{ctx} E_2 \,,\, E_1 : A$.

▶ **Corollary 22** (Merge is Associative). *If* $\Gamma \vdash (E_1 \,,\, E_2) \,,\, E_3 \Rightarrow A$ *and* $\Gamma \vdash E_1 \,,\, (E_2 \,,\, E_3) \Rightarrow A$*, then* $\Gamma \vdash (E_1 \,,\, E_2) \,,\, E_3 \backsimeq_{ctx} E_1 \,,\, (E_2 \,,\, E_3) : A$.

▶ **Corollary 23** (Coercions Preserve Semantics). *If* $A <: B \rightsquigarrow c_1$ *and* $A <: B \rightsquigarrow c_2$*, then* $\Delta \vdash \lambda x.\, c_1\, x \backsimeq_{log} \lambda x.\, c_2\, x : |A| \to |B|$.

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 [37] 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 [47] 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.

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 24 converts a queue to a type:

▶ **Definition 24.** $\mathcal{L} \to A$ 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,

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

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

$$\textsc{A-top}\ \frac{}{\mathcal{L} \vdash A \prec: \top \rightsquigarrow [\![\mathcal{L}]\!]_{\top} \circ \mathsf{top}} \qquad \textsc{A-arrNat}\ \frac{[\,] \vdash A \prec: A_1 \rightsquigarrow c_1 \qquad \mathcal{L} \vdash A_2 \prec: \mathsf{Nat} \rightsquigarrow c_2}{A, \mathcal{L} \vdash A_1 \to A_2 \prec: \mathsf{Nat} \rightsquigarrow c_1 \to c_2}$$

$$\textsc{A-rcdNat}\ \frac{\mathcal{L} \vdash A \prec: \mathsf{Nat} \rightsquigarrow c}{\{l\}, \mathcal{L} \vdash \{l : A\} \prec: \mathsf{Nat} \rightsquigarrow \{l : c\}} \qquad \textsc{A-andN1}\ \frac{\mathcal{L} \vdash A_1 \prec: \mathsf{Nat} \rightsquigarrow c}{\mathcal{L} \vdash A_1 \,\&\, A_2 \prec: \mathsf{Nat} \rightsquigarrow c \circ \pi_1}$$

$$\textsc{A-andN2}\ \frac{\mathcal{L} \vdash A_2 \prec: \mathsf{Nat} \rightsquigarrow c}{\mathcal{L} \vdash A_1 \,\&\, A_2 \prec: \mathsf{Nat} \rightsquigarrow c \circ \pi_2} \qquad \textsc{A-nat}\ \frac{}{[\,] \vdash \mathsf{Nat} \prec: \mathsf{Nat} \rightsquigarrow \mathsf{id}}$$

Figure 12 Algorithmic subtyping of NeColus

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. Let us illustrate the algorithm with 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.

$$\dfrac{\dfrac{\dfrac{D \qquad D'}{\{l\}, \mathsf{N}\,\&\,\mathsf{S}, \mathsf{N}\,\&\,\mathsf{S} \vdash \{l : \mathsf{N} \to \mathsf{N} \to \mathsf{N}\} \,\&\, \{l : \mathsf{S} \to \mathsf{S} \to \mathsf{S}\} \prec: \mathsf{N}\,\&\,\mathsf{S}}\ \textsc{A-and}}{\{l\} \vdash \{l : \mathsf{N} \to \mathsf{N} \to \mathsf{N}\} \,\&\, \{l : \mathsf{S} \to \mathsf{S} \to \mathsf{S}\} \prec: \mathsf{N}\,\&\,\mathsf{S} \to \mathsf{N}\,\&\,\mathsf{S} \to \mathsf{N}\,\&\,\mathsf{S}}\ \textsc{A-arr}(\textit{twice})}{\{l : \mathsf{N} \to \mathsf{N} \to \mathsf{N}\} \,\&\, \{l : \mathsf{S} \to \mathsf{S} \to \mathsf{S}\} \prec: \{l : \mathsf{N}\,\&\,\mathsf{S} \to \mathsf{N}\,\&\,\mathsf{S} \to \mathsf{N}\,\&\,\mathsf{S}\}}\ \textsc{A-rcd}$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{\mathsf{N}\,\&\,\mathsf{S} \prec: \mathsf{N}} \qquad \dfrac{\cdots}{\mathsf{N}\,\&\,\mathsf{S} \vdash \mathsf{N} \to \mathsf{N} \prec: \mathsf{N}}}{\mathsf{N}\,\&\,\mathsf{S}, \mathsf{N}\,\&\,\mathsf{S} \vdash \mathsf{N} \to \mathsf{N} \to \mathsf{N} \prec: \mathsf{N}}\ \textsc{A-arrNat}}{\{l\}, \mathsf{N}\,\&\,\mathsf{S}, \mathsf{N}\,\&\,\mathsf{S} \vdash \{l : \mathsf{N} \to \mathsf{N} \to \mathsf{N}\} \prec: \mathsf{N}}\ \textsc{A-rcdNat}}{\{l\}, \mathsf{N}\,\&\,\mathsf{S}, \mathsf{N}\,\&\,\mathsf{S} \vdash \{l : \mathsf{N} \to \mathsf{N} \to \mathsf{N}\} \,\&\, \{l : \mathsf{S} \to \mathsf{S} \to \mathsf{S}\} \prec: \mathsf{N}}\ \textsc{A-andN1}$$

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

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

Figure 13 Meta-functions of coercions

easily, completeness is much harder. The proof of completeness essentially follows that of Pierce [47] in that we need to show the algorithmic subtyping is reflexive and transitive.

Soundness of the Algorithm. The proof of soundness is straightforward.

▶ **Theorem 25** (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. ◀

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 26.** *$A \in \mathcal{U}(A)$*

Proof. By induction on the structure of A. ◀

▶ **Lemma 27.** *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 27 is not found in Pierce's proofs [47], which is crucial in Lemma 28, from which reflexivity (Lemma 29) follows immediately.

▶ **Lemma 28.** *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 29** (Reflexivity). *For every A there exists c such that $[\,] \vdash A \prec: A \rightsquigarrow c$.*

Proof. Immediate from Lemma 26 and Lemma 28. ◀

We omit the details of the proof of transitivity.

▶ **Lemma 30** (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 31** (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 Theorems 29 and 30 where appropriate. ◀

▶ Remark. Pierce's proof is wrong [47, pp. 20, Case F] in the case

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

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

▶ Remark. It is worth pointing out that the two coercions c and c' in Theorem 31 are contextually equivalent, which follows from Theorem 25 and Corollary 23.

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 [55] 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.

Breazu-Tannen et al. [10] proved the coherence of a coercion translation from Fun [13] extended with recursive types to System F by showing that any two typing derivations of the same judgement are normalizable to a unique normal derivation. Ghelli [20] 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 [59] proved the coherence of coercion translation from Moggi's computational lambda calculus [40] 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 calculus has general recursion. Biernacki and Polesiuk [8] considered the coherence problem of coercion semantics. Their criterion for coherence of the translation is *contextual equivalence* in the target calculus. They presented a construction of logical relations for establishing so constructed coherence for coercion semantics, 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 [54] has intersection types and a merge-like operator. 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. [15] proposed a coherent calculus with a special merge operator that works on functions only. More recently, Dunfield [23] shows significant expressiveness of type systems with intersection types and a merge operator. However his calculus lacks coherence. The limitation was addressed by Oliveira et al. [46], who introduced disjointness to ensure coherence. The combination of intersection types, a merge operator and parametric polymorphism, while achieving coherence was first studied in the F_i calculus [2]. Compared to prior work, NeColus simplifies type systems with disjoint intersection types by removing several restrictions. Furthermore, NeColus adopts a 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. [4]. 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. [5] showed how to type classes and mixins in a BCD-style record calculus with Bracha-Cook's merge operator [9]. 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 [47, 35, 52, 62]. Laurent [36] 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 [62], Bessai et al. [6] show a formally verified subtyping algorithm in Coq. Our Coq formalization follows a different idea based on Pierce's decision procedure [47], 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 [25]. 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 [65]), Scala does not support nested composition: programmers need to manually compose all the classes from multiple extensions. Generally speaking, systems that support family polymorphism usually require quite sophisticated mechanisms such as dependent types.

There are two approaches to family polymorphism: the original *object family* approach of Beta (e.g., virtual classes [38]) treats nested classes as attributes of objects of the family classes. Path-dependent types are used to ensure type safety for virtual types and virtual classes in the calculus *vc* [27]. 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. In the *class family* approach of Concord [34], Jx and J& [42, 43], nested classes and types are attributes of the family classes directly. Jx supports *nested inheritance*, a class family mechanism that allows nesting of arbitrary depth. J& is a language that supports *nested intersection*, building on top of Jx. Similar to 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. When conflicts arise, prefix types can be exploited to resolve the ambiguity. J&$_s$ [50] is an extension of the Java language that adds class sharing to J&. Saito et al. [57] identified a minimal, lightweight set of language features to enable family polymorphism, Corradi et al. [19] 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. More recently, a Java-like language called Familia [66] were proposed to combine subtyping polymorphism, parametric polymorphism and family polymorphism. The object and class family approaches have even been combined by the work on Tribe [16].

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 [12, 56] 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. It 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 [53]. The main challenge in the definition of the logical relation is the clause that relates type variables with arbitrary types. Careful measures are to be taken to avoid potential circularity due to impredicativity.[7] With the combination of parametric polymorphism and nested composition, an interesting application that we intend to investigate is native support for a highly modular form of *Object Algebras* [45, 7] and Visitors (or the finally tagless approach [14]).

Another direction for future work is to add mutable references, which would affect two aspects of our metatheory: type safety and coherence. For type safety, we expect that lessons learned from previous work on family polymorphism and mutability on OO apply to our work. For example, it is well-known that subtyping in the presence of mutable state often needs restrictions. Given such suitable restrictions we expect that type-safety in the presence of mutability still holds. For coherence, it would be a major technical challenge to adjust our coherence proof and its Coq mechanization: logical relations that account for mutable state (e.g., see Ahmed's thesis [1]) introduce significant complexity.

References

1 Amal Jamil Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.

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

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

3 Nada Amin, Adriaan Moors, and Martin Odersky. Dependent object types. In *Workshop on Foundations of Object-Oriented Languages*, 2012.

4 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *The journal of symbolic logic*, 48(04):931–940, 1983.

5 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.

6 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.

7 Xuan Bi and Bruno C. d. S. Oliveira. Typed first-class traits. In *European Conference on Object-Oriented Programming*, 2018.

8 Dariusz Biernacki and Piotr Polesiuk. Logical relations for coherence of effect subtyping. In *LIPIcs*, 2015.

9 Gilad Bracha and William R. Cook. Mixin-based inheritance. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1990.

10 Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, 1991.

11 Kim Bruce, Luca Cardelli, Giuseppe Castagna, Gary T Leavens, Benjamin Pierce, et al. On binary methods. In *Theory and Practice of Object Systems*, 1996.

12 Kim B. Bruce, Angela Schuett, and Robert van Gent. Polytoil: A type-safe polymorphic object-oriented language. In *European Conference on Object-Oriented Programming*, 1995.

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

14 Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509, 2009.

15 Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. In *LFP*, 1992.

16 David Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: More Types for Virtual Classes. In *AOSD*, 2007.

17 Adriana B Compagnoni and Benjamin C Pierce. Higher-order intersection types and multiple inheritance. *MSCS*, 6(5):469–501, 1996.

18 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 1981.

19 Andrea Corradi, Marco Servetto, and Elena Zucca. DeepFJig — modular composition of nested classes. *The Journal of Object Technology*, 11(2):1:1, 2012.

20 Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in f $\leq$. *MSCS*, 2(01):55, 1992.

21 Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *International Conference on Functional Programming*, 2000.

22 Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, 2006.

23 Joshua Dunfield. Elaborating intersection and union types. *Journal of Functional Programming*, 24:133–165, 2014.

24 Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In *Foundations of Software Science and Computation Structure (FoSSaCS)*, 2003.

25 Erik Ernst. Family polymorphism. In *European Conference on Object-Oriented Programming*, 2001.

26 Erik Ernst. Higher-order hierarchies. In *European Conference on Object-Oriented Programming*, 2003.

27 Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Symposium on Principles of Programming Languages*, 2006.

28 Facebook. Flow. `https://flow.org/`, 2014.

29 Kathleen Fisher and John Reppy. A typed calculus of traits. In *Workshop on Foundations of Object-Oriented Languages*, 2004.

30 Tim Freeman and Frank Pfenning. Refinement types for ML. In *Conference on Programming Language Design and Implementation*, 1991.

31 Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016.

32 Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, jun 1994.

33 David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23(2):167, 2010.

34 Paul Jolly, Sophia Drossopoulou, Christopher Anderson, and Klaus Ostermann. Simple dependent types: Concord. In *European Conference on Object-Oriented Programming Workshop on Formal Techniques for Java Programs (FTfJP)*, 2004.

35 Toshihiko Kurata and Masako Takahashi. Decidable properties of intersection type systems. *Typed Lambda Calculi and Applications*, pages 297–311, 1995.

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

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

38 O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1989.

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

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

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

42 Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

43 Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested Intersection for Scalable Software Composition. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006.

44 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.

45 Bruno C. d. S. Oliveira and William R. Cook. Extensibility for the masses. In *European Conference on Object-Oriented Programming*, 2012.

46 Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. Disjoint intersection types. In *International Conference on Functional Programming*, 2016.

47 Benjamin C Pierce. A decision procedure for the subtype relation on intersection types with bounded variables. Technical report, Carnegie Mellon University, 1989.
48 Gordon Plotkin. *Lambda-definability and logical relations*. Edinburgh University, 1973.
49 Garrel Pottinger. A type assignment for the strongly normalizable λ-terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 1980.
50 Xin Qi and Andrew C. Myers. Sharing classes between families. In *Conference on Programming Language Design and Implementation*, 2009.
51 Redhat. Ceylon. `https://ceylon-lang.org/`, 2011.
52 Jakob Rehof and Paweł Urzyczyn. Finite combinatory logic with intersection types. In *International Conference on Typed Lambda Calculi and Applications*, 2011.
53 John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP*, 1983.
54 John C Reynolds. Preliminary design of the programming language forsythe. Technical report, Carnegie Mellon University, 1988.
55 John C. Reynolds. The coherence of languages with intersection types. In *Lecture Notes in Computer Science (LNCS)*, pages 675–700. Springer Berlin Heidelberg, 1991.
56 Chieri Saito and Atsushi Igarashi. Matching *ThisType* to subtyping. In *Symposium on Applied Computing (SAC)*, 2009.
57 Chieri Saito, Atsushi Igarashi, and Mirko Viroli. Lightweight family polymorphism. *Journal of Functional Programming*, 18(03), 2007.
58 Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming*, 2003.
59 Jan Schwinghammer. Coherence of subsumption for monadic types. *Journal of Functional Programming*, 19(02):157, 2008.
60 Jeremy Siek, Peter Thiemann, and Philip Wadler. Blame and coercion: together again for the first time. In *Conference on Programming Language Design and Implementation*, 2015.
61 Richard Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65(2-3):85–97, 1985.
62 Rick Statman. A finite model property for intersection types. *Electronic Proceedings in Theoretical Computer Science*, 177:1–9, 2015.
63 W. W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of symbolic logic*, 32(2):198–212, 1967.
64 Philip Wadler. The expression problem. *Java-genericity mailing list*, 1998.
65 Yanlin Wang and Bruno C d S Oliveira. The expression problem, trivially! In *Proceedings of the 15th International Conference on Modularity*, 2016.
66 Yizhou Zhang and Andrew C. Myers. Familia: unifying interfaces, type classes, and family polymorphism. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2017.

A Some Definitions

▶ **Definition 32** (Type translation).

$$
\begin{aligned}
|\mathsf{Nat}| &= \mathsf{Nat} \\
|\top| &= \langle\rangle \\
|A \to B| &= |A| \to |B| \\
|A \,\&\, B| &= |A| \times |B| \\
|\{l : A\}| &= \{l : |A|\}
\end{aligned}
$$

▶ **Example 33** (Derivation of other direction of distribution).

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

▶ **Example 34** (Derivation of contravariant distribution).

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

B Full Type System of NeColus

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

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

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

$$
\textsc{S-andl}\ \dfrac{}{A_1 \,\&\, A_2 <: A_1 \rightsquigarrow \pi_1}
\qquad
\textsc{S-andr}\ \dfrac{}{A_1 \,\&\, A_2 <: A_2 \rightsquigarrow \pi_2}
\qquad
\textsc{S-and}\ \dfrac{A_1 <: A_2 \rightsquigarrow c_1 \qquad A_1 <: A_3 \rightsquigarrow c_2}{A_1 <: A_2 \,\&\, A_3 \rightsquigarrow \langle c_1, c_2 \rangle}
$$

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

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

$$
\textsc{S-topRcd}\ \dfrac{}{\top <: \{l : \top\} \rightsquigarrow \mathsf{top}_{\{l\}}}
$$

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

$$\text{T-TOP}\quad \frac{}{\Gamma \vdash \top \Rightarrow \top \rightsquigarrow \langle\rangle} \qquad \text{T-LIT}\quad \frac{}{\Gamma \vdash i \Rightarrow \mathsf{Nat} \rightsquigarrow i} \qquad \text{T-VAR}\quad \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A \rightsquigarrow x}$$

$$\text{T-APP}\quad \frac{\Gamma \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e_1 \qquad \Gamma \vdash E_2 \Leftarrow A_1 \rightsquigarrow e_2}{\Gamma \vdash E_1\, E_2 \Rightarrow A_2 \rightsquigarrow e_1\, e_2} \qquad \text{T-ANNO}\quad \frac{\Gamma \vdash E \Leftarrow A \rightsquigarrow e}{\Gamma \vdash E : A \Rightarrow A \rightsquigarrow e}$$

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

$$\text{T-RCD}\quad \frac{\Gamma \vdash E \Rightarrow A \rightsquigarrow e}{\Gamma \vdash \{l = E\} \Rightarrow \{l : A\} \rightsquigarrow \{l = e\}} \qquad \text{T-PROJ}\quad \frac{\Gamma \vdash E \Rightarrow \{l : A\} \rightsquigarrow e}{\Gamma \vdash E.l \Rightarrow A \rightsquigarrow e.l}$$

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

$$\text{T-ABS}\quad \frac{\Gamma, x : A \vdash E \Leftarrow B \rightsquigarrow e}{\Gamma \vdash \lambda x.\, E \Leftarrow A \to B \rightsquigarrow \lambda x.\, e} \qquad \text{T-SUB}\quad \frac{\Gamma \vdash E \Rightarrow B \rightsquigarrow e \qquad B <: A \rightsquigarrow c}{\Gamma \vdash E \Leftarrow A \rightsquigarrow c\, e}$$

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

$$\text{D-TOPL}\quad \frac{}{\top * A} \qquad \text{D-TOPR}\quad \frac{}{A * \top} \qquad \text{D-ARR}\quad \frac{A_2 * B_2}{A_1 \to A_2 * B_1 \to B_2} \qquad \text{D-ANDL}\quad \frac{A_1 * B \qquad A_2 * B}{A_1 \,\&\, A_2 * B}$$

$$\text{D-ANDR}\quad \frac{A * B_1 \qquad A * B_2}{A * B_1 \,\&\, B_2} \qquad \text{D-RCDEQ}\quad \frac{A * B}{\{l : A\} * \{l : B\}} \qquad \text{D-RCDNEQ}\quad \frac{l_1 \neq l_2}{\{l_1 : A\} * \{l_2 : B\}} \qquad \text{D-AXNATARR}\quad \frac{}{\mathsf{Nat} * A_1 \to A_2}$$

$$\text{D-AXARRNAT}\quad \frac{}{A_1 \to A_2 * \mathsf{Nat}} \qquad \text{D-AXNATRCD}\quad \frac{}{\mathsf{Nat} * \{l : A\}} \qquad \text{D-AXRCDNAT}\quad \frac{}{\{l : A\} * \mathsf{Nat}} \qquad \text{D-AXARRRCD}\quad \frac{}{A_1 \to A_2 * \{l : A\}}$$

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

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

$$\text{CTYP-EMPTY1}\quad \frac{}{[\cdot] : (\Gamma \Rightarrow A) \mapsto (\Gamma \Rightarrow A) \rightsquigarrow [\cdot]} \qquad \text{CTYP-APPL1}\quad \frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1 \to A_2) \rightsquigarrow \mathcal{D} \qquad \Gamma' \vdash E_2 \Leftarrow A_1 \rightsquigarrow e}{\mathcal{C}\, E_2 : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D}\, e}$$

$$\text{CTYP-APPR1}\quad \frac{\Gamma' \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e \qquad \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Leftarrow A_1) \rightsquigarrow \mathcal{D}}{E_1\, \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow e\, \mathcal{D}}$$

$$\text{CTYP-MERGEL1}\quad \frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1) \rightsquigarrow \mathcal{D} \qquad \Gamma' \vdash E_2 \Rightarrow A_2 \rightsquigarrow e \qquad A_1 * A_2}{\mathcal{C} \,,,\, E_2 : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1 \,\&\, A_2) \rightsquigarrow \langle \mathcal{D}, e\rangle}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\textsc{A-nat}\ \frac{}{[] \vdash \mathsf{Nat} \prec: \mathsf{Nat} \rightsquigarrow \mathsf{id}}
\qquad
\textsc{A-andN1}\ \frac{\mathcal{L} \vdash A_1 \prec: \mathsf{Nat} \rightsquigarrow c}{\mathcal{L} \vdash A_1 \,\&\, A_2 \prec: \mathsf{Nat} \rightsquigarrow c \circ \pi_1}
\qquad
\textsc{A-andN2}\ \frac{\mathcal{L} \vdash A_2 \prec: \mathsf{Nat} \rightsquigarrow c}{\mathcal{L} \vdash A_1 \,\&\, A_2 \prec: \mathsf{Nat} \rightsquigarrow c \circ \pi_2}$$

C Full Type System of λ_c

$\boxed{\Delta \vdash e : \tau}$ *(Target typing)*

$$\textsc{typ-unit}\ \frac{}{\Delta \vdash \langle\rangle : \langle\rangle}
\qquad
\textsc{typ-lit}\ \frac{}{\Delta \vdash i : \mathsf{Nat}}
\qquad
\textsc{typ-var}\ \frac{x : \tau \in \Delta}{\Delta \vdash x : \tau}
\qquad
\textsc{typ-abs}\ \frac{\Delta, x : \tau_1 \vdash e : \tau_2}{\Delta \vdash \lambda x.\, e : \tau_1 \to \tau_2}$$

$$\textsc{typ-app}\ \frac{\Delta \vdash e_1 : \tau_1 \to \tau_2 \qquad \Delta \vdash e_2 : \tau_1}{\Delta \vdash e_1\ e_2 : \tau_2}
\qquad
\textsc{typ-pair}\ \frac{\Delta \vdash e_1 : \tau_1 \qquad \Delta \vdash e_2 : \tau_2}{\Delta \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

$$\textsc{typ-capp}\ \frac{\Delta \vdash e : \tau \qquad c \vdash \tau \rhd \tau'}{\Delta \vdash c\, e : \tau'}
\qquad
\textsc{typ-rcd}\ \frac{\Delta \vdash e : \tau}{\Delta \vdash \{l = e\} : \{l : \tau\}}
\qquad
\textsc{typ-proj}\ \frac{\Delta \vdash e : \{l : \tau\}}{\Delta \vdash e.l : \tau}$$

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

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

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

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

$$\frac{}{\mathsf{dist}_{\{l\}} \vdash \{l : \tau_1\} \times \{l : \tau_2\} \rhd \{l : \tau_1 \times \tau_2\}}\ \textsc{cotyp-distRcd}$$

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

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

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

$$\frac{}{\mathsf{top}_{\{l\}}\ \langle\rangle \longrightarrow \{l = \langle\rangle\}}\ \textsc{step-topRcd} \qquad \frac{}{((c_1 \rightarrow c_2)\ v_1)\ v_2 \longrightarrow c_2\ (v_1\ (c_1\ v_2))}\ \textsc{step-arr}$$

$$\frac{}{\langle c_1, c_2\rangle\ v \longrightarrow \langle c_1\ v, c_2\ v\rangle}\ \textsc{step-pair} \qquad \frac{}{(\mathsf{dist}_{\rightarrow}\ \langle v_1, v_2\rangle)\ v_3 \longrightarrow \langle v_1\ v_3, v_2\ v_3\rangle}\ \textsc{step-distArr}$$

$$\frac{}{\mathsf{dist}_{\{l\}}\ \langle\{l = v_1\}, \{l = v_2\}\rangle \longrightarrow \{l = \langle v_1, v_2\rangle\}}\ \textsc{step-distRcd} \qquad \frac{}{\pi_1\ \langle v_1, v_2\rangle \longrightarrow v_1}\ \textsc{step-projL} \qquad \frac{}{\pi_2\ \langle v_1, v_2\rangle \longrightarrow v_2}\ \textsc{step-projR}$$

$$\frac{}{\{l : c\}\ \{l = v\} \longrightarrow \{l = c\ v\}}\ \textsc{step-crcd} \qquad \frac{}{(\lambda x.\ e)\ v \longrightarrow e[x \mapsto v]}\ \textsc{step-beta} \qquad \frac{}{\{l = v\}.l \longrightarrow v}\ \textsc{step-projRcd}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ \textsc{step-app1} \qquad \frac{e_2 \longrightarrow e_2'}{v_1\ e_2 \longrightarrow v_1\ e_2'}\ \textsc{step-app2} \qquad \frac{e_1 \longrightarrow e_1'}{\langle e_1, e_2\rangle \longrightarrow \langle e_1', e_2\rangle}\ \textsc{step-pair1} \qquad \frac{e_2 \longrightarrow e_2'}{\langle v_1, e_2\rangle \longrightarrow \langle v_1, e_2'\rangle}\ \textsc{step-pair2}$$

$$\frac{e \longrightarrow e'}{c\ e \longrightarrow c\ e'}\ \textsc{step-capp} \qquad \frac{e \longrightarrow e'}{\{l = e\} \longrightarrow \{l = e'\}}\ \textsc{step-rcd1} \qquad \frac{e \longrightarrow e'}{e.l \longrightarrow e'.l}\ \textsc{step-rcd2}$$