Revisiting Iso-Recursive Subtyping

YAODA ZHOU, The University of Hong Kong, China
BRUNO C. D. S. OLIVEIRA, The University of Hong Kong, China
JINXU ZHAO, The University of Hong Kong, China

The Amber rules are well-known and widely used for subtyping iso-recursive types. They were first briefly and informally introduced in 1985 by Cardelli in a manuscript describing the Amber language. Despite their use over many years, important aspects of the metatheory of the iso-recursive style Amber rules have not been studied in depth or turn out to be quite challenging to formalize.

This paper aims to revisit the problem of subtyping iso-recursive types. We start by introducing a novel declarative specification that we believe captures the "spirit" of Amber-style iso-recursive subtyping. Informally, the specification states that two recursive types are subtypes *if all their finite unfoldings are subtypes*. The Amber rules are shown to be sound with respect to this declarative specification. We then derive a *sound*, *complete* and *decidable* algorithmic formulation of subtyping that employs a novel *double unfolding* rule. Compared to the Amber rules, the double unfolding rule has the advantage of: 1) being modular; 2) not requiring reflexivity to be built in; and 3) leading to an easy proof of transitivity of subtyping. This work sheds new insights on the theory of subtyping iso-recursive types, and the new double unfolding rule has important advantages over the original Amber rules for both implementations and metatheoretical studies involving recursive types. All results are mechanically formalized in the Coq theorem prover. As far as we know, this is the first comprehensive treatment of iso-recursive subtyping dealing with unrestricted recursive types in a theorem prover.

CCS Concepts: • **Theory of computation** → **Type theory**; • **Software and its engineering** → **Object oriented languages**.

Additional Key Words and Phrases: Iso-recursive types, Formalization, Subtyping

ACM Reference Format:
Yaoda Zhou, Bruno C. d. S. Oliveira, and Jinxu Zhao. 2020. Revisiting Iso-Recursive Subtyping. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 223 (November 2020), 28 pages. https://doi.org/10.1145/3428291

1 INTRODUCTION

Recursive types are used in nearly all languages to define recursive data structures like sequences or trees. They are also used in Object-Oriented Programming every time a method needs an argument or return type of the enclosing class.

Recursive types come in two flavours: *equi-recursive types* and *iso-recursive types* [Crary et al. 1999]. With equi-recursive types a recursive type is equal to its unfolding. With iso-recursive types, a recursive type and its unfolding are only isomorphic. To convert between the (iso-)recursive type and its isomorphic unfolding explicit folding and unfolding constructs are necessary. The main advantage of equi-recursive types is convenience, as no explicit conversions are necessary.

Authors' addresses: Yaoda Zhou, Department of Computer Science, The University of Hong Kong, Hong Kong, China, ydzhou@cs.hku.hk; Bruno C. d. S. Oliveira, Department of Computer Science, The University of Hong Kong, Hong Kong, China, bruno@cs.hku.hk; Jinxu Zhao, Department of Computer Science, The University of Hong Kong, Hong Kong, China, jxzhao@cs.hku.hk.

2475-1421/2020/11-ART223
https://doi.org/10.1145/3428291

```
data List = Nil | Cons Int List

map :: (Int -> Int) -> List -> List
map f Nil          =
    Nil
map f (Cons x xs)  =
    Cons (f x) (map f xs)
```

```
class Shape {
   int area() {...}
   boolean compareArea(Shape s) {
      return s.area() == area();
   }
   Shape clone() {return new Shape();}
}
```

Fig. 1. Recursive types in Haskell (left) and Java (right).

However, a disadvantage is that algorithms for languages with equi-recursive types are quite complex. Furthermore, integrating equi-recursive types in type systems with advanced type features, while retaining desirable properties such as decidable type-checking, can be hard (or even impossible) [Colazzo and Ghelli 1999; Ghelli 1993; Solomon 1978].

Many languages adopt an iso-recursive formulation. The inconvenience of iso-recursive types is mostly eliminated by "hiding" the explicit folding and unfolding in other constructs. For example, in functional languages, such as Haskell or ML, iso-recursive types are provided via datatypes. Figure 1 (left) illustrates a simple recursive type in Haskell. The `List` datatype is recursive, as the `Cons` constructor requires a `List` as the second argument. Functions such as `map`, can then be defined by pattern matching. While there are no explicit folding or unfolding operations in the program above, every use of the constructors (`Nil` and `Cons`) triggers folding of the recursive type. Conversely, the patterns on `Nil` and `Cons` trigger unfolding of the recursive type. Similarly, in nominal Object-Oriented (OO) languages such as Java, iso-recursive types can be introduced in class definitions such as the one to the right of Figure 1. This class definition requires recursive types because both `compareArea` and `clone` need to refer to the enclosing class. Like the Haskell program above, there are no explicit uses of folding and unfolding. Instead, constructors trigger folding of the recursive type; while method calls (such as `area()`) trigger recursive type unfolding. The relationship between iso-recursive types, algebraic datatypes and pattern matching, and nominal OO class definitions is well-understood in the research literature [Lee et al. 2015; Pierce 2002; Stone and Harper 1996; Vanderwaart et al. 2003; Yang and Oliveira 2019].

The Amber rules are well-known and widely used for subtyping iso-recursive types. They were briefly and informally introduced in 1985 by Cardelli in a manuscript describing the Amber language [Cardelli 1985]. Later on, Amadio and Cardelli [1993] made a comprehensive study of the theory of recursive subtyping for a system with equi-recursive types employing Amber-style rules. One nice result of their study is a declarative model for specifying when two recursive types are in a subtyping relation. In essence, two (equi-)recursive types are subtypes if their infinite unfoldings are subtypes. Amadio and Cardelli's study remains to the day a standard reference for the theory of equi-recursive subtyping, although newer work simplifies and improves on the original theory [Brandt and Henglein 1997; Gapeyev et al. 2003]. Since then variants of the Amber rules have been employed multiple times in a variety of calculi and languages, but often in an iso-recursive setting [Bengtson et al. 2011; Chugh 2015; Duggan 2002; Lee et al. 2015; Swamy et al. 2011]. From this point onwards, in the context of this paper, whenever we use the term Amber rules we refer to a subtyping relation, modelling iso-recursive subtyping, with at least the following rules:

$$\frac{\Gamma, \alpha \le \beta \vdash A \le B}{\Gamma \vdash \mu\alpha.\, A \le \mu\beta.\, B}\ \text{S-Amber} \qquad \frac{\alpha \le \beta \in \Gamma}{\Gamma \vdash \alpha \le \beta}\ \text{S-Assumption} \qquad \frac{}{\Gamma \vdash A \le A}\ \text{S-Refl}$$

These rules are found in nearly all the literature modelling subtyping for iso-recursive types with the Amber rules. The first rule states that two recursive types are subtypes if their bodies are subtypes under the assumption that the two distinct recursive variables α and β are subtypes. The second rule states that two recursive variables are subtypes if they are related in the environment. We also consider the reflexivity rule to be an essential part of the formulation of iso-recursive style Amber rules for two different reasons. Without the reflexivity rule (or some similar rule) built-in, reflexivity cannot be proved. Indeed, it is well-known [Amadio and Cardelli 1993] that equal types with negative (or contravariant) recursive occurrences cannot be shown to be subtypes without reflexivity. A concrete example is $\mu\alpha.\ \alpha \to \mathsf{nat} <: \mu\alpha.\ \alpha \to \mathsf{nat}$. Secondly, the reflexivity rule is the main difference to an equi-recursive formulation employing the Amber rules. In an equi-recursive formulation, reflexivity is replaced by a much more powerful rule that employs an equivalence relation on types where two types are considered equal if their infinite unfoldings are equal. This enables showing, for instance, that $\mu\alpha.\ \mathsf{nat} \to \alpha <: \mu\alpha.\ \mathsf{nat} \to \mathsf{nat} \to \alpha$ holds in an equi-recursive formulation, while this clearly fails if we only employ syntactic equality as in the reflexivity rule.

The Amber rules are appealing due to their apparent simplicity, but their metatheory is not well studied. Clearly, from an implementation point of view, the Amber rules are rather simple and easy to implement. However, unlike an equi-recursive formulation, which has a clear declarative specification, there is no similar declarative specification for an iso-recursive formulation so far. Moreover, there are fundamental differences between equi-recursive and iso-recursive subtyping: while equi-recursive subtyping deals with infinite trees and is naturally understood in a coinductive setting [Brandt and Henglein 1997; Gapeyev et al. 2003], an Amber-style iso-recursive formulation deals with finite trees and ought to be understood in an inductive setting. Furthermore, important properties for algorithmic versions of the Amber rules are lacking or are quite difficult to prove. In particular, there is very little work in the literature regarding proof of transitivity for algorithmic formulations of the Amber rules. Indeed, the only proof for transitivity that we are aware of is by Bengtson et al. [2011]. However, the proof relies on a complex inductive argument, and attempts to formalize the proof in a theorem prover have been unsuccessful so far [Backes et al. 2014]. Finally, a fundamental lemma that arises in proofs of type preservation for calculi with iso-recursive subtyping is:

$$\text{If } \mu\alpha.\ A \le \mu\alpha.\ B \text{ then } [\alpha \mapsto \mu\alpha.\ A]\ A \le [\alpha \mapsto \mu\alpha.\ B]\ B$$

We call this lemma the *unfolding lemma*. The unfolding lemma plays a similar role in preservation to the substitution lemma (which is needed for proving preservation of beta-reduction), and is used to prove the case dealing with recursive type unfolding. The proof for the unfolding lemma is non-trivial, but there is also little work on proofs of this lemma for the Amber rules. While there are some interesting alternatives for iso-recursive subtyping [Hofmann and Pierce 1996; Ligatti et al. 2017], Amber-style subtyping strikes a good balance between expressive power and simplicity, and is widely used. Thus understanding Amber-style subtyping further is worthwhile.

This paper aims to revisit the problem of subtyping iso-recursive types. We start by introducing a novel declarative specification that we believe captures the "spirit" of Amber-style iso-recursive subtyping. Informally, the specification states that two recursive types are subtypes if all their finite unfoldings are subtypes. More formally, the subtyping rule for recursive types is:

$$\frac{\Gamma, \alpha \vdash [\alpha \mapsto A]^n\ A \le [\alpha \mapsto B]^n\ B \qquad \forall n = 1 \cdots \infty}{\Gamma \vdash \mu\alpha.\ A \le \mu\alpha.\ B}\ \text{S-Rec}$$

Here the notation $[\alpha \mapsto A]^n$ denotes the n-times finite unfolding of a type. Essentially, n times unfolding applies $n-1$ substitutions to the type A, and the rule checks that all n-times unfoldings are subtypes. Such a declarative formulation plays a similar role to Amadio and Cardelli's declarative specification for equi-recursive types. Because the specification is defined with respect to the finite unfoldings, this naturally leads to an inductive treatment of the theory. For example, the proof of transitivity of subtyping is fairly straightforward, with the more significant challenge being the unfolding lemma. With all the metatheory in place, proving subject-reduction for a typed lambda calculus with recursive types is a routine exercise. Furthermore, the Amber rules are shown to be sound with respect to this declarative specification.

The declarative specification of subtyping is not directly implementable because it has to consider all finite unfoldings. Furthermore, showing completeness of the Amber rules is hindered by the complexities involved their formalization. Therefore, to obtain a sound, complete and decidable algorithmic formulation, we follow a different route. We employ a novel *double unfolding* rule:

$$\frac{\Gamma, \alpha \vdash A \leq B \qquad \Gamma, \alpha \vdash [\alpha \mapsto A]\ A \leq [\alpha \mapsto B]\ B}{\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B}\ \text{S-Double}$$

This rule says that for determining if two recursive types are subtypes, checking 1-time and 2-times finite unfolding is enough. This rule accepts all valid subtyping statements that the Amber rules accept, but it has important advantages. In particular, the rule is modular in the sense that it does not require changes to other rules or definitions involved in subtyping. The environments are just a standard collection of type variables, and the rule for type variables is also standard. Consequently, proofs for properties such as transitivity only need to account for the new recursive case, while all the other cases remain essentially the same as in a subtyping relation without recursive types. In contrast, the Amber rules have a pervasive impact in the subtyping relation, which is the root cause of the difficulties in doing proofs such as transitivity. Moreover, an additional benefit is that reflexivity does not have to be built in, but it can be derived instead. Built-in reflexivity can be problematic in some settings, including calculi with record subtyping or intersection types. Such calculi can have "isomorphic" subtyping where two syntactically different types A and B can be subtypes of each other. For instance, in calculi with records, the types $\{x : Int, y : Bool\}$ and $\{y : Bool, x : Int\}$ are subtypes of each other. Avoiding built-in reflexivity makes the rules easier to apply in such settings. The main difficulty with the algorithmic formulation is proving soundness with respect to the declarative specification. For getting over this difficulty, we employ an inductive relation that captures valid *subtyping subderivations*.

To validate all our results we have mechanically formalized all our results in the Coq theorem prover. As far as we know this is the first comprehensive treatment of iso-recursive subtyping dealing with unrestricted recursive types in a theorem prover.

In summary, the contributions of this paper are:

- **A declarative specification for iso-recursive subtyping:** We propose a new declarative specification for iso-recursive subtyping, where two recursive types are subtypes if all the finite unfoldings are subtypes (Section 3).
- **Algorithmic subtyping with the double unfolding rule:** We show a sound, complete and decidable algorithmic formulation of subtyping employing a new double-unfolding rule (Section 4).
- **Induction on subderivations:** As part of the soundness proof for our algorithmic formulation we employ a novel technique of induction over subderivations of subtyping, which is independently useful as a proof technique (Section 4).
- **Soundness of the Amber rules:** We prove that the Amber rules are sound with respect to our new formulation of subtyping (Section 5).

- **Subject-reduction for a typed lambda calculus with recursive types:** To illustrate the applicability of our results we formalize a typed lambda calculus with recursive types and prove type preservation and progress (Section 3).
- **Mechanical formalization:** All the results are formalized in the Coq theorem prover and can be found at: https://github.com/juda/Iso-Recursive-Subtyping

2 OVERVIEW

This section provides an overview of the problem of iso-recursive subtyping and our approach. We first introduce some alternative formulations for iso-recursive subtyping and discuss some issues with the Amber rules. Then we present the key ideas of our work, including a novel declarative formulation of subtyping and the double unfolding rule.

2.1 Subtyping Recursive Types

Subtyping is a widely-used inclusion relation that compares two types. Many calculi have no types of "infinite" size. In such calculi comparing two types is relatively easy. However, with the existence of recursive types, comparing two types is no longer trivial. A recursive type $\mu\alpha.\ A$ usually contains itself as a subpart, represented by the type variable α. Therefore, a subtyping relation (or other form of comparison) needs to treat these types in a special way.

We choose to use a minimal set of types throughout this work for illustration. A type A, B, C, or D may refer to the primitive nat type, the top type $\top$, a function type $A \rightarrow B$, a type variable α or a recursive type $\mu\alpha.\ A$. The subtyping relations for the top type, primitive types and function types are standard:

$$\frac{}{A \leq \top} \qquad \frac{}{\mathsf{nat} \leq \mathsf{nat}} \qquad \frac{B_1 \leq A_1 \qquad A_2 \leq B_2}{A_1 \rightarrow A_2 \leq B_1 \rightarrow B_2}$$

Before diving into the design of subtyping relations for recursive types, we first look at some examples. We also discuss the role of the *unfolding lemma* in checking whether a subtyping relation between two recursive types is valid or not.

Example 1. Any type should be a subtype of itself, including

- $\mu\alpha.\ \alpha \rightarrow \alpha \leq \mu\alpha.\ \alpha \rightarrow \alpha$,
- $\mu\alpha.\ \alpha \rightarrow \mathsf{nat} \leq \mu\alpha.\ \alpha \rightarrow \mathsf{nat}$,
- $\mu\alpha.\ \mathsf{nat} \rightarrow \alpha \leq \mu\alpha.\ \mathsf{nat} \rightarrow \alpha$.

An important aspect to pay attention here is the negative occurrences of recursive type variables, which occur in the first two examples. The combination of contravariance of function types and recursive types is a key cause to some complexity which is necessary when subtyping recursive types, even for the case of equal types. Indeed, this is the key reason why in the Amber rules a reflexivity rule is needed. We come back to this point in Section 2.4.

Example 2. A second example is $\mu\alpha.\ \top \rightarrow \alpha \leq \mu\alpha.\ \mathsf{nat} \rightarrow \alpha$. This example illustrates *positive* recursive subtyping, since the recursive variables are used only in positive positions, and the two types are not equal. The left type is a function that consumes infinite values of any type, and the right type consumes infinite nat values. Hence the left type is more general than the right type.

Example 3. The type $\mu\alpha.\ \alpha \rightarrow \mathsf{nat}$ is *not* a subtype of $\mu\alpha.\ \alpha \rightarrow \top$. This final example serves the purpose of illustrating *negative* recursive subtyping, where recursive type variables occur in negative positions. If we ignore the recursive parts of these types, $A \rightarrow \mathsf{nat} \leq A \rightarrow \top$ holds for any type A. But that does not imply that $\mu\alpha.\ \alpha \rightarrow \mathsf{nat} \leq \mu\alpha.\ \alpha \rightarrow \top$, because the binder α on different sides refers to different types. If we unfold both types twice, we get:

$$((\mu\alpha.\alpha \to \mathsf{nat}) \to \mathsf{nat}) \to \mathsf{nat} \quad \text{v.s.} \quad ((\mu\alpha.\alpha \to \top) \to \top) \to \top$$

which should be rejected by the subtyping relation. Because of the contravariance of functions, we need to check not only that $\mathsf{nat} <: \top$ but also that $\top <: \mathsf{nat}$ (which does not hold).

The role of the unfolding lemma. In Example 3 we argued that subtyping should be rejected without actually defining a rule for subtyping of recursive types. The argument was that in such case subtyping should be rejected because unfolding the recursive type a few times leads to a subtyping relation that is going to be rejected by some other rule not involving recursive types. The unfolding lemma captures the essence of this argument formally:

$$\text{If } \mu\alpha.\, A \le \mu\alpha.\, B \text{ then } [\alpha \mapsto \mu\alpha.\, A]\, A \le [\alpha \mapsto \mu\alpha.\, B]\, B$$

It states that unfolding the types one time in a valid subtyping relation between recursive types always leads to a valid subtyping relation between the unfoldings. This property plays an important role in type soundness, and it essentially guarantees the type preservation of recursive type unfolding.

In the following subsections, we briefly review some possible designs for recursive subtyping.

2.2 A Rule That Only Works for Covariant Subtyping

As observed by Amadio and Cardelli [1993], a first idea to compare two recursive types is to use the following rules:

$$\frac{\Gamma, \alpha \vdash A \le B}{\Gamma \vdash \mu\alpha.\, A \le \mu\alpha.\, B} \qquad \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha \le \alpha}$$

which accept, for example, $\mu\alpha.\, \top \to \alpha \le \mu\alpha.\, \mathsf{nat} \to \alpha$ and $\mu\alpha.\, \alpha \to \alpha \le \mu\alpha.\, \alpha \to \alpha$. Unfortunately, these rules are unsound in the presence of negative recursive subtyping and contravariant subtyping for function types. We can easily derive the following invalid relation with those rules:

$$\mu\alpha.\, \alpha \to \mathsf{nat} \le \mu\alpha.\, \alpha \to \top$$

If we ignore the recursive symbol μ, it is not immediately obvious that the subtyping relation is problematic:

$$\alpha \to \mathsf{nat} \le \alpha \to \top$$

However, after unfolding the types twice the problem becomes obvious, as shown in Example 3:

$$((\mu\alpha.\, \alpha \to \mathsf{nat}) \to \mathsf{nat}) \to \mathsf{nat} \le ((\mu\alpha.\, \alpha \to \top) \to \top) \to \top$$

Generally speaking, these rules are sound for positive recursive subtyping. However, contravariant types and recursive type variables occurring in negative positions may allow *unsound* subtyping derivations, as shown above.

2.3 The Positive Restriction Rule

To fix the unsound rule in the presence of contravariant subtyping, we might restrict it with *positivity checks* on the types:

$$\frac{\Gamma, \alpha \vdash A \le B \qquad \text{non-neg}(\alpha, A) \qquad \text{non-neg}(\alpha, B)}{\Gamma \vdash \mu\alpha.\, A \le \mu\alpha.\, B}$$

where non-neg(α, A) is false when α occurs in negative positions of A. This restriction, which was also observed by Amadio and Cardelli [1993], solves the unsoundness problem and is employed in some languages and calculi [Backes et al. 2014]. The logic behind this restriction is that all the subderivations which encounter $\alpha \le \alpha$ (for some recursive type variable α) are valid. Since such subderivations only occur in positive (or covariant) positions, the left α represents $\mu\alpha.\ A$, and the right α represents $\mu\alpha.\ B$. Since the subtyping is covariant, the derivation $\mu\alpha.\ A \leq \mu\alpha.\ B$ is valid, and all subderivations $\alpha \leq \alpha$ are valid as well.

The main drawback of this rule is that no negative recursive subtyping is possible. It rejects some valid relations, such as $\mu\alpha.\ \top \rightarrow \alpha \leq \mu\alpha.\ \alpha \rightarrow \alpha$. Furthermore, at least without some form of reflexivity built-in, it even rejects subtyping of equal types with negative recursive variables, such as $\mu\alpha.\ \alpha \rightarrow \alpha \leq \mu\alpha.\ \alpha \rightarrow \alpha$.

2.4 The Amber Rules

Equi-recursive Amber rules. The Amber rules were introduced in the Amber language by Cardelli [1985]. Later, Amadio and Cardelli [1993] studied the metatheory for a subtyping relation that employs Amber-like rules. These rules are presented in Figure 2. The subtyping relation is declarative as the transitivity rule (rule OAmber-trans) is built-in. The rule OAmber-top and rule OAmber-arrow are standard. Rule OAmber-rec is the most prominent one, describing subtyping between two recursive types. The key idea in the Amber rules is to use *distinct* type variables for the two recursive types being compared (α and β). These two type variables are stored in the environment. Later, if a subtyping statement of the form $\alpha <: \beta$ is found, rule OAmber-assmp is used to check whether that pair is in the environment. The nice thing about rule OAmber-rec and rule OAmber-assmp is that they work very well for positive subtyping. Furthermore they rule out some bad cases with negative subtyping, such as $\mu\alpha.\ \alpha \rightarrow nat <: \mu\beta.\ \beta \rightarrow \top$. Unfortunately, rule OAmber-rec rules out too many cases with negative subtyping, including statements about equal types, such as $\mu\alpha.\ \alpha \rightarrow nat <: \mu\beta.\ \beta \rightarrow nat$. To compensate for this, rule OAmber-rec is complemented by a (generalization) of the reflexivity rule (rule OAmber-refl). In the case of Amadio and Cardelli's original rules, rule OAmber-rec comes with a non-trivial definition of equality $A = B$ (we refer to their paper for details). Such equality allows deriving statements such as $\mu\alpha.\ \mathsf{nat} \rightarrow \alpha = \mu\alpha.\ \mathsf{nat} \rightarrow \mathsf{nat} \rightarrow \alpha$ or $\mu\alpha.\ \mathsf{nat} \rightarrow \alpha = \mathsf{nat} \rightarrow \mu\alpha.\ \mathsf{nat} \rightarrow \alpha$, which is used to ensure that recursive types and their unfoldings are equivalent. That is, generally speaking, the following equality holds at the type-level:

$$\mu\alpha.\ A = [\alpha \mapsto \mu\alpha.\ A]\ A$$

In other words, the set of rules defines a subtyping relation for *equi-recursive types.* Amadio and Cardelli [1993] did a thorough study of the metatheory of such equi-recursive subtyping, including providing an intuitive specification for recursive subtyping. In essence two recursive types are subtypes if their infinite unfoldings are subtypes.

Iso-recursive Amber rules. Amadio and Cardelli's set of rules is more powerful than what is normally considered to be the folklore Amber rules for iso-recursive subtyping. Many typical presentations of the Amber rule simply use syntactic equality in reflexivity, which is less powerful, but is enough to express iso-recursive subtyping. In what follows we consider the folklore rules, where the equality ($A = B$) used in rule OAmber-refl is simplified by just considering syntactic equality. The iso-recursive rules can deal correctly with all the examples illustrated so far, accepting the various examples that we have argued should be accepted, and rejecting the other ones. Perhaps a small nitpicking point is the absence of well-formedness constraints in the subtyping rules. By modern day standards, this may look a little suspicious, but then again well-formedness of environments and types is typically standard and straightforward. Unfortunately, as it turns out, a suitable definition of well-formedness is non-trivial for Amber subtyping. We will come back to this issue in Section 5. Setting the issue of well-formedness aside for the moment, the Amber rules have some other important issues:

$\boxed{\Gamma \vdash A \leq B}$ *(Original Amber Rules)*

$$\textsc{OAmber-refl}\quad \frac{A = B}{\Gamma \vdash A \leq B} \qquad \textsc{OAmber-trans}\quad \frac{\Gamma \vdash A \leq B \qquad \Gamma \vdash B \leq C}{\Gamma \vdash A \leq C} \qquad \textsc{OAmber-assmp}\quad \frac{\alpha \leq \beta \in \Gamma}{\Gamma \vdash \alpha \leq \beta} \qquad \textsc{OAmber-top}\quad \frac{}{\Gamma \vdash A \leq \top}$$

$$\textsc{OAmber-arrow}\quad \frac{\Gamma \vdash B_1 \leq A_1 \qquad \Gamma \vdash A_2 \leq B_2}{\Gamma \vdash A_1 \to A_2 \leq B_1 \to B_2} \qquad \textsc{OAmber-rec}\quad \frac{\Gamma, \alpha \leq \beta \vdash A \leq B}{\Gamma \vdash \mu\alpha.\, A \leq \mu\beta.\, B}$$

Fig. 2. The complete Amber subtyping rules by Amadio and Cardelli [1993] for *equi-recursive* subtyping.

Reflexivity cannot be eliminated. The reflexivity rule is essential to the subtyping relation. As we have seen, one cannot even derive $\mu\alpha.\, \alpha \to nat \leq \mu\alpha.\, \alpha \to nat$ without the reflexivity rule, due to the contravariant positions of the variables. One possible fix is to add another rule that allows variable subtyping in contravariant positions:

$$\frac{\alpha \leq \beta \in \Gamma}{\Gamma \vdash \beta \leq \alpha}$$

However, such rule allows unsound subtypes, for instance, $\mu\alpha.\, \alpha \to \mathsf{nat} \leq \mu\alpha.\, \alpha \to \top$. In fact, adding this rule leads to a similar system to that in Section 2.2.

The reflexivity rule, if present in the subtyping relation, depends on a specific equivalence judgment. Simple systems might use syntactic equivalence or alpha-equivalence, yet those might be insufficient for other systems. For example, permutation of fields on record types should be considered as equivalent types, thus we may accept the following subtypes

$$\mu\alpha.\, \{x : \alpha, y : nat\} \to nat \leq \mu\alpha.\, \{y : nat, x : \alpha\} \to nat$$

However, if the built-in reflexivity employs only alpha-equivalence, such a subtyping derivation is rejected. To fix this it is natural to define the equivalence relation by requiring two types to be subtypes of each other. However, such definition would immediately lead to a circular dependency for recursive types. Therefore, one would need to explicitly define a reasonable set of equivalence rules (e.g. including alpha-equivalence and field permutations), with the sacrifice of simplicity and extensibility of a subtyping relation. The reader may refer to work from Ligatti et al. [2017] for a more extended discussion on the complications of having the reflexivity rule built-in.

Finding an algorithmic formulation: transitivity elimination is non-trivial. In the rules that Amadio and Cardelli [1993] use, and assuming that equivalence in reflexivity is just alpha-equivalence, simply dropping transitivity (rule OAmber-trans) to obtain an algorithmic formulation loses expressive power. A simple example that illustrates this is:

$$\frac{\alpha_1 \leq \alpha_2, \alpha_2 \leq \alpha_3 \vdash \alpha_1 \leq \alpha_2 \qquad \alpha_1 \leq \alpha_2, \alpha_2 \leq \alpha_3 \vdash \alpha_2 \leq \alpha_3}{\alpha_1 \leq \alpha_2, \alpha_2 \leq \alpha_3 \vdash \alpha_1 \leq \alpha_3}\ \textsc{does not hold!}$$

Such derivation is valid in a declarative formulation with transitivity, but invalid when transitivity is dropped. Therefore, either the declarative specification must be changed to eliminate “invalid” derivations, or the simply dropping transitivity will not work and some changes in the algorithmic rules are necessary.

Proofs of transitivity and other lemmas are hard. A related problem is proving that transitivity of an algorithmic system with Amber-style rules is hard. Surprisingly to us, despite the wide use of the Amber rules since 1985 for iso-recursive subtyping, there is very little work that describes transitivity proofs. Many works simply avoid the problem by considering only declarative rules with transitivity built-in [Cardone 1991; Lee et al. 2015; Pottier 2013]. The only proof that we are aware of for transitivity of an algorithmic formulation of the iso-recursive Amber rules is by Bengtson et al. [2011]. Some researchers have tried, but failed, to formalize this proof in Coq [Backes et al. 2014]. They found transitivity is hard to prove syntactically, and it requires a "very complicated inductive argument". Thus, they finally adopt the positive restriction, as we discussed in Section 2.3. We also tried to directly prove some of these properties in Coq with variations of the Amber rules, but none of them works properly.

Non-orthogonality of the Amber rules. Finally, the Amber rules interact with other subtyping rules. Besides requiring reflexivity, they require a specific kind of entries in the typing environment, which is different from typical entries in other subtyping relations. This affects other rules, and in particular it affects the proofs for cases that are not related to recursive types. For instance this is a key issue that we encountered when trying to prove transitivity and other properties. Furthermore, it also affects implementations, since adding the Amber rules to an existing implementation of subtyping requires changing existing definitions and some cases of the subtyping algorithm. In short, the Amber rules are not very modular: their addition has significant impact on existing definitions, rules, implementations and proofs.

2.5 Our Solution: A New Declarative Specification and the Double Unfolding Rule

While the Amber rules are simple, as we have argued, there are important issues with the rules. In particular developing the metatheory for the Amber rules is quite hard. Therefore, to provide a detailed account of the metatheory for iso-recursive subtyping we propose alternative definitions (both declarative and algorithmic) for subtyping of recursive types. The new formulation of subtyping has important advantages over the Amber rules: the new rules are more modular; they do not require reflexivity to be built-in; and transitivity and various other lemmas are easier to prove. Furthermore, we prove that the Amber rules are sound with respect to this new formulation.

The key idea. The key idea of the new rules is inspired by the rules presented for *covariant subtyping* in Section 2.2. The logic of the covariant rules is to approximate recursive subtyping using what we call a 1-time *finite unfolding*. We say that the unfolding is finite because we simply use α instead of using the recursive type itself during unfolding. If we apply finite unfoldings to all recursive types, we eventually end up having a comparison of two types representing finite trees. The covariant rules work fine in a setting with covariant subtyping only, but are unsound in a setting that also includes contravariant subtyping. A plausible question is then: can we fix these rules to become sound in the presence of contravariant subtyping?

The answer to this question is yes! Let us have a second look at the unsound counter-example that was presented in Section 2.2:

$$\mu\alpha.\ \alpha \to \mathsf{nat} \le \mu\alpha.\ \alpha \to \top$$

As we have argued, this subtyping statement should fail because unfolding the recursive type twice leads to an invalid subtyping derivation. However, with the 1-time finite unfolding used by the rules in Section 2.2, all that is checked is whether $\alpha \vdash \alpha \to \mathsf{nat} \le \alpha \to \top$ holds. Since such derivation does indeed hold, the rule *unsoundly* accepts $\mu\alpha.\ \alpha \to \mathsf{nat} \le \mu\alpha.\ \alpha \to \top$. The problem is that while the 1-time unfolding works, other n-times unfoldings do not. Therefore, an idea is to check whether other n-times unfoldings work as well to recover soundness.

Declarative subtyping. Our declarative subtyping rules build on the previous observation and only accept the subtyping relation between two recursive types if and only if *all* their n-times finite unfoldings are subtypes for any positive integer n:

$$\frac{\Gamma, \alpha \vdash [\alpha \mapsto A]^n\, A \le [\alpha \mapsto B]^n\, B \qquad \forall n = 1 \cdots \infty}{\Gamma \vdash \mu\alpha.\, A \le \mu\alpha.\, B}\ \text{S-Rec}$$

In comparison to the rules showed in Section 2.2, our subtyping rule S-REC has a stricter condition, by checking the subtyping relation for all n-times finite unfoldings, instead of only the 1-time finite unfolding. Such restriction eliminates the false positives on contravariant recursive types. The definition of n-times finite unfolding used in the rule is as follows:

Definition 1 (n-times finite unfolding)**.**

$$[\alpha \mapsto A]^n\, B := \underbrace{[\alpha \mapsto A]\,[\alpha \mapsto A] \cdots [\alpha \mapsto A]}_{(n-1)\ \text{times}}\, B$$

By definition, $[\alpha \mapsto A]^n\, A$ is the n-times finite unfolding of $\mu\alpha.\, A$. In other words, we execute $(n-1)$ times substitution of its body to itself. For example, $[\alpha \mapsto A]^1 A = A$, $[\alpha \mapsto A]^2\, A = [\alpha \mapsto A]\, A$, $[\alpha \mapsto A]^3\, A = [\alpha \mapsto A]\,[\alpha \mapsto A]\, A$, etc. We also slightly generalize the definition, to unfold a type B with another type A multiple times. This generalization is mainly used for proofs.

Algorithmic subtyping. An infinite amount of conditions is impossible to check algorithmically. However, it turns out that we only need to check 1-time and 2-times finite unfoldings to obtain an algorithmic formulation that is *sound*, *complete* and *decidable* with respect to the declarative formulation of subtyping. We can informally explain why 1-time and 2-times finite unfoldings are enough by looking again at the counter-example. The 2-times finite unfolding for the example is:

$$\alpha \vdash (\alpha \to \mathsf{nat}) \to \mathsf{nat} \le (\alpha \to \top) \to \top$$

When a recursive type variable in a negative position is unfolded twice, the types in the corresponding positive positions (i.e. the nat and $\top$) will now appear in both negative and positive positions. In turn, the subtyping relation now has to check both that $\mathsf{nat} <: \top$ (which is valid), and $\top <: \mathsf{nat}$ (which is invalid). Thus, the 2-times finite unfolding fails. In general, more finite unfoldings (3-times, 4-times, etc.) will only repeat the same checks that are done by the 1-time and 2-times finite unfolding, thus not contributing anything new to the subtyping check. Thus, the rule that we employ in the algorithmic formulation is the so-called *double unfolding* rule:

$$\frac{\Gamma, \alpha \vdash A \le B \qquad \Gamma, \alpha \vdash [\alpha \mapsto A]\, A \le [\alpha \mapsto B]\, B}{\Gamma \vdash \mu\alpha.\, A \le \mu\alpha.\, B}\ \text{S-Double}$$

As a final note, one may wonder if we can just check the 2-times finite unfolding (and do not do the 1-time finite unfolding check). Unfortunately this would lead to an unsound rule, as the following counter-example illustrates:

$$\mu\alpha.\, \mathsf{nat} \to \alpha \not\le \mu\alpha.\, \mathsf{nat} \to \mathsf{nat} \to \top$$

This derivation should fail because it violates the *unfolding lemma*:

$$\mathsf{nat} \to (\mu\alpha.\, \mathsf{nat} \to \alpha) \not\le \mathsf{nat} \to \mathsf{nat} \to \top$$

But the 2-times finite unfolding for this example ($\mathsf{nat} \to \mathsf{nat} \to \alpha \le \mathsf{nat} \to \mathsf{nat} \to \top$) is a valid subtyping statement! By checking only the 2-times finite unfolding, the subtyping statement is wrongly accepted. We must also check the 1-time finite unfolding ($nat \to \alpha \not\le nat \to nat \to \top$), which fails and is the reason why the double-unfolding rule rejects this example.

3 A CALCULUS WITH SUBTYPING AND RECURSIVE TYPES

In this section we will introduce a full calculus with declarative subtyping and recursive types. Our calculus is based on the simply typed lambda calculus extended with iso-recursive types and subtyping. This declarative system captures the idea that, with iso-recursive types, two recursive types are subtypes if all their finite unfoldings are subtypes. Notably we prove reflexivity, transitivity and the unfolding lemma.

3.1 Syntax and Well-Formedness

Syntax. The calculus that we model is a simply typed lambda calculus with subtyping. The syntax of types and contexts for this calculus is shown below.

$$\begin{array}{llcl}
\text{Types} & A, B, C, D & ::= & \mathsf{nat} \mid \top \mid A_1 \to A_2 \mid \alpha \mid \mu\alpha.\ A \\
\text{Expressions} & e & ::= & x \mid \mathsf{i} \mid e_1\ e_2 \mid \lambda x : A.\ e \mid \mathsf{unfold}\ [A]\ e \mid \mathsf{fold}\ [A]\ e \\
\text{Values} & v & ::= & \mathsf{i} \mid \lambda x : A.\ e \mid \mathsf{fold}\ [A]\ v \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, \alpha \mid \Gamma, x : A
\end{array}$$

Meta-variables A, B, C, D range over types. These types consist of: natural numbers (nat), the top type ($\top$), function types ($A \to B$), type variables (α) and recursive types ($\mu\alpha.\ A$). Expressions, denoted as e, include: natural numbers (i), applications ($e_1\ e_2$), lambda expressions ($\lambda x : A.\ e$). The expression unfold $[A]\ e$ is used to unfold the recursive type of an expression e; while fold $[A]\ e$ is used to fold the recursive type of an expression e. Some expressions are also values: natural numbers (i), lambda expressions ($\lambda x : A.\ e$) as well as fold expressions (fold $[A]\ v$) if their inner expressions are also values. The context is used to store variables with their type and type variables.

Well-formedness. The definition of a well-formed environment $\vdash \Gamma$ is standard, ensuring that all variables in the environment are distinct. The top of Figure 3 shows the judgement for well-formed types. A type is well-formed if all of its free variables are in the context. The rules of this judgement are mostly standard. The rule WFT-REC states that if the body of a recursive type is well-formed under an extended context then the recursive type is well-formed.

3.2 Subtyping

The bottom of Figure 3 shows the declarative subtyping judgement. Our subtyping rules are standard with the exception of the new rule for recursive types. Rule S-TOP states that any well-formed type A is a subtype of the $\top$ type. Rule S-VAR is a standard rule for type variables which are introduced when unfolding recursive types: variable α is a subtype of itself. The rule for function types (rule S-ARROW) is standard, but worth mentioning because it is contravariant on the input types. As illustrated in Section 2 (and various previous works), the interaction between recursive types and contravariance has been a key difficulty in the development of subtyping with recursive types. Finally, rule S-REC is most significant: it tells us that a recursive type $\mu\alpha.\ A$ is a subtype of $\mu\alpha.\ B$, if all their corresponding finite unfoldings are subtypes. Both $[\alpha \mapsto A]^n\ A$ and $[\alpha \mapsto B]^n\ B$ are used to denote n-times finite unfolding, as Definition 1 has illustrated.

3.3 Metatheory of Subtyping

The metatheory of the subtyping relation includes three essential properties: reflexivity, transitivity and the unfolding lemma.

A better induction principle for subtyping properties. The first challenge that we face when looking at the metatheory of subtyping with recursive types is to find adequate induction principles for various proofs. In particular the proofs of reflexivity and transitivity can be non-trivial a suitable

$\boxed{\Gamma \vdash A}$ *(Well-formed Type)*

$$\textsc{wft-nat}\ \frac{}{\Gamma \vdash \mathsf{nat}} \qquad \textsc{wft-Top}\ \frac{}{\Gamma \vdash \top} \qquad \textsc{wft-var}\ \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha} \qquad \textsc{wft-arrow}\ \frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \to A_2} \qquad \textsc{wft-rec}\ \frac{\Gamma, \alpha \vdash A}{\Gamma \vdash \mu\alpha.\ A}$$

$\boxed{\Gamma \vdash A \le B}$ *(Declarative subtyping)*

$$\textsc{S-nat}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{nat} \le \mathsf{nat}} \qquad \textsc{S-top}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash A \le \top} \qquad \textsc{S-var}\ \frac{\vdash \Gamma \qquad \alpha \in \Gamma}{\Gamma \vdash \alpha \le \alpha} \qquad \textsc{S-arrow}\ \frac{\begin{array}{c}\Gamma \vdash B_1 \le A_1 \\ \Gamma \vdash A_2 \le B_2\end{array}}{\Gamma \vdash A_1 \to A_2 \le B_1 \to B_2}$$

$$\textsc{S-rec}\ \frac{\Gamma, \alpha \vdash [\alpha \mapsto A]^n\ A \le [\alpha \mapsto B]^n\ B \quad \forall n = 1 \cdots \infty}{\Gamma \vdash \mu\alpha.\ A \le \mu\alpha.\ B}$$

Fig. 3. Well-formedness and subtyping rules.

induction principle. A first idea to prove both reflexivity and transitivity is to use induction on well-formed types. However, the problem of using this approach is that there is a mismatch between the well-formedness and subtyping rules for recursive types. The induction hypothesis that we get from rule wft-rec gives us a statement that works on 1-time finite unfoldings, whereas in the subtyping rule we have a premise expressed in terms of all finite unfoldings.

Fortunately, we can define an alternative variant of well-formedness that gives us a better induction principle. The idea is to replace rule wft-rec with a rule that expresses that if all finite unfoldings of a recursive type are well-formed then the recursive type is well-formed.

Definition 2. Rule wft-inf is defined as:

$$\textsc{wft-inf}\ \frac{\Gamma, \alpha \vdash [\alpha \mapsto A]^n\ A \quad \forall n = 1 \cdots \infty}{\Gamma \vdash \mu\alpha.\ A}$$

The two definitions of well-formedness are provably equivalent. In the proofs that follow, when we use induction on well-formed types, we use the variant with the rule wft-inf.

Reflexivity and transitivity. Next we prove reflexivity and transitivity. First of all, we know that subtyping is regular, i.e. subtyping implies well-formedness of context and types:

Lemma 3. Regularity: If $\Gamma \vdash A \le B$ then $\vdash \Gamma$ and $\Gamma \vdash A$ and $\Gamma \vdash B$.

Thanks to our standard context, the proofs of both reflexivity, transitivity are straightforward using the variant of well-formedness with rule wft-inf. This contrasts with the Amber rules [Cardelli 1985], where reflexivity needs to be built-in and the proof of transitivity is quite complex (and hard to mechanize on a theorem prover) [Backes et al. 2014; Bengtson et al. 2011].

Theorem 4. Reflexivity.

$$\text{If } \Gamma \vdash A \text{ then } \Gamma \vdash A \le A.$$

Theorem 5. Transitivity.

$$\text{If } \Gamma \vdash A \le B \text{ and } \Gamma \vdash B \le C \text{ then } \Gamma \vdash A \le C.$$

Unfolding lemma. Next, we turn to the unfolding lemma: if two recursive types are in a subtyping relation, then substituting themselves into their bodies preserves the subtyping relation. This lemma plays a crucial role in the proof of type preservation as we shall see in Section 3.5. However, the lemma cannot be proved directly: we need to prove a generalized lemma first.

Lemma 6. If

(1) $\Gamma, \alpha \vdash A \leq B$;
(2) $\Gamma \vdash \mu\alpha.\, C$ and $\Gamma \vdash \mu\alpha.\, D$;
(3) $\Gamma, \alpha \vdash [\alpha \mapsto C]^n\ A \leq [\alpha \mapsto D]^n\ B$ holds for all n,

then $\Gamma \vdash [\alpha \mapsto \mu\alpha.\, C]\ A \leq [\alpha \mapsto \mu\alpha.\, D]\ B$.

Proof. Induction on $\Gamma, \alpha \vdash A \leq B$. Cases rules S-NAT, S-TOP, and S-ARROW are simple.

- Rule S-VAR. Assume that both A and B are variable β. If $\beta \neq \alpha$, then the goal is proven directly. Otherwise, the third premise is still $\Gamma, \alpha \vdash [\alpha \mapsto C]^n\ \alpha \leq [\alpha \mapsto D]^n\ \alpha$, where n is arbitrary. The goal becomes $\Gamma \vdash \mu\alpha_1.\, C \leq \mu\alpha_1.\, D$. Then apply the rule for recursive types. The goal is equal to the third premise after alpha-conversion.
- Rule S-REC. Assume the A's shape is $\mu\alpha_1.\, A'$ and B's shape is $\mu\alpha_1.\, B'$. Then the third premise becomes $\Gamma, \alpha \vdash [\alpha \mapsto C]^n\ \mu\alpha_1.\, A' \leq [\alpha \mapsto D]^n\ \mu\alpha_1.\, B'$, which can be rewritten to $\Gamma, \alpha \vdash \mu\alpha_1.\, [\alpha \mapsto C]^n\ A' \leq \mu\alpha_1.\, [\alpha \mapsto D]^n\ B'$. The goal becomes $\Gamma \vdash [\alpha \mapsto \mu\alpha_2.\, C]\ \mu\alpha_1.\, A' \leq [\alpha \mapsto \mu\alpha_2.\, D]\ \mu\alpha_1.\, B'$, which can be rewritten to $\Gamma \vdash \mu\alpha_1.\, [\alpha \mapsto \mu\alpha_2.\, C]\ A' \leq \mu\alpha_1.\, [\alpha \mapsto \mu\alpha_2.\, D]\ B'$. By induction hypothesis, this goal is proven.

□

Lemma 6 captures the idea of finite approximation. It relates the boundless unfolding with limited unfolding. This lemma is a generalization of the unfolding lemma, and when $A = C$ and $B = D$, one easily obtains the unfolding lemma.

Lemma 7. Unfolding Lemma.

$$\text{If } \Gamma \vdash \mu\alpha.\, A \leq \mu\alpha.\, B \text{ then } \Gamma \vdash [\alpha \mapsto \mu\alpha.\, A]\ A \leq [\alpha \mapsto \mu\alpha.\, B]\ B.$$

3.4 Typing and Reduction Rules

Typing rules. As the top of Figure 4 shows, the typing rules are quite standard. Noteworthy are the rules involving recursive types. Rule TYPING-UNFOLD reveals that if e has type $\mu\alpha.\, A$ then, after unfolding, its type becomes $[\alpha \mapsto \mu\alpha.\, A]\ A$. Rule TYPING-FOLD says if e has type $[\alpha \mapsto \mu\alpha.\, A]\ A$, after folding, its type becomes $\mu\alpha.\, A$, with an additional type well-formedness check on $\mu\alpha.\, A$. The two constructs establish an isomorphism, which is used to deal with expressions with iso-recursive types. The last rule is the standard subsumption rule (rule TYPING-SUB).

Reduction. The bottom of Figure 4 shows the reduction rules, which are also quite standard. We only focus on the last three rules involving recursive types. Rule STEP-FLD cancels a pair of unfold and fold. Note that the two types A and B are not necessary the same. The last two rules (rule STEP-UNFOLD and rule STEP-FOLD) simply reduce the inner expressions for unfold's and fold's.

3.5 Type Soundness

In this subsection, we briefly illustrate how to prove type-soundness. The technique is mostly conventional, except for the fundamental use of the unfolding lemma in the preservation proof. Firstly, we need a conventional substitution lemma to deal with beta reduction in preservation:

Lemma 8. Substitution lemma. If $\Gamma_1, x : B, \Gamma_2 \vdash e : A$ and $\Gamma_2 \vdash e' : B$ then $\Gamma_1, \Gamma_2 \vdash [x \mapsto e']\ e : A$.

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

$$\textsc{typing-nat}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{i} : \mathsf{nat}} \qquad \textsc{typing-var}\ \frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{typing-sub}\ \frac{\Gamma \vdash e : A \qquad \Gamma \vdash A \le B}{\Gamma \vdash e : B} \qquad \textsc{typing-abs}\ \frac{\Gamma, x : A \vdash e : A_2}{\Gamma \vdash \lambda x : A.\ e : A_1 \to A_2}$$

$$\textsc{typing-app}\ \frac{\Gamma \vdash e_1 : A_1 \to A_2 \qquad \Gamma \vdash e_2 : A_1}{\Gamma \vdash e_1\ e_2 : A_2} \qquad \textsc{typing-unfold}\ \frac{\Gamma \vdash e : \mu\alpha.\ A}{\Gamma \vdash \mathsf{unfold}\ [\mu\alpha.\ A]\ e : [\alpha \mapsto \mu\alpha.\ A]\ A} \qquad \textsc{typing-fold}\ \frac{\Gamma \vdash e : [\alpha \mapsto \mu\alpha.\ A]\ A \qquad \Gamma \vdash \mu\alpha.\ A}{\Gamma \vdash \mathsf{fold}\ [\mu\alpha.\ A]\ e : \mu\alpha.\ A}$$

$\boxed{e_1 \hookrightarrow e_2}$ *(Reduction)*

$$\textsc{step-beta}\ \frac{}{(\lambda x : A.\ e_1)\ v_2 \hookrightarrow [x \mapsto v_2]\ e_1} \qquad \textsc{step-appl}\ \frac{e_1 \hookrightarrow e_1'}{e_1\ e_2 \hookrightarrow e_1'\ e_2} \qquad \textsc{step-appr}\ \frac{e_2 \hookrightarrow e_2'}{v_1\ e_2 \hookrightarrow v_1\ e_2'}$$

$$\textsc{step-fld}\ \frac{}{\mathsf{unfold}\ [A]\ (\mathsf{fold}\ [B]\ v) \hookrightarrow v} \qquad \textsc{step-unfold}\ \frac{e \hookrightarrow e'}{\mathsf{unfold}\ [A]\ e \hookrightarrow \mathsf{unfold}\ [A]\ e'} \qquad \textsc{step-fold}\ \frac{e \hookrightarrow e'}{\mathsf{fold}\ [A]\ e \hookrightarrow \mathsf{fold}\ [A]\ e'}$$

Fig. 4. Typing and Reduction Rules

Then we can proceed to the preservation and progress theorems.

Theorem 9. Preservation.

$$\text{If } \Gamma \vdash e : A \text{ and } e \hookrightarrow e' \text{ then } \Gamma \vdash e' : A.$$

Proof. By induction on $\Gamma \vdash e : A$. Other cases are trivial, except for

- Rule typing-app. In this case, e is decomposed into $e_1 \to e_2$. By inversion of $(e_1 \to e_2) \hookrightarrow e'$, we will get three sub-cases. Two of them are trivial, and for rule step-beta, Lemma 8 helps finish the case.
- Rule typing-unfold. In this case, e is decomposed into unfold $[\alpha.\ A]\ e$. By inversion, we will get two sub-cases. Case rule step-unfold is trivial. As for case rule step-fld, we do inversion again, raising two sub-cases (rule step-fold and rule step-fld). The former one is trivial. In latter case, we have premises $\Gamma \vdash \mathsf{fold}\ [A]\ v : B, \Gamma \vdash B \le \mu\alpha. C$ and goal $\Gamma \vdash v : [\alpha \mapsto \mu\alpha. C]\ C$. Doing induction on $\Gamma \vdash \mathsf{fold}\ [A]\ v : B$, by unfolding lemma (Lemma 7), we obtain a type C' such that $\Gamma \vdash v : [\alpha \mapsto \mu\alpha.\ C']\ C'$ and $\Gamma \vdash [\alpha \mapsto \mu\alpha.\ C']\ C' \le [\alpha \mapsto \mu\alpha.\ C]\ C$. By applying rule typing-sub, we prove our goal.

□

Theorem 10. Progress.

$$\text{If } \vdash e : A \text{ then } e \text{ is a value or exists } e', e \hookrightarrow e'.$$

4 ALGORITHMIC SUBTYPING WITH THE DOUBLE UNFOLDING RULE

In last section we introduced a declarative formulation of subtyping with recursive types. Unfortunately, such formulation is not directly implementable since the rule of subtyping for recursive checks against an infinite number of conditions (that all finite unfoldings are subtypes). In this section, we present a sound and complete algorithmic formulation of subtyping. This formulation

$\boxed{\Gamma \vdash_a A \leq B}$ *(Algorithmic subtyping)*

$$\textsc{SA-nat}\ \frac{\vdash \Gamma}{\Gamma \vdash_a \mathsf{nat} \leq \mathsf{nat}} \qquad \textsc{SA-top}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash_a A \leq \top} \qquad \textsc{SA-var}\ \frac{\vdash \Gamma \qquad \alpha \in \Gamma}{\Gamma \vdash_a \alpha \leq \alpha} \qquad \textsc{SA-arrow}\ \frac{\Gamma \vdash_a B_1 \leq A_1 \qquad \Gamma \vdash_a A_2 \leq B_2}{\Gamma \vdash_a A_1 \to A_2 \leq B_1 \to B_2}$$

$$\textsc{SA-rec}\ \frac{\Gamma, \alpha \vdash_a A \leq B \qquad \Gamma, \alpha \vdash_a [\alpha \mapsto A]\ A \leq [\alpha \mapsto B]\ B}{\Gamma \vdash_a \mu\alpha.\ A \leq \mu\alpha.\ B}$$

Fig. 5. Subtyping Rules for Algorithmic Type System

replaces the declarative rule S-rec by the double unfolding rule, which unfolds the recursive types 1-time and 2-times, respectively.

4.1 Syntax, Well-Formedness and Subtyping

The syntax and well-formedness of the algorithmic system share the same definition as the declarative system presented in Section 3.

Subtyping. Figure 5 shows the algorithmic subtyping judgment. All rules, except one for recursive types, remain the same as the declarative system. In the algorithmic subtyping, two recursive types are subtypes when: 1) their bodies are subtypes; and 2) unfolding the bodies one additional time preserves subtyping. In other words, checking 1-time and 2-times finite unfoldings rather than all finite unfoldings (as what rule S-rec does) is sufficient.

4.2 Reflexivity, Transitivity and Completeness

Our algorithmic subtyping simply relaxes the condition for recursive types while keeping the judgment form. Therefore, regularity, reflexivity and transitivity are easy to prove using similar techniques as the declarative system.

Lemma 11. Regularity: If $\Gamma \vdash_a A \leq B$ then $\Gamma \vdash A$ and $\Gamma \vdash B$.

Theorem 12. Reflexivity.

$$\text{If } \Gamma \vdash A \text{ then } \Gamma \vdash_a A \leq A.$$

Theorem 13. Transitivity.

$$\text{If } \Gamma \vdash_a A \leq B \text{ and } \Gamma \vdash_a B \leq C \text{ then } \Gamma \vdash_a A \leq C.$$

Note that, like the declarative system (and unlike the Amber rules), the transitivity proof is very simple with the double unfolding rule. The completeness of algorithmic subtyping is obvious, since the declarative system is has the same conditions of the algorithmic system (plus a few more).

Theorem 14. Completeness of algorithmic subtyping: If $\Gamma \vdash A \leq B$ then $\Gamma \vdash_a A \leq B$.

4.3 Soundness

The real challenge is the soundness of the algorithmic specification with respect to the declarative system. For soundness, we wish to prove that:

$$\text{If } \Gamma \vdash_a A \leq B \text{ then } \Gamma \vdash A \leq B.$$

The key problem is to show that finitely unfolding only one and two times is sufficient to guarantee that all finite unfoldings are sound. Although it is easy to give an informal argument as to why this is the case, as we did in Section 2, formalizing this argument is a whole different matter.

An overview of the key idea. The key idea to prove that 1-time and 2-times finite unfolding implies n-times finite unfolding is to capture this informal idea formally as a lemma:

$$\Gamma \vdash A \leq B \ \wedge \ \Gamma \vdash [\alpha \mapsto A]\ A \leq [\alpha \mapsto B]\ B \ \Rightarrow \ \Gamma \vdash [\alpha \mapsto A]^n\ A \leq [\alpha \mapsto B]^n\ B.$$

As we shall see this lemma is true but, unfortunately, it cannot be proved directly. The obvious attempt would be to do induction on $\Gamma \vdash A \leq B$. The essential problem is that we wish to analyse the different subcases for A and B, but we still want to use the original A and B in the substitutions. For instance, suppose that we have $A := nat \to A_1 \to A_2$ and $B := nat \to B_1 \to B_2$. Here $A_1 \to A_2$ and $B_1 \to B_2$ are contained in the type A and B. Now consider the case for function types $\Gamma \vdash A_1 \to A_2 \leq B_1 \to B_2$, which would occur in the proof. What we would like to have is the conclusion $\Gamma \vdash [\alpha \mapsto A]^n\ (A_1 \to A_2) \leq [\alpha \mapsto B]^n\ (B_1 \to B_2)$. However, what we get instead is $\Gamma \vdash [\alpha \mapsto (A_1 \to A_2)]^n\ (A_1 \to A_2) \leq [\alpha \mapsto (B_1 \to B_2)]^n\ (B_1 \to B_2)$. Therefore it is clear that we need some generalization of this lemma. A first idea is to generalize it as follows:

$$\begin{array}{lc} & \Gamma \vdash A \leq B \ \wedge \ \Gamma \vdash C \leq D \ \wedge \ \Gamma \vdash [\alpha \mapsto C]\ A \leq [\alpha \mapsto D]\ B \\ \Rightarrow & \Gamma \vdash [\alpha \mapsto C]^n\ A \leq [\alpha \mapsto D]^n\ B. \end{array}$$

Now it is possible to do induction on $\Gamma \vdash A \leq B$ without affecting the substituted types. However, this lemma is *false*. A counter-example is:

$$\begin{array}{lc} & \Gamma \vdash \top \to \alpha \leq \mathsf{nat} \to \alpha \ \wedge \ \Gamma \vdash \alpha \to \mathsf{nat} \leq \alpha \to \top \ \wedge \ \Gamma \vdash \top \to \alpha \to \mathsf{nat} \leq \mathsf{nat} \to \alpha \to \top \\ \not\Rightarrow & \Gamma \vdash \top \to (\alpha \to \mathsf{nat}) \to \mathsf{nat} \leq \mathsf{nat} \to (\alpha \to \top) \to \top. \end{array}$$

In this counter-example we choose $n = 2$. All the premises are satisfied, but the conclusion is false. Note that in the conclusion, because of the contravariance of function subtyping, we eventually require that $\Gamma \vdash \alpha \to \top \leq \alpha \to \mathsf{nat}$, which is clearly false.

Further analysis of the counter-example reveals an important problem of our first idea: the types A and C, and B and D are completely unrelated. However, we know that they ought to have some relationship. Indeed, we know that $\Gamma \vdash A \leq B$ should be a subderivation of $\Gamma \vdash C \leq D$. This is not the case in the counter-example. Thus, to capture such relationship precisely, we introduce a subderivation relation:

$$\Gamma_1, \Gamma_2 \vdash A \leq B \in_m C \leq D$$

This relation is key to obtain a suitable generalization of the lemma, without losing the relationship between A and C, and B and D. The relation states that $\Gamma_1, \Gamma_2 \vdash A \leq B$ is a positive or negative subderivation of $\Gamma_1 \vdash C \leq D$. A positive subderivation is a subderivation that arises from covariant uses of the subtyping relation. Conversely, a negative subderivation is a subderivation that arises from contravariant uses of the subtyping relation. The polarity of the subderivation is captured by the mode m, which can either be $+$ or $-$ denoting, respectively, a positive or negative subderivation. The following 3 examples illustrate the subderivation relation:

(1) $\Gamma_1, \cdot \vdash \alpha \leq \alpha \in_+ \top \to \alpha \leq \mathsf{nat} \to \alpha$
(2) $\Gamma_1, \cdot \vdash \mathsf{nat} \leq \top \in_- \top \to \alpha \leq \mathsf{nat} \to \alpha$
(3) $\Gamma_1, \cdot \vdash \alpha \to \mathsf{nat} \leq \alpha \to \top \notin_m \top \to \alpha \leq \mathsf{nat} \to \alpha$

The first example captures the fact that $\Gamma_1 \vdash \alpha \leq \alpha$ is a positive subderivation of $\Gamma_1 \vdash \top \to \alpha \leq \mathsf{nat} \to \alpha$. The second example illustrates a negative subderivation: $\Gamma_1 \vdash \mathsf{nat} \leq \top$ is a subderivation of $\Gamma_1 \vdash \top \to \alpha \leq \mathsf{nat} \to \alpha$. Note that, because of contravariance, the nat and $\top$ types occur in

$\boxed{\Gamma_1, \Gamma_2 \vdash A \leq B \in_m C \leq D}$ *(Subtyping Subderivation)*

$$\textsc{Der-refl}\ \frac{\Gamma_1 \vdash A \leq B}{\Gamma_1, \cdot \vdash A \leq B \in_+ A \leq B} \qquad \textsc{Der-funL}\ \frac{\Gamma_1, \Gamma_2 \vdash A_1 \to A_2 \leq B_1 \to B_2 \in_m C \leq D}{\Gamma_1, \Gamma_2 \vdash B_1 \leq A_1 \in_{\overline{m}} C \leq D}$$

$$\textsc{Der-funR}\ \frac{\Gamma_1, \Gamma_2 \vdash A_1 \to A_2 \leq B_1 \to B_2 \in_m C \leq D}{\Gamma_1, \Gamma_2 \vdash A_2 \leq B_2 \in_m C \leq D} \qquad \textsc{Der-rec}\ \frac{\Gamma_1, \Gamma_2 \vdash \mu\alpha.\, A \leq \mu\alpha.\, B \in_m C \leq D}{\Gamma_1, \Gamma_2, \alpha \vdash [\alpha \mapsto A]^n\, A \leq [\alpha \mapsto B]^n\, B \in_m C \leq D}$$

$\boxed{\Gamma \vdash^{\alpha}_{m} A \leq B}$ *(Negative Subtyping)*

$$\textsc{NTyp-var}\ \frac{}{\Gamma \vdash^{\alpha}_{-} \alpha \leq \alpha} \qquad \textsc{NTyp-funL}\ \frac{\Gamma \vdash^{\alpha}_{\overline{m}} B_1 \leq A_1}{\Gamma \vdash^{\alpha}_{m} A_1 \to A_2 \leq B_1 \to B_2} \qquad \textsc{NTyp-funR}\ \frac{\Gamma \vdash^{\alpha}_{m} A_2 \leq B_2}{\Gamma \vdash^{\alpha}_{m} A_1 \to A_2 \leq B_1 \to B_2}$$

$$\textsc{NTyp-rec}\ \frac{\Gamma, \alpha \vdash^{\alpha'}_{m} [\alpha \mapsto A]^n\, A \leq [\alpha \mapsto B]^n\, B \quad \forall n = 1 \cdots \infty}{\Gamma \vdash^{\alpha'}_{m} \mu\alpha.\, A \leq \mu\alpha.\, B}$$

Fig. 6. The subtyping derivation relation and negative subtyping relation.

a different side in the subderivation. Finally, the last example shows why the previous counter-example fails to be a subderivation: it is not possible to derive that $\Gamma_1 \vdash \alpha \to \mathsf{nat} \leq \alpha \to \top$ is a subderivation of $\Gamma_1 \vdash \top \to \alpha \leq \mathsf{nat} \to \alpha$.

An important special case of subderivations is: $\Gamma_1, \Gamma_2 \vdash \alpha \leq \alpha \in_- C \leq D$. That is when one of the negative subderivations of $\Gamma_1 \vdash C \leq D$ is $\Gamma_1, \Gamma_2 \vdash \alpha \leq \alpha$, for some α. In particular if $C := \mu\alpha.\, A$ and $D := \mu\alpha.\, B$, and $\Gamma_1, \Gamma_2 \vdash \alpha \leq \alpha$ occurs in a negative subderivation of $\Gamma_1 \vdash \mu\alpha.\, A \leq \mu\alpha.\, B$ then we can conclude that $A = B$ (i.e. the two recursive types must be equal). In essence 1-time and 2-times finite unfolding on types where $\alpha \leq \alpha$ occurs negatively will need to eventually check both that $A \leq B$ and $B \leq A$ hold, which can only be true when $A = B$.

The idea of subderivations of subtyping and the special case of negative subderivations containing $\Gamma_1, \Gamma_2 \vdash \alpha \leq \alpha$ plays a fundamental role in the proof of soundness, which we explain in detail next.

Subtyping subderivation. The full definition of the subtyping subderivation relation is shown in the top of Figure 6. Note in such relation we have two contexts: Γ_1 and Γ_2. Γ_1 is the original context for relation $\Gamma_1 \vdash C \leq D$, and Γ_2 is used to record new variables that appear in subderivations. The rule Der-refl is the base case: if types A and B are in subtyping relation, then the derivation tree starts from $\Gamma_1 \vdash A \leq B$. The rule Der-funL and rule Der-funR denote the cases where the subderivations arise from the contravariant and covariant cases of subtyping two function types, respectively. Note that in the contravariant case (rule Der-funL) the mode is flipped. The notation $\overline{m}$ denotes a simple function that flips the mode ($\overline{+} \equiv -$ and $\overline{-} \equiv +$). The rule Der-rec deals with recursive case, which is the most interesting one. If two recursive types are in subtyping relation then unfolding the bodies n times (for some n) is a subderivation of subtyping. The subtyping subderivation relation has the following useful (and expected) property:

Lemma 15. If $\Gamma_1, \Gamma_2 \vdash A \leq B \in_m C \leq D$ then $\Gamma_1 \vdash C \leq D$ and $\Gamma_1, \Gamma_2 \vdash A \leq B$.

Negative subtyping. Before we finalize the discussion of the soundness lemma, we need another auxiliary definition for capturing the notion of negative subtyping. Negative subtyping captures subtyping derivations where types contain a free variable α and the subtyping derivation $\Gamma \vdash \alpha \leq \alpha$ occurs negatively. This relation is useful to prove lemmas about subtyping with such subderivations. The relation is shown in the bottom of Figure 6. The rule NTyp-var is the base case: a subderivation $\Gamma \vdash \alpha \leq \alpha$ occurs negatively. The rule NTyp-funR states that if variable α occurs in some mode m in $\Gamma \vdash A_2 \leq B_2$, then it occurs in $\Gamma \vdash A_1 \to A_2 \leq B_1 \to B_2$ in the same mode. The rule NTyp-funL, similarly, states that if variable α occurs in $\Gamma \vdash A_2 \leq B_2$ in mode m, then it occurs in $\Gamma \vdash A_1 \to A_2 \leq B_1 \to B_2$ in the reverse mode. The rule NTyp-rec is the rule for recursive types. The subtyping derivation preserves the mode. The more interesting aspect of rule NTyp-rec is that it adds a new variable α' to the context (which is assumed to be distinct from α). One useful lemma about negative subtyping is:

Lemma 16. (Properties of negative subtyping)

(1) If $\Gamma \vdash^{\alpha}_{+} A \leq B$ and $\Gamma \vdash [\alpha \mapsto C]\ A \leq [\alpha \mapsto D]\ B$, then $\Gamma \vdash D \leq C$.
(2) If $\Gamma \vdash^{\alpha}_{-} A \leq B$ and $\Gamma \vdash [\alpha \mapsto D]\ A \leq [\alpha \mapsto C]\ B$, then $\Gamma \vdash D \leq C$.

If a derivation has negative subtyping derivations of α then we know that substitutions of α will actually be performed in matching positions in the type. Furthermore, because we know the polarity at which $\Gamma \vdash \alpha \leq \alpha$ occurs, then we can conclude that there is a subtyping relationship between the substituted types.

Another important property of the subtyping relation is:

Lemma 17. If $\Gamma \vdash A \leq B$ and $\Gamma \vdash B \leq A$ then $A = B$.

Equipped with both negative subtyping and the subtyping subderivation relations we can prove an important lemma:

Lemma 18. If

(1) $\Gamma_1, \Gamma_2 \vdash A \leq B \in_m C \leq D$;
(2) $\Gamma_1, \Gamma_2 \vdash^{\alpha}_{m} A \leq B$;
(3) α occurs in both C and D;
(4) $\Gamma_2 \vdash [\alpha \mapsto C]^n\ C \leq [\alpha \mapsto D]^n\ D$;
(5) $\Gamma_2 \vdash [\alpha \mapsto C]\ C \leq [\alpha \mapsto D]\ D$,

then $\Gamma_2 \vdash [\alpha \mapsto D]^n\ D \leq [\alpha \mapsto C]^n\ C$.

Proof. By induction on $\Gamma_1, \Gamma_2 \vdash A \leq B \in_m C \leq D$. The cases for function and recursive types (rule Der-funL, rule Der-funR and rule Der-rec) are proven by applying hypothesis, respectively. Thus we only talk about case rule Der-refl. In such case, we know that $m := +$, $A = C$ and $B = D$. Therefore, we have that $\Gamma_1 \vdash A \leq B$. Next, apply Lemma 16 to $\Gamma_1 \vdash^{\alpha}_{+} A \leq B$, getting $\Gamma_1 \vdash B \leq A$. Thus, we conclude $A = B$ by Lemma 17 and accordingly $C = D$. Finally, apply *reflexivity*. □

Soundness, finally! Having set up the basic infrastructure regarding the subtyping derivation and negative subtyping relations, we can finally focus on the goal of soundness. Lemma 18 is an important lemma that covers cases appearing in the soundness lemma when we have negative subtyping derivations. However, we still need a lemma dealing with subtyping derivations in general. With the help of Lemma 18, we prove:

Lemma 19. If

(1) $\Gamma_1, \Gamma_2 \vdash A \leq B$;
(2) $\Gamma_1, \Gamma_2 \vdash A \leq B \in_m C \leq D$;
(3) α occurs in both C and D

(4) $\Gamma_1 \vdash [\alpha \mapsto C]^n\ C \leq [\alpha \mapsto D]^n\ D$;
(5) $\Gamma_1 \vdash [\alpha \mapsto C]\ C \leq [\alpha \mapsto D]\ D$,

then

(1) $\Gamma_1, \Gamma_2 \vdash [\alpha \mapsto C]^{n+1}\ A \leq [\alpha \mapsto D]^{n+1}\ B$ when $m := +$;
(2) $\Gamma_1, \Gamma_2 \vdash [\alpha \mapsto D]^{n+1}\ A \leq [\alpha \mapsto C]^{n+1}\ B$ when $m := -$.

Now we can finally prove that 1-time and 2-times finite unfolding implies arbitrary unfolding:

Lemma 20. If $\Gamma \vdash A \leq B$ and $\Gamma \vdash [\alpha \mapsto A]\ A \leq [\alpha \mapsto B]\ B$ then $\Gamma \vdash [\alpha \mapsto A]^n\ A \leq [\alpha \mapsto B]^n\ B$.

Proof. By induction on n. For inductive step, apply Lemma 19 with $C := A, D := B, \Gamma_2 := \cdot, m := +$. □

The form of Lemma 20 is close to the shape of the infinite unfolding rule (rule S-REC) in declarative recursive types. Finally, we can prove the soundness theorem:

Theorem 21. Soundness of algorithmic subtyping.

$$\text{If } \Gamma \vdash_a A \leq B \text{ then } \Gamma \vdash A \leq B.$$

4.4 Decidability

Our subtyping rules are syntax-directed, leading to a relatively simple proof of decidability. We found that doing the proof/induction directly on the declarative system would lead to fewer cases in the proof of decidability. The reason is that the algorithmic rule for recursive types has 2 premises, whereas the declarative system has only 1. Thus, some surgical uses of soundness/completeness in the proof were enough to make it work, while saving us from dealing with extra cases compared to a proof using the algorithmic system instead.

Theorem 22. Decidability.

$$\text{If } \vdash \Gamma,\ \Gamma \vdash A \text{ and } \Gamma \vdash B \text{ then } \Gamma \vdash A \leq B \text{ or } \Gamma \vdash A \nleq B.$$

Proof. By nested induction on $\Gamma \vdash A$ and $\Gamma \vdash B$ (using the variant with rule WFT-INF). □

5 SOUNDNESS OF THE AMBER RULES

This section shows a variant of the Amber rules that is sound with respect to our new formulation of subtyping. The soundness lemma implies that if two types are subtypes under the Amber rules, they are subtypes under our new formulation. In other words our new subtyping relation is at least as good as the Amber rules in terms of expressiveness. To establish the soundness result we have to impose some well-formedness conditions. These conditions have been omitted in early formulations of the Amber rules (as mentioned in Section 2.4), but are necessary here to come up with precise results regarding the metatheory.

5.1 The Challenges of Well-Formedness for the Amber Rules

In the original Amber rules by Amadio and Cardelli [1993] (Figure 2) there are no well-formedness constraints. Unfortunately, defining such well-formedness constraints is not entirely trivial. Furthermore, for those interested in mechanical formalizations using theorem provers (as we are), such details need to be spelled out clearly. Well-formedness usually plays an important role in the metatheory, since some proofs can be more easily proved by considering well-formed types and environments only. One typical property of subtyping that we may hope to have is the so-called regularity of subtyping:

$$\text{If } \Gamma \vdash A \leq B \text{ then } \Gamma \vdash A \wedge \Gamma \vdash B.$$

which states that if a subtyping derivation is valid then the types are well-formed. Regularity is typically used in many other proofs, such as the proof of transitivity in algorithmic formulations. Note that, in the Amber rules, the rule for recursive subtypes uses two distinct type variables α and β in the recursive types. The use of such distinct type variables is crucial feature of the Amber rules and is used to prevent subderivations of the form $\Gamma \vdash \beta \leq \alpha$, where Γ only contains $\alpha \leq \beta$ but not $\beta \leq \alpha$. Otherwise, if such subderivations would be accepted, type soundness would be broken.

With the Amber rules an intuitive idea is that the subtyping environment consists of a sequence of pairs of types $\overline{\alpha \leq \beta}$ and that the α's are in scope on the type at the left-side of the subtyping relation (A), while the β's are in scope on the type at the right-side of the subtyping relation (B). Sadly, this idea is not that simple to realise. Note that in the subtyping rule of function types (rule Amber-arrow), the input arguments are swapped, so without any changes on the environment the type variables in the types would go out-of scope, and this breaks the regularity lemma. Furthermore, trying to perhaps swap the variables in the environment to keep them in-scope changes the meaning of the environment ($\alpha \leq \beta$ becomes $\beta \leq \alpha$). Trying to ensure that the α's are only in scope in one side of the relation, while the β's are only in scope on the other side, turns out to be quite tricky. Therefore, to make progress, we propose a weaker restriction in this section: we allow that both α's and β's are in scope for both types. Thus, the following derivation is valid with our variant of the Amber rules: $\alpha \leq \beta \vdash \alpha \rightarrow \beta \leq \top$. In other words, we accept some subtyping derivations that one would perhaps expect to be rejected. That is, in the Amber rules, if we have $\alpha \leq \beta$ in Γ, we would not expect that α and β appear on the same type. Rather we would expect that the α appears in one of the types, and β on the other one. However, accepting such subtyping derivations is not harmful: we can still prove the soundness of this variant with respect to our new formulation of subtyping.

5.2 Well-Formedness and Subtyping

In the Amber rules, the subtyping context stores pairs of distinct type variables. We use $\Delta := \cdot \mid \Delta, \alpha \leq \beta$ to denote the context for Amber rules. Figure 7 shows a set of standard Amber rules with a built-in reflexivity rule.

Well-formedness. A well-formed environment ($\vdash \Delta$) requires that all pairs of variables ($\alpha \leq \beta$) in the environment Δ are distinct. Well-formed types are almost standard, except that both α and β are considered declared by a pair ($\alpha \leq \beta$) in the context (rule WFAmber-varL and rule WFAmber-varR), and rule WFAmber-rec introduces a pair of fresh variables into the context, although the second variable is never used. Rule WFAmber-rec simply mimics the left-hand side derivation of rule Amber-rec of the Amber subtyping relation, as we shall see next.

Subtyping. The subtyping relation is almost the same as the original rules by Amadio and Cardelli [1993] in Figure 2. The noticeable difference is the addition of various well-formedness checks in various rules. For instance, base cases such as rule Amber-nat and rule Amber-top check whether the environments are well-formed. Moreover, in rule Amber-self we require the recursive type to be well-formed ($\Delta \vdash \mu\alpha.\ A$).

5.3 Towards Soundness: A Third Subtyping Relation Based on a Positive Restriction

To prove soundness with respect to our own formulation of subtyping we create an intermediate subtyping relation to make the proof easier. This intermediate relation, presented in Figure 8, is equivalent to the Amber rules in Figure 7. The key idea in this relation is to have two rules for recursive types: one rule (rule PosRes-rec) only accepts positive subtyping, which is inspired by monotonicity on recursive types [Amadio and Cardelli 1993; Appel and Felty 2000; Backes et al.

$\boxed{\Delta \vdash A}$ *(Well-formed Type of Amber rules)*

$$\textsc{WFAmber-nat}\ \frac{\vdash \Delta}{\Delta \vdash \mathsf{nat}} \qquad \textsc{WFAmber-Top}\ \frac{\vdash \Delta}{\Delta \vdash \top} \qquad \textsc{WFAmber-varL}\ \frac{\vdash \Delta \quad \alpha \le \beta \in \Delta}{\Delta \vdash \alpha} \qquad \textsc{WFAmber-varR}\ \frac{\vdash \Delta \quad \alpha \le \beta \in \Delta}{\Delta \vdash \beta}$$

$$\textsc{WFAmber-arrow}\ \frac{\Delta \vdash A_1 \quad \Delta \vdash A_2}{\Delta \vdash A_1 \rightarrow A_2} \qquad \textsc{WFAmber-rec}\ \frac{\Delta, \alpha \le \beta \vdash A \quad \beta \text{ is fresh}}{\Delta \vdash \mu\alpha.\ A}$$

$\boxed{\Delta \vdash_{amb} A \le B}$ *(Amber rules)*

$$\textsc{Amber-nat}\ \frac{\vdash \Delta}{\Delta \vdash_{amb} \mathsf{nat} \le \mathsf{nat}} \qquad \textsc{Amber-top}\ \frac{\vdash \Delta \quad \Delta \vdash A}{\Delta \vdash_{amb} A \le \top} \qquad \textsc{Amber-var}\ \frac{\vdash \Delta \quad \alpha \le \beta \in \Delta}{\Delta \vdash_{amb} \alpha \le \beta}$$

$$\textsc{Amber-arrow}\ \frac{\Delta \vdash_{amb} B_1 \le A_1 \quad \Delta \vdash_{amb} A_2 \le B_2}{\Delta \vdash_{amb} A_1 \rightarrow A_2 \le B_1 \rightarrow B_2} \qquad \textsc{Amber-rec}\ \frac{\Delta, \alpha \le \beta \vdash_{amb} A \le B}{\Delta \vdash_{amb} \mu\alpha.\ A \le \mu\beta.\ B} \qquad \textsc{Amber-self}\ \frac{\vdash \Delta \quad \Delta \vdash \mu\alpha.\ A}{\Delta \vdash_{amb} \mu\alpha.\ A \le \mu\alpha.\ A}$$

Fig. 7. A variant of the Amber rules, including well-formedness of types.

$\boxed{\alpha \in_m A \le_+ B}$ *(Positive restriction)*

$$\textsc{Posvar-nat}\ \frac{}{\alpha \in_m \mathsf{nat} \le_+ \mathsf{nat}} \qquad \textsc{Posvar-topL}\ \frac{}{\alpha \in_m A \le_+ \top} \qquad \textsc{Posvar-topR}\ \frac{}{\alpha \in_m \top \le_+ A} \qquad \textsc{Posvar-varX}\ \frac{}{\alpha \in_+ \alpha \le_+ \alpha} \qquad \textsc{Posvar-varY}\ \frac{\alpha \neq \beta}{\alpha \in_m \beta \le_+ \beta}$$

$$\textsc{Posvar-arrow}\ \frac{\alpha \in_{\overline{m}} B_1 \le_+ A_1 \quad \alpha \in_m A_2 \le_+ B_2}{\alpha \in_m A_1 \rightarrow A_2 \le_+ B_1 \rightarrow B_2} \qquad \textsc{Posvar-rec}\ \frac{\beta \in_m A \le_+ B \quad \alpha \in_+ A \le_+ B \quad \alpha \neq \beta}{\beta \in_m \mu\alpha.\ A \le_+ \mu\alpha.\ B} \qquad \textsc{Posvar-recself}\ \frac{\beta \notin fv(A)}{\beta \in_m \mu\alpha.\ A \le_+ \mu\alpha.\ A}$$

$\boxed{\Gamma \vdash A \le_+ B}$ *(Positive subtyping)*

$$\textsc{PosRes-nat}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{nat} \le_+ \mathsf{nat}} \qquad \textsc{PosRes-top}\ \frac{\vdash \Gamma \quad \Gamma \vdash A}{\Gamma \vdash A \le_+ \top} \qquad \textsc{PosRes-var}\ \frac{\vdash \Gamma \quad \alpha \in \Gamma}{\Gamma \vdash \alpha \le_+ \alpha} \qquad \textsc{PosRes-self}\ \frac{\vdash \Gamma \quad \Gamma \vdash \mu\alpha.\ A}{\Gamma \vdash \mu\alpha.\ A \le_+ \mu\alpha.\ A}$$

$$\textsc{PosRes-arrow}\ \frac{\Gamma \vdash B_1 \le_+ A_1 \quad \Gamma \vdash A_2 \le_+ B_2}{\Gamma \vdash A_1 \rightarrow A_2 \le_+ B_1 \rightarrow B_2} \qquad \textsc{PosRes-rec}\ \frac{\Gamma, \alpha \vdash A \le_+ B \quad \alpha \in_+ A \le_+ B}{\Gamma \vdash \mu\alpha.\ A \le_+ \mu\alpha.\ B}$$

Fig. 8. Positive Subtyping Rules

2014]; while the second rule (rule PosRes-self) is a reflexivity rule that accepts recursive types with negative recursive variables as well.

Most subtyping rules are identical to those of the Amber rules, and the only differences are rule PosRes-var and rule PosRes-rec. Rule PosRes-rec checks that the recursive type variable α satisfies the positive restriction relation $\alpha \in_+ \; A \leq_+ \; B$, while maintaining the same behavior as the Amber rules. This subtle relation may be interpreted as follows: $\alpha \in_+ A \leq_+ B$ means 1) $\alpha \leq \alpha$ does not appear in subderivations of $A \leq B$, or 2) $\alpha \leq \alpha$ appears in subderivations of $A \leq B$ only in positive positions. For example, $\alpha \in_+ \top \to \alpha \leq_+ \alpha \to \alpha$ holds; yet $\alpha \in_+ \mu\beta.\, \beta \to \alpha \leq_+ \mu\beta.\, \beta \to \alpha$ does not hold, since after unfolding for several times, α's on both sides appear on the same negative position. The following lemma reveals an important property that relates the positive restriction to double unfolding:

Lemma 23. If $\alpha \in_m A \leq_+ B$ and $\beta \in_+ A \leq_+ B$ then $\alpha \in_m [\beta \mapsto A]\, A \leq_+ [\beta \mapsto B]\, B$.

This lemma tells us that the positive restriction respects the mode on non-negative substitutions. As a result, the positive restriction not only reflects the Amber rules but also connects to our double-unfolding formulation of subtyping.

5.4 The Soundness Theorem

To show that the Amber subtyping is sound with respect to our subtyping relations, we need to translate the environments and types of Amber rules, as they are defined under different forms in our systems.

Definition 24. Translation of environments and types from the Amber rules.

$$\begin{array}{rlrl} |\cdot| &= \cdot & (\cdot)(A) &= A \\ |\Delta,\ \alpha \leq \beta| &= |\Delta|,\ \alpha & (\Delta,\ \alpha \leq \beta)(A) &= (\Delta)([\beta \mapsto \alpha]\ A) \end{array}$$

The translation functions, $|\cdot|$ and $(\cdot)(A)$, simply drop every second variable defined in the context Δ. For example, a subtyping judgment in the Amber system $\alpha \leq \beta \vdash \alpha \to \top \leq \beta \to \top$ is translated to $\alpha \vdash \alpha \to \top \leq \alpha \to \top$.

The relationship between the Amber subtyping and our subtyping with the positive restriction is shown by the following lemma, which is relatively straightforward:

Lemma 25. If $\Delta \vdash_{amb} A \leq B$ then $|\Delta| \vdash (\Delta)(A) \leq_+ (\Delta)(B)$.

We are now one step away from the soundness theorem: to prove that positive subtyping implies double unfolding subtyping. The main difference is on rule PosRes-rec, which corresponds to rule SA-rec in the double unfolding subtyping. The proof is tricky at the first glance, yet with Lemma 23, one can derive the following lemma:

Lemma 26. If $\alpha \in_+ A \leq_+ B$ and $\Gamma \vdash_a A \leq B$ then $\Gamma \vdash_a [\alpha \mapsto A]\, A \leq [\alpha \mapsto B]\, B$.

With the above lemma, the relation between positive subtyping and the algorithmic double unfolding subtyping is easy to establish:

Lemma 27. If $\Gamma \vdash A \leq_+ B$ then $\Gamma \vdash_a A \leq B$.

Combining Lemma 25, 26 and 27, we have

Theorem 28. Soundness of the Amber rules.

$$\text{If } \Delta \vdash_{amb} A \leq B \text{ then } |\Delta| \vdash_a (\Delta)(A) \leq (\Delta)(B).$$

Table 1. Paper-to-proofs correspondence guide.

Definition	File	Name in formalization	Notation in the paper
Well-formed Type (Figure 3)	definition.v	WF E A	$\Gamma \vdash A$
Well-formed Type (Definition 2)	definition.v	WFS E A	$\Gamma \vdash A$
Declarative subtyping (Figure 3)	definition.v	Sub E A B	$\Gamma \vdash A \leq B$
Typing (Figure 4)	definition.v	typing E e A	$\Gamma \vdash e : A$
Reduction (Figure 4)	definition.v	step e1 e2	$e_1 \hookrightarrow e_2$
Algorithmic subtyping (Figure 5)	definition.v	sub E A B	$\Gamma \vdash_a A \leq B$
Subtyping Subderivation (Figure 6)	definition.v	Der m E2 A B E1 C D	$\Gamma_1, \Gamma_2 \vdash A \leq B \in_m C \leq D$
Negative Subtyping (Figure 6)	definition.v	NTyp E m X A B	$\Gamma \vdash^{\alpha}_{m} A \leq B$
Well-formed Type of Amber rules (Figure 7)	amber_part_1.v	wf_amber E A	$\Delta \vdash A$
Amber rules (Figure 7)	amber_part_1.v	sub_amber E A B	$\Delta \vdash_{amb} A \leq B$
Positive restriction (Figure 8)	amber_part_1.v	posvar m X A B	$\alpha \in_m A \leq_+ B$
Positive subtyping (Figure 8)	amber_part_1.v	sub_amber2 E A B	$\Gamma \vdash A \leq_+ B$

Completeness. We have not proved completeness of the Amber rules with respect to our new formulation of subtyping. However, we conjecture that this result should hold. If completeness holds then our rules and the Amber rules have the same expressive power. However, showing the completeness of the Amber rules (or for that matter developing any of its associated metatheory) is quite challenging, and we have not managed to work out the proper formal argument for completeness. The soundness result that we have for the Amber rules is more important in practice than completeness, since it entails that a language design using our rules would not lose any expressive power compared to the Amber rules. Completeness would be a nice result to have (if it holds), but it does not have as much practical impact.

6 COQ PROOFS

We have chosen the Coq (8.10) proof assistant [The Coq Development Team 2019] to develop our formalization. The whole Coq formalization is built with a third-party library Metalib [1], which provides support for the locally nameless representation [Aydemir et al. 2008] to encode binders.

6.1 Definitions

All the definitions in the paper can be found in files *definition.v* and *amber_part_1.v*. Table 1 shows the correspondence of definitions between paper and the Coq artifacts. The file *definition.v* contains the definitions for STLC with an extension of iso-recursive types. It has definitions of well-formedness, subtyping (both declarative and algorithmic), typing, reduction, and additionally, negative subtyping and subtyping subderivation. The file *amber_part_1.v*, contains the definitions

[1]https://github.com/plclub/metalib

Table 2. Descriptions for the proof scripts.

Coq File	Theorems	Description
definition.v		The definitions of the SLTC extended with our recursive subtyping formulation.
infra.v	17	Infrastructure for locally nameless.
subtyping.v	3, 4, 5, 6, 7, 11, 12, 13, 14, 15, 16, 18, 19, 20, 21	Theorems about subtyping.
typesafety.v	8, 9, 10	Theorems about type soundness.
decidability.v	22	Theorems showing that our recursive subtyping is decidable.
amber_part_1.v	25	Definitions of Amber rules and positive restriction. Theorems showing that amber rules is sound w.r.t to the positive restriction.
amber_part_2.v	23, 26, 27, 28	Theorems showing that amber rules is sound w.r.t to our subtyping formulation.

for the Amber rules and the intermediate subtyping relation based on a positive restriction presented in Section 5.

For encoding variables and binders, we use the locally nameless representation to express all the types and terms. In paper, we use only *substitution* to represent *unfolding* of a recursive type. In the Coq proof, due to the use of the locally nameless representation, we also use of *opening* operation on pre-terms [Aydemir et al. 2008]. Furthermore, in the paper, we always use the same notation for well-formedness with rule WFT-REC or rule WFT-INF. In the Coq formalization, we have two distinct definitions of well-formedness, which are proved to be equivalent. WF E A is the one relation containing rule WFT-REC, and WFS E A is the other relation containing rule WFT-INF.

6.2 Lemmas and Theorems

Table 2 shows the descriptions for all the proof scripts. The theorems in Section 3 can be found at files *subtyping.v* and *typesafety.v*, the theorems in Section 4 can be found at files *subtyping.v* and *decidability.v*. Finally, the theorems in Section 5 can be found at files *amber_part_1.v* and *amber_part_2.v*.

An important difference between some of the lemma statements in the paper and the Coq proofs is that we make more use of modes in Coq. This change is done for readability purposes. In particular Lemma 16, 18 and 19 are stated without modes in the paper. For example, Lemma 16 is stated in the paper as:

(1) If $\Gamma \vdash_{+}^{\alpha} A \leq B$ and $\Gamma \vdash [\alpha \mapsto C]\ A \leq [\alpha \mapsto D]\ B$, then $\Gamma \vdash D \leq C$.
(2) If $\Gamma \vdash_{-}^{\alpha} A \leq B$ and $\Gamma \vdash [\alpha \mapsto D]\ A \leq [\alpha \mapsto C]\ B$, then $\Gamma \vdash D \leq C$.

but our Coq formalization uses some meta-functions on modes instead to formalize the same result. Using meta-functions on modes (Definition 29), the same lemma would look like:

If $\Gamma \vdash_{m}^{\alpha} A \leq B$ and $\Gamma \vdash [\alpha \mapsto C \oplus_m D]\ A \leq [\alpha \mapsto D \oplus_m C]\ B$, then $\Gamma \vdash D \leq C$.

Definition 29. Mode selector.

$$C \oplus_{+} D = C \quad C \oplus_{-} D = D$$

In the Coq proof, we defined some special notations for definitions representing n-times finite unfolding, and for the meta-functions on modes. Those definitions can be found in the file *definition.v*.

7 RELATED WORK

Throughout the paper we have already discussed some of the closest related work in detail. In this section we discuss other works on recursive subtyping.

Iso-recursive Amber rules. In Sections 2 and 5, we discussed Amadio and Cardelli [1993]'s study about recursive types. Their work is about equi-recursive types, which is enabled by a very expressive equivalence relation used in their reflexivity rule. Much of the follow-up work has employed a much weaker alpha-equivalence relation in the Amber rules, leading to an iso-recursive formulation of subtyping.

With respect to the metatheory of iso-recursive subtyping Amber rules, Bengtson et al. [2011]'s work is the closest to ours. They manually proved a full set of type safety properties, including the transitivity lemma for subtyping and the unfolding lemma (as a part of their inversion lemma). The transitivity lemma, "*perhaps the most difficult*" statement in their work, is proven with a complex inductive argument. For example, a subtyping chain of type variables, $\alpha_1 \le \alpha_2 \le \alpha_3$, is accepted by their transitivity statement, by means of adapting variable bindings in the contexts accordingly:

$$\frac{\Gamma[\alpha_1 \le \alpha_2] \vdash \alpha_1 \le \alpha_2 \qquad \Gamma[\alpha_2 \le \alpha_3] \vdash \alpha_2 \le \alpha_3}{\Gamma[\alpha_1 \le \alpha_3] \vdash \alpha_1 \le \alpha_3}$$

In other words, the subtyping judgments of their transitivity statement (used for proof) do not share the same context, which subtly captures the nature of context elements ($\alpha \le \beta$) in the Amber rules. Such technique involving inconsistent contexts is an uncommon practice, and it complicates the proof. Backes et al. [2014] attempted to formalize this transitivity proof in Coq, but they failed, stating that: "*The soundness of the Amber rule (Sub Rec) is hard to prove syntactically – in particular proving the transitivity of subtyping in the presence of the Amber rule requires a very complicated inductive argument, which only works for "executable" environments*".

Many other works avoid some of the complexity in the metatheory of the Amber rules by employing a declarative subtyping relation with transitivity built-in [Cardone 1991; Duggan 2002; Lee et al. 2015; Pottier 2013]. However, this leaves open the question of how to obtain a sound and complete algorithmic formulation, which as discussed in Sections 2 and 5, is non-trivial. Chugh [2015] observes the lack of some desirable properties (such as decidability) and difficulties of implementing languages modelling foundational aspects of Object-Oriented Programming when employing calculi with equi-recursive types. To address those difficulties he proposes a source calculus with iso-recursive types using the Amber rules, which enables decidability. He does not discuss transitivity of subtyping for the source calculus. Type-safety of the source calculus is shown via an elaboration into a target calculus with equi-recursive types and *F-bounded polymorphism* [Canning et al. 1989]. In general, those works employ elaboration and/or coercive subtyping, which leads to an alternative way to prove type-safety, and transitivity is either built-in or not discussed. In contrast, our metatheory comes with transitivity proofs, as well as a direct operational semantics for a calculus with iso-recursive types.

Other approaches to iso-recursive subtyping. Ligatti et al. [2017] propose an improvement to the Amber rules for iso-recursive subtyping. They observe that the Amber rules are sound, but incomplete with respect to type-safety. For example, the Amber rules reject $\mu\alpha.\ \{l : \mu\beta.\ \{l : \beta \to \mathsf{nat}\}, \{f : \mathsf{nat} \to \mathsf{nat}\}\} <: \mu\alpha.\ \{l : \alpha \to \mathsf{nat}\}$, but such a subtyping derivation does not violate type-safety (we assume a subtyping relation with record subtyping). Ligatti et al. [2017] propose a set of subtyping rules that is both sound and complete with respect to type-safety. The new formulation of subtyping proposed by us is also incomplete from the point of view of type-safety, since it remains close to the Amber rules. Our declarative formulation of subtyping is essentially following a syntactic approach to subtyping, whereas a formulation based on completeness with respect to type-safety is closer in spirit to *semantic subtyping* [Castagna and Frisch 2005]. While syntactic formulations are generally less expressive, their metatheory is usually simpler, and such formulations are also generally more modular. In particular, the new rules proposed by us for recursive subtyping are quite modular and can be added to an existing (syntactic) formulation of subtyping with minimal impact. In contrast, the more complete rules by Ligatti et al. [2017] require very specific environments for subtyping, and they also require some non-standard subtyping rules for *value-uninhabited* types. In terms of metatheory, their technique of *failing derivations* for proving reflexivity and transitivity is non-constructive and hard to formalize in a theorem prover.

For solving the conflict between contravariant types and recursive types, Hofmann and Pierce [1996] proposed an approach where only covariant types are allowed. In their subtyping rules, the inputs of function types must be the same. Later, Hosoya et al. [1998] gave an algorithm to prove transitivity and type soundness, but it still relies on a complicated environment where all of the components are pairs of structural recursive types. Thus, they have extra rules for contexts to obtain enough information for the subtyping assumptions. Featherweight Java [Igarashi et al. 2001], is another calculus that supports a form of iso-recursive types. Although there are no specific recursive type constructs, recursive types appear because class declarations can be recursive. An advantage of the Featherweight Java design is that recursive types are fairly easy to model, and modeling mutually recursive datatypes is straightforward. However, structural iso-recursive types, such as those in the Amber rules, allow for nested recursive types, which are not directly supported in Featherweight Java. Featherweight Java does support mutually recursive classes, so perhaps there is some general way to support such nested recursive types via an encoding.

Equi-recursive subtyping. Equi-recursive subtyping has been widely used in various calculi. With equi-recursive subtyping a recursive type is equivalent to its unfoldings. Amadio and Cardelli [1993]'s work provided the first theoretical foundation for equi-recursive types. Subsequent work by Brandt and Henglein [1997] and Gapeyev et al. [2003] improved and simplified the theory of Amadio and Cardelli [1993]'s study. In particular they advocated the use of coinduction for the metatheory of equi-recursive subtyping. Equi-recursive types play an important role in many areas. They have been employed for session types [Castagna et al. 2009; Chen et al. 2014; Gay and Hole 2005; Gay and Vasconcelos 2010], and Siek and Tobin-Hochstadt [2016] applied equi-recursive types in gradual typing. Dependent object types (DOT), the foundation of *Scala*, also considers a special form of equi-recursive type [Amin et al. 2016; Rompf and Amin 2016]. With conventional recursive types $\mu\alpha.\ A$, α stands for the recursive type itself. In DOT, the recursive type is of the form $\mu\ this.\ A$, where $this$ is the (run-time) self-reference. This construct, in combination with the form of dependent types supported in DOT allows for interesting applications that cannot be modelled with conventional recursive types. Nonetheless, DOT has to impose some contractiveness restrictions on the form of the recursive types for soundness, while no such restrictions are needed with iso-recursive types.

Mechanical formalizations with recursive subtyping. While to our knowledge there are no mechanical formalizations with the Amber rules, there are a few works trying to formalize other variants of recursive subtyping. Closest to our work is the Coq formalization by Backes et al. [2014]. They show a Coq proof for refinement types with a positive restriction for iso-recursive types. In fact, our positive subtyping formulation (Figure 8) is close to Backes et al. [2014]'s definition. However, our definition is more general since equal types with negative recursive occurrences are considered subtypes, whereas in their formulation recursive types with negative occurrences of recursive variables are forbidden. One subtle difference is that in their expressions syntax, unlike ours, unfold and fold expressions do not annotate a recursive type, which avoids the need for the unfolding lemma in the preservation proof, but it likely makes the type system undecidable. Appel and Felty [2000] gave a related Twelf proof of positive subtyping, where function types are invariant with respect to the input types of functions. Recently, based on big-step semantics, Amin and Rompf [2017] gave a formalization of DOT, which employs a special form of equi-recursive type. Danielsson and Altenkirch [2010], mixes induction and coinduction for proving properties of equi-recursive subtyping in Agda.

8 CONCLUSION

The Amber rules have been around for many years. They have been adapted and widely employed for iso-recursive formulations of subtyping. However the metatheory of Amber-style iso-recursive subtyping is not very well understood. In this work, we revisit the problem of iso-recursive subtyping and come up with novel declarative and algorithmic formulations of subtyping. We pay special attention to the metatheory, which is fully formalized in the Coq theorem prover. We believe that our work significantly improves the understanding of iso-recursive subtyping, and provides a platform for further developments in this area.

There are two interesting directions for future work. One obvious direction is to investigate the use of our novel formulation of iso-recursive subtyping in more complex subtyping relations. For instance it will be interesting to explore calculi with polymorphism, records, and intersection and union types. Another direction is to have a closer look at the alternative formulation of iso-recursive subtyping by Ligatti et al. [2017], and see whether the techniques developed in this paper can also help with a mechanical formalization of their work.

ACKNOWLEDGMENTS

We are grateful to anonymous reviewers that helped improving the presentation of our work. This work has been sponsored by Hong Kong Research Grant Council projects number 17210617 and 17209519.

REFERENCES

Roberto M Amadio and Luca Cardelli. 1993. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15, 4 (1993), 575–631.

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The essence of dependent object types. In *A List of Successes That Can Change the World*. Springer, 249–272.

Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 666–679.

Andrew W Appel and Amy P Felty. 2000. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 243–253.

Brian Aydemir, Arthur Charguéraud, Benjamin C Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering formal metatheory. *Acm sigplan notices* 43, 1 (2008), 3–15.

Michael Backes, Cătălin Hriţcu, and Matteo Maffei. 2014. Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations. *Journal of Computer Security* 22, 2 (2014), 301–353.

Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D Gordon, and Sergio Maffeis. 2011. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 2 (2011), 1–45.

Michael Brandt and Fritz Henglein. 1997. Coinductive axiomatization of recursive type equality and subtyping, Vol. 1210. 63–81. Full version in *Fundamenta Informaticae*, 33:309–338, 1998.

Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. 1989. F-Bounded Polymorphism for Object-Oriented Programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture* (Imperial College, London, United Kingdom) *(FPCA 1989)*. 8.

Luca Cardelli. 1985. Amber. In *LITP Spring School on Theoretical Computer Science*. Springer, 21–47.

Felice Cardone. 1991. Recursive types for Fun. *Theoretical Computer Science* 83, 1 (1991), 39–56.

Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Luca Padovani. 2009. Foundations of session types. In *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming.* 219–230.

Giuseppe Castagna and Alain Frisch. 2005. A Gentle Introduction to Semantic Subtyping. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '05).*

Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2014. On the preciseness of subtyping in session types. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming.* 135–146.

Ravi Chugh. 2015. IsoLATE: A type system for self-recursion. In *European Symposium on Programming Languages and Systems.* Springer, 257–282.

Dario Colazzo and Giorgio Ghelli. 1999. Subtyping recursive types in kernel fun. In *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158).* IEEE, 137–146.

Karl Crary, Robert Harper, and Sidd Puri. 1999. What is a Recursive Module?. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99).*

Nils Anders Danielsson and Thorsten Altenkirch. 2010. Subtyping, declaratively. In *International Conference on Mathematics of Program Construction.* Springer, 100–118.

Dominic Duggan. 2002. Type-safe linking with recursive DLLs and shared libraries. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 24, 6 (2002), 711–804.

Vladimir Gapeyev, Michael Levin, and Benjamin Pierce. 2003. Recursive Subtyping Revealed. *Journal of Functional Programming* 12, 6 (2003), 511–548. Preliminary version in *International Conference on Functional Programming (ICFP)*, 2000. Also appears as Chapter 21 of *Types and Programming Languages* by Benjamin C. Pierce (MIT Press, 2002).

Simon Gay and Malcolm Hole. 2005. Subtyping for session types in the pi calculus. *Acta Informatica* 42, 2-3 (2005), 191–225.

Simon J Gay and Vasco T Vasconcelos. 2010. Linear type theory for asynchronous session types. *Journal of Functional Programming* 20, 1 (2010), 19–50.

Giorgio Ghelli. 1993. Recursive types are not conservative over F≤. In *International Conference on Typed Lambda calculi and Applications.* Springer, 146–162.

Martin Hofmann and Benjamin C Pierce. 1996. Positive subtyping. *Information and Computation* 126, 1 (1996), 11–33.

Haruo Hosoya, Benjamin C Pierce, David N Turner, et al. 1998. Datatypes and subtyping. *Unpublished manuscript. Available http://www. cis. upenn. edu/˜ bcpierce/papers/index. html* (1998).

Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 3 (2001), 396–450.

Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin. 2015. A Theory of Tagged Objects. In *European Conference on Object-Oriented Programming (ECOOP).*

Jay Ligatti, Jeremy Blackburn, and Michael Nachtigal. 2017. On subtyping-relation completeness, with an application to iso-recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 39, 1 (2017), 1–36.

Benjamin C. Pierce. 2002. *Types and programming languages.* MIT press.

François Pottier. 2013. Syntactic soundness proof of a type-and-capability system with hidden state. *Journal of functional programming* 23, 1 (2013), 38–144.

Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications.* 624–641.

Jeremy G Siek and Sam Tobin-Hochstadt. 2016. The recursive union of some gradual types. In *A List of Successes That Can Change the World.* Springer, 388–410.

Marvin Solomon. 1978. Type definitions with parameters. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* 31–38.

Chris Stone and Robert Harper. 1996. *A Type-Theoretic Account of Standard ML 1996.* Technical Report CMU-CS-96-136. School of Computer Science, Carnegie Mellon University, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891.

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure Distributed Programming with Value-Dependent Types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011).*

The Coq Development Team. 2019. *Coq.* https://coq.inria.fr

Joseph C. Vanderwaart, Derek Dreyer, Leaf Petersen, Karl Crary, Robert Harper, and Perry Cheng. 2003. Typed Compilation of Recursive Datatypes. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation.* 98–108.

Yanpeng Yang and Bruno C. d. S. Oliveira. 2019. Pure iso-type systems. *Journal of Functional Programming* 29 (2019).