Revisiting Iso-Recursive Subtyping

YAODA ZHOU, The University of Hong Kong, China
JINXU ZHAO, The University of Hong Kong, China
BRUNO C. D. S. OLIVEIRA, 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 for 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 have equivalent expressive power to this declarative specification. We then show two variants of *sound*, *complete* and *decidable* algorithmic formulations of subtyping with respect to the declarative specification, which employ the idea of *double unfoldings*. Compared to the Amber rules, the double unfolding rules have the advantage of: (1) being modular; (2) not requiring reflexivity to be built in; (3) leading to an easy proof of transitivity of subtyping; and (4) being easily applicable to subtyping relations that are not antisymmetric (such as subtyping relations with record types). This work sheds new insights on the theory of subtyping iso-recursive types, and the new rules based on double unfoldings have important advantages over the original Amber rules involving recursive types. All results are mechanically formalized in the Coq 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, Jinxu Zhao, and Bruno C. d. S. Oliveira. 2018. Revisiting Iso-Recursive Subtyping. *J. ACM* 37, 4, Article 111 (August 2018), 55 pages. https://doi.org/10.1145/1122445.1122456

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; Jinxu Zhao, Department of Computer Science, The University of Hong Kong, Hong Kong, China, jxzhao@cs.hku.hk; Bruno C. d. S. Oliveira, Department of Computer Science, The University of Hong Kong, Hong Kong, China, bruno@cs.hku.hk.

0004-5411/2018/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

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].

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 [Abadi and Cardelli 1996; Bengtson et al. 2011; Chugh 2015; Duggan 2002; Lee et al. 2015; Swamy et al. 2011]. Perhaps most prominently the seminal work on "*A Theory of Objects*" by Abadi and Cardelli [1996] employs iso-recursive style Amber rules.

The Amber rules are appealing due to their apparent simplicity, but the metatheory for their iso-recursive formulation is not well studied. 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 iso-recursive 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. 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 for 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. The n times unfolding applies $n-1$ substitutions to the type A (the recursive type body), 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. Moreover, the Amber rules are shown to be equivalent (in terms of expressive power) to this declarative specification.

We also show alternative algorithmic formulations based on the idea of *double unfoldings*. We discuss two variants of rules for subtyping recursive types. The first variant, which we call the double unfolding rule, checks both 1-time and 2-times finite unfoldings. The second variant can be seen as an optimization that checks only 2-times finite unfoldings, by tracking the names of the recursive types to avoid the 1-time finite unfolding check. We call the second variant *nominal unfolding*. Both rules accept all valid subtyping statements that the Amber rules accept, but they have important advantages. In particular the rules with double unfoldings:

- **Enable modular proofs.** The new subtyping rules for recursive types are modular in the sense that proofs for properties such as transitivity or reflexivity only need to account for the new recursive case. All the other cases remain essentially the same as in a subtyping relation without recursive types. Key to this form of modularity is the use standard environments, which are just a collection of type variables.
- **Have easy proofs of transitivity of subtyping.** A particular consequence of the previous point is an easy proof for transitivity, which has been a stumbling block in the past for the iso-recursive Amber rules. 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. To our knowledge the only transitivity proof for the Amber rules is due to Bengtson et al. [2011], and the proof is quite intricate, relying on a complex inductive argument.
- **Do not require built-in reflexivity.** An additional benefit is that reflexivity does not have to be built in, but it can be derived instead. In the Amber rules built-in reflexivity is necessary to deal with contravariant occurrences of recursive type variables.
- **Are applicable to antisymmetric subtyping relations.** Built-in reflexivity can be problematic in some settings, including calculi with record subtyping or intersection/union types. Such calculi can have "isomorphic" subtyping where two syntactically different types A and B can be subtypes of each other. In other words the subtyping relation is not antisymmetric. Avoiding built-in reflexivity makes the rules easier to apply in such settings. As we show, the double unfolding rules can deal with record types easily.

The focus of our work is on iso-recursive subtyping rules that enable easy metatheory, and improving the understanding of Amber-style iso-recursive subtyping. Therefore our work will be useful to those interested on the theory of recursive types, as well as for formalizations of calculi using iso-recursive subtyping. Formalizations can benefit from our work to easily develop calculi with recursive types and prove important properties, such as transitivity, decidability and type soundness. While the rules based on double unfolding rules are algorithmic and therefore can be used in implementations, our focus is not on efficient algorithms. For implementations, the use of the Amber rules may still be preferable if efficiency is an important concern. Moreover, there are alternatives to the Amber rules, such as the complete rules by Ligatti et al. [2017], which may be preferable for extra expressive power in the subtyping relation, as well as efficient algorithms.

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:

Table 1. Some key theorems in the paper.

	Reflexivity	Transitivity	Unfolding Lemma
Amber Rules	Built-in	Corollary 58	Corollary 59
Finite Unfolding	Theorem 5	Theorem 6	Lemma 8
Double Unfolding	Theorem 15	Theorem 16	Lemma 24
Nominal Unfolding	Theorem 25	Theorem 26	Lemma 27
Weakly Positive Subtyping	Theorem 39	Theorem 41	Lemma 42

- **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 in Section 3.
- **Algorithmic subtyping based on double unfoldings:** We show two sound, complete and decidable algorithmic formulations of subtyping employing the idea of double unfoldings in Section 4. The first formulation uses a so-called double unfolding rule, and the second formulation uses a so-called nominal unfolding rule.
- **Equivalence to the Amber rules:** We prove that the Amber rules are equivalent, in terms of expressive power, to our new formulation of subtyping in Section 5.
- **Subject-reduction for a typed lambda calculus with recursive types and record types:** To illustrate the applicability of our results to calculi with subtyping relations that are not antisymmetric, we formalize a typed lambda calculus with recursive types as well as record types and prove type preservation and progress in Section 6.
- **Subtyping with a Weakly Positive Restriction:** In addition to the Amber rules and the finite and the rules based on double unfoldings, we also give another equivalent formulation of subtyping based on a weakly positive restriction of recursive variables. This variant captures precisely a folklore observation that the Amber rules express two situations where a recursive variable can be a subtype of another type: positive subtyping and reflexivity. This variant, presented as part of Section 5, is used as an intermediate step to prove the equivalence between the Amber rules and a formulation using double unfolding.
- **Mechanical formalization:** All the results are formalized in the Coq theorem prover and can be found at: https://github.com/juda/Iso-Recursive-Subtyping

Table 1 and Figure 1 summarize some key lemmas and theorems of this paper. In particular, it shows that all five formulations of subtyping presented in this paper are equivalent in terms of expressive power.

This article is a significantly expanded version of a conference paper [Zhou et al. 2020]. There are several improvements with respect to the conference version. First of all, we considerably simplify the proof of soundness theorem between double and finite unfoldings, and provide a proof of the unfolding lemma directly using the double unfolding rules. In the original soundness proof, a special relation capturing valid subtyping derivations was used. In the new proof, this relation and proof technique is no longer used, greatly simplifying the proof. Secondly, we prove that the Amber rules are complete with respect to the double unfolding formalization. Consequently, all five subtyping formulations are shown to be equivalent (Figure 1). The conference version only shows the soundness of the Amber rules with respect to finite unfoldings, but not their completeness. Thirdly we present a variant of double unfoldings, called nominal unfoldings, which are more efficient that double unfoldings and yet preserve all of the key advantages in terms of the development of the meta-theory of the original double unfolding rules. Finally, the material in Section 6, showing that our new rules can be applied to calculi with record types, is new. This

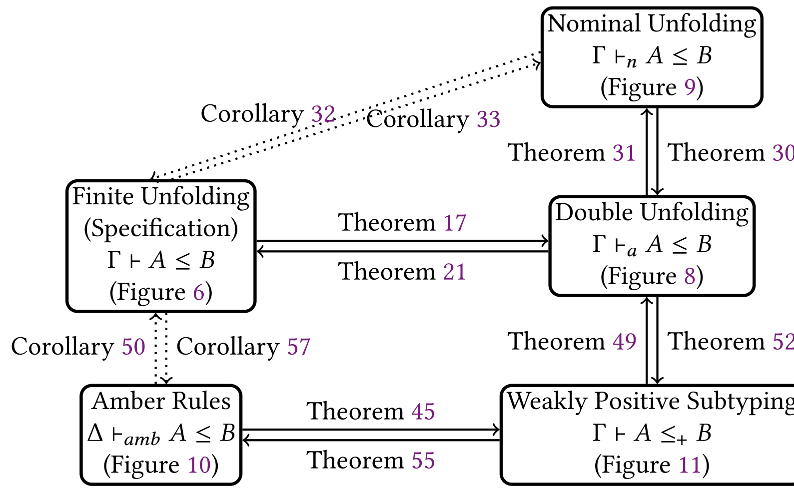


Fig. 1. A diagram with the soundness and completeness lemmas in this work.

is interesting because it shows that rules based on double unfoldings can deal smoothly with subtyping relations that are not antisymmetric, unlike the Amber rules.

2 OVERVIEW

This section provides an overview of the problem of iso-recursive subtyping and our results. We first briefly review applications of iso-recursive subtyping, 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 two algorithmic variants based on double unfoldings. Finally, we show how the double unfolding rule can be employed in calculi with record types.

2.1 Applications of Iso-Recursive Types

Before we move to our work we first briefly review some of the applications of iso-recursive types. Many programming languages adopt an iso-recursive formulation. In practice, 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, a flavour of iso-recursive types is provided via datatypes.

Figure 2 (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, 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 2. 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

```
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. 2. Recursive types in Haskell (left) and Java (right).

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].

2.2 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 another 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 $A \to B$, a type variable α or a recursive type $\mu\alpha.\ A$. The subtyping rules for the top type, primitive types and function types are standard:

$$\frac{}{A \le \top} \qquad \frac{}{\mathsf{nat} \le \mathsf{nat}} \qquad \frac{B_1 \le A_1 \qquad A_2 \le B_2}{A_1 \to A_2 \le B_1 \to B_2}$$

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[1]

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

An important aspect to pay attention to 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 will come back to this point in Section 2.5.

Example 2. A second example is $\mu\alpha.\ \top \to \alpha \le \mu\alpha.\ \mathsf{nat} \to \alpha$. 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 $\mu\alpha.\ \alpha \to \mathsf{nat}$ is *not* a subtype of $\mu\alpha.\ \alpha \to \top$. 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, $A \to \mathsf{nat} \le A \to \top$ holds for

[1]We assume that recursive types have lower priority. That is, $\mu\alpha.\ \top \to \alpha$ means $\mu\alpha.\ (\top \to \alpha)$ not $(\mu\alpha.\ \top) \to \alpha$.

any type A. But that does not imply that $\mu\alpha.\ \alpha \to \mathsf{nat} \leq \mu\alpha.\ \alpha \to \top$, because the type variable 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 $\mathsf{nat} \leq \top$ but also that $\top \leq \mathsf{nat}$ (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:

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

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.

2.3 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:

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

which accept, for example, $\mu\alpha.\ \top \to \alpha \leq \mu\alpha.\ \mathsf{nat} \to \alpha$ and $\mu\alpha.\ \alpha \to \alpha \leq \mu\alpha.\ \alpha \to \alpha$. 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:

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

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

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

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

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

Generally speaking, these rules are sound for positive recursive subtyping. However, contravariant recursive types, where the recursive type variables occur in negative positions, may allow *unsound* subtyping statements, as shown above.

2.4 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:

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

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 $\alpha \leq \alpha$ (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 statement $\mu\alpha.\ A \leq \mu\alpha.\ B$ is valid, and all substatements $\alpha \leq \alpha$ 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 $\mu\alpha.\ \top \to \alpha \leq \mu\alpha.\ \alpha \to \alpha$. Furthermore, at least without some form of reflexivity built-in, it even rejects subtyping of equal types with negative recursive variables, such as $\mu\alpha.\ \alpha \to \alpha \leq \mu\alpha.\ \alpha \to \alpha$.

2.5 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 3. 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 $\alpha \leq \beta$ 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 $\mu\alpha.\ \alpha \to \mathsf{nat} \leq \mu\beta.\ \beta \to \top$. Unfortunately, rule OAmber-rec rules out too many cases with negative subtyping, including statements about equal types, such as $\mu\alpha.\ \alpha \to \mathsf{nat} \leq \mu\beta.\ \beta \to \mathsf{nat}$. 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 $\mu\alpha.\ \mathsf{nat} \to \alpha = \mu\alpha.\ \mathsf{nat} \to \mathsf{nat} \to \alpha$ or $\mu\alpha.\ \mathsf{nat} \to \alpha = \mathsf{nat} \to \mu\alpha.\ \mathsf{nat} \to \alpha$, 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 a variant of syntactic equality[2] in reflexivity, which is less powerful, but it 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

[2]More precisely, in a setting where binders and variables are encoded using names, alpha-equivalence is used. In settings where De Bruijn indices are used, it amounts to syntactic equality.

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

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

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

Fig. 3. The complete Amber subtyping rules by Amadio and Cardelli [1993] for *equi-recursive* subtyping.

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:

Reflexivity cannot be eliminated. The reflexivity rule is essential to the subtyping relation. As we have seen, one cannot even derive $\mu\alpha.\ \alpha \to \mathsf{nat} \leq \mu\alpha.\ \alpha \to \mathsf{nat}$ 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 2.3.

The reflexivity rule, if present in the subtyping relation, depends on a specific equivalence judgment. Simple systems with antisymmetric subtyping relations might use syntactic equivalence or alpha-equivalence. Yet syntactic or alpha-equivalence 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 subtyping statement:

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

However, if the built-in reflexivity employs only alpha-equivalence, such a subtyping statement may be rejected. For instance if record types are modelled as sequences in the abstract syntax (which is quite common [Pierce 2002]), then the two records $\{x : \alpha, y : \mathsf{nat}\}$ and $\{y : \mathsf{nat}, x : \alpha\}$ will be syntactically different. In this case the subtyping relation *is not* antisymmetric. That is both $\{x : \alpha, y : \mathsf{nat}\} \leq \{y : \mathsf{nat}, x : \alpha\}$ and $\{y : \mathsf{nat}, x : \alpha\} \leq \{x : \alpha, y : \mathsf{nat}\}$ are true, but the two types are not equal. Thus, a (strict) reflexivity rule employing syntactic equality is not adequate in such cases. For record types it may be possible to avoid this issue by using a different representation in the abstract syntax. For instance, we could try to model record types instead as finite maps from field names to types. Then equality of finite maps could have the expected properties for equality and a standard reflexivity rule could suffice. However, other type system features, such as union ($A \vee B$) and intersection types ($A \wedge B$) [Barbanera et al. 1995; Coppo et al. 1981; Pottinger 1980], would pose similar challenges. In those type systems we wish to have $A \wedge B$ and $B \wedge A$ to be equivalent types, for example. A change of representation of abstract syntax does not seem to help for such features.

The reader may refer to work by Ligatti et al. [2017] for a more extended discussion on the complications of having the reflexivity rule built-in. We will also come back to this point in Section 2.8.

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:

$$\frac{\alpha_1 \leq \alpha_2, \alpha_2 \leq \alpha_3 \vdash \alpha_1 \leq \alpha_2 \qquad \alpha_1 \leq \alpha_2, \alpha_2 \leq \alpha_3 \vdash \alpha_2 \leq \alpha_3}{\alpha_1 \leq \alpha_2, \alpha_2 \leq \alpha_3 \vdash \alpha_1 \leq \alpha_3} \text{ DOES NOT HOLD!}$$

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 that proving transitivity of an algorithmic formulation 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 [Abadi and Cardelli 1996; 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, as it requires a "very complicated inductive argument". Thus, they finally adopt the positive restriction, as we discussed in Section 2.4. 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, *most importantly*, proofs.

2.6 A New Declarative Specification for Iso-Recursive Subtyping

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. As a first step towards understanding the essence of the Amber rules we provide a new declarative specification of iso-recursive subtyping in terms of finite unfoldings. We prove that the Amber rules are equivalent (sound and complete) 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 2.3. 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 2.3:

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

As we have argued, this subtyping statement should fail because unfolding the recursive type twice leads to an invalid subtyping statement. However, with the 1-time finite unfolding used by the rules in Section 2.3, all that is checked is whether $\alpha \vdash \alpha \to \mathsf{nat} \leq \alpha \to \top$ holds. Since such statement does hold, the rule *unsoundly* accepts $\mu\alpha.\ \alpha \to \mathsf{nat} \leq \mu\alpha.\ \alpha \to \top$. 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:

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

In comparison to the rules showed in Section 2.3, 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 $\mu\alpha.\ A$, but we use a slight generalization (mainly for proofs) to unfold a type B with another type A multiple times. For example, for the recursive type $\mu\alpha.\ \mathsf{nat} \to \alpha$, the one-time finite unfolding is $\mathsf{nat} \to \alpha$ and the two times finite unfolding is $\mathsf{nat} \to \mathsf{nat} \to \alpha$. Note that the zero-times finite unfolding of a recursive type $\mu\alpha.\ A$ would be the recursive type itself, according to our terminology. In our definition of n-times finite unfolding we start counting from 1 and we apply the definition to the recursive type body (rather than the recursive type itself). In other words, we execute $(n-1)$ times substitutions (where n corresponds to the arity of the finite unfolding) of the body of the recursive type 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. The counting scheme for the n-times finite unfolding definition may look at little odd. One may expect the more natural looking definition where the body is unfolded n times instead of $n-1$ times. However, using n times instead of $n-1$ would disagree with our terminology for finite unfoldings of recursive types. For instance, the one-time unfolding of $\mu\alpha.\ \mathsf{nat} \to \alpha$ is $\mathsf{nat} \to \alpha$, and does *zero* (not one!) substitutions in the body.

In rule S-REC, the number of times that the left and the right types are unfolded is exactly the same. One may wonder if it makes sense to consider cases where we would unfold the recursive types a different number of types on the left and on the right. We believe that such approach would lead to a type unsound rule, and that it is important that the number of finite unfoldings is the same. For instance, consider $\mu\alpha.\ \mathsf{nat} \to \alpha \leq \mu\alpha.\ \mathsf{nat} \to \mathsf{nat} \to \top$. In this case if we choose to unfold the body of the left recursive type $n+1$ times and the body of the right recursive type only n times (for all n) then we would get a valid subtyping statement. However, those two types should not be subtypes since if we apply the unfolding lemma we would obtain: $\mathsf{nat} \to (\mu\alpha.\ \mathsf{nat} \to \alpha) \leq \mathsf{nat} \to \mathsf{nat} \to \top$. The latter is not a valid subtyping statement.

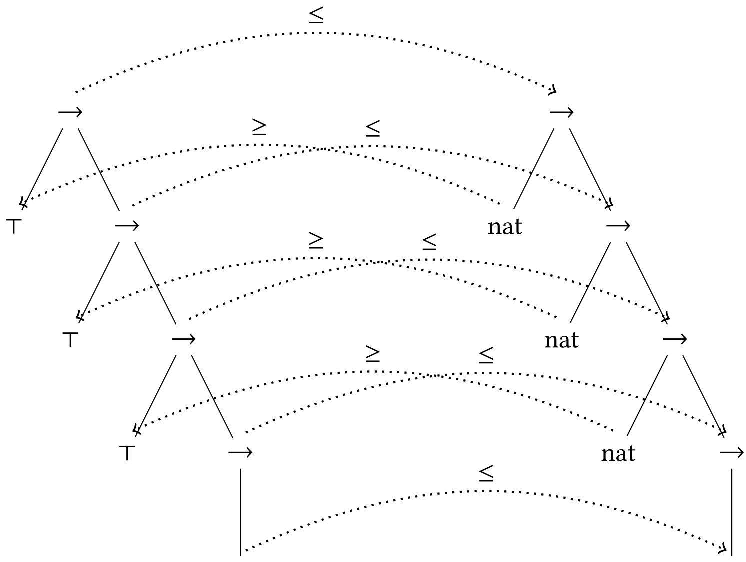


Fig. 4. Tree model for equi-recursive subtyping.

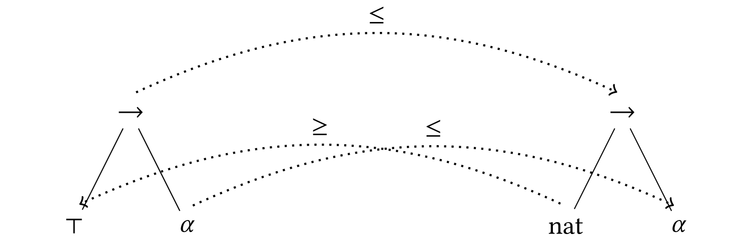
Contrasting Equi and Iso-Recursive Types. It is useful to contrast the rule S-REC and its formulation in terms of finite unfoldings to Amadio and Cardelli's specification of equi-recursive subtyping in terms of infinite unfoldings of the recursive types. In Amadio and Cardelli's work they use the notion of *finite approximation* of a tree, which is closely related to the idea of finite unfoldings. A simplified[3] specification of equi-recursive subtyping in terms of subtyping of infinite trees can be reformulated in terms of finite unfoldings as:

$$\frac{\Gamma \vdash [\alpha \to A]^{\infty}\ A \le [\alpha \to B]^{\infty}\ B}{\Gamma \vdash \mu\alpha.\ A \le \mu\alpha.\ B}\ \textsc{S-Equi}$$

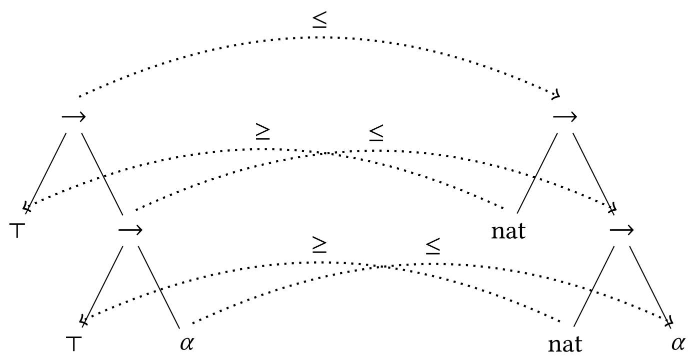
Where the notation $[\alpha \to A]^{\infty}\ A$ denotes applying infinite substitutions to A. In other words, we define the equi-recursive comparison by just one comparison on the limit case, which will potentially compare two infinite trees. With rule S-EQUI subtyping statements such as $\mu\alpha.\ \top \to \alpha \le \mu\alpha.\ \mathsf{nat} \to \alpha$ hold, just like with the rule S-REC for iso-recursive subtyping. However, unlike iso-recursive subtyping, subtyping statements such as $\mu\alpha.\ \top \to \alpha \le \mu\alpha.\ \mathsf{nat} \to \mathsf{nat} \to \alpha$ also hold, since we unfold both trees to the limit. Figure 4 visualizes the tree model equi-recursive subtyping. Note that the figure applies to both $\mu\alpha.\ \top \to \alpha \le \mu\alpha.\ \mathsf{nat} \to \alpha$ and $\mu\alpha.\ \top \to \alpha \le \mu\alpha.\ \mathsf{nat} \to \mathsf{nat} \to \alpha$, since in both cases the infinite unfoldings of the trees in the subtyping statements are the same.

Instead of a single comparison in the limit case, the rule S-REC for iso-recursive subtyping requires infinitely many comparisons, one for each n-time unfolding. For example, Figure 5 visualizes the comparisons for $\mu\alpha.\ \top \to \alpha \le \mu\alpha.\ \mathsf{nat} \to \alpha$ in the iso-recursive model. In the figure we show only the first 3 comparisons, which would correspond to the 1-time, 2-times and 3-times finite unfoldings respectively. However, there would be an infinite number of such comparisons for all n-times finite unfoldings. Using rule S-REC the subtyping statement $\mu\alpha.\ \top \to \alpha \le \mu\alpha.\ \mathsf{nat} \to \mathsf{nat} \to \alpha$ fails, unlike in the equi-recursive model. It is easy to see why this is the case. Since rule S-REC requires

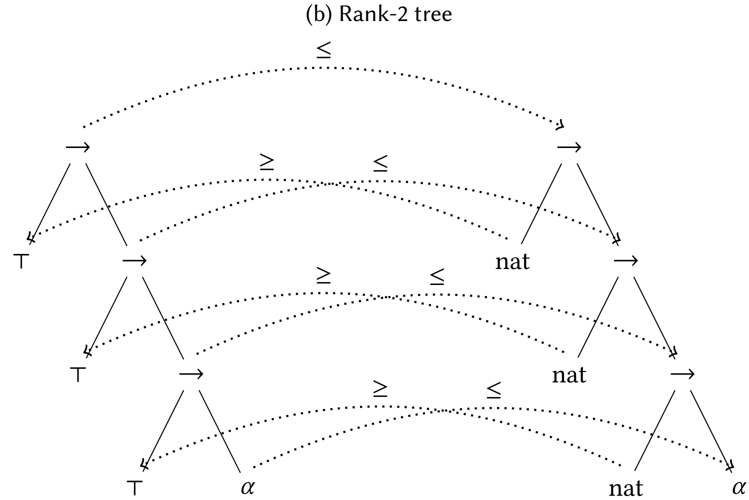
[3]This definition is simplified because the rule S-EQUI compares only two recursive types. In general, in equi-recursive formulations, any two types (recursive or not) can be unfolded and compared. For instance $\mathsf{nat} \to (\mu\alpha.\ \mathsf{nat} \to \alpha) \le \mu\alpha.\ \mathsf{nat} \to \alpha$ should hold, since the infinite unfoldings of the two types are the same.



(a) Rank-1 tree



(b) Rank-2 tree



(c) Rank-3 tree

Fig. 5. Tree model for iso-recursive subtyping for the first 3 finite unfoldings for $\mu\alpha.\ \top \to \alpha \le \mu\alpha.\ nat \to \alpha$.

that *all* comparisons are successful, to show that two recursive types are not subtypes it is enough to show that one of the finite comparisons fails. For example, the comparison of one-time finite unfoldings, which amounts to $\top \to \alpha \le \mathsf{nat} \to \mathsf{nat} \to \alpha$, fails. Therefore, we can see that rule S-REC rejects $\mu\alpha.\ \top \to \alpha \le \mu\alpha.\ \mathsf{nat} \to \mathsf{nat} \to \alpha$.

2.7 Algorithmic Subtyping: Double and Nominal Unfoldings

An infinite amount of conditions is impossible to check algorithmically. Therefore, we must find alternative formulations that are algorithmic for implementations. As we will show, a suitable formulation with iso-recursive Amber rules is equivalent to our declarative specification. Thus, the Amber rules can in principle serve as a foundation for an implementation. However, there are reasons to seek for alternative algorithmic rules. Most importantly, as we have argued in Section 2.5, the Amber rules are hard to work with in proofs and metatheory. Therefore, to provide a detailed account of the metatheory for iso-recursive subtyping we propose alternative algorithmic definitions for subtyping of recursive types. The new formulations of subtyping have 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 new rules are also equivalent with respect to the declarative specification of iso-recursive subtyping and the Amber rules.

Double Unfoldings. 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 in Section 2.3. The 2-times finite unfolding for the example is:

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

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 $\mathsf{nat} \leq \top$ (which is valid), and $\top \leq \mathsf{nat}$ (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:

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

With this rule 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:

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

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

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

But the 2-times finite unfolding for this example ($\mathsf{nat} \to \mathsf{nat} \to \alpha \leq \mathsf{nat} \to \mathsf{nat} \to \top$) 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 ($\mathsf{nat} \to \alpha \nleq \mathsf{nat} \to \mathsf{nat} \to \top$), which fails and is the reason why the double unfolding rule rejects this example.

Nominal Unfoldings. The double unfolding rule is interesting because it directly relates to the declarative formulation using finite unfoldings. However, the double unfolding rules have exponential time complexity due to the two premises for both (1-time and 2-times) finite unfoldings. At first, the 1-time finite unfolding appears unnecessary, since the 2-times unfolding seems to do all the checks of the 1-time finite unfolding. Unfortunately, as our previous counter-example has shown, the 1-time finite unfolding check cannot be avoided, due to some spurious subtyping that exists when using only the 2-times finite unfolding. In an implementation, there are potentially some approaches to avoid the cost of the extra 1-time finite unfolding check. For example, we can store the result of one-time finite unfolding during subtype checking, and reuse that result as part of subtype checking of the double unfoldings. This would avoid recomputation and lead to a more efficient algorithm. However, it would be nicer to address this issue of the double unfoldings in the formalism itself.

For avoiding the 1-time finite unfolding in the double unfolding rule, we propose a variant of the rule. Having understood the nature of the spurious subtyping problem that appeared in our counter-example using only 2-times finite unfoldings, the key idea to solve the problem is simple. We track the name of the recursive variables during double unfoldings to avoid accidental subtyping. Our approach is to add an extra label with the recursive variable name of the recursive type. This regulates the structure of the derivation tree. Formally, our nominal unfolding rule is:

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

Compared to the double unfolding rule, our nominal unfolding rule only has one premise. More importantly, it avoids the spurious subtyping problem. In our new nominal unfolding rule, we do not need the extra check for the one-time finite unfolding (checking $\Gamma, \alpha \vdash A \leq B$). The derivation tree below reflects the change for our simpler counter-example of the double unfolding rule (without the extra one-time finite unfolding check):

$$\cfrac{\cfrac{\mathsf{nat} \leq \mathsf{nat} \qquad \{\alpha : \mathsf{nat} \to \alpha\} \leq \mathsf{nat} \to \top\ \text{(fails!)}}{\mathsf{nat} \to \{\alpha : \mathsf{nat} \to \alpha\} \leq \mathsf{nat} \to \mathsf{nat} \to \top}}{\mu\alpha.\ \mathsf{nat} \to \alpha \leq \mu\alpha.\ \mathsf{nat} \to \mathsf{nat} \to \top}$$

The presence of the extra label means that we now get $\{\alpha : \mathsf{nat} \to \alpha\} \leq \mathsf{nat} \to \top$ (which fails) instead of $\mathsf{nat} \to \alpha \leq \mathsf{nat} \to \top$ (which succeeds). In other words, the presence of the nominal label avoids the need for the extra one-time finite unfolding check to rule out the counter-example.

Discussion. As we shall see, both the double and the nominal unfolding rules are easy to work with in terms of proofs and metatheory, and the nominal unfolding rules can even simplify some proofs due to the single premise. The double unfolding rule is directly inspired by the finite unfolding specification. The nominal unfolding rule additionally employs the idea of tracking the recursive type variable as a label to avoid spurious subtyping that arises from double unfoldings. Therefore, it can avoid the extra 1-time finite unfolding check. In the nominal unfolding rule it is interesting to observe that the names of recursive type variables play an important role, just as in the Amber rules. However, in the Amber rules, we use distinct type variable names and track the subtyping relation between those variables. In the nominal unfolding rule we use the same type variable name, which is sufficient to identify types that originate from the double unfolding substitutions. Therefore, spurious subtyping when using only double unfoldings can be avoided in the nominal unfolding rule.

As a final remark, in follow up work to the work in this article, we have encountered some settings where nominal unfolds and double unfoldings are not equivalent. In particular, in a setting with intersection types $A \wedge B$ [Barbanera et al. 1995; Coppo et al. 1981; Pottinger 1980], the nominal unfolding rule works well, but the double unfolding rule accepts more subtyping statements, which invalidates the unfolding lemma. It appears that the issue is related to subtyping relations that allow multiple ways to derive the same subtyping statement. In systems with intersection types, for instance, there are multiple overlapping rules to deal with intersections. Therefore, in some settings the nominal label seems to be not just useful to perform an optimization, but also to ensure the correctness of subtyping. Nonetheless, we have not encountered any setting yet where the nominal unfolding rule does not work.

Some Final Implementation Considerations. The double unfolding and nominal unfolding rules are primarily designed with the goal of leading to a simple metatheory and proofs. Both rules employ substitutions which, if used directly in an implementation, have significant performance penalties. To avoid substitutions one possibility would be to adopt *explicit substitutions* [Abadi et al. 1991], which are a standard solution to avoid the performance penalties associated with substitutions. Another possibility would be to adopt some ideas in the implementation approach proposed by Ligatti et al. [2017]. Although Ligatti et al.'s rules have different expressive power compared to the Amber rules and our rules, they also employ substitutions. They present an optimized $O(mn)$ algorithm that avoids the use of substitutions, and we believe that it should be possible to adopt some of those ideas to implement double/nominal unfoldings. Finally, a simple optimization for both double and nominal unfoldings is to avoid substitutions in *positive* positions. As Section 2.3 discusses for covariant subtyping using $\Gamma \vdash A \leq B$ in the premise of the recursive subtyping rule is sound. Thus, we should not need to substitute recursive type variables that are found in positive positions, which avoids extra subtype checks of the substituted types. In other words, we could have the variant (here for nominal unfoldings):

$$\frac{\Gamma, \alpha \vdash [\alpha \mapsto \{\alpha : A\}]^{+}\, A \leq [\alpha \mapsto \{\alpha : B\}]^{+}\, B}{\Gamma \vdash \mu\alpha.\, A \leq \mu\alpha.\, B}\ \text{S-Nominal+}$$

The idea is to employ a polarized form of substitution $[\alpha \mapsto A]^{m}\, B$, which is parametrized by a positive (+) or negative (−) mode m. This form of substitution would only perform substitutions at negative occurrences of type variables. Thus, the special case of covariant subtyping would behave equivalently to the rule presented in Section 2.3. We leave the development and proof of correctness for an efficient algorithm for future work.

2.8 A Calculus with Recursive Record Types

As an illustration of the advantages of our rules, in Section 6 we show an application to a calculus with records and iso-recursive types.

In Section 2.5, we have discussed that the Amber rules cannot deal well with some forms of subtyping. In particular, the reflexivity rule is limiting when the subtyping relation is not antisymmetric. In the context of subtyping, antisymmetry is the property that if two types are both subtypes of each other, then the two types are (syntactically) equal. More formally:

$$\Gamma \vdash A \leq B \wedge \Gamma \vdash B \leq A \Rightarrow A = B$$

In simple subtyping relations, such as for instance a simply typed lambda calculus extended with the top type and recursive types, this property holds. For instance, the calculus in Section 3 has an antisymmetric subtyping relation.

Unfortunately, many languages contain subtyping relations that are not antisymmetric. For instance, if a language contains some form of record types (which includes essentially all OOP languages), then the subtyping relation is not antisymmetric. In the example below, the subtyping statement

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

should hold, since $\{x : \alpha, y : \mathsf{nat}\}$ and $\{y : \mathsf{nat}, x : \alpha\}$ are subtypes of each other. However, the two types are not syntactically equal. In such a setting, the use of the Amber rules would require that, instead of using syntactic equality in the reflexivity rule, we should use an equivalence relation on types. However, we cannot simply define equivalence to be:

$$\Gamma \vdash A \sim B \;:=\; \Gamma \vdash A \leq B \land \Gamma \vdash B \leq A$$

because then the reflexivity rule would become (by a simple unfolding of the equivalence definition):

$$\frac{\Delta \vdash A \leq B \quad \Delta \vdash B \leq A}{\Delta \vdash A \leq B}\text{Amber-Refl-Wrong}$$

which would lead to a circular (and ill-behaved) subtyping relation. Instead, a separate equivalence relation needs to be defined to ensure that record types are equivalent up-to permutation. But adding such a separate relation on types would add complexity, since we would need a new set of rules and theorems about such relation.

In contrast, with the double unfolding rules, because reflexivity is not built-in, we can simply define the equivalence relation above ($\Gamma \vdash A \sim B$) via subtyping. Thus, the double unfolding rules do not require a separate definition of equivalence, and they also do not rely on the subtyping relation being antisymmetric. The calculus in Section 6 illustrates the addition of records and records types to the calculus in Section 3. This addition has minimal impact of the calculus and metatheory: the proof techniques are similar, except that instead of syntactic equality we use our equivalence definition for types when proving the unfolding lemma.

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: term variables (x), 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 (Figure 6), ensuring that all variables in the environment are distinct. In a well-formed environment, repetition of variables is not allowed and the order of variables are not important. Note that, throughout the paper, we adopt the convention that variables are distinct. For instance, in rule WFT-REC the α introduced in Γ is distinct from other variables in Γ. In our Coq formalization the use of a locally

$\boxed{\vdash \Gamma}$ *(Well-Formed Environment)*

$$\text{WFE-EMPTY}\ \frac{}{\vdash \cdot} \qquad \text{WFE-SUB}\ \frac{\vdash \Gamma \quad \alpha \notin \Gamma}{\vdash \Gamma, \alpha} \qquad \text{WFE-TYP}\ \frac{\vdash \Gamma \quad x \notin \Gamma \quad \Gamma \vdash A}{\vdash \Gamma, x : A}$$

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

$$\text{WFT-NAT}\ \frac{}{\Gamma \vdash \mathsf{nat}} \qquad \text{WFT-TOP}\ \frac{}{\Gamma \vdash \top} \qquad \text{WFT-VAR}\ \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha} \qquad \text{WFT-ARROW}\ \frac{\Gamma \vdash A_1 \quad \Gamma \vdash A_2}{\Gamma \vdash A_1 \to A_2} \qquad \text{WFT-REC}\ \frac{\Gamma, \alpha \vdash A}{\Gamma \vdash \mu\alpha.\ A}$$

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

$$\text{S-NAT}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{nat} \le \mathsf{nat}} \qquad \text{S-TOP}\ \frac{\vdash \Gamma \quad \Gamma \vdash A}{\Gamma \vdash A \le \top} \qquad \text{S-VAR}\ \frac{\vdash \Gamma \quad \alpha \in \Gamma}{\Gamma \vdash \alpha \le \alpha} \qquad \text{S-ARROW}\ \frac{\Gamma \vdash B_1 \le A_1 \quad \Gamma \vdash A_2 \le B_2}{\Gamma \vdash A_1 \to A_2 \le B_1 \to B_2}$$

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

Fig. 6. Well-formedness and subtyping rules.

nameless [Charguéraud 2011] encoding for binders makes such informal conventions precise. The top of Figure 6 also 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 6 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 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 the 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 without a suitable 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:

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

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 $\Gamma \vdash A \leq B$ then $\vdash \Gamma$ and $\Gamma \vdash A$ and $\Gamma \vdash B$.

Another important property of our subtyping rules is that the order of variables in contexts is irrelevant. That is we can always permute whole portions of the environment:

Lemma 4. If $\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4 \vdash A \leq B$ then $\Gamma_1, \Gamma_3, \Gamma_2, \Gamma_4 \vdash A \leq B$.

Thanks to our standard context, the proofs of both reflexivity and 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 5. Reflexivity.

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

Theorem 6. Transitivity.

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

Proof. From Lemma 3 we know all types and the environment are well-formed. Do induction on $\Gamma \vdash B$ (with the rule WFT-INF).

- Rule WFT-NAT: Do inversion on both two subtyping statements, and we know that A is nat and C is nat or $\top$.
- Rule WFT-Top: Do inversion on $\Gamma \vdash \top \leq C$, and we know that C is $\top$.
- Rule WFT-VAR: Do inversion on both two subtyping statements, and we know that A is α and C is α or $\top$.
- Rule WFT-ARROW: Assume $B := B_1 \to B_2$.
  - Do inversion on $\Gamma \vdash B_1 \to B_2 \leq C$, we know C is $\top$ or $C := C_1 \to C_2$. The former one is solved immediately. From the latter one, we obtain $\Gamma \vdash C_1 \leq B_1$ and $\Gamma \vdash B_2 \leq C_2$.
  - Do inversion on $\Gamma \vdash A \leq B_1 \to B_2$, we know $A := A_1 \to A_2$ and obtain $\Gamma \vdash B_1 \leq A_1$ and $\Gamma \vdash A_2 \leq B_2$.
  - Now the goal is $\Gamma \vdash A_1 \to A_2 \leq C_1 \to C_2$. Applying the arrow rule, what we need to prove are $\Gamma \vdash C_1 \leq A_1$ and $\Gamma \vdash A_2 \leq C_2$. The two goals can be solved by the induction hypotheses.

- Rule WFT-INF: Assume $B := \mu\alpha.\ B'$.
  - Firstly, it is worthwhile stating the induction hypothesis that we get from rule WFT-INF explicitly: $\forall n\ A\ C,\ \Gamma, \alpha \vdash A \le [\alpha \mapsto B']^n\ B' \ \wedge\ \Gamma, \alpha \vdash [\alpha \mapsto B']^n\ B' \le C \ \Rightarrow\ \Gamma, \alpha \vdash A \le C$.
  - Do inversion on $\Gamma \vdash \mu\alpha.\ B' \le C$, we know C is $\top$ or $C := \mu\alpha.\ C'$. The former one is solved immediately. From the latter one, we obtain $\forall n,\ \Gamma, \alpha \vdash [\alpha \mapsto B']^n\ B' \le [\alpha \mapsto C']^n\ C'$.
  - Do inversion on $\Gamma \vdash A \le \mu\alpha.\ B'$, we know $A := \mu\alpha.\ A'$ and obtain $\forall n,\ \Gamma, \alpha \vdash [\alpha \mapsto A']^n\ A' \le [\alpha \mapsto B']^n\ B'$.
  - Now the goal is $\Gamma \vdash \mu\alpha.\ A' \le \mu\alpha.\ C'$. Applying the rule for recursive types, what we need to prove is $\forall n,\ \Gamma, \alpha \vdash [\alpha \mapsto A']^n\ A' \le [\alpha \mapsto C']^n\ C'$, which can be solved by the induction hypothesis.

□

Modularity of the proofs. Note that in our transitivity proof, all the cases, except for the recursive case, are standard and essentially the same as in a calculus without recursive types. In other words the proof is modular in the sense that existing cases of the proof are not significantly affected by the addition of recursive types. Other proofs, such as reflexivity or weakening, are similarly modular in the same sense. Existing proofs for previous formulations of iso-recursive subtyping [Bengtson et al. 2011; Ligatti et al. 2017] and in particular transitivity proofs are non-modular, and require significant changes after the addition of recursive types. We discuss this in more detail in Section 8.

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 7. If

(1) $\Gamma_1, \alpha, \Gamma_2 \vdash A \le B$;
(2) $\Gamma_1, \Gamma_2 \vdash \mu\alpha_1.\ C$ and $\Gamma_1, \Gamma_2 \vdash \mu\alpha_1.\ D$;
(3) α does not occur free in C and D;
(4) $\Gamma_1, \alpha, \Gamma_2 \vdash [\alpha \mapsto C]^n\ A \le [\alpha \mapsto D]^n\ B$ holds for all n,

then $\Gamma_1, \Gamma_2 \vdash [\alpha \mapsto \mu\alpha_1.\ C]\ A \le [\alpha \mapsto \mu\alpha_1.\ D]\ B$.

Proof. Induction on $\Gamma_1, \alpha, \Gamma_2 \vdash A \le B$. Cases rules S-NAT, S-TOP, and S-ARROW are simple.

- Rule S-VAR. Assume that A and B are variable β. If $\beta \neq \alpha$, then the goal is proven directly. Otherwise, the fourth premise is $\Gamma_1, \alpha, \Gamma_2 \vdash [\alpha \mapsto C]^n\ \alpha \le [\alpha \mapsto D]^n\ \alpha$, where n is arbitrary. The goal becomes $\Gamma_1, \Gamma_2 \vdash \mu\alpha_1.\ C \le \mu\alpha_1.\ D$. Then we can apply the rule for recursive types. Note that in the context, the order of variables is unimportant (see Lemma 4) we can permute the context without affecting the correctness. Therefore, the goal is equal to the fourth premise after context permutation and alpha-conversion between α_1 and α, which is possible due to the premise (3). Note also, that premise (3) can be derived from premise (2), but we explicitly show it as a premise due to the role in the proof.
- Rule S-REC. Assume that the shape of A is $\mu\alpha_2.\ A'$ and the shape of B is $\mu\alpha_2.\ B'$.
  - The fourth premise becomes $\forall n',\ \Gamma_1, \alpha, \Gamma_2 \vdash [\alpha \mapsto C]^n\ \mu\alpha_2.\ A' \le [\alpha \mapsto D]^n\ \mu\alpha_2.\ B'$, which can be rewritten to $\forall n',\ \Gamma_1, \alpha, \Gamma_2 \vdash \mu\alpha_2.\ [\alpha \mapsto C]^n\ A' \le \mu\alpha_2.\ [\alpha \mapsto D]^n\ B'$.
  - The goal becomes $\Gamma_1, \Gamma_2 \vdash [\alpha \mapsto \mu\alpha_1.\ C]\ \mu\alpha_2.\ A' \le [\alpha \mapsto \mu\alpha_1.\ D]\ \mu\alpha_2.\ B'$, which can be rewritten to $\Gamma_1, \Gamma_2 \vdash \mu\alpha_2.\ [\alpha \mapsto \mu\alpha_1.\ C]\ A' \le \mu\alpha_2.\ [\alpha \mapsto \mu\alpha_1.\ D]\ B'$.
  - If we apply rule S-REC to the goal, we get: $\forall n,\ \Gamma_1, \Gamma_2, \alpha_2 \vdash [\alpha_2 \mapsto ([\alpha \mapsto \mu\alpha_1.\ C]\ A')]^n\ ([\alpha \mapsto \mu\alpha_1.\ C]\ A') \le [\alpha_2 \mapsto ([\alpha \mapsto \mu\alpha_1.\ D]\ B')]^n\ ([\alpha \mapsto \mu\alpha_1.\ D]\ B')$.

- We rewrite the goal above, getting: $\forall n,\ \Gamma_1, \Gamma_2, \alpha_2 \vdash [\alpha \mapsto \mu\alpha_1.\, C]\, [\alpha_2 \mapsto A']^n\, A' \le [\alpha \mapsto \mu\alpha_1.\, D]\, [\alpha_2 \mapsto B']^n\, B'$.
- The induction hypothesis is complex, so we write it here explicitly for readability of the proof: $\forall n,\ (\forall n',\ \Gamma_1, \alpha, \Gamma_2 \vdash [\alpha \mapsto C]^n [\alpha_2 \mapsto A']^{n'}\, A' \le [\alpha \mapsto D]^n [\alpha_2 \mapsto B']^{n'}\, B') \Rightarrow \Gamma_1, \alpha_2, \Gamma_2 \vdash [\alpha \mapsto \mu\alpha_1.\, C]\, [\alpha_2 \mapsto A']^n\, A' \le [\alpha \mapsto \mu\alpha_1.\, D]\, [\alpha_2 \mapsto B']^n\, B'$.
- By applying context permutation and induction hypothesis to the goal, we get: $\forall n,\ \forall n',\ \Gamma_1, \alpha, \Gamma_2 \vdash [\alpha \mapsto C]^n [\alpha_2 \mapsto A']^{n'}\, A' \le [\alpha \mapsto D]^n [\alpha_2 \mapsto B']^{n'}\, B'$, which can be proven by the inversion of fourth premise due to the fact that substitution is commutative.

□

Lemma 7 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 8. Unfolding Lemma.

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

3.4 Typing and Reduction Rules

Typing rules. As the top of Figure 7 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 7 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 necessarily 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 (via Lemma 10). Firstly, we need a conventional substitution lemma to deal with beta reduction in preservation:

Lemma 9. Substitution lemma. If $\Gamma_1, x : B, \Gamma_2 \vdash e : A$ and $\Gamma_2 \vdash e' : B$ then $\Gamma_1, \Gamma_2 \vdash [x \mapsto e']\, e : A$.

Then we show how the unfolding lemma is used in the proof on type soundness, via an inversion of typing lemma for fold expressions:

Lemma 10. Inversion of typing for fold expressions: If $\Gamma \vdash \text{fold}\ [A]\ e : S$ and $\Gamma \vdash S \le \mu\alpha.\, B$, then $\exists T,\ \Gamma \vdash e : [\alpha \mapsto \mu\alpha.\, T]\, T \wedge \Gamma \vdash [\alpha \mapsto \mu\alpha.\, T]\, T \le [\alpha \mapsto \mu\alpha.\, B]\, B$.

Proof. Do induction on $\Gamma \vdash \text{fold}\ [A]\ e : S$.

- Rule TYPING-FOLD: the premises become $\Gamma \vdash e : [\alpha \mapsto \mu\alpha.\, A']\, A'$ (assume $A = \mu\alpha.\, A'$) and $\Gamma \vdash \mu\alpha.\, A' \le \mu\alpha.\, B$. In such situation, let $T = A'$, we achieve the goal by applying the unfolding lemma (Lemma 8).
- Rule TYPING-SUB: trivial by applying transitivity (Theorem 6).

□

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

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

$$\frac{\Gamma \vdash e : \mu\alpha.\ A}{\Gamma \vdash \mathsf{unfold}\ [\mu\alpha.\ A]\ e : [\alpha \mapsto \mu\alpha.\ A]\ A}\ \textsc{typing-unfold} \qquad \frac{\Gamma \vdash e : [\alpha \mapsto \mu\alpha.\ A]\ A \qquad \Gamma \vdash \mu\alpha.\ A}{\Gamma \vdash \mathsf{fold}\ [\mu\alpha.\ A]\ e : \mu\alpha.\ A}\ \textsc{typing-fold}$$

$$\frac{\Gamma \vdash e_1 : A_1 \to A_2 \qquad \Gamma \vdash e_2 : A_1}{\Gamma \vdash e_1\ e_2 : A_2}\ \textsc{typing-app} \qquad \frac{\Gamma \vdash e : A \qquad \Gamma \vdash A \le B}{\Gamma \vdash e : B}\ \textsc{typing-sub}$$

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

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

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

Fig. 7. Typing and reduction rules.

Finally, we can proceed to the preservation and progress theorems, and the proof strategy is quite standard.

Theorem 11. Preservation.

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

PROOF. By induction on $\Gamma \vdash e : A$. Most cases are trivial or standard, except for

- Rule TYPING-UNFOLD. In this case, e is decomposed into unfold $[\mu\alpha.\ A]\ e$, and our goal is to prove $\Gamma \vdash e' : [\alpha \mapsto \mu\alpha.\ A]\ A$.
  By inversion on unfold $[\mu\alpha.\ A]\ e \hookrightarrow e'$, we will get two sub-cases.
  - The case for rule STEP-UNFOLD is trivial: e' continues to decompose into unfold $[\mu\alpha.\ A]\ e'$. By applying rule TYPING-UNFOLD and induction hypothesis, we achieve the goal.
  - As for case involving rule STEP-FLD, the first premise becomes $\Gamma \vdash$ fold $[A']\ v : \mu\alpha.\ A$. Then we do the inversion on the first premise again, get two sub-cases. The first case is same as the goal. The second case, raised by rule TYPING-SUB, needs some extra work: what we get now are $\Gamma \vdash$ fold $[A']\ v : S$ and $\Gamma \vdash S \le \mu\alpha.\ A$. Then we apply Lemma 10 (where the unfolding lemma is used) and rule TYPING-SUB to achieve the goal.

□

Theorem 12. Progress.

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

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

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

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

Fig. 8. Algorithmic subtyping.

4 ALGORITHMIC SUBTYPING

In the 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 types checks against an infinite number of conditions (that all finite unfoldings are subtypes). In this section, we first present two sound and complete algorithmic formulations of subtyping. This formulation replaces the declarative rule S-REC by rules based on double unfoldings. A first rule, which we call the double unfolding rule, unfolds the recursive types 1-time and 2-times, respectively. This double unfolding rule relates different other subtyping formulations in this paper, playing a significant role as a hub, as shown in Figure 1. We then give another algorithmic variant, using the nominal unfolding rule, and finally prove our subtyping rules for iso-recursive types are decidable.

4.1 Syntax, Well-Formedness and Subtyping

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

Well-Formedness. In the algorithmic version, we use $\Gamma \vdash A$ to represent that A is well-formed. The rules of $\Gamma \vdash A$ are the same as the top of Figure 6. Similarly to Section 3, we define an alternative variant of well-formedness with the rule WFT-RECUR to give us better induction principles for the proofs.

Definition 13. Rule WFT-RECUR is defined as:

$$\frac{\Gamma, \alpha \vdash A \qquad \Gamma, \alpha \vdash [\alpha \mapsto A]\ A}{\Gamma \vdash \mu\alpha.\ A}\ \textsc{wft-recur}$$

Subtyping. Figure 8 shows the algorithmic subtyping judgment. All the rules, except the one for recursive types, remain the same as the declarative system. In algorithmic subtyping, rule SA-REC states that 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 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 to those used in the declarative system.

Lemma 14. Regularity: If $\Gamma \vdash_a A \leq B$ then $\vdash \Gamma$ and $\Gamma \vdash A$ and $\Gamma \vdash B$.

Theorem 15. Reflexivity.

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

Theorem 16. 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 has the same conditions of the algorithmic system (plus a few more).

Theorem 17. Completeness of algorithmic subtyping.

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

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.

Finding the right generalization for soundness. 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 with such an approach is that we wish to analyze 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 $A := \mathsf{nat} \to A_1 \to A_2$ and $B := \mathsf{nat} \to B_1 \to B_2$. 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 as a subcase 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

$$\Gamma \vdash [\alpha \mapsto (A_1 \to A_2)]^n\ (A_1 \to A_2) \leq [\alpha \mapsto (B_1 \to B_2)]^n\ (B_1 \to B_2)$$

In other words, what gets substituted are not the original types A and B, but only a part of those types ($A_1 \to A_2$ and $B_1 \to B_2$) that is being considered by the current case. Therefore, it is clear that we need some generalization of this lemma. A first idea is to generalize it as follows:

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

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:

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

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 $\Gamma \vdash \alpha \rightarrow \top \leq \alpha \rightarrow \mathsf{nat}$, which is clearly false.

By further analyzing the counter-example, we can see that the influence of contravariance on variables is not reflected in such a lemma. Therefore, our generalized soundness lemma should deal with type variables at contravariant positions and covariant positions respectively, but under the same pattern. In other words we need a pair of lemmas: one to deal with covariance, and another to deal with contravariance.

The generalized lemma. Learning from the lessons of the failed attempts at soundness we reach to the following lemma, which holds:

Lemma 18. If,

(1) $\Gamma \vdash A \leq B$;
(2) $\Gamma \vdash C \leq D$;
(3) $\Gamma \vdash [\alpha \mapsto C]^n\ C \leq [\alpha \mapsto D]^n\ D$.

then

(1) $\Gamma \vdash [\alpha \mapsto C]\ A \leq [\alpha \mapsto D]\ B$ implies $\Gamma \vdash [\alpha \mapsto C]^{n+1}\ A \leq [\alpha \mapsto D]^{n+1}\ B$ and
(2) $\Gamma \vdash [\alpha \mapsto D]\ A \leq [\alpha \mapsto C]\ B$ implies $\Gamma \vdash [\alpha \mapsto D]^{n+1}\ A \leq [\alpha \mapsto C]^{n+1}\ B$.

Proof. By induction on $\Gamma \vdash A \leq B$.

- Case rule S-var: In such case $A = B = \beta$. If $\beta \neq \alpha$, we prove the goal trivially. Otherwise,
  - ⋆ Goal (1): We want to prove $\Gamma \vdash [\alpha \mapsto C]^n\ C \leq [\alpha \mapsto D]^n\ D$, which can be obtained from premise (3).
  - ⋆ Goal (2), We have premises $\Gamma \vdash C \leq D$ by premise (2) and $\Gamma \vdash D \leq C$ from the condition of goal (2), thus $C = D$ by Lemma 19. Goal (2) is proven by reflexivity.
- Case rule S-arrow: In such case $A = A_1 \rightarrow A_2$ and $B = B_1 \rightarrow B_2$.
  - ⋆ Goal (1): We need to prove $\Gamma \vdash [\alpha \mapsto C]^{n+1}\ (A_1 \rightarrow A_2) \leq [\alpha \mapsto D]^{n+1}\ (B_1 \rightarrow B_2)$, which can be rewritten as $\Gamma \vdash ([\alpha \mapsto C]^{n+1}\ A_1) \rightarrow ([\alpha \mapsto C]^{n+1}\ A_2) \leq ([\alpha \mapsto D]^{n+1}\ B_1) \rightarrow ([\alpha \mapsto D]^{n+1}\ B_2)$. By applying rule S-arrow, we need to prove $\Gamma \vdash [\alpha \mapsto D]^{n+1}\ B_1 \leq [\alpha \mapsto C]^{n+1}\ A_1$ and $\Gamma \vdash [\alpha \mapsto C]^{n+1}\ A_2 \leq [\alpha \mapsto D]^{n+1}\ B_2$. The former one can be proved by using the induction hypothesis arising from goal (2), while the latter one can be proved by using the induction hypothesis arising from goal (1).
  - ⋆ Goal (2): We need to prove $\Gamma \vdash [\alpha \mapsto D]^{n+1}\ (A_1 \rightarrow A_2) \leq [\alpha \mapsto C]^{n+1}\ (B_1 \rightarrow B_2)$, which can be rewritten as $\Gamma \vdash ([\alpha \mapsto D]^{n+1}\ A_1) \rightarrow ([\alpha \mapsto D]^{n+1}\ A_2) \leq ([\alpha \mapsto C]^{n+1}\ B_1) \rightarrow ([\alpha \mapsto C]^{n+1}\ B_2)$. By applying rule S-arrow, we need to prove $\Gamma \vdash [\alpha \mapsto C]^{n+1}\ B_1 \leq [\alpha \mapsto D]^{n+1}\ A_1$ and $\Gamma \vdash [\alpha \mapsto D]^{n+1}\ A_2 \leq [\alpha \mapsto C]^{n+1}\ B_2$. The former one can be proved by using the induction hypothesis arising from goal (1), while the latter one can be proved by using the induction hypothesis arising from goal (2).
- Case rule S-rec: Now we assume $A = \mu\alpha'.\ A'$ and $B = \mu\alpha'.\ B'$. Since in such case, we do not need to consider the contravariance, we will just show how to prove goal (1). Goal (2) can be proved using the same approach.
  - The condition arising from the goal (1) becomes $\Gamma \vdash [\alpha \mapsto C]\ \mu\alpha'.\ A' \leq [\alpha \mapsto D]\ \mu\alpha'.\ B'$, which can be rewritten as $\Gamma \vdash \mu\alpha'.\ [\alpha \mapsto C]\ A' \leq \mu\alpha'.\ [\alpha \mapsto D]\ B'$.
  - After inversion, we get $\forall n', \Gamma \vdash [\alpha' \mapsto ([\alpha \mapsto C]\ A')]^{n'} [\alpha \mapsto C]\ A' \leq [\alpha' \mapsto ([\alpha \mapsto D)]\ B']^{n'} [\alpha \mapsto D]\ B'$, which can be rewritten as $\forall n', \Gamma \vdash [\alpha \mapsto C][\alpha' \mapsto A']^{n'}\ A' \leq [\alpha \mapsto D][\alpha' \mapsto B']^{n'}\ B'$.
  - The goal now is $\Gamma \vdash [\alpha \mapsto C]^{n+1}\ \mu\alpha'.\ A' \leq [\alpha \mapsto D]^{n+1}\ \mu\alpha'.\ B'$, which can be rewritten as $\Gamma \vdash \mu\alpha'.\ [\alpha \mapsto C]^{n+1}\ A' \leq \mu\alpha'.\ [\alpha \mapsto D]^{n+1}\ B'$.

- Applying rule S-REC on the goal, we get $\forall n',\ \Gamma, \alpha' \vdash [\alpha' \mapsto ([\alpha \mapsto C]^{n+1}\ A')]^{n'} [\alpha \mapsto C]^{n+1}\ A' \leq [\alpha' \mapsto ([\alpha \mapsto D]^{n+1} B')]^{n'} [\alpha \mapsto D]^{n+1}\ B'$, which can be rewritten as $\forall n',\ \Gamma, \alpha' \vdash [\alpha \mapsto C]^{n+1} [\alpha' \mapsto A']^{n'}\ A' \leq [\alpha \mapsto D]^{n+1} [\alpha' \mapsto B']^{n'}\ B'$.
- Finally, we apply the induction hypothesis, to prove goal (1).

□

Compared with our last failed attempt, there is an extra condition (condition 3). More importantly, there are now two conclusions. These conclusions basically express two different lemmas. One lemma, with all the conditions and conclusion (1), and another lemma with all conditions and conclusion (2). Conclusion (1) covers covariant uses of the lemma, whereas conclusion (2) covers contravariant uses of the lemma. Note that when we apply the lemma in our soundness theorem, we have that $A = C$ and $B = D$. Those types will then become different as the subcases of type A and B are processed. For covariant cases, A is a portion of the type C, and B is a portion of the type D. Conclusion (1) covers this, and we can see that we are substituting C in A and D in B. However, the contravariance of function types will flip the input types being checked for subtyping. This means that in effect, A is now a portion of D (in a contravariant position in D) and B is a portion of C (in a contravariant position in C). Goal (2) captures such nuance and provides a formulation for the lemma that deals with subparts of C and D, which are in contravariant positions.

The proof of Lemma 18 relies on the following property of the subtyping relation:

Lemma 19. Antisymmetry of declarative subtyping: If $\Gamma \vdash A \leq B$ and $\Gamma \vdash B \leq A$ then $A = B$.

Also, from Lemma 18, we now can prove:

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

Proof. Do induction on n. For the base case, we simply apply the premise (1). For the induction case, we apply Lemma 18 with $C = A$ and $D = B$, then apply induction hypothesis. □

The form of Lemma 20 is close to the shape of the infinite unfolding rule (rule S-REC) for 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 The Unfolding Lemma for the Double Unfolding Rules

In Section 3, we showed how to prove the unfolding lemma for the declarative system. It turns out that the unfolding lemma can also be proved relatively easily for the algorithmic system using a technique similar to that employed in the proof of soundness in Section 4.3. A direct proof of the unfolding lemma is useful for language designers wishing to skip the declarative system, and formulate only an algorithmic version.

Lemma 18 provides an interesting (and necessary) lemma for proving soundness between double and finite unfoldings. For that lemma a key insight is that we need two forms: one for dealing with contravariant cases, and another to deal with covariant cases. Inspired by this insight, we are able to prove the unfolding lemma directly for the double unfolding rules, using a similar technique. Firstly we need a lemma similar to Lemma 19, but for the algorithmic relation:

Lemma 22. Antisymmetry of algorithmic subtyping: If $\Gamma \vdash_a A \leq B$ and $\Gamma \vdash_a B \leq A$ then $A = B$.

Then we can formulate the generalized lemma that is needed to prove the unfolding lemma as follows:

Lemma 23. If

(1) $\Gamma_1, \alpha, \Gamma_2, \vdash_a A \le B$;
(2) $\Gamma_1, \alpha, \Gamma_2, \vdash_a C \le D$;
(3) $\Gamma_1, \Gamma_2 \vdash_a \mu\alpha.\, C \le \mu\alpha.\, D$;

then

(1) $\Gamma_1, \alpha, \Gamma_2 \vdash_a [\alpha \mapsto C]\ A \le [\alpha \mapsto D]\ B$ implies $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\, C]\ A \le [\alpha \mapsto \mu\alpha.\, D]\ B$ and
(2) $\Gamma_1, \alpha, \Gamma_2 \vdash_a [\alpha \mapsto D]\ A \le [\alpha \mapsto C]\ B$ implies $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\, D]\ A \le [\alpha \mapsto \mu\alpha.\, C]\ B$.

Proof. Note that premise (2) can be obtained by inversion of premise (3). We explicitly show it here just for convenience. The whole proof follows a similar structure to Lemma 18: we proceed by induction on $\Gamma_1, \alpha, \Gamma_2 \vdash_a A \le B$.

- Case rule SA-var: In such case $A = B = \beta$. If $\beta \neq \alpha$, we simply achieve the goal. Otherwise,
  - ⋆ Goal (1): We want to prove $\Gamma_1, \Gamma_2 \vdash_a \mu\alpha.\, C \le \mu\alpha.\, D$, which is actually premise (3).
  - ⋆ Goal (2): From the condition of goal (2), we have $\Gamma_1, \alpha, \Gamma_2, \vdash_a D \le C$, which is the inverse of premise (2). Thus, we get $C = D$ by Lemma 22. Goal (2) is proven by reflexivity.
- Case rule SA-arrow: In such case $A = A_1 \to A_2$ and $B = B_1 \to B_2$.
  - ⋆ Goal (1):
    - – We need to prove $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\, C]\ (A_1 \to A_2) \le [\alpha \mapsto \mu\alpha.\, D]\ (B_1 \to B_2)$, which can be rewritten as $\Gamma_1, \Gamma_2 \vdash_a ([\alpha \mapsto \mu\alpha.\, C]\ A_1) \to ([\alpha \mapsto \mu\alpha.\, C]\ A_2) \le ([\alpha \mapsto \mu\alpha.\, D]\ B_1) \to ([\alpha \mapsto \mu\alpha.\, D]\ B_2)$.
    - – By applying rule SA-arrow, we need to prove $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\, D]\ B_1 \le [\alpha \mapsto \mu\alpha.\, C]\ A_1$ and $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\, C]\ A_2 \le [\alpha \mapsto \mu\alpha.\, D]\ B_2$.
    - – The former one can be proved by using induction hypothesis arising from goal (2), while the latter one can be proved by using induction hypothesis arising from goal (1).
  - ⋆ Goal (2):
    - – We need to prove $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\, D]\ (A_1 \to A_2) \le [\alpha \mapsto \mu\alpha.\, C]\ (B_1 \to B_2)$, which can be rewritten as $\Gamma_1, \Gamma_2 \vdash_a ([\alpha \mapsto \mu\alpha.\, D]\ A_1) \to ([\alpha \mapsto \mu\alpha.\, D]\ A_2) \le ([\alpha \mapsto \mu\alpha.\, C]\ B_1) \to ([\alpha \mapsto \mu\alpha.\, C]\ B_2)$.
    - – By applying rule SA-arrow, we need to prove $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\, C]\ B_1 \le [\alpha \mapsto \mu\alpha.\, D]\ A_1$ and $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\, D]\ A_2 \le [\alpha \mapsto \mu\alpha.\, C]\ B_2$.
    - – The former one can be proved by using induction hypothesis arising from goal (1), while the latter one can be proved by using induction hypothesis arising from goal (2).
- Case rule SA-rec: Now we assume $A = \mu\alpha'.\, A'$ and $B = \mu\alpha'.\, B'$. Since in such case, we do not need to consider the contravariance, we will just show how to prove goal (1). Goal (2) can be proven with the same approach.
  - – The goal now is $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\, C]\ \mu\alpha'.\, A' \le [\alpha \mapsto \mu\alpha.\, D]\ \mu\alpha'.\, B'$, which can be rewritten as $\Gamma_1, \Gamma_2 \vdash_a \mu\alpha'.\, [\alpha \mapsto \mu\alpha.\, C]\ A' \le \mu\alpha'.\, [\alpha \mapsto \mu\alpha.\, D]\ B'$.
  - – The condition arising from the goal (1) becomes $\Gamma_1, \alpha, \Gamma_2 \vdash_a [\alpha \mapsto C]\ \mu\alpha'.\, A' \le [\alpha \mapsto D]\ \mu\alpha'.\, B'$, which can be rewritten as $\Gamma_1, \alpha, \Gamma_2 \vdash_a \mu\alpha'.\, [\alpha \mapsto C]\ A' \le \mu\alpha'.\, [\alpha \mapsto D]\ B'$.
  - – Do inversion on this condition and reorder the context and substitution, we get two new conditions: $\Gamma_1, \alpha, \Gamma_2, \alpha' \vdash_a [\alpha \mapsto C]\ A' \le [\alpha \mapsto D]\ B'$ and $\Gamma_1, \alpha, \Gamma_2, \alpha' \vdash_a [\alpha \mapsto C][\alpha' \mapsto A']\ A' \le [\alpha \mapsto D][\alpha' \mapsto B']\ B'$.
  - – Because of the double unfolding rule, we will have two induction hypotheses, which are
    - ⋆ I.H.(1), which comes from 1-time unfolding : $\Gamma_1, \alpha, \Gamma_2, \vdash_a C \le D \Rightarrow \Gamma_1, \Gamma_2 \vdash_a \mu\alpha.\, C \le \mu\alpha.\, D \Rightarrow \Gamma_1, \alpha, \Gamma_2, \alpha' \vdash_a [\alpha \mapsto C]\ A' \le [\alpha \mapsto D]\ B' \Rightarrow \Gamma_1, \Gamma_2, \alpha' \vdash_a [\alpha \mapsto \mu\alpha.\, C]\ A' \le [\alpha \mapsto \mu\alpha.\, D]\ B'$.

  - ⋆ I.H.(2), which comes from 2-times unfolding : $\Gamma_1, \alpha, \Gamma_2, \vdash_a C \leq D \Rightarrow \Gamma_1, \Gamma_2 \vdash_a \mu\alpha.\, C \leq \mu\alpha.\, D \Rightarrow \Gamma_1, \alpha, \Gamma_2, \alpha' \vdash_a [\alpha \mapsto C][\alpha' \mapsto A']\, A' \leq [\alpha \mapsto D][\alpha' \mapsto B']\, B' \Rightarrow \Gamma_1, \Gamma_2, \alpha' \vdash_a [\alpha \mapsto \mu\alpha.\, C][\alpha' \mapsto A']\, A' \leq [\alpha \mapsto \mu\alpha.\, D][\alpha' \mapsto B']\, B'$.
- Apply rule SA-REC on the goal, we obtain two sub-goals: $\Gamma_1, \Gamma_2, \alpha' \vdash_a [\alpha \mapsto \mu\alpha.\, C]\, A' \leq [\alpha \mapsto \mu\alpha.\, D]\, B'$ and $\Gamma_1, \Gamma_2, \alpha' \vdash_a [\alpha' \mapsto ([\alpha \mapsto \mu\alpha.\, C]\, A')][\alpha \mapsto \mu\alpha.\, C]\, A' \leq [\alpha' \mapsto ([\alpha \mapsto \mu\alpha.\, D]\, B')][\alpha \mapsto \mu\alpha.\, D]\, B'$.
- For the former one, we apply the I.H.(1). As for the latter one, after rewriting the goal to $\Gamma_1, \Gamma_2, \alpha' \vdash_a [\alpha \mapsto \mu\alpha.\, C][\alpha' \mapsto A']\, A' \leq [\alpha \mapsto \mu\alpha.\, D][\alpha' \mapsto B']\, B'$, we apply the I.H.(2).

□

Like Lemma 18, in Lemma 23 the two conclusions are basically reflecting two lemmas: one for covariant uses (when A is a part of C and B is a part of D), and another for contravariant uses (when A is a part of D and B is a part of C). By letting $C := A$, $D := B$, we easily obtain:

Lemma 24. Unfolding Lemma.

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

4.5 Nominal Unfoldings

In this subsection, we will describe the nominal unfolding rule, which is another algorithmic variant equivalent to declarative subtyping. Compared with the double unfolding rules, nominal unfoldings have better efficiency (since only one premise is needed), while eliminating spurious subtyping derivations that arise with double unfoldings (see example in Section 2.7).

Syntax and well-formedness. The syntax of contexts for this calculus is the same as Section 3. For the syntax of types, based on the syntax in Section 3, we extend it with *labelled types* $\{\alpha : A\}$. Labelled types can be viewed as a simple form of nominal types. They are essentially a pair that contains a name (or type variable) α and a type.

The well-formedness $\Gamma \vdash A$ is also defined as Section 3, but for recursive types and labelled types (the top of Figure 9). To get a better induction hypothesis, we slightly modify the form of well-formed recursive types, as rule WFT-NOMINAL shows. As before, rule WFT-NOMINAL is proven to be equivalent to rule WFT-REC. The first premise $\Gamma, \alpha \vdash A$ might appear redundant at first glance, but it is indeed necessary, because from the second premise $\Gamma, \alpha \vdash [\alpha \mapsto \{\alpha : A\}]\, A$, we cannot derive $\Gamma, \alpha \vdash A$, which is the insurance with respect to the correctness of substitution during the proof. Meanwhile, since we introduce labelled types, as rule WFT-LABEL shows, a labelled type is well-formed if its inner type is well-formed.

Subtyping. The bottom of Figure 9 shows the definition of subtyping with the nominal unfolding rule. We denote subtyping for nominal unfoldings as $\Gamma \vdash_n A \leq B$. Rules SN-NAT, SN-TOP, SN-VAR, and SN-ARROW are the same as the corresponding double unfolding subtyping rules. Rule SN-RCD is new, stating that a labelled type is a subtype of another labelled type if the two types are labelled with the same name and $A \leq B$.

Rule SN-REC, the nominal unfolding rule, is the most interesting one. This rule follows an idea quite similar to the double unfolding rule. The body of the recursive type is unfolded twice. However, for the innermost unfolding, the type that we substitute is not the type of the body directly. Instead, we use a labelled type, where the label has the same name as the recursive variable α, and the type that is labelled is the body of the recursive type. In other words, instead of using the double unfolding $[\alpha \mapsto A]\, A$ we use $[\alpha \mapsto \{\alpha : A\}]\, A$. The label is crucial to avoid spurious subtyping derivations, and it is also the reason why in the nominal unfolding formulation we do not need to check the subtyping of single unfoldings as well. In the double unfolding rule, there is an extra

$\boxed{\Gamma \vdash A}$ *(Well-Formed Type, Selected Rules)*

$$\text{WFT-LABEL}\ \frac{\Gamma \vdash A}{\Gamma \vdash \{\alpha : A\}} \qquad \text{WFT-NOMINAL}\ \frac{\Gamma, \alpha \vdash A \qquad \Gamma, \alpha \vdash [\alpha \mapsto \{\alpha : A\}]\ A}{\Gamma \vdash \mu\alpha.\ A}$$

$\boxed{\Gamma \vdash_n A \leq B}$ *(Nominal Subtyping)*

$$\text{SN-NAT}\ \frac{\vdash \Gamma}{\Gamma \vdash_n \mathsf{nat} \leq \mathsf{nat}} \qquad \text{SN-TOP}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash_n A \leq \top} \qquad \text{SN-VAR}\ \frac{\vdash \Gamma \qquad \alpha \in \Gamma}{\Gamma \vdash_n \alpha \leq \alpha} \qquad \text{SN-ARROW}\ \frac{\Gamma \vdash_n B_1 \leq A_1 \qquad \Gamma \vdash_n A_2 \leq B_2}{\Gamma \vdash_n A_1 \to A_2 \leq B_1 \to B_2}$$

$$\text{SN-REC}\ \frac{\Gamma, \alpha \vdash_n [\alpha \mapsto \{\alpha : A\}]\ A \leq [\alpha \mapsto \{\alpha : B\}]\ B}{\Gamma \vdash_n \mu\alpha.\ A \leq \mu\alpha.\ B} \qquad \text{SN-RCD}\ \frac{\Gamma \vdash_n A \leq B}{\Gamma \vdash_n \{\alpha : A\} \leq \{\alpha : B\}}$$

Fig. 9. Well-formedness and subtyping rules for nominal unfoldings.

premise that checks the single unfolding and prevents certain cases of spurious subtyping. The absence of this extra premise also makes some of the metatheory simpler.

Basic properties. All the proofs about reflexivity, transitivity and unfolding lemma for nominal unfoldings are almost the same as double unfoldings, since both subtyping rules are based on two times finite unfoldings. We list all the theorems here and skip the details (the reader can consult our mechanized proofs for full details).

Theorem 25. Reflexivity.

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

Theorem 26. Transitivity.

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

Lemma 27. Unfolding Lemma.

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

Another important property is that, from the nominal unfolding rules, we can derive 1-time finite unfoldings. This lemma is important to show that nominal unfoldings subsume the double unfolding rule:

Lemma 28. If $\Gamma, \alpha \vdash_n [\alpha \mapsto \{\alpha : A\}]\ A \leq [\alpha \mapsto \{\alpha : B\}]\ B$ then $\Gamma, \alpha \vdash_n A \leq B$.

4.6 Equivalence between Nominal Unfoldings and Double Unfoldings

The subtyping relation presented in Section 4.5 is equivalent to a subtyping relation that uses the double unfolding rules for recursive types. This equivalence is not surprising, since nominal unfoldings are essentially the double unfolding rule with an extra label and without the one time finite unfolding premise. Lemma 28 and some other similar auxiliary lemmas are used to formulate the equivalence between the two encodings. The most interesting aspect of the equivalence proof is that we need to translate types for the nominal unfolding formulation into types of the double-unfolding formulation. Such a translation is necessary because nominal unfoldings require labelled types, which do not exist in the double unfolding formulation. Thus, the translation simply erases the labels.

Definition 29. The *erase* ($\searrow$) function is defined as:

$$
\begin{array}{rcl}
\mathsf{nat}\searrow & = & \mathsf{nat} \\
\top\searrow & = & \top \\
\alpha\searrow & = & \alpha \\
(A \to B)\searrow & = & A\searrow \to B\searrow \\
(\mu\alpha.\, A)\searrow & = & \mu\alpha.\, A\searrow \\
\{\alpha : A\}\searrow & = & A\searrow
\end{array}
$$

With the erasure function we can conclude that our nominal unfoldings are equivalent to double unfoldings with the following two lemmas:

Theorem 30. If $\Gamma \vdash_n A \leq B$ then $\Gamma \vdash_a A\searrow \leq B\searrow$.

Theorem 31. If $\Gamma \vdash_a A \leq B$ then $\Gamma \vdash_n A \leq B$.

For Theorem 30 we wish to show that all valid subtyping statements using nominal unfoldings are also valid under the double unfolding formulation. To show this result we have to apply the erasure function to the types, since the types in the nominal unfolding formulation may contain labels. For Theorem 31 no erasure function is necessary since the types in the double unfolding formulation are a subset of those in the nominal unfolding formulation. Thus, they can be directly mapped. As a consequence of the two theorems above, our nominal unfoldings are also sound and complete with respect to our specification using finite unfoldings.

Corollary 32. If $\Gamma \vdash_n A \leq B$ then $\Gamma \vdash A\searrow \leq B\searrow$.

Corollary 33. If $\Gamma \vdash A \leq B$ then $\Gamma \vdash_n A \leq B$.

4.7 Decidability

Our subtyping rules are decidable. We have already proved the equivalence between the rules employing nominal, double and finite unfoldings. Since both the nominal and the double unfolding rules are syntax directed, they provide a useful foundation to prove decidability. We have proved decidability based on our nominal rule and a measure that is based on the depth of the unfolded tree. A similar proof should be possible using the double unfolding rule, except that with the double unfolding rule there is some extra work because of the extra 1-time finite unfolding premise.

Our subtyping rules are based on substitution, which can increase the size of types after an unfolding. Therefore, a straightforward induction on the size of types will not work. A first idea may be doing induction lexicographically on a pair with the number of nesting of recursive binders, and the size of types. The logic is that, after a nominal unfolding, the recursive binder that we are going to unfold will not reappear again. However, this does not quite work because the bodies of recursive types can contain other recursive types and the substitutions may introduce new copies of those recursive types. Thus, the subtyping rule for recursive types does not necessarily reduce the number of recursive binders. Consider, for instance, the following example:

$$\mu\alpha.\, \mu\beta.\, \alpha \to \beta$$

After the nominal unfolding and α-conversion, the type will become:

$$\mu\beta.\, \{\alpha : \mu\beta'.\, \alpha \to \beta'\} \to \beta$$

which does not decrease the number of recursive binders. Nevertheless, if we continue to process the types using nominal unfolding, we will finally reach a type without any recursive binders. After a few more steps in the subtyping derivation, we obtain:

$$\{\alpha : \mu\beta'.\, \alpha \to \beta'\} \to \{\beta : \{\alpha : \mu\beta'.\, \alpha \to \beta'\} \to \beta\}$$

and the inner recursive types $\mu\beta'.\ \alpha \to \beta'$ no longer contain recursive types in their bodies, and we will finally obtain types free of recursive types after another round of nominal unfolding.

Measure based on the depth of the unfolded tree. To provide a measure that decreases at every nominal unfolding, we define a function based on the depth of the expanded tree of a type. This function essentially simulates the unfolding process of the tree using nominal unfoldings and allow us to obtain an (over-)approximation of the depth of the (fully) unfolded tree.

Definition 34. The *height* of a type A in a context Ψ ($\Psi := \cdot \mid \Psi, \alpha \mapsto i$, where i represents a natural number), written $height_\Psi(A)$, is defined as follows:

$$
\begin{array}{lcll}
height_\Psi(\mathrm{nat}) & = & 0 & \\
height_\Psi(\top) & = & 0 & \\
height_\Psi(A_1 \to A_2) & = & max(height_\Psi(A_1),\ height_\Psi(A_2)) + 1 & \\
height_\Psi(\{\alpha : A\}) & = & height_\Psi(A) + 1 & \\
height_\Psi(\alpha) & = & \Psi(\alpha) & (\alpha \in \Psi) \\
height_\Psi(\alpha) & = & 0 & (\alpha \notin \Psi) \\
height_\Psi(\mu\alpha.\ A) & = & \text{let } i = height_{\Psi,\ \alpha\mapsto 0}(A) \text{ in } height_{\Psi,\ \alpha\mapsto i+1}(A) + 1 &
\end{array}
$$

The two interesting cases in the *height* function are for recursive variables and recursive types. For a recursive variable, if it can be found in the context, we retrieve the corresponding height associated with the recursive variable from the context, whereas we return 0 if α is not in the context. Note that, when performing subtyping on two closed types (which is always the case in the subsumption rule) the latter case never happens. However, to make *height* total we have to consider this case too, and therefore our height function applies even to types which are not well-formed. For a recursive type, we firstly compute the height of its body by assuming that the height of its binder is 0. In other words i is the height of the one time finite unfolding. Then we compute the height of the body again, but this time assuming that the height of its binder is $i + 1$ (i.e. the size of the one-time unfolding plus 1). This basically computes the overall height of the nominal unfolding. Since we compute the height two times for a recursive type, our *height* function is convex: its second derivative with respect to the number of recursive types is non-negative thus a linear over-approximation is impossible.

Finally, the measure of a type A is defined as $height(A)$, which is the height of expanded tree when the context Ψ is empty.

Decidability. With the new measure, now we can prove the decidability lemma. For a non-recursive type, it is obvious that the height of a conclusion from any inputs is strictly greater than the height of its any premises. For a recursive type, the measure will decrease by 1 after a nominal unfolding. In other words, what we want to show is

$$height(\mu\alpha.\ A) - 1 = height_{\alpha\mapsto height_{\alpha\mapsto 0}(A)+1}(A) = height([\alpha \mapsto \{\alpha \mapsto A\}]\ A).$$

Firstly, it is easy to observe that $height_{\alpha\mapsto 0}(A) = height(A)$, because for a variable, it is either found at the context, which is $\Psi(\alpha) = 0$, or not found at the context, which will return 0. Then, when we try to compute $height([\alpha \mapsto \{\alpha : A\}]\ A)$, since α is substituted by $\{\alpha : A\}$ and α is not in the context, the formula can be rewritten as $height_{\alpha\mapsto height(\{\alpha:A\})}(A)$, in which we do not try to proceed with the substitution, but just return the result from the context. We can continue to rewrite this formula as $height_{\alpha\mapsto height(A)+1}(A)$. Through $height_{\alpha\mapsto 0}(A) = height(A)$, the formula is equal to $height_{\alpha\mapsto height_{\alpha\mapsto 0}(A)+1}(A)$. Therefore, we have proven our proposition. Next we can prove that this measure suffices to show the termination of subtyping with nominal unfoldings:

Lemma 35. If $max(height(A),\ height(B)) \le k$, then $\Gamma \vdash_n A \le B$ or not $\Gamma \vdash_n A \le B$.

Proof. Do induction on k, A and B, respectively. □

Let $k = max(height(A),\ height(B))$, we obtain:

Theorem 36. Termination:

For any inputs Γ, A and B, we either have $\Gamma \vdash_n A \leq B$ or not $\Gamma \vdash_n A \leq B$.

Finally, from termination and the soundness and completeness of subtyping based on nominal unfoldings with respect to subtyping based on finite unfoldings we can conclude that our specification of iso-recursive subtyping is decidable.

Corollary 37. Decidability: Our specification for iso-recursive subtyping is decidable.

5 EQUIVALENCE TO THE AMBER RULES

This section shows a variant of the Amber rules that is equivalent, in terms of expressive power, to our new formulation of subtyping. We prove the equivalence via soundness and completeness theorems between the two formulations of subtyping. The soundness lemma implies that if two types are subtypes under the Amber rules, they are subtypes under our new formulation. The completeness lemma implies that if two types are subtypes under our new formulation, they are subtypes under the Amber rules. With both lemmas we can conclude that our formulation and the Amber rules have the same expressiveness. To establish the soundness and completeness results we have to impose some well-formedness conditions. These conditions have been omitted in early formulations of the Amber rules (as mentioned in Section 2.5), 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 3) there are no well-formedness constraints. Unfortunately, defining such well-formedness constraints is not entirely trivial. Furthermore, for those interested in mechanical formalization 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:

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

which states that if a subtyping statement is valid then the context and 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 types uses two distinct type variables α and β in the recursive types. The use of such distinct type variables is a 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 type variables $\overline{\alpha \leq \beta}$ 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 in 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 in 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 in the other side, turns out to be quite tricky. Therefore, to make progress, we propose a weaker restriction in this section: we allow both α's and β's to be in scope for both types. Thus, the following subtyping statement is valid with our variant of the Amber rules: $\alpha \leq \beta \vdash \alpha \to \beta \leq \top$. In other words, we accept some subtyping statements that one would perhaps expect to be ill-formed or rejected. That is, in the Amber rules, if we have $\alpha \leq \beta$ in Γ, we would not expect that α and β appear in the same type. Rather we would expect that the α appears in one of the types, and β in the other one. However, accepting such subtyping statements is not harmful: we can still prove the soundness and completeness 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 10 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. With our definition of well-formed types regularity is easy to obtain:

Lemma 38. Regularity: If $\Delta \vdash_{amb} A \leq B$ then $\vdash \Delta$ and $\Delta \vdash A$ and $\Delta \vdash B$.

Subtyping. The subtyping relation is almost the same as the original rules by Amadio and Cardelli [1993] in Figure 3. 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 A Third Subtyping Relation Based on a Weakly Positive Restriction

To prove the soundness and completeness 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 11, is equivalent to the Amber rules in Figure 10. The key idea in this relation is to have a rule for recursive types (rule PosRes-rec), which only accepts weakly positive subtyping. This formulation is inspired by the existing positive formulation of subtyping for recursive types [Amadio and Cardelli 1993; Appel and Felty 2000; Backes et al. 2014], but it is more general.

In essence, what we mean by weakly positive subtyping is that we can never find a contravariant subderivation $\alpha \leq \alpha$, where α is a recursive type variable, for *non-equal* recursive types. For instance this excludes $\mu\alpha.\alpha \to \mathsf{nat} \leq \mu\alpha.\alpha \to \top$, since here α is used contravariantly, and $\alpha \leq \alpha$ would appear as a subderivation. Notice, however, that weakly positive subtyping still allows subtyping of recursive types with negative occurrences of the recursive type variable in two cases:

- **Equal types:** If the recursive types are equal, then weakly positive subtyping still considers the two types to be subtypes. For instance $\mu\alpha.\alpha \to \mathsf{nat} \leq \mu\alpha.\alpha \to \mathsf{nat}$, is a valid subtyping statement.

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

$$\text{WFAmber-nat}\ \frac{\vdash \Delta}{\Delta \vdash \mathsf{nat}} \qquad \text{WFAmber-Top}\ \frac{\vdash \Delta}{\Delta \vdash \top} \qquad \text{WFAmber-varl}\ \frac{\vdash \Delta \qquad \alpha \leq \beta \in \Delta}{\Delta \vdash \alpha} \qquad \text{WFAmber-varr}\ \frac{\vdash \Delta \qquad \alpha \leq \beta \in \Delta}{\Delta \vdash \beta}$$

$$\text{WFAmber-arrow}\ \frac{\Delta \vdash A_1 \qquad \Delta \vdash A_2}{\Delta \vdash A_1 \to A_2} \qquad \text{WFAmber-rec}\ \frac{\Delta, \alpha \leq \beta \vdash A \qquad \beta \text{ is fresh}}{\Delta \vdash \mu\alpha.\ A}$$

$\boxed{\Delta \vdash_{amb} A \leq B}$ *(Amber Rules)*

$$\text{Amber-nat}\ \frac{\vdash \Delta}{\Delta \vdash_{amb} \mathsf{nat} \leq \mathsf{nat}} \qquad \text{Amber-top}\ \frac{\vdash \Delta \qquad \Delta \vdash A}{\Delta \vdash_{amb} A \leq \top} \qquad \text{Amber-arrow}\ \frac{\Delta \vdash_{amb} B_1 \leq A_1 \qquad \Delta \vdash_{amb} A_2 \leq B_2}{\Delta \vdash_{amb} A_1 \to A_2 \leq B_1 \to B_2}$$

$$\text{Amber-var}\ \frac{\vdash \Delta \qquad \alpha \leq \beta \in \Delta}{\Delta \vdash_{amb} \alpha \leq \beta} \qquad \text{Amber-rec}\ \frac{\Delta, \alpha \leq \beta \vdash_{amb} A \leq B}{\Delta \vdash_{amb} \mu\alpha.\ A \leq \mu\beta.\ B} \qquad \text{Amber-self}\ \frac{\vdash \Delta \qquad \Delta \vdash \mu\alpha.\ A}{\Delta \vdash_{amb} \mu\alpha.\ A \leq \mu\alpha.\ A}$$

Fig. 10. A variant of the Amber rules, including well-formedness of types.

- **The recursive type variable is a subtype of $\top$:** If a recursive type variable appears negatively, but the only (negative) subderivations are of the form $\alpha \leq \top$, then that is allowed in weakly positive subtyping. For instance $\mu\alpha.\top \to \alpha \leq \mu\alpha.\alpha \to \alpha$ is a valid weakly positive subtyping statement.

These two exceptions are why we use the term "weakly" to characterize such formulation of subtyping. In contrast, existing formulations of positive subtyping, such as that described in Section 2.4 or originally described by Amadio and Cardelli [1993] do not make such exceptions and would reject the subtyping statements that we have described above.

Well-formedness and weakly positive relation. Well-formed types are the same as in Figure 6. To examine whether a type variable occurs positively in a subtyping relation, we define a weakly positive restriction relation $\alpha \in_m A \leq B$ at the top of Figure 11. Here, $\alpha \in_m A \leq B$ means that: type variable α occurs in the derivation $A \leq B$ with a mode m, where a mode m is either positive (+) or negative (-)[4]. This relation checks that every instance of $\alpha \leq \alpha$ in the proof derivation of $A \leq B$ is found in a positive position inside the proof (rule Pos-varx). Moreover, for every subderivation of $A \leq B$ with shape $\mu\beta.\ A' \leq \mu\beta.\ B'$ either 1) $A' = B'$ and α is not free in A' (rule Posvar-recself), or 2) $\beta \in_+ A' \leq B'$ (rule Posvar-rec).

For example, $\alpha \in_+ \top \to \alpha \leq \alpha \to \alpha$ holds, since the only instance of $\alpha \leq \alpha$ occurs positively and there are no recursive types inside, so the second condition does not apply. To see the need for the second condition, consider:

$$\beta \in_+ \mu\alpha.\ \alpha \to \beta \leq \mu\alpha.\ \alpha \to \beta$$

which might seem to hold according to the syntax, since β appears only in positive positions. However, it is rejected by both rule Posvar-rec and rule Posvar-recself. Rule Posvar-rec requires that α also appears positively in subderivations, which does not hold in this example. The reason

[4]Note that $\in_m$ is just part of the syntax of the relation, rather than a separate operator.

$\boxed{\alpha \in_m A \leq B}$ *(Weakly Positive Restriction)*

$$\frac{}{\alpha \in_m \mathsf{nat} \leq \mathsf{nat}}\ \textsc{Posvar-nat} \qquad \frac{}{\alpha \in_m A \leq \top}\ \textsc{Posvar-topL} \qquad \frac{}{\alpha \in_m \top \leq A}\ \textsc{Posvar-topR}$$

$$\frac{\alpha \in_{\overline{m}} B_1 \leq A_1 \qquad \alpha \in_m A_2 \leq B_2}{\alpha \in_m A_1 \to A_2 \leq B_1 \to B_2}\ \textsc{Posvar-arrow}$$

$$\frac{}{\alpha \in_+ \alpha \leq \alpha}\ \textsc{Posvar-varX} \qquad \frac{\alpha \neq \beta}{\alpha \in_m \beta \leq \beta}\ \textsc{Posvar-varY} \qquad \frac{\beta \notin fv(A)}{\beta \in_m \mu\alpha.\ A \leq \mu\alpha.\ A}\ \textsc{Posvar-recself}$$

$$\frac{\beta \in_m A \leq B \qquad \alpha \in_+ A \leq B \qquad \alpha \neq \beta}{\beta \in_m \mu\alpha.\ A \leq \mu\alpha.\ B}\ \textsc{Posvar-rec}$$

$\boxed{\Gamma \vdash A \leq_+ B}$ *(Weakly Positive Subtyping)*

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{nat} \leq_+ \mathsf{nat}}\ \textsc{PosRes-nat} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash A \leq_+ \top}\ \textsc{PosRes-top} \qquad \frac{\vdash \Gamma \qquad \alpha \in \Gamma}{\Gamma \vdash \alpha \leq_+ \alpha}\ \textsc{PosRes-var}$$

$$\frac{\Gamma \vdash B_1 \leq_+ A_1 \qquad \Gamma \vdash A_2 \leq_+ B_2}{\Gamma \vdash A_1 \to A_2 \leq_+ B_1 \to B_2}\ \textsc{PosRes-arrow}$$

$$\frac{\Gamma, \alpha \vdash A \leq_+ B \qquad \alpha \in_+ A \leq B}{\Gamma \vdash \mu\alpha.\ A \leq_+ \mu\alpha.\ B}\ \textsc{PosRes-rec} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash \mu\alpha.\ A}{\Gamma \vdash \mu\alpha.\ A \leq_+ \mu\alpha.\ A}\ \textsc{PosRes-self}$$

Fig. 11. Weakly Positive Subtyping Rules.

we pose such restriction is because unfolding both types results in the following judgment

$$\beta \in_+ (\mu\alpha.\ \alpha \to \beta) \to \beta \leq (\mu\alpha.\ \alpha \to \beta) \to \beta$$

where a negative occurrence of $\beta \leq \beta$ would appear in a subderivation. A similar issue happens whenever $\alpha \leq \alpha$ appears negatively and the recursive types are not equal to each other.

There are also some noteworthy points in the other rules for the weakly positive restriction relation. In rule Posvar-arrow, for the contravariant types, we switch their mode by a flip operation $\bar{m}$: $\overline{+} = -$ and $\overline{-} = +$. Rule Posvar-varY states that if α is not equal to β, we do not care what the mode for β is. Rule Posvar-topR, at first glance, looks suspicious, since it seems to indicate that $\top \leq A$ is valid. In this rule the choice of notation for the relation, using $\leq$, may be a little misleading. Although normally we follow the derivation of the subtyping relation, the mode is determined by the position and not by whether the two types are subtypes. The addition of rule Posvar-topR is not harmful: the relation is always accompanied by weakly positive subtyping derivations, and $\top \leq A$ never occurs in such subtyping derivations. The reason to include rule Posvar-topR is that we wish that our weakly positive restriction relation is symmetric: if $\alpha \in_m A \leq B$ then $\alpha \in_m B \leq A$. This symmetry property is important for the proof of Lemma 47.

Subtyping. Most subtyping rules are identical to those of the Amber rules, and the only differences are rule PosRes-var, rule PosRes-rec and rule PosRes-self. The rule PosRes-var is similar to our formulations, checking whether two variables are same. The latter two rules state that: 1) two recursive types are subtypes if they are equal (rule PosRes-self); or 2) the recursive variable satisfies the weakly positive restriction and the two bodies are subtypes (rule PosRes-rec).

Basic properties. Reflexivity is straightforward since we have explicit reflexivity built-in for recursive types.

Theorem 39. Reflexivity.

$$\text{If } \vdash \Gamma \text{ and } \Gamma \vdash A \text{ then } \Gamma \vdash A \leq_+ A.$$

As for transitivity, because we have the weakly positive restriction for recursive subtyping, the proof is a bit complex. We need to prove an auxiliary lemma in advance:

Lemma 40. If

(1) $\Gamma \vdash A \leq_+ B$;
(2) $\Gamma \vdash B \leq_+ C$;
(3) $\alpha \in_m A \leq B$;
(4) $\alpha \in_m B \leq C$,

then $\Gamma \vdash A \leq_+ C$ and $\alpha \in_m A \leq C$.

Proof. Induction on $\Gamma \vdash B$. □

Then we can have the transitivity theorem.

Theorem 41. Transitivity.

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

Proof. Induction on $\Gamma \vdash B$. For the recursive case, apply lemma 40, we have all premises. □

Finally, it is also possible to prove the unfolding lemma for weakly positive subtyping:

Lemma 42. Unfolding Lemma.

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

The proof employs similar techniques to those used for the soundness lemma (Lemma 49). We skip the details here.

5.4 The Soundness Theorem

To show that Amber subtyping is sound with respect to weakly positive subtyping and the double unfolding rules, we need to translate the environments and types used in the Amber formulation, since they have different forms.

Definition 43. Translation of environments and types from the Amber rules.

$$|\cdot| = \cdot \qquad\qquad (\cdot)(A) = A$$
$$|\Delta,\ \alpha \leq \beta| = |\Delta|,\ \alpha \qquad (\Delta,\ \alpha \leq \beta)(A) = (\Delta)([\beta \mapsto \alpha]\ A)$$

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 $\alpha \vdash \alpha \to \top \leq \alpha \to \top$.

Before showing the relationship between the Amber subtyping and our subtyping with the positive restriction, we must prove an important auxiliary lemma:

Lemma 44. If $\Delta \vdash_{amb} A \leq B$ and $(\alpha \leq \beta) \in \Delta$, then

(1) $\alpha \notin fv(B)$ and $\beta \notin fv(A)$ implies $\alpha \in_+ (\Delta)(A) \leq (\Delta)(B)$ and
(2) $\alpha \notin fv(A)$ and $\beta \notin fv(B)$ implies $\alpha \in_- (\Delta)(A) \leq (\Delta)(B)$.

Proof. Do induction on $\Delta \vdash_{amb} A \leq B$.

- Rule Amber-var: In such case $A = \alpha'$ and $B = \beta'$.
  - ⋆ Goal (1): If $\alpha \neq \alpha'$, we achieve the goal (recall that $\alpha \in_m \alpha' \leq \alpha'$ always holds for any mode m). Otherwise, we know that $\alpha = \alpha'$. Since $(\alpha \leq \beta) \in \Delta$, the goal becomes $\alpha \in_+ \alpha \leq \alpha$.
  - ⋆ Goal (2): $\alpha \notin fv(A)$ implies $\alpha \neq \alpha'$.
- Rule Amber-rec: Assume $A = \mu\alpha'.\ A'$ and $B = \mu\beta'.\ B'$, then the goal becomes $\alpha \in_m (\Delta)(\mu\alpha'.\ A') \leq (\Delta)(\mu\beta'.\ B')$ (m is $+$ and $-$, respectively, for two goals), which can be rewritten as $\alpha \in_m \mu\alpha'.\ (\Delta)(A') \leq \mu\beta'.\ (\Delta)(B')$. Note that here, in the new goal, we use different bound variables α' and β' in the μ binders to match with the names of the free variables that are added to the environment Δ next. However, we can freely rename the bound variables to the same variable, and we will indeed rename β' below to α', since the rules for recursive types in the weakly positive subtyping require the same bound variable names[5].
  For convenience, let us denote $\Delta, (\alpha' \leq \beta')$ as Δ'. Then the induction hypotheses become:
  (1) $\forall \alpha\ \beta,\ (\alpha \leq \beta) \in \Delta' \Rightarrow \alpha \notin fv(B') \Rightarrow \beta \notin fv(A') \Rightarrow \alpha \in_+ (\Delta')(A') \leq (\Delta')(B')$ and
  (2) $\forall \alpha\ \beta,\ (\alpha \leq \beta) \in \Delta' \Rightarrow \alpha \notin fv(A') \Rightarrow \beta \notin fv(B') \Rightarrow \alpha \in_- (\Delta')(A') \leq (\Delta')(B')$.
  - ⋆ For goal (1): we apply rule Posvar-rec, then we need to check if $\alpha' \in_+ (\Delta')(A') \leq (\Delta')(B')$ and $\alpha \in_+ (\Delta')(A') \leq (\Delta')(B')$. Both cases can be solved by applying induction hypothesis (1).
  - ⋆ For goal (2): we apply rule Posvar-rec, then we need to check if $\alpha' \in_- (\Delta')(A') \leq (\Delta')(B')$ and $\alpha \in_+ (\Delta')(A') \leq (\Delta')(B')$. For the former one we apply induction hypothesis (2), and for the latter one we apply induction hypothesis (1).
- Rule Amber-self: Assume $A = B = \mu\alpha'.\ A'$. From the condition of the goal, we know that $\alpha \notin fv(A')$ always holds, thus $\alpha \in_m (\Delta)(A) \leq (\Delta)(B)$ is true for any mode m.

□

With the help of Lemma 44, we can prove that if two types are subtypes under the Amber rules, they are also subtypes under weakly positive subtyping:

Theorem 45. If $\Delta \vdash_{amb} A \leq B$ then $|\Delta| \vdash (\Delta)(A) \leq_+ (\Delta)(B)$.

Proof. Do induction on $\Delta \vdash_{amb} A \leq B$. We show only the more interesting case for recursive types.

- Rule Amber-rec: Assume $A = \mu\alpha'.\ A'$ and $B = \mu\beta'.\ B'$. The goal becomes $|\Delta| \vdash (\Delta)(\mu\alpha'.\ A') \leq (\Delta)(\mu\beta'.\ B')$, which can be rewritten as $|\Delta| \vdash \mu\alpha'.\ (\Delta)(A') \leq \mu\beta'.\ (\Delta)(B')$. Apply rule PosRes-rec, the first premise $|\Delta, \alpha \leq \beta| \vdash (\Delta)(A') \leq (\Delta)(B')$ can be solved by induction hypothesis. Since we know $\Delta, \alpha' \leq \beta' \vdash_{amb} A' \leq B'$, we apply Lemma 44 to it. By obtaining $\alpha' \in_+ (\Delta)(A') \leq (\Delta)(B')$, we can solve the second premise.

□

We are now one step away from the soundness theorem: to prove that the weakly 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 requires the following lemma which reveals an important property to prove that the weakly positive subtyping implies double unfolding subtyping:

Lemma 46. 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.

[5] In Coq, using the locally nameless representation, bound variables are represented as De Bruijn indices, and only free variables use names. Thus renaming is unnecessary.

The proof of Lemma 46, as other substitution lemmas that we have showed before, requires a generalization. Such a generalization is a bit tricky, since we allow equal types in the positive restriction. For readers interested in the details of the generalization, we refer to our mechanized proof. This lemma is important because it shows that, with Lemma 46 proved, we can derive the following lemma, which relates weakly positive subtyping to our algorithmic subtyping relation in the double unfolding form:

Lemma 47. If $\Gamma \vdash_a A \leq B$ and $\Gamma \vdash_a C \leq D$, then

(1) $\alpha \in_+ A \leq B$ implies $\Gamma \vdash_a [\alpha \mapsto C]\ A \leq [\alpha \mapsto D]\ B$ and
(2) $\alpha \in_- B \leq A$ implies $\Gamma \vdash_a [\alpha \mapsto D]\ A \leq [\alpha \mapsto C]\ B$.

Proof. Do induction on $\Gamma \vdash_a A \leq B$. We only show how to prove the function case.

- Rule SA-arrow: Assume $A = A_1 \to A_2$ and $B = B_1 \to B_2$.
  - ⋆ Goal (1):
    - ∗ The goal becomes $\Gamma \vdash_a [\alpha \mapsto C]\ (A_1 \to A_2) \leq [\alpha \mapsto D]\ (B_1 \to B_2)$, which can be rewritten as $\Gamma \vdash_a ([\alpha \mapsto C]\ A_1) \to ([\alpha \mapsto C]\ A_2) \leq ([\alpha \mapsto D]\ B_1) \to ([\alpha \mapsto D]\ B_2)$.
    - ∗ The condition from Goal (1) becomes $\alpha \in_+ A_1 \to A_2 \leq B_1 \to B_2$. By inversion, we obtain $\alpha \in_- B_1 \leq A_1$ and $\alpha \in_+ A_2 \leq B_2$.
    - ∗ Apply rule SA-arrow on the goal, we need to prove: $\Gamma \vdash_a [\alpha \mapsto D]\ B_1 \leq [\alpha \mapsto C]\ A_1$ and $\Gamma \vdash_a [\alpha \mapsto C]\ A_2 \leq [\alpha \mapsto D]\ B_2$.
    - ∗ For the latter one, we apply the induction hypothesis (1).
    - ∗ For the former one, we apply the induction hypothesis (2). However, we need to prove $\alpha \in_- A_1 \leq B_1$. Recall that the positive restriction is commutative, so from $\alpha \in_- B_1 \leq A_1$ we can prove $\alpha \in_- A_1 \leq B_1$.
  - ⋆ Goal (2):
    - ∗ The goal becomes $\Gamma \vdash_a [\alpha \mapsto D]\ (A_1 \to A_2) \leq [\alpha \mapsto C]\ (B_1 \to B_2)$, which can be rewritten as $\Gamma \vdash_a ([\alpha \mapsto D]\ A_1) \to ([\alpha \mapsto D]\ A_2) \leq ([\alpha \mapsto C]\ B_1) \to ([\alpha \mapsto C]\ B_2)$.
    - ∗ The condition from Goal (1) becomes $\alpha \in_- B_1 \to B_2 \leq A_1 \to A_2$. By inversion, we obtain $\alpha \in_+ A_1 \leq B_1$ and $\alpha \in_- B_2 \leq A_2$.
    - ∗ Apply rule SA-arrow on the goal, we need to prove: $\Gamma \vdash_a [\alpha \mapsto C]\ B_1 \leq [\alpha \mapsto D]\ A_1$ and $\Gamma \vdash_a [\alpha \mapsto D]\ A_2 \leq [\alpha \mapsto C]\ B_2$.
    - ∗ For the latter one, we apply the induction hypothesis (2).
    - ∗ For the former one, we apply the induction hypothesis (1). However, we need to prove $\alpha \in_+ B_1 \leq A_1$. Because the positive restriction is commutative, from $\alpha \in_+ A_1 \leq B_1$ we can prove $\alpha \in_+ B_1 \leq A_1$.

□

Corollary 48. 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 Corollary 48, the relation between positive restriction and the algorithmic double unfolding subtyping is easy to establish:

Theorem 49. If $\Gamma \vdash A \leq_+ B$ then $\Gamma \vdash_a A \leq B$.

Combining Lemmas 21, 45 and 49, we have

Corollary 50. Soundness of the Amber rules with respect to the declarative formulation.

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

5.5 The Completeness Theorem

The completeness theorem, to some degree, is more difficult than soundness theorem. Because the Amber rules are more complex in terms of shape than the double unfolding rule, more cases need to be discussed when we do induction on a simpler formulation.

Firstly, let us consider how to convert the double unfolding rule to weakly positive subtyping. The double unfolding rule and weakly positive subtyping share the same context, which means the only source of difference comes from the treatment of recursive types. For weakly positive subtyping, the following inversion lemma is useful:

Lemma 51. If $\Gamma \vdash A \leq_+ B$ and $\Gamma \vdash C \leq_+ D$, then

(1) $\Gamma \vdash [\alpha \mapsto C]\ A \leq_+ [\alpha \mapsto D]\ B$ implies $\alpha \in_+ A \leq B$ or $C = D$;
(2) $\Gamma \vdash [\alpha \mapsto D]\ A \leq_+ [\alpha \mapsto C]\ B$ implies $\alpha \in_- A \leq B$ or $C = D$.

This lemma states that if after substitution the subtyping relation is preserved, then either C and D are equal; or the type variable respects the weakly positive restriction.

Now we can prove that weakly positive subtyping is complete with respect to the double unfolding formulation.

Theorem 52.

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

Proof. Induction on $\Gamma \vdash_a A \leq B$. All cases are straightforward except when A is $\mu\alpha.\ A'$ and B is $\mu\alpha.\ B'$. By induction hypothesis, we know that $\Gamma \vdash A' \leq_+ B'$. By applying lemma 51 with $A := A'$, $B := B'$, $C := A'$, $D := B'$, and mode +, we get that either $\alpha \in_+ A' \leq B'$ or $A' = B'$. For the former case, we apply rule PosRes-rec. For the latter case, we apply reflexivity. □

The translation from weakly positive subtyping to the Amber rules is quite tricky due to the different shapes of the contexts. To illustrate the difficulty consider the following subtyping statement using weakly positive subtyping:

$$\Gamma, \alpha \vdash \top \to \alpha \leq_+ \mathsf{nat} \to \alpha$$

where the environment binds the type variable α. For proving the subtyping relationship, we need to prove:

$$\Gamma, \alpha \vdash \alpha \leq_+ \alpha$$

However, if we want to prove the same statement using the Amber rules, we need to change the relationship to:

$$\Delta, \alpha \leq \beta \vdash_{amb} \top \to \alpha \leq \mathsf{nat} \to \beta$$

and

$$\Delta, \alpha \leq \beta \vdash_{amb} \alpha \leq \beta$$

Note that, in weakly positive subtyping, we only need to store the free variables in the environment, while in the Amber rules, we have more variables and store the subtyping relationship between those variables as well.

The recipe of the conversion is to first generate a bundle of variables and match them to existing variables. Then we determine the mode for each variable in weakly positive subtyping, which helps us to allocate every pair of generated variables. After converting the context and types to the form of Amber rules, we prove that they preserve the subtyping relationship under the Amber rules.

As a second example, assume that we want to convert the following judgment into an Amber judgment

$$\alpha, \beta \vdash \beta \to \alpha \leq_+ \beta \to \alpha$$

$\boxed{\Gamma \vdash A \leq_{+} B \rhd \Pi}$ *(Position Allocation)*

$$\frac{}{\cdot \vdash A \leq_{+} B \rhd \cdot}\ \textsc{mono-nil} \qquad \frac{\alpha \in_{m} A \leq B \qquad \Gamma \vdash A \leq_{+} B \rhd \Pi}{\Gamma, \alpha \vdash A \leq_{+} B \rhd \Pi, m}\ \textsc{mono-cons}$$

Fig. 12. Position Allocation for Weakly Positive Subtyping

we first generate new variables α', β' and assume the subtyping relations $\alpha \leq \alpha'$, $\beta \leq \beta'$. Then we examine the positivity of both variables and find out that these relations hold

$$\alpha \in_{+} \beta \to \alpha \leq \beta \to \alpha \text{ and } \beta \in_{-} \beta \to \alpha \leq \beta \to \alpha$$

In the next step, we substitute the variables in the typing judgment, according to the mode and location. If the variable is in the left-hand side and occurs positively, or right-hand side and occurs negatively, we keep the variable as it is. Otherwise, we substitute the variable with its corresponding one ($\alpha \mapsto \alpha'$ and $\beta \mapsto \beta'$). After these steps, the final result becomes a normal Amber judgment, which has the same meaning of the initial judgment:

$$\alpha \leq \alpha', \beta \leq \beta' \vdash_{amb} \beta' \to \alpha \leq \beta \to \alpha'$$

We prove that this subtyping relation holds under the Amber rules.

Position Allocation. As Figure 12 shows, we define a relation that relates each variable to a mode. The mode in Π has a one-to-one correspondence to the variables in Γ in the same order. The definition of Π is

$$\Pi := \cdot \mid \Pi, + \mid \Pi, -$$

Note that it is not necessarily the case that Π is unique. For example, a variable that never occurs can be accepted by both modes, therefore its corresponding element in Π can be any mode.

Definition 53. Generation of a bundle of fresh variables.

$$\langle \Gamma \rangle := \{(\alpha \leq \beta) \mid \forall \alpha \in \Gamma,\ \beta \text{ is fresh}\}$$

After we have a list of pairs of variables (denoted as $\langle \Gamma \rangle$) and the mode for each variable, we design a function that converts the types according to our information. Note that $\langle \Gamma \rangle$ has same form as the contexts in the Amber setting.

Definition 54. We design a function $convert(\Delta, \Pi, A, m)$ for converting types from weakly positive subtyping setting to the Amber setting, which takes four inputs: a context for Amber formulation, a stack of modes, a type A and a mode. This function returns the converted type as output. Note that the Π is computed as Figure 12 shown, thus its length is equal to the length of Δ.

$$convert(\Delta, \Pi, A, m) = \begin{cases} A & \text{If } \Delta \text{ and } \Pi \text{ are empty.} \\ convert(\Delta', \Pi', [\alpha \mapsto \beta]\ A, m) & \text{If } \Delta = \Delta', \alpha \leq \beta \text{ and } \Pi = \Pi', m \\ convert(\Delta', \Pi', A, m) & \text{If } \Delta = \Delta', \alpha \leq \beta \text{ and } \Pi = \Pi', \text{flip } m \end{cases}$$

We can now state the completeness theorem with Definition 54, where the subtyping relation of weakly positive subtyping preserves under the Amber rules.

Theorem 55. Completeness of the Amber rules: If $\Gamma \vdash A \leq_{+} B$ and $\Gamma \vdash A \leq_{+} B \rhd \Pi$, denoted $\langle \Gamma \rangle$ as Δ, then

$$\Delta \vdash_{amb} convert(\Delta, \Pi, A, -) \leq convert(\Delta, \Pi, B, +).$$

For simplicity, we skip the procedure of the proof for this theorem. Theorem 55 has the longest mechanized proof in the presented paper, which relies on plenty of auxiliary lemmas distinguishing whether two recursive types are equal carefully.

The theorem involves some manipulation of the context and types, due to the inconsistency of contexts between our system and the Amber rules. However, it is very easy to obtain a simple form of corollary where the contexts are empty:

Corollary 56.

$$\text{If } \cdot \vdash A \leq_+ B \text{ then } \cdot \vdash_{amb} A \leq B.$$

The statement is less general than Theorem 55, but it does reveal that the programmer cannot distinguish between our algorithm and the Amber one, since in the subsumption rule, the subtyping judgment always starts with an empty subtyping context. That is, type variables in the double unfolding formulations, and subtyping relations between type variables in the Amber formulations are only introduced by the subtyping relation, and not by the typing relation. The only information that should be in the context during the subsumption rule is the type information for variables.

Combining Lemmas 17, 52 and 56, we have

Corollary 57. Completeness of the Amber rules with respect to the declarative formulation.

$$\text{If } \cdot \vdash A \leq B \text{ then } \cdot \vdash_{amb} A \leq B.$$

Finally, with the equivalence theorems, transitivity and unfolding lemma for our formulations (Lemmas 57, 50, 6 and 24), we can claim the Amber rules are transitive and satisfy the unfolding lemma.

Corollary 58. Transitivity of the Amber rules.

$$\text{If } \cdot \vdash_{amb} A \leq B \text{ and } \cdot \vdash_{amb} B \leq C \text{ then } \cdot \vdash_{amb} A \leq C.$$

Corollary 59. Unfolding lemma for the Amber rules.

$$\text{If } \cdot \vdash_{amb} \mu\alpha.\ A \leq \mu\alpha.\ B \text{ then } \cdot \vdash_{amb} [\alpha \mapsto \mu\alpha.\ A]\ A \leq [\alpha \mapsto \mu\alpha.\ B]\ B.$$

Notably, for transitivity, it is interesting to observe that transitivity holds under an empty environment. In Section 2, we discussed the issues with transitivity and showed a counter-example. That counter-example does not apply to our transitivity lemma because it uses non-empty environments. Therefore a possible "fix" to the declarative formulation in Figure 3 is to restrict the transitivity rule to use only empty environments.

6 A CALCULUS WITH RECORDS

So far we considered calculi where the subtyping relation is antisymmetric. For instance, for the calculus presented in Section 4, Lemmas 19 and 22 hold. Both the Amber rules and the new rules proposed by us work well for antisymmetric subtyping relations. However, as explained in Section 2, applying the Amber rules in subtyping relations that are not antisymmetric is non-trivial due to the built-in reflexivity rule. The purpose of this section is to show that, unlike the Amber rules, the double unfolding rules can be easily applied to subtyping relations that are not antisymmetric. In this section we show the type-soundness for an extension of the calculus in Sections 3 and 4 with records and records types, which leads to a subtyping relation that is not antisymmetric when record types are represented as a sequence of pairs of labels and types.

Types	A, B, C, D	$::=$	$\mathsf{nat} \mid \top \mid A_1 \to A_2 \mid \alpha \mid \mu\alpha.\, A \mid \boldsymbol{\{l_i : A_i\ ^{i \in 1 \cdots n}\}}$
Expressions	e	$::=$	$x \mid \mathsf{i} \mid e_1\ e_2 \mid \lambda x : A.\ e \mid \mathsf{unfold}\ [A]\ e \mid \mathsf{fold}\ [A]\ e \mid \boldsymbol{\{l_i = e_i\ ^{i \in 1 \cdots n}\}} \mid \boldsymbol{e.l}$
Values	v	$::=$	$\mathsf{i} \mid \lambda x : A.\ e \mid \mathsf{fold}\ [A]\ v \mid \boldsymbol{\{l_i = v_i\ ^{i \in 1 \cdots n}\}}$

$\boxed{\Gamma \vdash A}$ *(Well-Formed Type (with Record Types))*

$$\textsc{wft-recur}\quad \frac{\Gamma, \alpha \vdash A \qquad \Gamma, \alpha \vdash [\alpha \mapsto A]\ A}{\Gamma \vdash \mu\alpha.\ A} \qquad\qquad \textsc{wft-rcd}\quad \frac{\Gamma \vdash A_i}{\Gamma \vdash \{l_i : A_i\ ^{i \in 1 \cdots n}\}}$$

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

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

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

$$\textsc{SA-rcd}\quad \frac{\{l_i\ ^{i \in 1 \cdots n}\} \subseteq \{k_j\ ^{j \in 1 \cdots m}\} \qquad k_j = l_i \text{ implies } \Gamma \vdash_\alpha A_j \leq B_i}{\Gamma \vdash_a \{k_j : A_j\ ^{j \in 1 \cdots m}\} \leq \{l_i : B_i\ ^{i \in 1 \cdots n}\}}$$

Fig. 13. Well-formedness and subtyping rules for record types.

6.1 Syntax, Well-Formedness and Subtyping

Syntax. The syntax of the calculus is:

Natural numbers, arrow types, the top type, type variables and recursive types are the same as before (Section 3.1). The additional syntax related to records and record types is highlighted with a bold font. The notation of record types is $\{l_i : A_i\ ^{i \in 1 \cdots n}\}$. Every label has an associated a type and all labels are required to be distinct. A record expression has the form of $\{l_i = e_i\ ^{i \in 1 \cdots n}\}$, and $e.l$ is the record projection expression.

Well-Formedness. In the type system with record types, we use $\Gamma \vdash A$ to represent that A is well-formed. The rules of $\Gamma \vdash A$ include most of the rules at the top of Figure 6. The rule WFT-RCD is new and ensures the well-formedness of record types. Similarly to Section 4, we use rule WFT-RECUR for recursive types.

Subtyping. Our subtyping rules follow the rules in Figure 8, but are extended with an algorithmic formulation of record subtyping. The definition of record subtyping (rule SA-RCD) is standard [Pierce 2002]: a record type A is a subtype of another record type B when: 1) all the labels in A are a subset of the labels in B; and 2) the field types of the corresponding labels are subtypes.

6.2 Metatheory of Subtyping

Subtyping is reflexive, transitive and the unfolding lemma holds.

Reflexivity and transitivity. After adding record types, reflexivity and transitivity are still preserved.

Theorem 60. Reflexivity

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

Theorem 61. Transitivity

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

Unfolding lemma. Unlike the proof for the unfolding lemma in Section 4, we cannot rely on the antisymmetry lemma (Lemma 22) for proving the unfolding lemma. Instead of alpha-equivalence or syntactic equality, we introduce a weaker form of equivalence.

Definition 62 (Equivalence)**.**

$$\Gamma \vdash_a A \sim B \;:=\; \Gamma \vdash_a A \leq B \wedge \Gamma \vdash_a B \leq A$$

With Definition 62, two record types $\{x : Int, y : Bool\}$ and $\{y : Bool, x : Int\}$ are considered to be equivalent: the only difference of these two types is that one type is a permutation of the other type. In other words, the equivalence shows that the order in which the labels appear in a record type does not matter.

One essential lemma is

Lemma 63. If $\Gamma \vdash_a A \leq B$ and $\Gamma \vdash_a C \sim D$, then $\Gamma \vdash_a [\alpha \mapsto C]\ A \leq [\alpha \mapsto D]\ B$.

This lemma states that if two types are subtypes, then after substituting a recursive type variable α with two equivalent types, the subtyping relationship is preserved. The proof of this lemma is straightforward. With Lemma 63, we can prove our core lemma, as we did before.

Lemma 64. If

(1) $\Gamma_1, \alpha, \Gamma_2 \vdash_a A \leq B$;
(2) $\Gamma_1, \alpha, \Gamma_2 \vdash_a C \leq D$;
(3) $\Gamma_1, \Gamma_2 \vdash_a \mu\alpha.\, C \leq \mu\alpha.\, D$;

then

(1) $\Gamma_1, \alpha, \Gamma_2 \vdash_a [\alpha \mapsto C]\ A \leq [\alpha \mapsto D]\ B$ implies $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\, C]\ A \leq [\alpha \mapsto \mu\alpha.\, D]\ B$ and
(2) $\Gamma_1, \alpha, \Gamma_2 \vdash_a [\alpha \mapsto D]\ A \leq [\alpha \mapsto C]\ B$ implies $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\, D]\ A \leq [\alpha \mapsto \mu\alpha.\, C]\ B$.

Proof. By induction on $\Gamma_1, \alpha, \Gamma_2 \vdash_a A \leq B$. Other cases are the same as proof of Lemma 23, except for:

- Case rule SA-var: In such case $A = B = \beta$. If $\alpha \neq \beta$, the goal is trivial.
  - Otherwise, for goal (1), we want to prove $\Gamma_1, \Gamma_2 \vdash_a \mu\alpha.\, C \leq \mu\alpha.\, D$, which is actually premise (3).
  - For goal (2), we have $\Gamma_1, \alpha, \Gamma_2 \vdash_a C \leq D$ from premise (2), and $\Gamma_1, \alpha, \Gamma_2 \vdash_a D \leq C$ from the condition of goal (2), thus $\Gamma_1, \alpha, \Gamma_2 \vdash_a C \sim D$. By Lemma 63, we get $\Gamma_1, \alpha, \Gamma_2 \vdash_a [\alpha \mapsto D]\ D \leq [\alpha \mapsto C]\ C$. As a result, we have $\Gamma_1, \Gamma_2 \vdash_a \mu\alpha.\, D \leq \mu\alpha.\, C$.

□

Finally, we can prove the unfolding lemma:

Lemma 65. Unfolding Lemma

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

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

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

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

$$\frac{\Gamma \vdash e : [\alpha \mapsto \mu\alpha.\ A]\ A \qquad \Gamma \vdash \mu\alpha.\ A}{\Gamma \vdash \mathsf{fold}\ [\mu\alpha.\ A]\ e : \mu\alpha.\ A}\ \textsc{typing-fold} \qquad \frac{\text{for each } i \quad \Gamma \vdash e_i : A_i}{\Gamma \vdash \{l_i = e_i\ ^{i \in 1 \cdots n}\} : \{l_i : A_i\ ^{i \in 1 \cdots n}\}}\ \textsc{typing-rcd}$$

$$\frac{\Gamma \vdash e : \{l_i : A_i\ ^{i \in 1 \cdots n}\}}{\Gamma \vdash e.l_i : A_i}\ \textsc{typing-proj}$$

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

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

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

$$\frac{}{\{l_i = v_i\ ^{i \in 1 \cdots n}\}.l_j \hookrightarrow v_j}\ \textsc{step-projrcd} \qquad \frac{e \hookrightarrow e'}{e.l_j \hookrightarrow e'.l_j}\ \textsc{step-proj}$$

$$\frac{e_j \hookrightarrow e_j'}{\{l_i = v_i\ ^{i \in 1 \cdots j-1},\ l_j = e_j,\ l_k = e_k\ ^{k \in j+1 \cdots n}\} \hookrightarrow \{l_i = v_i\ ^{i \in 1 \cdots j-1},\ l_j = e_j',\ l_k = e_k\ ^{k \in j+1 \cdots n}\}}\ \textsc{step-rcd}$$

Fig. 14. Typing and reduction rules for record types

A final remark is that the same technique that we employ here to prove the unfolding lemma could have been used in the calculus in Section 4 as well. In other words, we do not need to rely on the antisymmetry lemmas in Section 4. We opted to present the two techniques in the paper to also emphasize the difference between antisymmetric and non-antisymmetric relations, since for the Amber rules such difference is quite important.

6.3 Type Soundness

We use the same typing and reduction rules as Section 3.4, extended with extra rules for records and record types.

Table 2. Paper-to-proofs correspondence guide (without record types).

Definition	File (in src/ folder)	Name in Coq	Notation
Well-formed Type (Figure 6)	Rules.v	WFA E A	$\Gamma \vdash A$
Well-formed Type (Definition 2)	Rules.v	WFS E A	$\Gamma \vdash A$
Well-formed Type (Definition 13)	Rules.v	WF E A	$\Gamma \vdash A$
Well-formed Type (Figure 9)	NominalUnfolding.v	Nominal.WFS E A	$\Gamma \vdash A$
Declarative subtyping (Figure 6)	Rules.v	Sub E A B	$\Gamma \vdash A \leq B$
Typing (Figure 7)	Rules.v	typing E e A	$\Gamma \vdash e : A$
Reduction (Figure 7)	Rules.v	step e1 e2	$e_1 \hookrightarrow e_2$
Double unfolding rule (Figure 8)	Rules.v	sub E A B	$\Gamma \vdash_a A \leq B$
Nominal unfolding rule (Figure 9)	NominalUnfolding.v	Nominal.Sub E A B	$\Gamma \vdash_n A \leq B$
Well-formed Type (Figure 10)	AmberBase.v	wf_amber E A	$\Delta \vdash A$
Amber rules (Figure 10)	AmberBase.v	sub_amber E A B	$\Delta \vdash_{amb} A \leq B$
Weakly Positive restriction (Figure 11)	AmberBase.v	posvar m X A B	$\alpha \in_m A \leq B$
Weakly Positive subtyping (Figure 11)	PositiveBase.v	wk_sub E A B	$\Gamma \vdash A \leq_+ B$
Weakly Positive subtyping (Definition 69)	AmberBase.v	sub_amber2 E A B	$\Gamma \vdash_u A \leq_+ B$

Typing. As the top of Figure 14 shows, we have two typing rules for record types. Rule TYPING-RCD states that a record is well-typed if we know that all its fields are well-typed. Rule TYPING-PROJ checks that the record that we are projecting from is well-typed, and contains the field label that we are projecting.

Reduction. As the bottom of Figure 14 shows, we have three reduction rules for record types. Rule STEP-PROJRCD retrieves a component of a record. Rule STEP-PROJ reduces the record expression being projected. Rule STEP-RCD implements a left-to-right evaluation order to reduce a record.

Type Soundness. The proof technique of proving type-soundness is conventional, without any special approach, except for the use of the unfolding lemma in preservation (just as in Section 3). Therefore, we can directly prove preservation and progress.

Theorem 66. Preservation.

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

Theorem 67. Progress.

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

7 COQ PROOFS

We have chosen the Coq (8.13) proof assistant [The Coq Development Team 2019] to develop our formalization. The whole Coq formalization is built with a third-party library Metalib[6], which provides support for the locally nameless representation [Aydemir et al. 2008] to encode binders.

Table 3. Paper-to-proofs correspondence guide (with record types).

Definition	File (in src_extension/ folder)	Name in Coq	Notation
Well-formed Type (Figure 13)	definition.v	WF E A	$\Gamma \vdash A$
Subtyping (Figure 13)	definition.v	Sub E A B	$\Gamma \vdash A \leq B$
Typing (Figure 14)	definition.v	typing E e A	$\Gamma \vdash e : A$
Reduction (Figure 14)	definition.v	step e1 e2	$e_1 \hookrightarrow e_2$

7.1 Definitions

Simply Typed Lambda Calculus (STLC) with iso-recursive types. The folder *src* includes all the Coq proofs about STLC extended with iso-recursive subtyping, which is the calculus described in Sections 3, 4 and 5. All the definitions in the paper can be found in files *Rules.v*, *AmberBase.v* and *NominalUnfolding.v*. Table 2 shows the correspondence of definitions between the paper and the Coq artifacts. The file *Rules.v* contains the definitions for our type system. It has definitions of well-formedness, subtyping (both finite and double unfoldings), typing, and reduction. The file *AmberBase.v*, contains the definitions for the Amber rules and the intermediate subtyping relation based on a weakly positive restriction presented in Section 5. The file *NominalUnfolding.v* contains all the definitions and proofs involving nominal unfoldings, expect for the decidability proof, which is contained in file *Decidability.v*.

For encoding variables and binders, we use the locally nameless representation to express all the types and terms. In the 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, rule WFT-INF, rule WFT-RECUR and rule WFT-NOMINAL. In the Coq formalization, we have four distinct definitions of well-formedness, which are proved to be equivalent.

Simply Typed Lambda Calculus (STLC) with iso-recursive types and record types. The folder *src_extension* includes all the Coq proofs about STLC with iso-recursive subtyping and record types, which corresponds to the calculus in Section 6. The folder structure is similar, except that we move the unfolding lemma to a new file named *unfolding.v*. All the definitions in the paper can be found in files *definition.v*. Table 3 shows the correspondence of definitions between the paper and the Coq artifacts.

7.2 Lemmas and Theorems

Table 4 shows the descriptions for all the proof scripts in Section 3, Section 4 and Section 5. For succinctness, we briefly describe all the lemmas and theorems, annotating them with related subtyping formulation inside the brackets. In Table 4, *Finite* represents our specification, *Double* represents the double unfolding rule, *Nominal* represents the nominal unfolding rule, *Positive* represents the weakly positive subtyping, and *Amber* represents the Amber rules.

Table 5 shows the descriptions for all the proof scripts in Section 6.

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, all variants of the unfolding lemma in the paper are presented without modes in the paper. Figure 15 illustrates the difference between the formulations with and without modes

[6] https://github.com/plclub/metalib. Note that currently (February 2022), *Metalib* library only supports Coq (<=8.10), thus some modifications are needed. More precisely, all *omega* tactics should be replaced by *lia* tactics.

Table 4. Descriptions for the proof scripts.

Theorems	Description	Files (in src/ folder)	Name in Coq
Theorem 5	Reflexivity (Finite)	FiniteUnfolding.v	refl
Theorem 6	Transitivity (Finite)	FiniteUnfolding.v	Transitivity
Lemma 8	Unfolding lemma (Finite)	FiniteUnfolding.v	unfolding_lemma
Theorem 11	Preservation	Typesafety.v	preservation
Theorem 12	Progress	Typesafety.v	progress
Theorem 15	Reflexivity (Double)	FiniteUnfolding.v	refl_algo
Theorem 16	Transitivity (Double)	FiniteUnfolding.v	trans_algo
Theorem 17	Completeness (Double)	FiniteUnfolding.v	completeness
Theorem 21	Soundness (Double)	DoubleUnfolding.v	soundness
Lemma 24	Unfolding lemma (Double)	DoubleUnfolding.v	unfolding_lemma_version2
Theorem 25	Reflexivity (Nominal)	NominalUnfolding.v	Nominal.sub_refl
Theorem 26	Transitivity (Nominal)	NominalUnfolding.v	Nominal.Transitivity
Lemma 27	Unfolding lemma (Nominal)	NominalUnfolding.v	Nominal.unfolding_lemma
Theorem 30	Nominal to Double	NominalUnfolding.v	nominal_to_double
Theorem 31	Double to Nominal	NominalUnfolding.v	double_to_nominal
Corollary 32	Soundness (Nominal)	NominalUnfolding.v	nominal_to_finite
Corollary 33	Completeness (Nominal)	NominalUnfolding.v	finite_to_nominal
Theorem 36	Decidability	Decidability.v	decidability
Theorem 39	Reflexivity (Positive)	AmberBase.v	sub_amber2_refl
Theorem 41	Transitivity (Positive)	PositiveBase.v	sub_amber2_trans
Lemma 42	Unfolding lemma (Positive)	PositiveBase.v	unfolding_for_pos
Theorem 45	Amber to Positive	AmberBase.v	sub_amber_to_amber_2
Theorem 49	Positive to Double	AmberSoundness.v	sub_amber_2_to_sub
Corollary 50	Soundness (Amber)	AmberSoundness.v	amber_soundness2
Theorem 52	Double to Positive	PositiveSubtyping.v	sub_to_amber2
Theorem 55	Positive to Amber	AmberCompleteness.v	amber_complete_aux
Corollary 57	Completeness (Amber)	AmberCompleteness.v	amber_complete2
Corollary 58	Transitivity (Amber)	AmberCompleteness.v	amber_transitivity
Corollary 59	Unfolding lemma (Amber)	AmberCompleteness.v	amber_unfolding

for the unfolding lemma (note that the premise (2) is redundant since it is the inversion of the premise (3), thus in the Coq code we drop this premise while in the paper presentation we keep it for readability). Our Coq formalization uses some meta-functions on modes instead to formalize the same result. Using meta-functions on modes (Definition 68), the same lemma would look like the right part of Figure 15.

Definition 68. Mode selector.

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

Table 5. Descriptions for the proof scripts (complement).

Theorems	Description	Files (in src_extension/ folder)	Name in Coq
Theorem 60	Reflexivity	subtyping.v	sub_refl
Theorem 61	Transitivity	subtyping.v	Transitivity
Lemma 64	Unfolding lemma	unfolding.v	unfolding_lemma
Theorem 66	Preservation	typesafety.v	preservation
Theorem 67	Progress	typesafety.v	progress

Lemma 23 in paper:
If

(1) $\Gamma_1, \alpha, \Gamma_2 \vdash_a A \leq B$;
(2) $\Gamma_1, \alpha, \Gamma_2 \vdash_a C \leq D$;
(3) $\Gamma_1, \Gamma_2 \vdash_a \mu\alpha.\ C \leq \mu\alpha.\ D$;

then

(1) $\Gamma_1, \alpha, \Gamma_2 \vdash_a [\alpha \mapsto C]A \leq [\alpha \mapsto D]B$ implies $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\ C]A \leq [\alpha \mapsto \mu\alpha.\ D]B$ and
(2) $\Gamma_1, \alpha, \Gamma_2 \vdash_a [\alpha \mapsto D]A \leq [\alpha \mapsto C]B$ implies $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\ D]A \leq [\alpha \mapsto \mu\alpha.\ C]B$.

Lemma 23 in Coq:

If

(1) $\Gamma_1, \alpha, \Gamma_2 \vdash_a A \leq B$;
(2) $\Gamma_1, \Gamma_2 \vdash_a \mu\alpha.\ C \leq \mu\alpha.\ D$;
(3) $\Gamma_1, \alpha, \Gamma_2 \vdash_a [\alpha \mapsto C \oplus_m D]A \leq [\alpha \mapsto D \oplus_m C]B$

then
$\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\ C \oplus_m D]A \leq [\alpha \mapsto \mu\alpha.\ D \oplus_m C]B$

Fig. 15. Comparison between paper and Coq statements for Lemma 23.

In the Coq proof, we also 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 *Rules.v*.

Another important difference is in the decidability proof. Unlike the paper proof, where in the context we store the variable names as keys, in the Coq proof we employ De Bruijn indices to represent all recursive variables stored in the context.

7.3 Alternative Weakly Positive Subtyping

During the proof of completeness of Amber rules, we found that the built-in reflexivity in the weakly positive subtyping disturbs the computation of position allocation for recursive types. Thus, in the mechanized proof, we use an alternative (Definition 69) for weakly positive subtyping to compute the mode more precisely: the default positive mode for equal recursive types.

The key idea is to unify rule PosRes-self and rule PosRes-rec into one rule, then we "hide" the problematic reflexivity subtly by rule PosRes-recalt:

Definition 69. An alternative rule for checking if two recursive types are subtypes in weakly positive subtyping:

$$\textsc{PosRes-recalt}\quad \frac{\Gamma, \alpha \vdash A \leq_+ B \qquad \beta \text{ is fresh} \qquad \beta \in_+ \mu\alpha.\ A \leq \mu\alpha.\ B}{\Gamma \vdash \mu\alpha.\ A \leq_+ \mu\alpha.\ B}$$

Denoting $\Gamma \vdash_u A \leq_+ B$ as the weakly positive subtyping with the alternative rule PosRes-recalt for recursive types, we show that it has same expressiveness as the original definition of weakly positive subtyping (Figure 11):

Lemma 70. The two representations of weakly positive subtyping are equivalent:

$$\Gamma \vdash_u A \leq_+ B \Leftrightarrow \Gamma \vdash A \leq_+ B.$$

7.4 Variable Generation

Another difficulty worth mentioning is generating a bundle of variables in Definition 53. Such definition actually does two things: (1) generate a set of fresh variables; (2) match every fresh variable with an existing variable. This is a bit involved in Coq.

File *src/AmberCompleteness.v* gives the details showing how to solve this issue. We iterate each variable (denote as α) in context Γ, generate a fresh variable β and store both variables. One possibility is that the name of α might be used in previous stored set of variables. In that case, we generate one more fresh variable and store it. After that, we have a set of mixed variables containing all variables in context Γ and the number of new fresh variables is the same as the size of context Γ. All the variables in the set are distinct. Then we filter variables that belong to Γ and match them with variables in Γ one by one. Finally, we have a valid $\langle\Gamma\rangle$, as Definition 53 describes.

8 DISCUSSION AND RELATED WORK

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

Iso-recursive Amber rules. In Sections 2 and 5, we discussed Amadio and Cardelli [1993]'s work on 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 with the 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, $\alpha_1 \leq \alpha_2 \leq \alpha_3$, is accepted by their transitivity statement, by means of adapting variable bindings in the contexts accordingly:

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

In other words, the subtyping judgments of their transitivity statement (used for their proof) do not share the same context, which subtly captures the nature of context elements ($\alpha \leq \beta$) 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 [Abadi and Cardelli 1996; 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.

Complete 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. Besides the complications due to the presence of a reflexivity rule, they find that a source of incompleteness of the Amber rules comes from complications with recursive type unrolling. The two rules for subtyping recursive types employed by Ligatti et al. are:

$$\frac{S, \mu\alpha.\ A \le \mu\beta.\ B \vdash [\alpha \mapsto \mu\alpha.\ A]\ A \le [\beta \mapsto \mu\beta.\ B]\ B}{S \vdash \mu\alpha.\ A \le \mu\beta.\ B}\,\text{L17-rec1}$$

$$\frac{(\mu\alpha.\ A \le \mu\beta.\ B) \in S}{S \vdash \mu\alpha.\ A \le \mu\beta.\ B}\,\text{L17-rec2}$$

The basic idea is that subtyping environments S track all subtyping relations between recursive types that have already been observed. Rule L17-rec1 is the rule that is triggered if $\mu\alpha.\ A \le \mu\beta.\ B$ has not been observed yet. In that case $\mu\alpha.\ A \le \mu\beta.\ B$ is simply added to the environment and the recursive type variables are directly replaced by the recursive types in the bodies. In rule L17-rec2, if $\mu\alpha.\ A \le \mu\beta.\ B$ is already in the environment, then we know that the two recursive types are in a subtyping relation and we can terminate. One similarity to the double unfolding and the nominal unfolding rules is that both Ligatti et al.'s rules and our rules employ one substitution for each type. However, in Ligatti et al.'s rules we substitute the recursive type variable with the recursive type directly, whereas in our rules we use a finite unfolding: that is we use the body of the recursive type instead. Ligatti et al.'s rules are more powerful than both the Amber rules and all the rules presented in this paper, including our declarative formulation with finite unfoldings, as well as the double and nominal unfolding rules. The simplest example that illustrates the different expressive power between our rules and the rules by Ligatti et al. [2017] is perhaps $\mu\alpha.\ \alpha \le \mu\alpha.\ (\mu\beta.\ \alpha)$. With Ligatti et al.'s rules this is a valid subtyping statement, as illustrated by the following derivation:

$$\dfrac{\dfrac{\dfrac{\mu\alpha.\ \alpha \le \mu\alpha.\ (\mu\beta.\ \alpha) \in \{\mu\alpha.\ \alpha \le \mu\alpha.\ (\mu\beta.\ \alpha),\ \mu\alpha.\ \alpha \le \mu\beta.\ (\mu\alpha.\ (\mu\beta.\ \alpha))\}}{\mu\alpha.\ \alpha \le \mu\alpha.\ (\mu\beta.\ \alpha),\ \mu\alpha.\ \alpha \le \mu\beta.\ (\mu\alpha.\ (\mu\beta.\ \alpha)) \vdash \mu\alpha.\ \alpha \le \mu\alpha.\ (\mu\beta.\ \alpha)}\,\text{L17-rec2}}{\mu\alpha.\ \alpha \le \mu\alpha.\ (\mu\beta.\ \alpha) \vdash \mu\alpha.\ \alpha \le \mu\beta.\ (\mu\alpha.\ (\mu\beta.\ \alpha))}\,\text{L17-rec1}}{\vdash \mu\alpha.\ \alpha \le \mu\alpha.\ (\mu\beta.\ \alpha)}\,\text{L17-rec1}$$

In contrast, the rules based on finite unfoldings reject such subtyping statement. For instance, here is the failed derivation with the double unfolding rules:

$$\dfrac{\dfrac{\text{Derivation fails here}}{\alpha \vdash \alpha \le \mu\beta.\ \alpha} \qquad \dfrac{\text{Derivation fails here}}{\alpha \vdash \alpha \le \mu\beta.\ (\mu\beta.\ \alpha)}}{\vdash \mu\alpha.\ \alpha \le \mu\alpha.\ (\mu\beta.\ \alpha)}\,\text{SA-rec}$$

The source of the difference in terms of expressive power between our rules (as well as the Amber rules) and Ligatti et al.'s rules is related to the treatment of subtyping between type variables and recursive types. In the failed derivation with the double unfolding rule we can see that the derivation fails when we encounter a subtyping statement of the form $\alpha \le \mu\beta.\ A$. That is when we try to compare a recursive type variable with a recursive type. In both our rules and the Amber rules, such statements are always rejected, since the recursive type variables are opaque and the structure of the recursive type denoted by the type variable is not known. In some sense with the Amber rules and our rules recursive type variables act similarly to nominal types, and comparing them with a recursive type that happens to have a structurally compatible shape will fail. In Ligatti et al.'s rules, because type variables are always replaced by the recursive type, the structure of the recursive type is known (or transparent) and then the subtyping rules for recursive types can be used instead. In addition to the simple example that we describe above, Ligatti et al. [2017] have identified two larger examples that demonstrate that their rules can derive subtyping statements that the Amber rules cannot:

- $\mu n.\{\mathsf{sub} : (\mu i'.\{\mathsf{sub} : i' \rightarrow \mathsf{unit}\}) \rightarrow \mathsf{unit}, \mathsf{min} : \mathsf{unit} \rightarrow \mathsf{int}\} \le \mu i.\{\mathsf{sub} : i \rightarrow \mathsf{unit}\}$;
- $\mu a.(((\mu b.((b + \mathsf{nat}) + a)) + \mathsf{nat}) + a) \le \mu c.((c + \mathsf{real}) + c)$.

As they observe, accepting such subtyping statements does not violate type-safety. The two examples above are also rejected by our rules, for similar reasons to the simpler example above. The failure to derive such subtyping statements is expected since we proved that our rules are equivalent in terms of expressive power to the Amber rules, which reject them as well.

In addition to rules L17-rec1 and L17-rec2, some non-standard subtyping rules for *value-uninhabited* types are also needed for achieving the completeness of subtyping with respect to type safety. If a type is value-uninhabited then every expression of that type diverges. In other words, value-uninhabited types are treated as bottom types ($\perp$). As Ligatti et al. explained, if we do not care about subtyping completeness with respect to type-safety, we can ignore the extra subtyping rules for value-uninhabited types, and still get additional expressive power over the Amber rules. From the point of view of type-safety, the new formulations of subtyping proposed by us are also incomplete, since they have the same expressive power as 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 extensible. To achieve their goal of a complete formulation of subtyping with respect to type safety, Ligatti et al. [2017] had to develop several new proof techniques to accomplish this goal. For instance one of the techniques developed in their work is induction on *failing derivations*, which requires defining an explicit relation that captures failed derivations of subtyping. A further complicating factor is the non-standard form of environments S required by rules L17-rec1 and L17-rec2, which must contain entries of the form $\mu\alpha.\ A \le \mu\beta.\ B$. This is in contrast to our rules, which all employ standard environments with type variables only. Both of these mean that the subtyping metatheory is significantly different from conventional formulations of subtyping. In Ligatti et al.'s work, most important theorems, such as transitivity or reflexivity, are proved by doing induction on failing derivations. For example, their transitivity theorem is proved via an auxiliary theorem called *strong subtyping transitivity* of the form:

$$\frac{S \vdash \tau_1 \le \tau_3 \text{ is not derivable} \qquad \vdash \tau_1 \le \tau_2}{\vdash \tau_2 \le \tau_3 \text{ is not derivable}}$$

This theorem relies on the failed derivations relation and leads to a transitivity proof that is quite different from conventional transitivity proofs for subtyping. In contrast, our transitivity theorem (as well as other lemmas such as reflexivity) and proofs are standard. For instance, as we show in Theorem 6, our transitivity proof is modular in the sense that proofs for the cases of non-recursive type constructs (such as function types) are essentially the same as for a subtyping relation without recursive types. In other words the addition of recursive types using our rules has little impact on existing proofs[7]. This is not the case in Ligatti et al.'s work since their rules for recursive subtyping as well as their proof techniques for showing completeness of subtyping with respect to type-safety require new proof techniques and proofs, and even new theorem statements. In addition, all our proofs have been formalized in a theorem prover, whereas Ligatti et al.'s proofs have not been mechanically formalized yet.

Other approaches to iso-recursive subtyping. 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 types 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 for 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 11) 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. 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

[7]Our locally nameless [Charguéraud 2011] based Coq proofs follow a similar style to Chargueraud's proofs for System $F_{<:}$ in https://www.chargueraud.org/softs/ln.

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.

9 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. More practically, the double unfolding rule and nominal unfolding rule are easy to integrate in existing calculi and this work presents the proof techniques needed to prove standard properties (such as transitivity and type soundness). Moreover, we show that it is easy to employ our algorithmic subtyping rules to subtyping relations that are not antisymmetric.

Investigating the use of our novel formulation of iso-recursive subtyping in more complex subtyping relations is an interesting direction for future work. For instance, it will be interesting to explore calculi with polymorphism, intersection/union types as well as calculi with bounded quantification. Investigating optimal algorithms for Amber-style iso-recursive subtyping is also an interesting direction for future work. Finally, 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 the anonymous OOPSLA and TOPLAS reviewers for their valuable comments that helped to improve the presentation of our work. John Tang Boyland provided us with valuable feedback and he produced an alternative formalization for the calculus presented in Sections 3 and 4 in the SASyLF prover [Aldrich et al. 2008]. His SASyLF formalization is included in our supplementary materials. We also thank Litao Zhou for his helpful discussion about the decidability proof, that helped us finally propose a good measure for the expanded tree. This work has been sponsored by Hong Kong Research Grant Council projects number 17209519 and 17209520.

REFERENCES

Martin Abadi and Luca Cardelli. 1996. *A theory of objects*. Springer Science & Business Media.

Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. 1991. Explicit Substitutions. *J. Funct. Program.* 1, 4 (1991), 375–416.

Jonathan Aldrich, Robert J. Simmons, and Key Shin. 2008. SASyLF: an educational proof assistant for language theory. In *Functional and Declarative Programming in Education (FDPE'08)*. ACM, 31–40.

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.

Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. 1995. Intersection and Union Types: Syntax and Semantics. *Information and Computation* 119, 2 (June 1995), 202–230.

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 pages.

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)*.

Arthur Charguéraud. 2011. The Locally Nameless Representation. *Journal of Automated Reasoning* (2011), 1–46. 10.1007/s10817-011-9225-2.

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*. 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.

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 1981. Functional Characters of Solvable Terms. *Mathematical Logic Quarterly* 27, 2-6 (1981), 45–58.

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$\leq$. 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.

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

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).

Yaoda Zhou, Bruno C. d. S. Oliveira, and Jinxu Zhao. 2020. Revisiting Iso-Recursive Subtyping. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 223 (Nov. 2020), 28 pages.