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 be sound and complete with respect to this declarative specification. We then show another *sound*, *complete* and *decidable* algorithmic formulation of subtyping that employs a novel *double unfolding* rule. Compared to the Amber rules, the double unfolding rule has the advantage of: (1) being modular; (2) not requiring reflexivity to be built in; (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 double unfolding rule has important advantages over the original Amber rules for both implementations and metatheoretical studies involving recursive types. All results are mechanically formalized in the Coq theorem prover.

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), 38 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

```
data List = Nil | Cons Int List

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

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

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

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

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

The Amber rules are well-known and widely used for subtyping iso-recursive types. They were briefly and informally introduced in 1985 by Cardelli in a manuscript describing the Amber language [Cardelli 1985]. Later on, Amadio and Cardelli [1993] made a comprehensive study of the theory of recursive subtyping for a system with equi-recursive types employing Amber-style rules. One nice result of their study is a declarative model for specifying when two recursive types are in a subtyping relation. In essence, two (equi-)recursive types are subtypes if their infinite unfoldings are subtypes. Amadio and Cardelli's study remains to the day a standard reference for the theory of equi-recursive subtyping, although newer work simplifies and improves on the original theory [Brandt and Henglein 1997; Gapeyev et al. 2003]. Since then variants of the Amber rules have been employed multiple times in a variety of calculi and languages, but often in an iso-recursive setting [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. Indeed, the only proof for transitivity that we are aware of is by Bengtson et al. [2011]. However, the proof relies on a complex inductive argument, and attempts to formalize the proof in a theorem prover have been unsuccessful so far [Backes et al. 2014]. Finally, a fundamental lemma that arises in proofs of type preservation for calculi with iso-recursive subtyping is:

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

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:

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

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

We also show an alternative algorithmic formulation based on the so-called *double unfolding* rule, which only checks 1-time and 2-times finite unfoldings. This rule accepts all valid subtyping statements that the Amber rules accept, but it has important advantages. In particular the double unfolding rule has 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).

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

In summary, the contributions of this paper are:

- **A declarative specification for iso-recursive subtyping:** We propose a new declarative specification for iso-recursive subtyping, where two recursive types are subtypes if all the finite unfoldings are subtypes in Section 3.

Table 1. Some key theorems in the paper.

	Reflexivity	Transitivity	Unfolding Lemma
Amber Rules	Built-in	N/A	N/A
Finite Unfolding	Theorem 4	Theorem 5	Lemma 7
Double Unfolding	Theorem 13	Theorem 14	Lemma 22
Weakly Positive Subtyping	Theorem 25	Theorem 27	Lemma 28

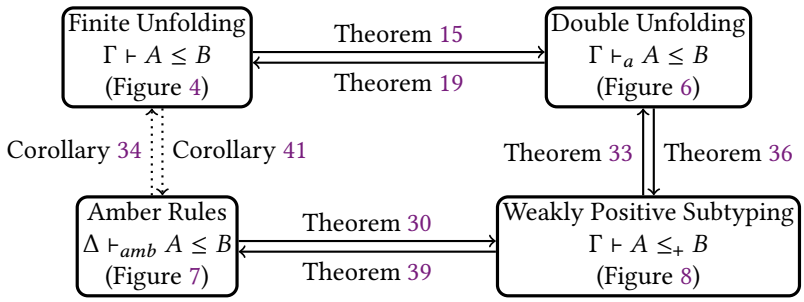

Fig. 2. A diagram with the soundness and completeness lemmas in this work.

- **Algorithmic subtyping with the double unfolding rule:** We show a sound, complete and decidable algorithmic formulation of subtyping employing a new double unfolding rule in Section 4.
- **Soundness and completeness of the Amber rules:** We prove that the Amber rules are sound and complete with respect 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 double unfolding rules, we also give another equivalent formulation of subtyping based on a weakly positive restriction of recursive variables. This variant is used as an intermediate step to prove the soundness and completeness between the Amber rules and a formulation using double unfolding. This variant is presented as part of Section 5.
- **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 2 summarize some key lemmas and theorems of this paper.

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 double unfolding formalization. Consequently, all four subtyping formulations are shown to be equivalent (Figure 2). The conference version only shows the soundness of the Amber rules with respect to finite unfoldings, but not their completeness. Finally, the material in Section 6, showing that our novel double unfolding can be applied to calculi

with record types, is new. This is interesting because it shows that the double unfolding rule 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 approach. We first introduce some alternative formulations for iso-recursive subtyping and discuss some issues with the Amber rules. Then we present the key ideas of our work, including a novel declarative formulation of subtyping and the double unfolding rule. Finally, we show how the double unfolding rule can be employed in calculi with record types.

2.1 Subtyping Recursive Types

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

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

An important aspect to pay attention 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 2.4.

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 any type A. But that does not imply that $\mu\alpha.\ \alpha \to \mathsf{nat} \le \mu\alpha.\ \alpha \to \top$, 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.

2.2 A Rule That Only Works for Covariant Subtyping

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

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

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 occurr in negative positions, may allow *unsound* subtyping statements, as shown above.

2.3 The Positive Restriction Rule

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

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

where $\text{non-neg}(\alpha, A)$ 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 $\mu\alpha.\ \alpha \to \alpha \leq \mu\alpha.\ \alpha \to \alpha$.

2.4 The Amber Rules

Equi-recursive Amber rules. The Amber rules were introduced in the Amber language by Cardelli [1985]. Later, Amadio and Cardelli [1993] studied the metatheory for a subtyping relation that employs Amber-like rules. These rules are presented in Figure 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 $\mu\alpha.\ \alpha \to nat \leq \mu\beta.\ \beta \to \top$. Unfortunately, rule OAmber-rec rules out too many cases with negative subtyping, including statements about equal types, such as $\mu\alpha.\ \alpha \to nat \leq \mu\beta.\ \beta \to nat$. 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[1] in reflexivity, which is less powerful, but is enough to express iso-recursive subtyping. In what follows we consider the folklore rules, where the equality ($A = B$) used in rule OAmber-refl is simplified by just considering syntactic equality. The iso-recursive rules can deal correctly with all the examples illustrated so far, accepting the various examples that we have argued should be accepted, and rejecting the other ones. Perhaps a small nitpicking point is the absence of well-formedness constraints in the subtyping rules. By modern day standards, this may look a little suspicious, but then again well-formedness of environments and types is typically standard and straightforward. Unfortunately, as it turns out, a suitable definition of well-formedness is non-trivial for Amber subtyping. We will come back to this issue in Section 5. Setting the issue of well-formedness aside for the moment, the Amber rules have some other important issues:

[1]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)*

$$\textsc{OAmber-refl}\ \frac{A = B}{\Gamma \vdash A \leq B} \qquad \textsc{OAmber-trans}\ \frac{\Gamma \vdash A \leq B \qquad \Gamma \vdash B \leq C}{\Gamma \vdash A \leq C}$$

$$\textsc{OAmber-assmp}\ \frac{\alpha \leq \beta \in \Gamma}{\Gamma \vdash \alpha \leq \beta} \qquad \textsc{OAmber-top}\ \frac{}{\Gamma \vdash A \leq \top}$$

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

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

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

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

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

The reflexivity rule, if present in the subtyping relation, depends on a specific equivalence judgment. Simple systems 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:

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

However, if the built-in reflexivity employs only alpha-equivalence, such a subtyping statement is rejected. In this case the subtyping relation *is not* antisymmetric. That is both $\{x : \alpha, y : nat\} \leq \{y : nat, x : \alpha\}$ and $\{y : nat, x : \alpha\} \leq \{x : \alpha, y : nat\}$ are true, but the two types are not equal. Thus, a (strict) reflexivity rule employing syntactic equality is not adequate in such cases. The reader may refer to work from Ligatti et al. [2017] for a more extended discussion on the complications of having the reflexivity rule built-in. We will also come back to this point in Section 2.6.

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

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

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

Proofs of transitivity and other lemmas are hard. A related problem is 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 2.3. We also tried to directly prove some of these properties in Coq with variations of the Amber rules, but none of them works properly.

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

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

While the Amber rules are simple, as we have argued, there are important issues with the rules. In particular developing the metatheory for the Amber rules is quite hard. Therefore, to provide a detailed account of the metatheory for iso-recursive subtyping we propose alternative definitions (both declarative and algorithmic) for subtyping of recursive types. The new formulation of subtyping has important advantages over the Amber rules: the new rules are more modular; they do not require reflexivity to be built-in; and transitivity and various other lemmas are easier to prove. Furthermore, we prove that the Amber rules are sound 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 2.2. The logic of the covariant rules is to approximate recursive subtyping using what we call a 1-time *finite unfolding*. We say that the unfolding is finite because we simply use α instead of using the recursive type itself during unfolding. If we apply finite unfoldings to all recursive types, we eventually end up having a comparison of two types representing finite trees. The covariant rules work fine in a setting with covariant subtyping only, but are unsound in a setting that also includes contravariant subtyping. A plausible question is then: can we fix these rules to become sound in the presence of contravariant subtyping?

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

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

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 2.2, 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:

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

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

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

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

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

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

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

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:

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

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

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

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

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

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 ($nat \to \alpha \not\leq nat \to nat \to \top$), which fails and is the reason why the double unfolding rule rejects this example.

2.6 A Calculus with Recursive Record Types

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

In Section 2.4, 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:

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

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

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

should hold, since $\{x : \alpha, y : nat\}$ and $\{y : nat, x : \alpha\}$ 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 use the equivalence relation above ($\Gamma \vdash A \sim B$). 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.

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

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

Well-formedness. The definition of a well-formed environment $\vdash \Gamma$ is standard, ensuring that all variables in the environment are distinct. The top of Figure 4 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 4 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 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.

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

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

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

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

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

Fig. 4. Well-formedness and subtyping rules.

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

Definition 2. Rule WFT-INF is defined as:

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

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

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

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

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 4. Reflexivity.

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

Theorem 5. Transitivity.

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

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

Lemma 6. If

(1) $\Gamma_1, \alpha, \Gamma_2 \vdash A \leq B$;
(2) $\Gamma_1, \Gamma_2 \vdash \mu\alpha.\ C$ and $\Gamma_1, \Gamma_2 \vdash \mu\alpha.\ D$;
(3) $\Gamma_1, \alpha, \Gamma_2 \vdash [\alpha \mapsto C]^n\ A \leq [\alpha \mapsto D]^n\ B$ holds for all n,

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

Proof. Induction on $\Gamma_1, \alpha, \Gamma_2 \vdash A \leq B$. 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 third premise is still $\Gamma_1, \alpha, \Gamma_2 \vdash [\alpha \mapsto C]^n\ \alpha \leq [\alpha \mapsto D]^n\ \alpha$, where n is arbitrary. The goal becomes $\Gamma_1, \Gamma_2 \vdash \mu\alpha_1.\ C \leq \mu\alpha_1.\ D$. Then apply the rule for recursive types. The goal is equal to the third premise after alpha-conversion.
- Rule S-REC. Assume that the A's shape is $\mu\alpha_1.\ A'$ and B's shape is $\mu\alpha_1.\ B'$. Then the third premise becomes $\Gamma_1, \alpha, \Gamma_2 \vdash [\alpha \mapsto C]^n\ \mu\alpha_1.\ A' \leq [\alpha \mapsto D]^n\ \mu\alpha_1.\ B'$, which can be rewritten to $\Gamma_1, \alpha, \Gamma_2 \vdash \mu\alpha_1.\ [\alpha \mapsto C]^n\ A' \leq \mu\alpha_1.\ [\alpha \mapsto D]^n\ B'$. The goal becomes $\Gamma_1, \Gamma_2 \vdash [\alpha \mapsto \mu\alpha_2.\ C]\ \mu\alpha_1.\ A' \leq [\alpha \mapsto \mu\alpha_2.\ D]\ \mu\alpha_1.\ B'$, which can be rewritten to $\Gamma_1, \Gamma_2 \vdash \mu\alpha_1.\ [\alpha \mapsto \mu\alpha_2.\ C]\ A' \leq \mu\alpha_1.\ [\alpha \mapsto \mu\alpha_2.\ D]\ B'$. By induction hypothesis, this goal is proven.

□

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

Lemma 7. Unfolding Lemma.

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

3.4 Typing and Reduction Rules

Typing rules. As the top of Figure 5 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 5 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. Firstly, we need a conventional substitution lemma to deal with beta reduction in preservation:

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

Then we can proceed to the preservation and progress theorems.

Theorem 9. Preservation.

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

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

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

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

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

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

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

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

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

Fig. 5. Typing and reduction rules.

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

□

Theorem 10. Progress.

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

4 ALGORITHMIC SUBTYPING WITH THE DOUBLE UNFOLDING RULE

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 present a sound and complete algorithmic formulation of subtyping. This formulation replaces the declarative rule S-REC by the double unfolding rule, which unfolds the recursive types 1-time and 2-times, respectively.

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

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

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

Fig. 6. Subtyping Rules for Algorithmic Type System

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 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 4. Similarly to Section 3, we define an alternative variant of well-formedness with the rule rule WFT-RECUR to give us better induction principles for the proofs.

Definition 11. Rule WFT-RECUR is defined as:

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

Subtyping. Figure 6 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 12. Regularity: If $\Gamma \vdash_a A \leq B$ then $\vdash \Gamma$ and $\Gamma \vdash A$ and $\Gamma \vdash B$.

Theorem 13. Reflexivity.

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

Theorem 14. 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 15. 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 analyse the different subcases for A and B, but we still want to use the original A and B in the substitutions. For instance, suppose that we have $A := \mathit{nat} \to A_1 \to A_2$ and $B := \mathit{nat} \to B_1 \to B_2$. 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:

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

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:

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

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

By further analyzing the counter-example, we can see that the influence of contravariance on variables 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 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 16. If

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

then

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

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

- Case rule SA-var: In such case $A = B = \alpha$. For goal (1), we want to prove $\Gamma \vdash [\alpha \mapsto C]^n \leq [\alpha \mapsto D]^n$, which is actually premise (3) (with $n-1$). For 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 17. Goal (2) is proven by reflexivity.
- Case rule SA-arrow: In such case $A = A_1 \to A_2$ and $B = B_1 \to B_2$. By construction, we need to prove $\Gamma \vdash [\alpha \mapsto D]^n\ B_1 \leq [\alpha \mapsto C]^n\ A_1$ and $\Gamma \vdash [\alpha \mapsto C]^n\ A_2 \leq [\alpha \mapsto D]^n\ B_2$. 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).
- Case rule SA-rec: Apply the induction hypothesis.

□

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 16 relies on the following property of the subtyping relation:

Lemma 17. Antisymmetry of declarative subtyping: If $\Gamma \vdash A \leq B$ and $\Gamma \vdash B \leq A$ then $A = B$.

From Lemma 16, by letting $C := A$, $D := B$, we have

Lemma 18. If $\Gamma \vdash A \leq B$ and $\Gamma \vdash [\alpha \mapsto A]\ A \leq [\alpha \mapsto B]\ B$, then $\forall n,\ \Gamma \vdash [\alpha \mapsto A]^n\ A \leq [\alpha \mapsto B]^n\ B$.

The form of Lemma 18 is close to the shape of the infinite unfolding rule (rule S-rec) in declarative recursive types. Finally, we can prove the soundness theorem:

Theorem 19. 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 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 16 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 17, but for the algorithmic relation:

Lemma 20. 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 21. 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. 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 16: we proceed by induction on $\Gamma_1, \alpha, \Gamma_2 \vdash_a A \leq B$.

- Case rule SA-var: In such case $A = B = \alpha$. 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), from the condition of goal (2), we have $\Gamma_1, \alpha, \Gamma_2, \vdash_a D \leq C$, which is the inverse of premise (2). Thus $C = D$ by Lemma 20. Goal (2) is proven by reflexivity.
- Case rule SA-arrow: In such case $A = A_1 \to A_2$ and $B = B_1 \to B_2$. By construction, we need to prove $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\ D]B_1 \leq [\alpha \mapsto \mu\alpha.\ C]A_1$ and $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\ C]A_2 \leq [\alpha \mapsto \mu\alpha.\ D]B_2$. 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).
- Case rule SA-rec: Apply induction hypothesis.

□

Like Lemma 16, in Lemma 21 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 22. 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 Decidability

The subtyping relation is decidable. Our subtyping rules are syntax-directed, leading to a relatively simple proof of decidability. We found that doing the proof/induction directly on the declarative

system would lead to fewer cases in the proof of decidability. The reason is that the algorithmic rule for recursive types has 2 premises, whereas the declarative system has only 1. Thus, some surgical uses of soundness/completeness in the proof were enough to make it work, while saving us from dealing with extra cases compared to a proof using the algorithmic system instead.

Theorem 23. Decidability.

$$\text{If } \vdash \Gamma,\ \Gamma \vdash A \text{ and } \Gamma \vdash B \text{ then } \Gamma \vdash A \leq B \text{ or } \Gamma \vdash A \not\leq B.$$

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

5 SOUNDNESS AND COMPLETENESS OF THE AMBER RULES

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

5.1 The Challenges of Well-Formedness for the Amber Rules

In the original Amber rules by Amadio and Cardelli [1993] (Figure 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 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 on the type at the right-side of the subtyping relation (B). Sadly, this idea is not that simple to realise. Note that in the subtyping rule of function types (rule AMBER-ARROW), the input arguments are swapped, so without any changes on the environment the type variables in the types would go out-of scope, and this breaks the regularity lemma. Furthermore, trying to perhaps swap the variables in the environment to keep them in-scope changes the meaning of the environment ($\alpha \leq \beta$ becomes $\beta \leq \alpha$). Trying to ensure that the α's are only in scope in one side of the relation, while the β's are only in scope on the other side, turns out to be quite tricky. Therefore, to make progress, we propose a weaker restriction in this section: we allow 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 on the same type. Rather we would expect that the α appears in one of the types, and β on the other one. However, accepting such subtyping 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 7 shows a set of standard Amber rules with a built-in reflexivity rule.

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

Lemma 24. 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 8, is equivalent to the Amber rules in Figure 7. 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.
- **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.

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

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

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

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

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

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

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

These two exceptions are why we use "weakly" to characterize such formulation of subtyping. In contrast, existing formulations of positive subtyping, such as that described in Section 2.3 or originally described by Amadio and Cardelli [1993] do not make such exceptions and would reject the subtyping statements that we described above.

Well-formedness and subtyping. Well-formed types are the same as in Figure 4. Most subtyping rules are identical to those of the Amber rules, and the only differences are rule PosRes-var and rule PosRes-rec. Rule PosRes-rec checks that the recursive type variable α satisfies the weakly positive restriction relation $\alpha \in_+ \mu\alpha.\ A \leq \mu\alpha.\ B$, while maintaining the same behavior as the Amber rules. This subtle relation may be interpreted as follows: $\alpha \in_+ A \leq B$ means 1) $\alpha \leq \alpha$ does not appear in subderivations of $A \leq B$, or 2) $\alpha \leq \alpha$ appears in subderivations of $A \leq B$ only in positive positions, including unfolding of recursive types. For example, $\alpha \in_+ \top \to \alpha \leq \alpha \to \alpha$ holds. A second example is

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

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 β also appears positively in subderivations, which does not hold in this example. The reason we pose such restriction is because, for instance, 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 negative $\beta \leq \beta$ would appear in its subderivation. A similar issue happens whenever $\alpha \leq \alpha$ appears negatively and the recursive types are not equal to each other.

Note that weakly positive subtyping does not have reflexivity built-in. Instead, reflexivity is derivable using existing rules:

Lemma 25. Reflexivity.

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

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

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

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

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

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

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

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

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

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

Fig. 8. Weakly Positive Subtyping Rules

Proof. Induction on $\Gamma \vdash A$. For the recursive case (rule PosRes-rec), the first premise can be satisfied by induction hypothesis. On the other hand, since β is fresh, the second premise can be satisfied by rule Posvar-recself. □

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

Lemma 26. 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 27. 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 26, we have all premises. □

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

Lemma 28. 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 33). 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 29. Translation of environments and types from the Amber rules.

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

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

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

Lemma 30. If $\Delta \vdash_{amb} A \leq B$ then $|\Delta| \vdash (\Delta)(A) \leq_+ (\Delta)(B)$.

We are now one step away from the soundness theorem: to prove that 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 31. 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. This lemma is important because it shows that, with Lemma 31 proved, we can derive the following lemma, which relates weakly positive subtyping to our algorithmic subtyping relation in the double unfolding form:

Lemma 32. If $\alpha \in_+ A \leq B$ and $\Gamma \vdash_a A \leq B$ then $\Gamma \vdash_a [\alpha \mapsto A]\ A \leq [\alpha \mapsto B]\ B$.

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

Theorem 33. If $\Gamma \vdash A \leq_+ B$ then $\Gamma \vdash_a A \leq B$.

Combining Lemma 19, 30 and 33, we have

Corollary 34. Soundness of the Amber rules with respect to 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 35. 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 36.

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

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 35 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 constructor 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 environmen 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

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

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

$$\alpha \in_+ \beta \to \alpha \leq \beta \to \alpha \ \wedge\ \beta \in_- \beta \to \alpha \leq \beta \to \alpha$$

$\boxed{\Gamma \vdash A \leq_+ B \rhd \Pi}$ *(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. 9. Position Allocation for Positive Subtyping

In the next step, we substitute the variables 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 to 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 9 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 37. 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 38. 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 9 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 38, where the subtyping relation of weakly positive subtyping preserves under the Amber rules.

Theorem 39. 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

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

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 40.

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

The statement is less general than Theorem 39, 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 forumalations 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 15, 36 and 40, we have

Corollary 41. Completeness of the Amber rules with respect to declarative formulation.

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

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 17 and 20 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.

6.1 Syntax, Well-Formedness and Subtyping

Syntax. The syntax of the calculus is:

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

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 represents 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 4. 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 6, 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.

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

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

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

$$\textsc{sa-nat}\ \frac{\vdash \Gamma}{\Gamma \vdash_a \mathsf{nat} \leq \mathsf{nat}} \qquad \textsc{sa-top}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash_a A \leq \top} \qquad \textsc{sa-var}\ \frac{\vdash \Gamma \qquad \alpha \in \Gamma}{\Gamma \vdash_a \alpha \leq \alpha}$$

$$\textsc{sa-arrow}\ \frac{\Gamma \vdash_a B_1 \leq A_1 \qquad \Gamma \vdash_a A_2 \leq B_2}{\Gamma \vdash_a A_1 \to A_2 \leq B_1 \to B_2}$$

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

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

Fig. 10. Well-formedness and subtyping rules for record types.

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

Theorem 42. Reflexivity

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

Theorem 43. 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 20) for proving the unfolding lemma. Instead of alpha-equivalence or syntactic equality, we introduce a weaker form of equivalence.

Definition 44 (Equivalence)**.**

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

With Definition 44, 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 45. 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 45, we can prove our core lemma, as we did before.

Lemma 46. 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 $\Gamma_1, \alpha, \Gamma_2 \vdash_a A \leq B$.

- Case rule SA-VAR: In such case $A = B = \alpha$. 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 premise (2) $\Gamma_1, \alpha, \Gamma_2 \vdash_a C \leq D$ 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 45, we get $\Gamma_1, \alpha, \Gamma_2 \vdash_a [\alpha \mapsto D]\ D \leq [\alpha \mapsto C]\ C$. As a result, $\Gamma_1, \Gamma_2 \vdash_a \mu\alpha.\ D \leq \mu\alpha.\ C$.
- Case rule SA-ARROW: In such case $A = A_1 \to A_2$ and $B = B_1 \to B_2$. By construction, we need to prove $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\ D]\ B_1 \leq [\alpha \mapsto \mu\alpha.\ C]\ A_1$ and $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\ C]\ A_2 \leq [\alpha \mapsto \mu\alpha.\ D]\ B_2$. The former one can be proved by using the induction hypothesis arising from goal (2), while the latter one can be proved by using induction hypothesis arising from goal (1).
- Case rules SA-REC and SA-RCD: Apply induction hypothesis.

□

Finally, we can prove the unfolding lemma:

Lemma 47. Unfolding Lemma

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

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.

Typing. As the top of Figure 11 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 11 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 48. Preservation.

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

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

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

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

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

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

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

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

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

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

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

Fig. 11. Typing and reduction rules for record types

Theorem 49. Progress.

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

7 COQ PROOFS

We have chosen the Coq (8.10) proof assistant [The Coq Development Team 2019] to develop our formalization. The whole Coq formalization is built with a third-party library Metalib[2], which provides support for the locally nameless representation [Aydemir et al. 2008] to encode binders.

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

Definition	File (in src/ folder)	Name in Coq	Notation
Well-formed Type (Figure 4)	definition.v	WFA E A	$\Gamma \vdash A$
Well-formed Type (Definition 2)	definition.v	WFS E A	$\Gamma \vdash A$
Well-formed Type (Definition 11)	definition.v	WF E A	$\Gamma \vdash A$
Declarative subtyping (Figure 4)	definition.v	Sub E A B	$\Gamma \vdash A \leq B$
Typing (Figure 5)	definition.v	typing E e A	$\Gamma \vdash e : A$
Reduction (Figure 5)	definition.v	step e1 e2	$e_1 \hookrightarrow e_2$
Algorithmic subtyping (Figure 6)	definition.v	sub E A B	$\Gamma \vdash_a A \leq B$
Well-formed Type (Figure 7)	amber_part_1.v	wf_amber E A	$\Delta \vdash A$
Amber rules (Figure 7)	amber_part_1.v	sub_amber E A B	$\Delta \vdash_{amb} A \leq B$
Weakly Positive restriction (Figure 8)	amber_part_1.v	posvar m X A B	$\alpha \in_m A \leq B$
Weakly Positive subtyping (Figure 8)	amber_part_1.v	sub_amber2 E A B	$\Gamma \vdash A \leq_+ B$

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

Definition	File (in src_extension/ folder)	Name in Coq	Notation
Well-formed Type (Figure 10)	definition.v	WF E A	$\Gamma \vdash A$
Subtyping (Figure 10)	definition.v	Sub E A B	$\Gamma \vdash A \leq B$
Typing (Figure 11)	definition.v	typing E e A	$\Gamma \vdash e : A$
Reduction (Figure 11)	definition.v	step e1 e2	$e_1 \hookrightarrow e_2$

7.1 Definitions

SLTC 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 and 4. All the definitions in the paper can be found in files *definition.v* and *amber_part_1.v*. Table 2 shows the correspondence of definitions between the paper and the Coq artifacts. The file *definition.v* contains the definitions for our type system. It has definitions of well-formedness, subtyping (both declarative and algorithmic), typing, and reduction. The file *amber_part_1.v*, contains the definitions for the Amber rules and the intermediate subtyping relation based on a weakly positive restriction presented in Section 5.

For encoding variables and binders, we use the locally nameless representation to express all the types and terms. In 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 rules WFT-REC, WFT-INF, and WFT-RECUR. In the Coq formalization, we have three distinct definitions of well-formedness, which are proved to be equivalent. WFA E A is the relation containing rule WFT-REC, WFS E A is the relation containing rule WFT-INF, and WF E A is the relation containing rule WFT-RECUR.

SLTC 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.

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

Table 4. Descriptions for the proof scripts.

Theorems	Description	Files (in src/ folder)	Name in Coq
Theorem 4	Reflexivity for declarative subtyping	subtyping.v	refl
Theorem 5	Transitivity for declarative subtyping	subtyping.v	Transitivity
Lemma 7	Unfolding lemma for declarative subtyping	subtyping.v	unfolding_lemma
Theorem 9	Preservation	typesafety.v	preservation
Theorem 10	Progress	typesafety.v	progress
Theorem 13	Reflexivity for algorithmic subtyping	subtyping.v	refl_algo
Theorem 14	Transitivity for algorithmic subtyping	subtyping.v	trans_algo
Theorem 15	Completeness of algorithmic subtyping	subtyping.v	completeness
Theorem 19	Soundness of algorithmic subtyping	subtyping2.v	soundness
Lemma 22	Unfolding lemma for algorithmic subtyping	subtyping2.v	unfolding_lemma_version2
Theorem 23	Decidability	decidability.v	decidability
Theorem 25	Reflexivity for weakly positive subtyping	amber_part_1.v	sub_amber2_refl
Theorem 27	Transitivity for weakly positive subtyping	subtyping3.v	sub_amber2_trans
Lemma 28	Unfolding lemma for weakly positive subtyping	subtyping3.v	unfolding_for_pos
Theorem 30	Translation from Amber subtyping to weakly positive subtyping	amber_part_1.v	sub_amber_to_amber_2
Theorem 33	Translation from weakly positive subtyping to algorithmic subtyping	amber_part_2.v	sub_amber_2_to_sub
Corollary 34	Soundness of the Amber rules	amber_part_2.v	amber_soundness2
Theorem 36	Translation from algorithmic subtyping to weakly positive subtyping	subtyping4.v	sub_to_amber2
Theorem 39	Translation from weakly positive subtyping to Amber subtyping	amber_part_3.v	amber_complete_aux
Corollary 41	Completeness of the Amber rules	amber_part_3.v	amber_complete2

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

Theorems	Description	Files (in src_extension/ folder)	Name in Coq
Theorem 42	Reflexivity	subtyping.v	sub_refl
Theorem 43	Transitivity	subtyping.v	Transitivity
Lemma 46	Unfolding lemma	unfolding.v	unfolding_lemma
Theorem 48	Preservation	typesafety.v	preservation
Theorem 49	Progress	typesafety.v	progress

Lemma 21 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 21 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. 12. Comparison between paper and Coq statements for Lemma 21.

7.2 Lemmas and Theorems

Table 4 shows the descriptions for all the proof scripts in Section 3, Section 4 and Section 5. 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 12 illustrates the difference between the formulations with and without modes for the unfolding lemma. Our Coq formalization uses some meta-functions on modes instead to formalize the same result. Using meta-functions on modes (Definition 50), the same lemma would look like the right part of Figure 12.

Definition 50. Mode selector.

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

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 *definition.v*.

7.3 Variables Generation

Another difficulty worth to mention is generating a bundle of variables in Definition 37. 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/amber_part_3.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 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 37 describes.

8 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.

Other approaches to iso-recursive subtyping. Ligatti et al. [2017] propose an improvement to the Amber rules for iso-recursive subtyping. They observe that the Amber rules are sound, but

incomplete with respect to type-safety. For example, the Amber rules reject $\mu\alpha.\ \alpha \leq \mu\alpha.\ (\mu\beta.\ \alpha)$, but such a subtyping derivation does not violate type-safety. Ligatti et al. [2017] propose a set of subtyping rules that is both sound and complete with respect to type-safety. The new formulation of subtyping proposed by us is also incomplete from the point of view of type-safety, since it has 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 modular. In particular, the new rules proposed by us for recursive subtyping are quite modular and can be added to an existing (syntactic) formulation of subtyping with minimal impact. In contrast, the more complete rules by Ligatti et al. [2017] require very specific environments for subtyping, and they also require some non-standard subtyping rules for *value-uninhabited* types. In terms of metatheory, their technique of *failing derivations* for proving reflexivity and transitivity is non-constructive and hard to formalize in a theorem prover.

For solving the conflict between contravariant types and recursive types, Hofmann and Pierce [1996] proposed an approach where only covariant types are allowed. In their subtyping rules, the inputs of function types must be the same. Later, Hosoya et al. [1998] gave an algorithm to prove transitivity and type soundness, but it still relies on a complicated environment where all of the components are pairs of structural recursive types. Thus, they have extra rules for contexts to obtain enough information for the subtyping assumptions. Featherweight Java [Igarashi et al. 2001], is another calculus that supports a form of iso-recursive types. Although there are no specific recursive type constructs, recursive types appear because class declarations can be recursive. An advantage of the Featherweight Java design is that recursive types are fairly easy to model, and modeling mutually recursive 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 8) is close to Backes et al. [2014]'s definition. However, our definition is more general since equal types with negative recursive occurrences are considered subtypes, whereas in their formulation recursive types with negative occurrences of recursive variables are forbidden. Appel and Felty [2000] gave a related Twelf proof of positive subtyping, where function types are invariant with respect to the input types of functions. Recently, based on big-step semantics, Amin and Rompf [2017] gave a formalization of DOT, which employs a special form of equi-recursive type. Danielsson and Altenkirch [2010], mixes induction and coinduction for proving properties of equi-recursive subtyping in Agda.

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

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

ACKNOWLEDGMENTS

We are grateful to the anonymous OOPSLA reviewers and Yaozhu Sun 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. This work has been sponsored by Hong Kong Research Grant Council projects number 17209519 and 17209520.

REFERENCES

M. Abadi and L. Cardelli. 1996. *A Theory of Objects*. New York.

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.

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)*.

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

on Functional Programming (ICFP 2011).

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

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

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

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.