QuickSub: Efficient Iso-Recursive Subtyping

LITAO ZHOU, University of Hong Kong, China
BRUNO C. D. S. OLIVEIRA, University of Hong Kong, China

Many programming languages need to check whether two recursive types are in a subtyping relation. Traditionally recursive types are modelled in two different ways: equi- or iso- recursive types. While efficient algorithms for subtyping equi-recursive types are well studied for simple type systems, efficient algorithms for iso-recursive subtyping remain understudied.

In this paper we present `QuickSub`: an efficient and simple to implement algorithm for iso-recursive subtyping. `QuickSub` has the same expressive power as the well-known iso-recursive Amber rules. The worst case complexity of `QuickSub` is $O(nm)$, where m is the size of the type and n is the number of recursive binders. However, in practice, the algorithm is *nearly linear* with the worst case being hard to reach. Consequently, in many common cases, `QuickSub` can be several times faster than alternative algorithms. We validate the efficiency of `QuickSub` with an empirical evaluation comparing it to existing equi-recursive and iso-recursive subtyping algorithms. We prove the correctness of the algorithm and formalize a simple calculus with recursive subtyping and records. For this calculus we also show how type soundness can be proved using `QuickSub`. All the results have been formalized and proved in the Coq proof assistant.

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

Additional Key Words and Phrases: Recursive types, Subtyping, Algorithm

ACM Reference Format:
Litao Zhou and Bruno C. d. S. Oliveira. 2025. QuickSub: Efficient Iso-Recursive Subtyping. *Proc. ACM Program. Lang.* 9, POPL, Article 33 (January 2025), 33 pages. https://doi.org/10.1145/3704869

1 Introduction

There are two main approaches to model recursive types in type systems: equi-recursive types and iso-recursive types. Iso-recursive types [Crary et al. 1999] treat a recursive type and its unfolding as different types. The recursive type and its unfolding are related by inserting term level fold/unfold constructs. Equi-recursive types [Morris 1968] treat recursive types and their unfoldings as equal:

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

since they represent the same infinite tree [Amadio and Cardelli 1993]. Equi-recursive equivalence is powerful and useful to type check many programs without requiring explicit fold and unfold annotations. This can be convenient for programming, since there is no need to change the term structure. In addition, equi-recursive subtyping is also well-studied [Amadio and Cardelli 1993].

While equi-recursive types look appealing, there are some important considerations when choosing whether to adopt them. The powerful form of equivalence or subtyping comes at a cost. As it is well-known from the literature [Brandt and Henglein 1998], equi-recursive equivalence and subtyping requires *coinductive* reasoning. Coinductive reasoning leads to complications in the metatheory, as well as to relatively high algorithmic complexity. Thus, there has been significant

Authors' Contact Information: Litao Zhou, University of Hong Kong, Hong Kong, China, ltzhou@cs.hku.hk; Bruno C. d. S. Oliveira, University of Hong Kong, Hong Kong, China, bruno@cs.hku.hk.

ACM 2475-1421/2025/1-ART33
https://doi.org/10.1145/3704869

research effort on efficient algorithms for both equi-recursive subtyping [Gapeyev et al. 2002; Kozen et al. 1993] and equivalence [Cardone and Coppo 1991; Coppo 1985] for simple type systems. Gapeyev et al. presented an efficient algorithm for checking equi-recursive subtyping with at most $O(n^2)$ recursive calls, where n is the number of recursive binders. Kozen et al. reduced the subtyping problem to the problem of testing automata emptiness and achieved quadratic time complexity. Therefore, it can be said that efficient algorithms for equi-recursive subtyping have been foundationally well understood in the literature, despite the overall computational costs still being expensive.

Equi-recursive algorithms are also non-trivial to extend with more advanced type system features. For instance, the interaction between equi-recursive types and type constructors, which is necessary for modeling ML-style recursive modules [Crary et al. 1999], is complex: it is unknown whether type equivalence remains decidable in that setting. For another example, the combination of bounded quantification and equi-recursive subtyping introduces significant complications [Colazzo and Ghelli 2005; Ghelli 1993; Jeffrey 2001], and requires rather complex metatheory.

Iso-recursive types are less convenient but, on the other hand, they: 1) are easier to scale to more advanced features; 2) have comparatively simpler metatheory; and, 3) are *perceived* as having computationally less complex operations. The first two points are backed up by strong evidence in the literature. For instance, follow-up work on recursive modules has adopted iso-recursive types to obtain more practical module systems with decidable type equivalence [Dreyer 2005; Dreyer et al. 2001; Russo 2001], since the interaction of iso-recursive types and type constructors is simpler. The interaction between bounded quantification and iso-recursive typing is also simple, as illustrated by Zhou et al. [2023], leading to a natural extension of $F_{<:}$ [Cardelli and Wegner 1985] with iso-recursive types. In addition, despite being less convenient, iso-recursive types are known to have the same expressive power as equi-recursive types [Abadi and Fiore 1996; Zhou et al. 2024].

The point about computational complexity (3) deserves more discussion. Since most formulations of iso-recursive types remain *inductive*, iso-recursive subtyping algorithms only have to deal with finite trees. Rossberg [2023] argues that is an important point to consider for performance sensitive applications as, in principle, it leads to more efficient implementations of equivalence and subtyping. Rossberg is motivated by the use of recursive types in WebAssembly (Wasm) and type-safe low-level languages in general. He argues that, in those performance sensitive settings, iso-recursive types are a significantly more attractive choice compared to equi-recursive types. His work presents an efficient formulation of *declared* iso-recursive types, which are adopted in practice by Wasm. This declared formulation of iso-recursive types is very efficient as it only requires subtyping to be checked at the declaration of recursive types.

Unlike equi-recursive types, efficient subtyping algorithms for iso-recursive types have received less attention in the literature. For traditional iso-recursive types the most well-known formulation of subtyping employs the Amber rules [Cardelli 1985, 1993]. In contrast to Rossberg's more restrictive declared subtyping, the Amber rules follow a purely structural approach where a recursive type is related to any other recursive type with a compatible structure. Furthermore, the Amber rules are inductively defined, which, according to Rossberg, should be an advantage in obtaining an algorithmic formulation. However, a naive implementation of the Amber rules has exponential time complexity, which is *worse* than the complexity reported for the best-known algorithms for equi-recursive subtyping. The culprit for the cost of the Amber rules is the need for a built-in reflexivity rule. While several alternative formulations of subtyping with equivalent expressive power to the Amber rules exist in the literature [Zhou et al. 2022b], and can avoid built-in reflexivity, none of these formulations is efficient. Surprisingly, there is little work on efficient formulations of (non-declared) iso-recursive subtyping. A notable exception is the work by Ligatti et al. [2017],

which presents a more powerful formulation of iso-recursive subtyping, using coinductive techniques similar to those adopted in equi-recursive subtyping. Ligatti et al. designed an algorithm with efficiency in mind. However, the use of coinductive reasoning still leads to relatively high computational costs in practice.

In this paper we present `QuickSub`: an efficient algorithm for iso-recursive subtyping, which is also simple to implement. We prove that `QuickSub` has the same expressive power as the iso-recursive Amber rules. The worst case complexity of `QuickSub` is $O(nm)$, where m is the size of the type and n is the number of recursive binders. However, the worst case is hard to reach. Many common cases are linear in practice. For instance, for positive recursive types – which are the only kinds of recursive types supported in languages like Coq [Coq Development Team 2024] or Agda [Norell 2007], and the most common case in functional programming – the algorithm is linear in practice. Moreover, there are still many cases with negative recursive types where the algorithm remains linear. Consequently, `QuickSub` can be several times faster (sometimes by orders of magnitude) than alternative algorithms, in those cases.

The efficiency of `QuickSub` is validated via an empirical evaluation comparing it to existing equi-recursive and iso-recursive subtyping algorithms implemented in OCaml [Leroy et al. 2021]. In addition we formalize a simple calculus with recursive subtyping and records. For this calculus we also show how type soundness, transitivity of subtyping and the unfolding lemma [Zhou et al. 2022b] can be proved using `QuickSub`.

We believe that our work validates the perceived intuition that (inductive) iso-recursive subtyping is, in practice, computationally simpler than formulations based on coinduction. Furthermore, it provides a practical and effective algorithm that can be employed in applications where performance is an important consideration or when equi-recursive subtyping is impractical or undecidable.

In summary, the contributions of this paper are:

- **An efficient algorithm for iso-recursive subtyping.** We introduce `QuickSub`, a novel algorithm for iso-recursive subtyping that significantly improves efficiency over existing algorithmic formulations.
- **Equivalence proof to the Amber rules.** We prove that `QuickSub` is equivalent in expressive power to the well-known iso-recursive Amber rules.
- **A direct proof of type soundness of the subtyping rules.** We provide a direct type soundness proof for a calculus that employs iso-recursive types and `QuickSub` subtyping, without relying on any equivalence results to other existing iso-recursive subtyping rules.
- **Extension to record types.** We extend `QuickSub` to $\texttt{QuickSub}^{\{\}}$ that handles record subtyping, broadening the applicability and flexibility of the algorithm.
- **Coq formalization, OCaml implementation and empirical evaluation.** We provide a mechanical formalization and proofs for all the results in Coq. We also evaluate an OCaml implementation of the algorithm and discuss its performance.

2 Overview

This section provides background of existing approaches to iso-recursive subtyping, and their drawbacks in terms of efficiency. Then it introduces key ideas leading to `QuickSub`.

2.1 Subtyping Iso-Recursive Types

For recursive types it is common and useful to have subtyping. The metatheory of both equi-recursive subtyping [Amadio and Cardelli 1993; Brandt and Henglein 1998; Danielsson and Altenkirch 2010; Gapeyev et al. 2002] and iso-recursive subtyping [Cardelli 1985; Ligatti et al. 2017; Zhou et al. 2020, 2022b] has been studied in the literature. In this paper, we focus on iso-recursive

subtyping. Before diving into the rules for subtyping iso-recursive types, we first look at some examples to see what kind of recursive types can be in a subtyping relation. We assume standard structural subtyping rules for basic type constructs, such as:

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

The subtyping rule for function types is contravariant in the argument type and covariant in the return type.

With iso-recursive subtyping, it is expected that if two recursive types are subtypes, then their unfoldings should also be subtypes, which can be expressed as follows:

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

This property is called the *unfolding lemma* in the literature [Zhou et al. 2022b], and plays a key role in the type soundness proof in calculi with iso-recursive subtyping. We will reason about the unfolding lemma formally for `QuickSub` in Section 3. Readers may refer to Zhou et al. [2022b] for a more comprehensive discussion on this property. Here we use the unfolding lemma to identify valid subtyping relations between various examples.

The examples are categorized by different polarities (positive and negative) of recursive variables. Simply put, a type variable occurs *positively* if it appears on the left side of an even number of arrows ($\to$) in a function type, and *negatively* if it appears on the left side of an odd number of arrows. For example, in the type $\alpha \to (\beta \to \gamma)$, α and β occurs negatively, while γ occurs positively. We will see that the polarity of recursive variables plays a crucial role in determining the subtyping relation between recursive types.

Positive recursive subtyping. Consider $\mu\alpha.\ \top \to \alpha \leq \mu\alpha.\ \mathsf{nat} \to \alpha$, where the recursive variable α appears positively. The left type can be regarded as a function that consumes infinite values of any type, and the right type consumes infinite nat values. Therefore, the left type is more general than the right type. The unfolding lemma is helpful to validate this subtyping statement:

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

After unfolding, we have to check that $\mathsf{nat} <: \top$, which is true. If we continue applying additional unfoldings, the subtyping statement would remain valid and simply require more $\mathsf{nat} <: \top$ substatements to be validated. When the recursive variable only appears in positive positions, the subtyping relation is straightforward: it suffices to compare the recursive bodies for subtyping.

Negative recursive subtyping. Next, we consider $\mu\alpha.\ \alpha \to \mathsf{nat} \not\leq \mu\alpha.\ \alpha \to \top$, where the type on the left can be seen as an object that takes itself and produces a nat value. In contrast, the type on the right takes itself and produces a top value. The subtyping statement above *does not hold*, as a term of the left type cannot be used where the right type is expected. The type on the right expects an input of an object capable of producing any values, but the type on the left only produces nat values. This inconsistency can be discovered by unfolding the recursive types twice:

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

Due to the contravariant comparison in function types, we need to check not only $\mathsf{nat} \leq \top$ but also $\top \leq \mathsf{nat}$, which does not hold. It can be seen that negative occurrences of recursive variables prevent many subtyping statements that should hold when the recursive variables are considered as free variables, as in the positive recursive subtyping case. Indeed, they make the subtyping relation "almost" equality. Since the reflexivity property is expected for subtyping, subtyping judgments like $\mu\alpha.\ \alpha \to \mathsf{nat} \leq \mu\alpha.\ \alpha \to \mathsf{nat}$ should still hold.

At this point it is useful to introduce the concept of *equivalent* and *strict* subtyping, which will play an important role in `QuickSub`. By equivalent subtyping, we mean that two types are subtypes

of each other, i.e. A and B are equivalent if $A \leq B$ and $B \leq A$. In negative recursive subtyping, recursive types that have equivalent bodies should also be equivalent and therefore in the subtyping relation. By strict subtyping, we mean that A is a subtype of B but B is not a subtype of A.

Special cases in negative recursive subtyping. Including equivalent subtyping is not the end of the story for subtyping negative recursive types. For example, consider $\mu\alpha.\ \top \to \alpha \leq \mu\alpha.\ \alpha \to \alpha$, which is a valid strict subtyping relation with negative recursive types. Despite the negative occurrence of α in the right type, if we unroll the two recursive types, what we need to compare is

$$\top \to (\mu\alpha.\ \top \to \alpha) \leq (\mu\alpha.\ \alpha \to \alpha) \to (\mu\alpha.\ \alpha \to \alpha)$$

and the contravariant comparison $\mu\alpha.\ \alpha \to \alpha \leq \top$ clearly holds.

Handling negative recursive subtyping can be quite tricky. There are many invalid subtyping statements due to contravariance of function types, but we must still allow for the special cases such as reflexivity and subtyping with $\top$ types to hold. Existing approaches to iso-recursive subtyping have introduced various strategies to navigate these complexities and proposed several sets of inference rules for subtyping iso-recursive types. However, these solutions often entail a compromise on efficiency, as we will discuss in the next section.

Nested recursive subtyping. The problem of subtyping iso-recursive types becomes even more complicated when nested recursive types are involved. The polarity of variables becomes less clear in the presence of nested recursive types. Consider comparing the following two types:

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

The variable β appears to be in a positive position in the recursive body of the two types. However, due to the existence of a variable α appearing in a negative position, the two types are not related by subtyping. To see this, we unfold the $\mu\alpha.\ \alpha \to \beta$ twice and compare the two types:

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

As highlighted by the gray color, the variable β appears in a negative position in the recursive body after unfolding. Since the recursive bodies for the β variable are strict subtypes, the two types are not subtypes. This example shows that nested recursive types can hide negative occurrences of recursive variables as "apparently" positive ones. Thus, it is difficult to test for the positivity of recursive variables, and the subtyping rules need to be carefully designed to handle such cases.

2.2 Algorithmic Iso-Recursive Subtyping

There are several existing approaches that provide algorithms for iso-recursive subtyping. Next we give an overview of existing approaches and their issues in terms of efficiency.

Amber-style iso-recursive subtyping. The Amber rules have long been known and used for subtyping iso-recursive types. They were initially introduced informally by Cardelli [1985] in the Amber programming language, and later formally studied by Amadio and Cardelli for subtyping equi-recursive types. For subtyping iso-recursive types, variants of Amadio and Cardelli's rules have been widely used in many different calculi and programming languages [Abadi and Cardelli 1996; Abadi and Fiore 1996; Bengtson et al. 2011; Chugh 2015; Duggan 2002; Lee et al. 2015; Swamy et al. 2011]. The key rules are the rules Amber-var and Amber-rec to compare recursive types.

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

For example, the derivation below shows an application of the Amber rules for subtyping the positive recursive subtyping example that we have seen before:

$$\dfrac{\dfrac{\dfrac{\alpha \le \beta \vdash_{amb} \mathsf{nat} \le \top \qquad \dfrac{\alpha \le \beta \in \alpha \le \beta}{\alpha \le \beta \vdash_{amb} \alpha \le \beta}\ \textsc{Amber-var}}{\alpha \le \beta \vdash_{amb} \top \to \alpha \le \mathsf{nat} \to \beta}\ \textsc{Amber-arrow}}{\cdot \vdash_{amb} \mu\alpha.\ \top \to \alpha \le \mu\beta.\ \mathsf{nat} \to \beta}\ \textsc{Amber-rec}$$

As discussed before, due to the contravariance of function subtyping, many negative recursive types should be prevented from the subtyping relation. For example, $\mu\alpha.\ \alpha \to \mathsf{nat}$ is not a subtype of $\mu\beta.\ \beta \to \top$. The Amber rules deal with this issue nicely with rule Amber-rec, as shown below:

$$\dfrac{\dfrac{\ldots \quad \alpha \le \beta \vdash_{amb} \beta \not\le \alpha}{\alpha \le \beta \vdash_{amb} \alpha \to \mathsf{nat} \not\le \beta \to \top}\ \textsc{Amber-arrow}}{\cdot \vdash_{amb} \mu\alpha.\ \alpha \to \mathsf{nat} \not\le \mu\beta.\ \beta \to \top}\ \textsc{Amber-rec}$$

However, such a design also prevents negative recursive types from being subtypes of themselves, which should be allowed due to reflexivity. Therefore, an explicit reflexivity rule Amber-self is needed to handle this case. The presence of rule Amber-self introduces a non-deterministic choice in the subtyping algorithm, since this rule overlaps with rule Amber-rec. This results in a significant performance overhead in the presence of nested recursive types. The built-in reflexivity rule also makes it difficult for the subtyping relation to be extended to non-antisymmetric subtyping relations [Ligatti et al. 2017; Zhou et al. 2022b], such as record subtyping, as discussed in Section 4.

Another technical issue with the Amber rules is that the recursive variable names need to be carefully chosen. When using the rule Amber-rec, the names for the two recursive type variables must be different. This is not conventional in practice, as programmers may write their own names for recursive types or compilers may generate variable names for anonymous recursive types in a way that does not satisfy this requirement. If one considers an internal representation of recursive types, such as de Bruijn indices [De Bruijn 1972] or named representations, then in order to use the Amber rules, one needs to traverse the recursive body every time using rule Amber-rec and rename the variables to a fresh free variable, which leads to a bottleneck in performance as well.

Cardelli [1993] considered an alternative algorithmic formulation of the Amber rules using de Bruijn indices that avoids the issue of variable renaming and non-deterministic choices of recursive subtyping rules [Cardelli 1993, Appendix G.2]. Essentially he defined a new set of rules to compute the "ties" between the recursion variables, which decides whether the recursive types are positive or not before comparing the recursive bodies. However, as we have discussed, with negative recursive subtyping, there can be more complex cases than just equivalence. Since Cardelli [1993]'s algorithmic rules differ a lot from the original form of the Amber rules and there are no formal proofs for the algorithmic rules, it is not clear whether the rules still have the same expressive power as the Amber rules. Moreover, computing the ties requires several extra traversals for the recursive types, and the equivalence is checked by running the subtyping algorithm twice, i.e. checking $A \le B$ and $B \le A$ for $A \approx B$, which still incurs a performance overhead.

Subtyping by nominal unfolding. Despite the Amber rules being widely used in practice, the metatheory for the iso-recursive subtyping Amber rules has not been well studied until recently [Zhou et al. 2020, 2022b]. In Zhou et al.'s work, they proposed a new specification for iso-recursive subtyping rules by *finite unfolding*, and showed its equivalence to the Amber rules. They have also formulated an *algorithmic* subtyping relation, called *nominal unfolding*, to deal with iso-recursive subtyping. The nominal unfoldings rules are easy to work with formally and extend to other subtyping relations, such as records, intersection types [Zhou et al. 2022a], and bounded

quantification [Zhou et al. 2023] with mechanized proofs in Coq. The nominal unfolding rules are:

$$\text{SN-VAR}\ \frac{\vdash \Gamma \qquad \alpha \in \Gamma}{\Gamma \vdash_n \alpha \le \alpha} \qquad \text{SN-REC}\ \frac{\Gamma, \alpha \vdash_n [\alpha \mapsto A^{\alpha}]\ A \le [\alpha \mapsto B^{\alpha}]\ B}{\Gamma \vdash_n \mu\alpha.\ A \le \mu\alpha.\ B} \qquad \text{SN-LABEL}\ \frac{\Gamma \vdash_n A \le B}{\Gamma \vdash_n A^{\alpha} \le B^{\alpha}}$$

The key design for the nominal unfolding rules is that recursive type bodies need to be unfolded by labeled types A^{α} instead of just variables α. Labeled types basically help with the double unfolding of the recursive type (as seen in the premise of rule SN-REC), so that the contravariant occurrences of recursive variables can be checked with an extra unfolding. The labels provide a distinct nominal identity to the unfolded form of recursive types so that they can only be compared to unfoldings of the same recursive type. For example, checking the previous positive subtyping example with the nominal unfolding rules follows the derivation below:

$$\dfrac{\dfrac{\alpha \vdash_n \mathsf{nat} \le \top \qquad \dfrac{\dfrac{}{\alpha \vdash_n \alpha \le \alpha}\,\text{SN-VAR}}{\alpha \vdash_n \top \to \alpha \le \mathsf{nat} \to \alpha}\,\text{SN-ARROW}}{\alpha \vdash_n (\top \to \alpha)^{\alpha} \le (\mathsf{nat} \to \alpha)^{\alpha}}\,\text{SN-LABEL}}{\alpha \vdash_n \top \to (\top \to \alpha)^{\alpha} \le \mathsf{nat} \to (\mathsf{nat} \to \alpha)^{\alpha}}\,\text{SN-ARROW}}{\cdot \vdash_n \mu\alpha.\ \top \to \alpha \le \mu\alpha.\ \mathsf{nat} \to \alpha}\,\text{SN-REC}$$

The nominal unfolding rules differ from the Amber rules in several ways. Firstly, recursive type variables are not required to be distinct, which makes the rules more convenient for reasoning and fits better with various representations of binders. Secondly, in terms of metatheory, the nominal unfolding rules do not have a built-in reflexivity rule, which makes the subtyping relation applicable to subtyping relations that are not antisymmetric. Zhou et al. proved that the nominal unfolding rules are type sound, and have the same expressive power as the iso-recursive Amber rules via an intermediate iso-recursive subtyping formulation, called *weakly positive subtyping*. This formulation is discussed in Section 3.2 to establish the expressive power of `QuickSub`.

Despite these results, the nominal rules are not an efficient algorithm to use in practice. As one can see in rule SN-REC, the unfolding of recursive types can cause an exponential blowup of size in terms of the depth of nested recursive variables. The weakly positive subtyping formulation also has its implementation issues, as we will discuss after introducing them in Section 3.2.

Complete iso-recursive subtyping. As we have seen, the Amber rules, as well as equivalent formulations such as the nominal unfolding rules, lack an efficient algorithmic implementation. There are alternatives to the Amber rules though, that may have a different expressive power but come with efficient algorithms. One such alternative is by Ligatti et al. [2017]. They find that the Amber rules are sound, but incomplete with respect to type-safety. In other words, there can be more subtyping statements that are not covered by the Amber rules, but are still type-safe in the setting of iso-recursive types. These mainly come from recursive type unrolling. The rules for subtyping recursive types employed by Ligatti et al. are:

$$\frac{S, \mu\alpha.\ A \le \mu\beta.\ B \vdash [\alpha \mapsto \mu\alpha.\ A]\ A \le [\beta \mapsto \mu\beta.\ B]\ B}{S \vdash \mu\alpha.\ A \le \mu\beta.\ B}\text{L17-REC1} \qquad \frac{(\mu\alpha.\ A \le \mu\beta.\ B) \in S}{S \vdash \mu\alpha.\ A \le \mu\beta.\ B}\text{L17-REC2}$$

The basic idea is that subtyping environments S track all subtyping relations between recursive types that have already been observed. When deciding whether two recursive types are in a subtyping relation, the environment is consulted first to see if the relation has been observed before (rule L17-REC2). If not, the relation is added to the environment and the recursive type variables are replaced by the recursive types in the bodies (rule L17-REC1). Rule L17-REC1 resembles similar designs in equi-recursive subtyping rules [Brandt and Henglein 1998; Gapeyev et al. 2002],

and pushes the iso-recursive subtyping relation one step further to also consider recursive type unrolling. For example, the subtyping relation $\mu\alpha.\ \top \to \alpha \leq \mu\alpha.\ \top \to (\mu\beta.\ \mathsf{nat} \to \beta)$ is accepted by their rules, but not by the Amber rules. For a detailed discussion we refer to Zhou et al. [2022b]'s paper for a comparison between the Amber rules and the rules by Ligatti et al..

Rule L17-REC1 may look costly in terms of performance, as it requires unfolding the recursive types with themselves. However, in practice, it can be implemented by compressing all representations of μ-types into an "unroll table" that maps recursive type variables to their unrolled counterparts. In Ligatti et al.'s paper, one such algorithm is presented and shown to have a complexity of $O(mn)$ where m is the number of recursive type variables and n is the size of the types in the two types being compared, whichever is larger. Ligatti et al. informally argued that this algorithm is equivalent to the complete rules, but they did not prove this result formally.

2.3 Key Ideas Towards an Efficient Algorithm

The goal of `QuickSub` is to design an efficient algorithm that has the same expressive power as the Amber rules. Since Amber-style subtyping strikes a good balance between expressive power and simplicity, and has been widely adopted, we believe having an efficient and equivalent algorithm for the Amber rules is of practical significance.

In the rest of this section, we introduce the key design ideas behind the `QuickSub` algorithm with two goals in mind: being efficient to implement, while also being equivalent in terms of expressive power to the Amber rules. We will use the subtyping examples in previous sections to illustrate these key ideas. The actual final `QuickSub` algorithm will be introduced in Section 3, exploiting and refining the ideas described in this section.

Tracking polarities. The first observation from the previous examples is that, if the recursive variables all appear in positive positions, then we can just compare the recursive bodies directly. For example, for $\mu\alpha.\ \top \to \alpha \leq \mu\alpha.\ \mathsf{nat} \to \alpha$, we could simply compare the bodies of the recursive types for subtyping. This can be described by the following rule:

$$\frac{\Psi, \alpha \vdash A \leq B \quad \alpha \notin \mathsf{NegVar}(A, B)}{\Psi \vdash \mu\alpha.\ A \leq \mu\alpha.\ B}\ \text{QS-REC-1}$$

where $\mathsf{NegVar}(A, B)$ is the set of negative recursive variables in A and B. In this way, we do not need to unfold the recursive types as in the nominal unfolding rules, and can directly compare the bodies of the recursive types. The rule can be further optimized by collecting $\mathsf{NegVar}(A, B)$ and checking the inclusion of α on the fly, as we will show in Section 3.

Tracking strict subtypes. Rule QS-REC-1 is not sufficient to handle negative recursive types. A recursive type with negative occurrences of recursive variables, such as $\mu\alpha.\ \alpha \to \mathsf{nat}$, though not a subtype of $\mu\alpha.\ \alpha \to \top$, is still a subtype of itself due to reflexivity. The Amber rules deal with this by having an extra reflexivity rule AMBER-SELF, which is not ideal for an efficient algorithm, since it requires backtracking. To avoid backtracking, our solution is to distinguish strict subtyping from equivalence. This idea can be described by the following rules:

$$\frac{\Psi, \alpha \vdash A \lessapprox B \quad \alpha \notin \mathsf{NegVar}(A, B)}{\Psi \vdash \mu\alpha.\ A \lessapprox \mu\alpha.\ B}\ \text{QS-REC-2A} \qquad \frac{\Psi, \alpha \vdash A \approx B \quad \alpha \in \mathsf{NegVar}(A, B)}{\Psi \vdash \mu\alpha.\ A \approx \mu\alpha.\ B}\ \text{QS-REC-2B}$$

Instead of using a unified symbol $\leq$ to indicate successful subtype checking, the checking result is now represented as a metavariable $\lessapprox \in \{<, \approx\}$ for strict subtyping and equivalence. In other words, we do not interpret the subtyping rules as a function that compares two types for subtyping and returns a boolean. Instead, we would interpret subtyping as a function that compares two types and gives us three possible results: 1) the two types are *strict* subtypes ($<$); 2) the two types are

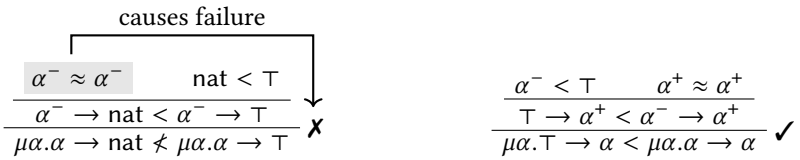


Fig. 1. Illustration of negative recursive subtyping

equivalent subtypes ($\approx$); or 3) subtyping fails. Rule QS-REC-2A accepts both equivalence and strict subtyping for positive recursive types, while rule QS-REC-2B only accepts equivalence for negative recursive types. This way, we can avoid backtracking for reflexivity, since the choice of the rules is deterministic based on whether α is in $\text{NegVar}(A, B)$.

Accordingly, the subtype result needs to be tracked throughout the rules. For example, in base cases of the inference rules we have $\text{nat} \approx \text{nat}$ and $A < \top$ when $A \neq \top$, rather than the generic $\text{nat} \leq \text{nat}$ or $A \leq \top$. The subtype result also needs to be propagated when combining subtyping results. For example, the subtyping rule for function types would be:

$$\frac{\Psi \vdash B_1 \lessapprox_1 A_1 \quad \Psi \vdash A_2 \lessapprox_2 B_2}{\Psi \vdash A_1 \to A_2 \ (\lessapprox_1 \bullet \lessapprox_2)\ B_1 \to B_2}\ \text{QS-ARROW-ATTEMPT}$$

Here $\lessapprox_1 \bullet \lessapprox_2$ is a composition function that combines the subtyping results $\lessapprox_1$ and $\lessapprox_2$ to a new subtyping result. It works by checking if both subtyping results are $\approx$. If so, the composition is $\approx$; otherwise, it is $<$. Tracking the equality and strict subtype information precisely in the subtyping rules also avoids the need for running the subtyping algorithm twice as in Cardelli [1993]'s algorithm for Amber rules.

Tracking polarity is not enough: negative variables meet $\top$ types. The rules QS-REC-2A and QS-REC-2B, however, still miss some subtyping relations that are valid in the Amber rules. Consider $\mu\alpha.\ \top \to \alpha \leq \mu\alpha.\ \alpha \to \alpha$, which is a valid subtyping statement. This statement is not covered by the rules we have proposed so far, as the recursive variable α appears negatively. In a richer subtyping relation there might be more types other than $\top$ to be considered. For example, in the presence of bottom types $\bot$, it is also possible to have $\mu\alpha.\alpha \to \alpha <: \mu\alpha.\bot \to \alpha$. With intersection types, denoted as $A \,\&\, B$ for intersecting A and B, we may also have $\mu\alpha.(\top \& \top) \to \alpha <: \mu\alpha.\alpha \to \alpha$. To achieve a general solution to negative recursive subtyping, it is not effective to just identify those special cases and come up with ad-hoc rules to include them into the subtyping relation.

To have a solution that can scale up and be able to deal with other features that can make the situation even more complicated, we need to look deep into the reason why negative recursive subtyping prevents certain subtyping relations. A key observation in `QuickSub` is that: if the recursive variable appears at a negative position and *is compared with another recursive variable*, then the only supertype for its corresponding recursive type is itself and $\top$.

We illustrate this idea in Figure 1. We use superscripts to indicate the polarity of the recursive variables. For example, in comparing the recursive bodies of $\mu\alpha.\ \alpha \to \text{nat} \not\leq \mu\alpha.\ \alpha \to \top$, the recursive variable α is compared to itself at a negative position, so α^- will generate an equality constraint that the recursive type must be equivalent to itself. In contrast, in the second example, α^- is compared with $\top$ instead of another recursive variable, so it is allowed for the recursive type to be in a strict subtyping relation. We call the collection of such variables that are compared with another recursive variable negatively the *equality variable set*, since they essentially pose a constraint on the subtyping derivation that certain recursive types must be equivalent. By refining the NegVar function in rules QS-REC-2A and QS-REC-2B to collect only the equality variable sets, the negative recursive subtyping problem can be handled effectively, as we will detail in Section 3.

$\boxed{\Psi \vdash_{\oplus} A \lessapprox B}$ *(QuickSub Subtyping)*

$$\text{QS-NAT}\ \frac{}{\Psi \vdash_{\oplus} \mathsf{nat} \approx_{\emptyset} \mathsf{nat}} \qquad \text{QS-TOPEQ}\ \frac{}{\Psi \vdash_{\oplus} \top \approx_{\emptyset} \top} \qquad \text{QS-TOPLT}\ \frac{A \neq \top}{\Psi \vdash_{\oplus} A < \top}$$

$$\text{QS-VARPOS}\ \frac{\alpha^{\oplus} \in \Psi}{\Psi \vdash_{\oplus} \alpha \approx_{\emptyset} \alpha} \qquad \text{QS-VARNEG}\ \frac{\alpha^{\overline{\oplus}} \in \Psi}{\Psi \vdash_{\oplus} \alpha \approx_{\{\alpha\}} \alpha}$$

$$\text{QS-RECLT}\ \frac{\Psi, \alpha^{\oplus} \vdash_{\oplus} A_1 < A_2}{\Psi \vdash_{\oplus} \mu\alpha.\, A_1 < \mu\alpha.\, A_2} \qquad \text{QS-RECEQ}\ \frac{\Psi, \alpha^{\oplus} \vdash_{\oplus} A_1 \approx_S A_2 \qquad \alpha \notin S}{\Psi \vdash_{\oplus} \mu\alpha.\, A_1 \approx_S \mu\alpha.\, A_2}$$

$$\text{QS-RECEQIN}\ \frac{\Psi, \alpha^{\oplus} \vdash_{\oplus} A_1 \approx_S A_2 \qquad \alpha \in S}{\Psi \vdash_{\oplus} \mu\alpha.\, A_1 \approx_{((S \cup \mathrm{FV}(A_1)) \setminus \{\alpha\})} \mu\alpha.\, A_2} \qquad \text{QS-ARROW}\ \frac{\Psi \vdash_{\overline{\oplus}} A_2 \lessapprox_1 A_1 \qquad \Psi \vdash_{\oplus} B_1 \lessapprox_2 B_2}{\Psi \vdash_{\oplus} A_1 \to A_2\ (\lessapprox_1 \bullet \lessapprox_2)\ B_1 \to B_2}$$

$$\approx_{S_1} \bullet \approx_{S_2} \ = \ \approx_{S_1 \cup S_2} \qquad \approx_{\emptyset} \bullet < \ = \ <$$
$$< \bullet < \ = \ < \qquad < \bullet \approx_{\emptyset} \ = \ <$$

Fig. 2. The QuickSub subtyping rules.

3 Efficient Subtyping Algorithm

In this section, we work with a minimal set of types to explain the key idea of QuickSub. The syntax of types, as well as other constructs used in the subtyping algorithm, is:

Types	A, B	::=	$\mathsf{nat} \mid \top \mid A_1 \to A_2 \mid \alpha \mid \mu\alpha.\, A$
Subtyping results	$\lessapprox$	::=	$< \mid \approx_S$
Polarity modes	$\oplus$	::=	$+ \mid -$
Subtyping contexts	Ψ	::=	$\cdot \mid \Psi, \alpha^{\oplus}$
Equality variable sets	S	::=	$\emptyset \mid \{\alpha_1, \ldots, \alpha_n\}$

Meta-variables A, B range over types. Types include base types (nat), the top type ($\top$), function types ($A_1 \to A_2$), type variables (α), and recursive types ($\mu\alpha.\, A$). We will explain other constructs in the syntax when they are introduced in the rules.

3.1 QuickSub Subtyping

Figure 2 presents the QuickSub algorithm as syntax-directed inference rules. The subtyping judgement $\Psi \vdash_{\oplus} A \lessapprox B$ determines whether type A is a subtype of type B under the context Ψ, parameterized by a polarity mode $\oplus \in \{+, -\}$. The notation that we employ is similar to conventional subtyping. However, $\lessapprox$ is not simply a piece of notation, but instead it is a parameter of the relation. Moreover, when interpreted algorithmically, this parameter is an *output* result. The output result $\lessapprox \in \{<, \approx_S\}$ is a metavariable ranging over two possibilities: strict subtypes ($<$) and or equivalent types ($\approx_S$). The equivalent result also carries an *equality variable set* S, which tracks a set of variables used to check equality constraints during subtyping. Tracking the subtype results and the equality variable sets are key ingredients to the design of our efficient algorithm.

Subtyping base types. Base types nat and $\top$ are equal to themselves (rules QS-NAT and QS-TOPEQ), while types that are not equal to the $\top$ type are strict subtypes of $\top$ (rule QS-TOPLT). Since there are no common free variables in base types, in rules QS-NAT and QS-TOPEQ the equality variable set is empty, as indicated by the subscript $\approx_{\emptyset}$.

Tracking negative recursive variables. As briefly mentioned in Section 2.3, QuickSub uses the equality variable set to track those negative variables that pose an equality constraint on the

recursive types. To make QuickSub efficient, the equality variable set S is collected dynamically in the subtyping derivation. Since the equality variable sets are only used to represent equality constraints, when the subtyping result is strict subtyping, the algorithm can fail immediately, so they are only tracked as part of the equality result, which we denote as $\approx_S$. When the rules are interpreted as an algorithm, the equality variable set is also part of the output result.

Specifically, the equality variable set for two types in a subtyping relation is the set of those variables that are compared with themselves in a negative position of their recursive bodies, as spotted by the rule QS-VARNEG. When the variables are compared in a positive position, they are not included in the set, as indicated by rule QS-VARPOS. For example, in the first example of Figure 1, the variable α appears negatively in both $\mu\alpha.\ \alpha \to \mathsf{nat}$ and $\mu\alpha.\ \alpha \to \top$, so that rule QS-VARNEG is used and returns $\alpha^+ \vdash_- \alpha \approx_{\{\alpha\}} \alpha$, assuming the initial polarity mode is positive. In contrast, in the second example $\mu\alpha.\ \top \to \alpha < \mu\alpha.\ \alpha \to \alpha$, of Figure 1, the variable α is either compared with a top type or appears positively in the recursive type, so that no equality constraints are generated.

$$\alpha^+ \vdash_- \alpha < \top \quad \text{(by QS-TOPLT)} \quad \text{and} \quad \alpha^+ \vdash_+ \alpha \approx_\emptyset \alpha \quad \text{(by QS-VARPOS)}$$

Subtyping function types. The equality variable set becomes useful in rule QS-ARROW. We present the full definition of function $\bullet$ at the bottom of Figure 2. This definition extends the subtyping result composition function, informally introduced in Section 2.3, to include the equality variable set. The function is partial and for those cases where the composition is undefined, the subtyping checking will fail. Basically there are two cases when $\bullet$ returns a valid result: (1) When both subtyping statements are equality ($\approx$), the output mode is also equality, and the equality variable set is the union of the two sets; (2) When one of the subtyping statements returns a strict subtype result ($<$), the output result must also be strict, and it is required that the equality variable set for the other subtyping statement is empty. In other words, a non-empty equality variable set indicates that the types they are related to cannot be composed with strict subtyping counterparts, as we will show in the example below. When the subtyping result is strict, the equality variable set must be empty for the composition to be valid.

Continuing with the example $\mu\alpha.\ \alpha \to \mathsf{nat} \not\leq \mu\alpha.\ \alpha \to \top$ in Figure 1, we wish to detect at an early stage that the negative variable α makes subtyping fail. Unlike using substitution (as in the nominal unfolding) or other techniques, QuickSub traverses the types *only once* without substitutions. The two subtyping premises in the rule QS-ARROW are

$$\alpha^+ \vdash_- \alpha \approx_{\{\alpha\}} \alpha \qquad\qquad \alpha^+ \vdash_+ \mathsf{nat} < \top \tag{1}$$

However, the composition of the two subtyping statements $\approx_{\{\alpha\}} \bullet <$ is undefined, which makes the subtyping statement fail. This indicates that the negative variable α requires an equality constraint on all the types it appears in, which contradicts the strict subtyping result given by the second premise. We will reason about the correctness of the $\bullet$ function formally in Section 3.2.

Subtyping recursive types. The rules QS-RECLT, QS-RECEQ, and QS-RECEQIN are the most interesting rules in the algorithm. All three rules start by comparing the recursive body of the two types in the current context extended by a binding that maps the current recursive variable to the current polarity mode, as can be seen from the first premise. Based on the result of subtyping the two recursive type bodies, different rules are chosen. If the subtyping result is strict, the subtyping result in the conclusion is also strict (rule QS-RECLT). If the subtyping result is equivalent and the recursive variable α is not in the equality variable set, the subtyping result in the conclusion is also equivalent. Moreover the equality variable set is propagated through (rule QS-RECEQ). In this case, there are no constraints on the recursive variable. The presence of the recursive variable in the set, indicates that there is an equality constraint on the variable. Thus, the recursive bodies of

the two types must be equivalent (rule QS-RECEQIN). In this case, the update of the variable set is more complicated. The free variables in the recursive body are first included, and then the current recursive binder variable is excluded since it is no longer a free variable in the recursive type. The reason why the free variables $\mathsf{FV}(A_1)$ are included is related to nested recursive types. To illustrate this, recall the example for nested recursive types:

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

Variable β causes the subtyping statement fail, since it can appear negatively after unfolding the inner recursive type $\mu\alpha.\ \alpha \to \beta$ twice. This can be seen from the following derivation in `QuickSub`:

$$\dfrac{\dfrac{\text{QS-TOPLT}\ \ \beta^+ \vdash_- \mathsf{nat} < \top \qquad \text{QS-ARROW}\ \dfrac{\text{QS-VARNEG}\ \ \beta^+, \alpha^+ \vdash_- \alpha \approx_{\{\alpha\}} \alpha \qquad \text{QS-VARPOS}\ \ \beta^+, \alpha^+ \vdash_+ \beta \approx_{\emptyset} \beta}{\text{QS-RECEQIN}\ \dfrac{\beta^+, \alpha^+ \vdash_+ \alpha \to \beta \approx_{\{\alpha\}} \alpha \to \beta \qquad \alpha \in \{\alpha\}}{\beta^+ \vdash_+ \mu\alpha.\ \alpha \to \beta \approx_{\{\beta\}} \mu\alpha.\ \alpha \to \beta}}}{\beta^+ \vdash_+ \top \to (\mu\alpha.\ \alpha \to \beta) \not\leq \mathsf{nat} \to (\mu\alpha.\ \alpha \to \beta)}\ \text{QS-ARROW}\ \text{✗}}{\cdot \vdash_+ \mu\beta.\ \top \to (\mu\alpha.\ \alpha \to \beta) \not\leq \mu\beta.\ \mathsf{nat} \to (\mu\alpha.\ \alpha \to \beta)}\ \text{QS-REC...}$$

As we highlight in the derivation, the subtyping judgement fails due to the undefined composition $< \bullet \approx_{\{\beta\}}$ in the rule QS-ARROW rule. The failure can be further traced back to the $\approx_{\{\beta\}}$ result after applying the rule QS-RECEQIN rule. Note that the equality variable set $\{\beta\}$ is computed from $(\{\alpha\} \cup \mathsf{FV}(\alpha \to \beta))\backslash\{\alpha\}$, where $\mathsf{FV}(\alpha \to \beta) = \{\alpha, \beta\}$. Without the union of $\mathsf{FV}(A_1)$ in rule QS-RECEQIN, the subtyping derivation above would have succeeded since the inner recursive type $\mu\alpha.\ \alpha \to \beta$ would return $\approx_{\emptyset}$ and the composition in rule QS-ARROW would be valid.

On the other hand, $\mu\alpha.\ \alpha \to \beta$ is equivalent to itself, so it is not feasible to reject this subtyping statement at a stage when only the $\mu\alpha.\ \alpha \to \beta$ part is compared. The subtyping will fail only when $\mu\alpha.\ \alpha \to \beta$ is composed with other strict subtypes, which is the case above. Nonetheless, we still expect the following equivalent subtyping statement to succeed:

$$\dfrac{\dfrac{\text{QS-TOPEQ}\ \ \beta^+ \vdash_- \top_{\emptyset} \approx \top \qquad \dfrac{\cdots}{\beta^+ \vdash_+ \mu\alpha.\ \alpha \to \beta \approx_{\{\beta\}} \mu\alpha.\ \alpha \to \beta} \qquad \approx_{\emptyset} \bullet \approx_{\{\beta\}} = \approx_{\{\beta\}}}{\text{QS-RECEQIN}\ \dfrac{\beta^+ \vdash_+ \top \to (\mu\alpha.\ \alpha \to \beta) \approx_{\{\beta\}} \top \to (\mu\alpha.\ \alpha \to \beta) \qquad \beta \in \{\beta\}}{\cdot \vdash_+ \mu\beta.\ \top \to (\mu\alpha.\ \alpha \to \beta) \approx_{\emptyset} \mu\beta.\ \top \to (\mu\alpha.\ \alpha \to \beta)}}\ \text{QS-ARROW}$$

Therefore, we still allow equivalent recursive types with negative variables to be subtypes of each other, as described in rule QS-RECEQIN. However, we refine the equality variable set so that the variables in negative positions can be precisely tracked, and leave the rejection of strict subtyping to the composition stage in rule QS-ARROW. Overall, the interaction between the equality variable set and the subtyping composition function $\bullet$ leads to an efficient algorithm that can handle all iso-recursive subtyping cases effectively.

Analysis of the algorithm performance. For the rules in Figure 2 to be an algorithm, the context Ψ, types A, B and polarity mode $\oplus$ are considered as inputs, and the subtyping result $\lessapprox$ is the output of the algorithm. Since the inference rules only describe the successful cases, the algorithm is defined as a partial function and returns a false result indicating non-subtypes when no rules are applicable. It can be verified that the rules are *structurally recursive* on the types, and that the size of the inputs will decrease strictly on every recursive call. As a result, the problem of exponential blowup as seen in nominal unfolding rules does not exist. Moreover, the choice of the rules is deterministic.

For example, the choice of rules QS-RECLT, QS-RECEQ, and QS-RECEQIN is decided by the result of subtyping for the recursive bodies. Because of the side conditions checking whether the current recursive binder variable is in the equality variable set, there are no performance costs due to backtracking between different inference rules as seen in the Amber rules.

Despite these optimizations, there is still a performance bottleneck in rule QS-RECEQIN of the algorithm when it is repeatedly called on nested recursive types. As we described earlier, the equality variable set S is updated by including the free variables of the recursive body and excluding the current recursive binder variable. Computing the free variables and the set union operation can be costly in the worst case. However, this can be optimized by collecting the free variables on the fly and using efficient data structures for set operations. As we will see in Section 5, by using imperative boolean arrays to represent the equality variable set in the `QuickSub` implementation, the time complexity of set union operation is linear with respect to the number of variables in the set, which in the worst case is the total number of recursive variables n in the type. Therefore, assuming the size of the type is m, the worst case complexity of the algorithm is $O(mn)$, which remains the same as the complexity of Ligatti et al.'s ML implementation for complete iso-recursive subtyping.

Compared to the nominal rules or the Amber rules, our `QuickSub` algorithm is more efficient in terms of complexity. The nominal rules have exponential complexity due to the use of substitution for unfolding. Similarly, the Amber rules, also have an exponential complexity on the depth of recursive variables due to a non-deterministic choice of rules for subtyping recursive types, and possibly extra overhead of variable renaming. Another point to note is that the $O(mn)$ complexity is worst case complexity. In practice it is not common to have a large number of negative appearing variables in a type, since these types easily tend make the subtyping judgement fail as we have shown in the examples above. As we will show in Section 5, for practical subtype checking tasks, our algorithm is already efficient to use and scales well to large recursive types, outperforming the other formulations of recursive subtyping.

3.2 Correctness

To prove the correctness of the algorithm we show that `QuickSub` is equivalent to the Amber rules for iso-recursive subtyping and other equivalent variants defined in previous work. Among these variants we find the weakly positive subtyping rules [Zhou et al. 2022b] most convenient to develop an equivalence proof with `QuickSub`. The weakly positive subtyping rules were originally proposed as an intermediate step to prove the equivalence between the nominal unfolding rules and the Amber rules. Zhou et al. [2022b] have shown that all the three sets of rules are equivalent to each other. Therefore, by proving the equivalence between `QuickSub` and the weakly positive subtyping rules, we can establish the equivalence between `QuickSub` and *all* of these variants.

Weakly positive subtyping rules. We present the weakly positive subtyping rules by Zhou et al. [2022b] in Figure 3. The rules have two components: weakly positive restriction $\alpha \in_{\oplus} A \leq B$ and weakly positive subtyping $\Gamma \vdash_p A \leq B$.

The subtyping part is mostly structural. Rule POSRES-REC compares two recursive types by structurally checking the recursive body, but requires that the recursive variable α satisfies the weakly positive restriction $\alpha \in_+ A \leq B$. As we shall explain later, the weakly positive restriction considers more carefully the polarity of the type variable α than standard formulations of positive subtyping [Amadio and Cardelli 1993]. For recursive types that satisfy such restriction, the subtyping can be checked simply by comparing the recursive bodies. There is also a built-in reflexivity rule POSRES-SELF for comparing negative equivalent recursive types, for a similar reason to that already discussed for the Amber rules.

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

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

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

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

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

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

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

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

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

Fig. 3. Weakly Positive Subtyping Rules.

The weakly positive restriction describes whether a type variable α occurs in a derivation of $A \leq B$ with a specific polarity $\oplus$ (either positive or negative). This relation ensures that given a variable α in question, every instance of $\alpha \leq \alpha$ in the derivation of $A \leq B$ only occurs in a positive position (rule Posvar-varx), so that invalid contravariant subderivations are prevented. For recursive types, the weakly positive restriction is satisfied in two cases: either the recursive bodies are equal and α is not free in $\mu\beta.\, A$ (rule Posvar-recself), or the recursive variable β is found in a weakly positive position inside the proof and α satisfies the weakly positive restriction in the recursive bodies (rule Posvar-rec).

To see why rule Posvar-rec needs the second condition, consider the following example:

$$\beta \in_{+} \mu\alpha.\, \alpha \to \beta \leq \mu\alpha.\, \alpha \to \beta$$

which will be rejected by both rules Posvar-recself and Posvar-rec, since $\beta \in \mathsf{FV}(\alpha \to \beta)$ and $\alpha \in_{+} \alpha \to \beta \leq \alpha \to \beta$ is not derivable. Otherwise, unfolding the recursive type would lead to

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

with a negative subderivation $\beta \in_{-} \beta \leq \beta$, which leads to an inconsistency in the rules. Overall, the weakly positive subtyping restriction, as well as the subtyping rules ensures that iso-recursive subtyping is correctly handled. For a detailed discussion on the weakly positive subtyping rules, we refer the readers to Zhou et al.'s work.

Relating `QuickSub` *to the weakly positive restriction.* The weakly positive restriction provides a way to formally reason about the equality variable sets in `QuickSub`. Indeed, we can show that the `QuickSub` context Ψ captures the weakly positive information of all the variables in the context.

Depending on whether the variable $\alpha^{\oplus'} \in \Psi$ has the same polarity as the polarity $\oplus$ of the current subtyping judgment, we can determine whether α appears weakly positively or negatively in the two types A and B, as described in the following theorems.

Theorem 3.1 (Relation to weakly positive restrictions (strict subtyping)). If $\Psi \vdash_\oplus A < B$, then
(1) $\alpha^{\oplus} \in \Psi$ implies $\alpha \in_+ A \leq B$
(2) $\alpha^{\overline{\oplus}} \in \Psi$ implies $\alpha \in_- A \leq B$

This also holds for the case of $\approx_S$, but only in the case where the variable α is not in the equality variable set S. When $\alpha \in S$, indicating there is a negative equality $\alpha \leq \alpha$ in the derivation of $A \approx B$ (see rule QS-VARNEG), the weakly positive restriction no longer holds. Note that we use $\notin_\oplus$ to denote the logical negation of $\in_\oplus$, and that $\notin_+$ is not equivalent to $\in_-$.

Theorem 3.2 (Relation to weakly positive restrictions (equivalence)). If $\Psi \vdash_\oplus A \approx_S B$, then
(1) $\alpha^{\oplus} \in \Psi$ and $\alpha \notin S$ implies $\alpha \in_+ A \leq B$
(2) $\alpha^{\overline{\oplus}} \in \Psi$ and $\alpha \notin S$ implies $\alpha \in_- A \leq B$
(3) $\alpha^{\oplus} \in \Psi$ and $\alpha \in S$ implies $\alpha \notin_+ A \leq B$
(4) $\alpha^{\overline{\oplus}} \in \Psi$ and $\alpha \in S$ implies $\alpha \notin_- A \leq B$

For the sake of completeness we also prove a theorem for the case when α is a fresh variable that does not appear in Ψ. In this case, it is guaranteed that α appears weakly positively and negatively in the subtyping judgment $A \leq B$.

Theorem 3.3 (Relation to weakly positive restrictions (fresh variables)). If $\Psi \vdash_\oplus A \lessapprox B$, then for any $\alpha \notin \mathrm{dom}\ \Psi$ and any polarity $\oplus'$, $\alpha \in_{\oplus'} A \leq B$.

Soundness of `QuickSub` *to weakly positive subtyping.* We can now show that the `QuickSub` rules are sound with respect to the weakly positive subtyping rules.

Theorem 3.4 (Soundness of `QuickSub` to Weakly Positive Subtyping). If $\Psi \vdash_\oplus A \lessapprox B$, then $|\Psi| \vdash_p A \leq B$, where $|\Psi|$ removes all the polarity annotations from the context Ψ.

The proof is straightforward by induction on the derivation of $\Psi \vdash_\oplus A \lessapprox B$. Most of the cases are direct by induction hypotheses, except for the recursive type cases $\Psi \vdash_\oplus \mu\alpha.\, A \lessapprox \mu\alpha.\, B$. For case QS-RECLT and QS-RECEQ, by Theorem 3.1 (1) and Theorem 3.2 (1), we know that $\alpha \in_+ A \leq B$, so that rule POSRES-REC can be applied. For case QS-RECEQIN, by Theorem 3.5 the two recursive types are equal, so that we can apply the rule POSRES-SELF to complete the proof. The proof of Theorem 3.5 is straightforward by induction on the subtyping derivation.

Theorem 3.5 (`QuickSub` equivalence implies equality). If $\Psi \vdash_\oplus A \approx_S B$, then $A = B$.

Completeness of `QuickSub` *to weakly positive subtyping.* We wish to prove the following theorem to show that `QuickSub` is complete with respect to the weakly positive subtyping rules:

Theorem 3.6 (Completeness of `QuickSub`). If $\cdot \vdash_p A \leq B$, then there exists $\lessapprox$, such that $\cdot \vdash_+ A \lessapprox B$.

The proof of Theorem 3.6 is not straightforward by induction, as the empty context $\cdot$ needs to be generalized to handle recursive cases. However, the context cannot be generalized arbitrarily. For example, consider the weakly positive subtyping judgment $\alpha \vdash_p \alpha \to \mathsf{nat} \leq \alpha \to \top$. Annotating α with positive polarity leads to an invalid judgment in `QuickSub`:

$$\frac{\alpha^+ \vdash_- \alpha \approx_{\{\alpha\}} \alpha \qquad \alpha^+ \vdash_+ \mathsf{nat} < \top}{\alpha^+ \vdash_+ \alpha \to \mathsf{nat} \not< \alpha \to \top}\ \text{QS-ARROW}\ \text{✗}$$

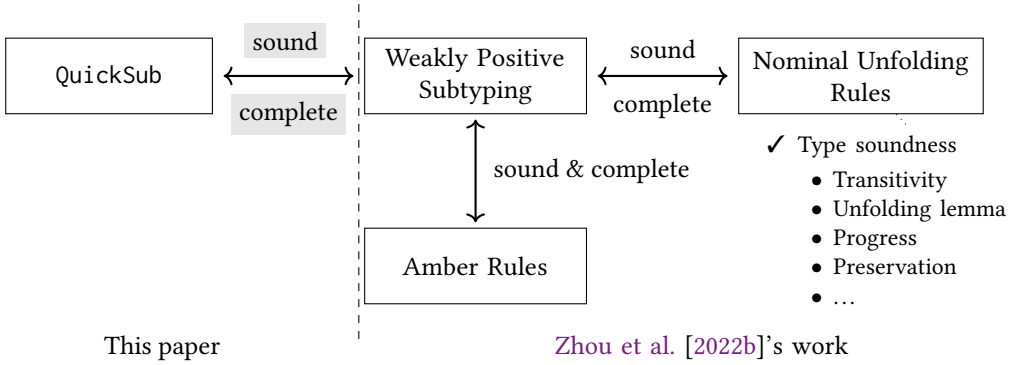


Fig. 4. Type soundness proof framework for QuickSub, indirectly

This suggests that in the generalized lemma we need to specify how the weakly positive subtyping context Γ, that only contains variables, is related to the QuickSub context Ψ in the conclusion, which also assigns polarities to variables. To this end we define a restriction *well bound context* to ensure that all the variables in the context Ψ are consistent with the polarity of the subtyping judgment up to the weakly positive restriction, so that the context can be safely generalized.

Definition 3.7 (Well bound context). A context Ψ is well bound with respect to a polarity $\oplus$ and two types A and B, denoted as $\Psi \in_{\oplus} A \leq B$, if for any $\alpha^{\oplus'} \in \Psi$, $\alpha \in_{\oplus' + \oplus} A \leq B$, where $\oplus' + \oplus$ is $+$ if $\oplus'$ and $\oplus$ are the same, and $-$ otherwise.

The generalized lemma is then stated as follows:

Lemma 3.8 (Generalized completeness of QuickSub). If $\Gamma \vdash_p A \leq B$, and there exists Ψ such that dom Γ = dom Ψ, and $\Psi \in_{\oplus} A \leq B$, then there exists $\lessapprox$, such that $\Psi \vdash_+ A \lessapprox B$.

The proof of the generalized lemma is by induction on the derivation of $\Gamma \vdash_p A \leq B$. Most of the cases follow the induction hypotheses directly. In the cases for recursive types we need to show that the extended context $(\Psi, \alpha^{\oplus})$ is well bound, which is guaranteed by the weakly positive conditions in the weakly positive subtyping premise. The most interesting case is the PosRes-arrow case, in which we get two QuickSub results $B_1 \lessapprox_1 A_1$ and $B_2 \lessapprox_2 A_2$ from the induction hypothesis. We need to show that $\lessapprox_1 \bullet \lessapprox_2$ does not cause trouble so that rule QS-arrow can be applied. Thanks to the well bound context restriction, those cases where $\lessapprox_1 \bullet \lessapprox_2$ are undefined can be ruled out by contradiction. We refer the reader to the Coq formalization for the detailed proof.

3.3 Direct Proof of Type Soundness

We have shown that QuickSub is equivalent to the weakly positive subtyping rules in the previous section. Since Zhou et al. [2022b] have proven that the weakly positive subtyping rules are sound and complete to the nominal unfolding rules, and that the nominal unfolding rules are type sound, we can conclude that QuickSub is also type sound. This proof is illustrated in Figure 4, which we refer to as an *indirect* proof of type soundness, as the proof relies on the development of other equivalent subtyping rules. In this section we provide a *direct* proof of type soundness for QuickSub.

Why a direct proof? While the indirect proof shown in Figure 4 has effectively established type soundness by linking QuickSub to established rules, there are strong reasons to also study a direct proof for QuickSub. Firstly, proving type soundness via equivalence to other rules involves significant additional complexity. This can be observed by the fact that useful properties of the

Amber rules that contribute to type soundness, such as transitivity of subtyping and unfolding lemma, which we will state shortly, are proven indirectly by showing their equivalence to the nominal unfolding rules [Zhou et al. 2022b]. Whether there exists a direct proof for other equivalent rules is not clear. It is also non-trivial to prove the equivalence between various iso-recursive subtyping rules without going through an intermediate equivalent system that is carefully defined for the purpose of proving equivalence, such as the weakly positive subtyping rules. Each of the six soundness or completeness proofs in Figure 4 requires significant reasoning and many auxiliary lemmas. Therefore, although only two soundness and completeness proofs highlighted in gray in Figure 4 are needed to prove type soundness for `QuickSub`, the overall proof framework still has a high level of complexity.

Moreover, having a direct proof makes `QuickSub` easier to extend in the future with new features. If all we cared were the features that are already present in Zhou et al. [2022b]'s work, then an indirect proof framework like Figure 4 would be sufficient. However, most of the time one would like to study iso-recursive subtyping with other features, such as records, polymorphism, bounded quantification [Zhou et al. 2023], and intersection [Zhou et al. 2022a] or union types. In this case, if one wants to add new features to `QuickSub` using an indirect proof, it would require modifying the underlying systems, proving type soundness within those systems, and updating the equivalence proofs. As there are many complexities involved in all these steps, this would be a large burden. In contrast, with a direct type soundness proof, each new feature can be added and assessed for type soundness in isolation, simplifying the proof development process considerably.

A calculus with iso-recursive types and `QuickSub` *subtyping.* We develop a simple calculus to show how a direct proof of type soundness can be developed for `QuickSub`. The syntax, typing rules and reduction rules are presented in Figure 5. Meta-variables e range over expressions, with conventional syntax for variables (x), integers (n), applications ($e_1\ e_2$), lambda abstractions ($\lambda x : A.\ e$), unfolding ($\mathsf{unfold}\ [A]\ e$) and folding ($\mathsf{fold}\ [A]\ e$) expressions for iso-recursive types.

The typing rules are standard. For base cases $\vdash \Gamma$ checks the well-formedness of the context Γ, which requires that all the variable types in the context are well-formed. A fold expression constructs a recursive type (rule Typing-fold), while an unfold expression opens a recursive type to its unfolding (rule Typing-unfold). One specialized rule is the subsumption rule Typing-sub, which calls `QuickSub` with an empty context $\cdot$ and positive polarity $+$ to check the subtyping relation between types A and B. Both strict subtyping and equivalence are accepted in the subsumption rule.

The reduction rules for the calculus are also standard. In rules step-fold and step-unfold the expressions inside the fold and unfold constructs are reduced until they reach a value. The fold of values are treated as values, while for unfold expressions with fold values, the rule step-fld is applied to eliminate the pair of annotations.

Unfolding lemma. We prove the type soundness of the calculus using progress and preservation. As identified by previous works on iso-recursive subtyping [Ligatti et al. 2017; Zhou et al. 2022b] the key to the type soundness proof is the *unfolding lemma*:

Lemma 3.9 (Unfolding lemma). If $\cdot \vdash_+ \mu\alpha.\ A \lessapprox \mu\alpha.\ B$, then $\cdot \vdash_+ A[\alpha \mapsto \mu\alpha.\ A] \lessapprox B[\alpha \mapsto \mu\alpha.\ B]$

The unfolding lemma ensures the preservation property for iso-recursive typing in the presence of subtyping. To illustrate this, consider an expression $\mathsf{unfold}\ [\mu\alpha.\ B]\ (\mathsf{fold}\ [\mu\alpha.\ A]\ v)$. The derivation below shows the typing of this expression.

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & x \mid \mathsf{n} \mid e_1\ e_2 \mid \lambda x : A.\ e \mid \mathsf{unfold}\ [A]\ e \mid \mathsf{fold}\ [A]\ e \\
\text{Values} & v & ::= & \mathsf{n} \mid \lambda x : A.\ e \mid \mathsf{fold}\ [A]\ v \\
\text{Typing Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

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

$$\textsc{typing-nat}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{i} : \mathsf{nat}} \qquad \textsc{typing-var}\ \frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{typing-app}\ \frac{\Gamma \vdash e_1 : A_1 \to A_2 \qquad \Gamma \vdash e_2 : A_1}{\Gamma \vdash e_1\ e_2 : A_2}$$

$$\textsc{typing-abs}\ \frac{\Gamma, x : A_1 \vdash e : A_2}{\Gamma \vdash \lambda x : A_1.\ e : A_1 \to A_2} \qquad \textsc{typing-fold}\ \frac{\Gamma \vdash e : [\alpha \mapsto \mu\alpha.\ A]\ A}{\Gamma \vdash \mathsf{fold}\ [\mu\alpha.\ A]\ e : \mu\alpha.\ A}$$

$$\textsc{typing-unfold}\ \frac{\Gamma \vdash e : \mu\alpha.\ A}{\Gamma \vdash \mathsf{unfold}\ [\mu\alpha.\ A]\ e : [\alpha \mapsto \mu\alpha.\ A]\ A} \qquad \textsc{typing-sub}\ \frac{\Gamma \vdash e : A \qquad \cdot \vdash_+ A \lessapprox B}{\Gamma \vdash e : B}$$

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

$$\textsc{Typing-unfold}\ \frac{\textsc{Typing-sub}\ \dfrac{\textsc{Typing-fold}\ \dfrac{\cdot \vdash v : A[\alpha \mapsto \mu\alpha.\ A] \qquad \ldots}{\cdot \vdash \mathsf{fold}\ [\mu\alpha.\ A]\ v : \mu\alpha.\ A} \qquad \cdot \vdash_+ \mu\alpha.\ A \lessapprox \mu\alpha.\ B}{\cdot \vdash \mathsf{fold}\ [\mu\alpha.\ A]\ v : \mu\alpha.\ B} \qquad \ldots}{\cdot \vdash \mathsf{unfold}\ [\mu\alpha.\ B]\ (\mathsf{fold}\ [\mu\alpha.\ A]\ v) : B[\alpha \mapsto \mu\alpha.\ B]}$$

By inversion we know that after reduction using rule STEP-FLD, the result v has the type $A[\alpha \mapsto \mu\alpha.\ A]$, and that $\mu\alpha.\ A \lessapprox \mu\alpha.\ B$. To prove preservation that v has the type $B[\alpha \mapsto \mu\alpha.\ B]$ after reduction, the unfolding lemma is necessary.

The unfolding lemma is proved as a corollary of the generalized lemma below:

Lemma 3.10 (Generalized unfolding lemma). If $\Psi_2 \vdash_\oplus C \lessapprox_1 D$, then

(1) $\Psi_1, \alpha^\oplus, \Psi_2 \vdash_\oplus A \lessapprox_2 B$ and $\Psi_1, \alpha^\oplus, \Psi_2 \in_\oplus A \le B$ implies that there exists $\lessapprox'$, such that $\Psi_1, \Psi_2 \vdash_\oplus A[\alpha \mapsto C] \lessapprox' B[\alpha \mapsto D]$.
(2) $\Psi_1, \alpha^{\overline{\oplus}}, \Psi_2 \vdash_\oplus A \lessapprox_2 B$ and $\Psi_1, \alpha^{\overline{\oplus}}, \Psi_2 \in_\oplus A \le B$ implies that there exists $\lessapprox'$, such that $\Psi_1, \Psi_2 \vdash_\oplus A[\alpha \mapsto D] \lessapprox' B[\alpha \mapsto C]$.

The generalized unfolding lemma basically states that one can substitute variable α in the types $A \lessapprox_2 B$ in a subtyping judgment with a pair of types $C \lessapprox_1 D$ that are also related by subtyping. When the polarity of α is consistent with the polarity of the subtyping judgment, the substitution can be done covariantly, and contravariantly otherwise, as expressed by the two sub-cases in the lemma. The two sub-cases need to be proved simultaneously, as in rule QS-ARROW the substitution direction may be swapped in the contravariant condition. For a similar reason as in the proof of

the generalized completeness lemma (Lemma 3.8), we need to generalize the unfolding lemma to a well bound context, so that problematic cases like the undefined composition $\lessapprox_1 \bullet \lessapprox_2$ in case QS-Arrow are avoided. Note that for defining the well bound context condition, only the weakly positive restriction (the upper part of Figure 3) is needed, not the full weakly positive subtyping rules. With Lemma 3.10 there is no need to go through QuickSub's soundness or completeness to the weakly positive subtyping rules as we did in Section 3.2 for type soundness. For the detailed proof of the generalized unfolding lemma, we refer the readers to the Coq mechanization.

Next, we show how Lemma 3.10 is used to prove the unfolding lemma. There are two cases for the result of $\cdot \vdash_+ \mu\alpha.\, A \lessapprox \mu\alpha.\, B$. When $\lessapprox = <$, by inversion we know $\alpha^+ \vdash_+ A < B$. By Theorem 3.1 we know that $\alpha \in_+ A \le B$ so that the well bound context condition is satisfied. By setting $\Psi_1 = \cdot$, $\Psi_2 = \cdot$, $\oplus = +$, $C = \mu\alpha.\, A$, $D = \mu\alpha.\, B$, $\lessapprox_1 = \lessapprox_2 = <$, in Lemma 3.10, we can derive the unfolding lemma. When $\lessapprox = \approx$, we directly know that $A = B$, and the unfolding lemma holds by reflexivity:

Theorem 3.11 (Reflexivity). For any closed type A, there exists S, such that $\cdot \vdash_+ A \approx_S A$.

Transitivity. To prove type safety directly it is also required that the subtyping relation must be transitive. For algorithmic subtyping, transitivity is not a built-in inference rule and therefore needs to be proved as a theorem. The proof of transitivity for the Amber rules used to be a challenging task and required intricate proof techniques [Bengtson et al. 2011]. Zhou et al. [2022b] showed that with the nominal unfolding rules, they can have an easy proof of transitivity. We also prove the transitivity theorem for QuickSub directly.

Theorem 3.12 (Transitivity). If $\Psi \vdash_\oplus A \lessapprox_1 B$ and $\Psi \vdash_\oplus B \lessapprox_2 C$, then $\Psi \vdash_\oplus A\ (\lessapprox_1 \circ \lessapprox_2)\ C$, where

$$< \circ < = <, \quad < \circ \approx_S = <, \quad \approx_S \circ < = <, \quad \approx_{S_1} \circ \approx_{S_2} = \approx_{S_1 \cup S_2}$$

The sequential composition operator $\circ$ can be regarded as a total function version of the composition operator $\bullet$ that does not rule out the case where $<$ is composed with $\approx_S$ where $S \neq \emptyset$. The transitivity theorem is proved by induction on the intermediate type B and then case analysis on the two subtyping relations in the conventional way.

Type Soundness. With the unfolding lemma and the transitivity theorem for QuickSub, the type soundness of the calculus in Figure 5 can be proved in a standard way.

Theorem 3.13 (Progress). For any expression e and type A, if $\cdot \vdash e : A$ then either e is a value or there exists an expression e' such that $e \hookrightarrow e'$.

Theorem 3.14 (Preservation). For any expression e and type A, if $\cdot \vdash e : A$ and $e \hookrightarrow e'$ then $\cdot \vdash e' : A$.

4 Extension to Records

For practical uses of QuickSub, it is useful to extend the subtyping rules to features that are common in programming languages. In this section we show how QuickSub can be extended to handle record types with iso-recursive subtyping. We present an extension of QuickSub, which we call QuickSub$^{\{\}}$ and show its use in a calculus with records, iso-recursive types and subtyping.

4.1 Record Subtyping

We start by revisiting the standard subtyping rules for record types [Pierce 2002], which can be written as:

$$\frac{\{l_i\ ^{i\in 1\cdots n}\} \subseteq \{k_j\ ^{j\in 1\cdots m}\} \qquad k_j = l_i \text{ implies } A_j \le B_i}{\{k_j : A_j\ ^{j\in 1\cdots m}\} \le \{l_i : B_i\ ^{i\in 1\cdots n}\}}\text{S-RCD}$$

where $\{l_i : A_i\ ^{i\in 1\cdots m}\}$ and $\{k_j : B_j\ ^{j\in 1\cdots n}\}$ are two record types with m and n fields respectively. Two record types are in a subtyping relation when the fields of the right type are a subset of the left type, and the types of the corresponding fields are subtypes.

$\boxed{\Psi \vdash_{\oplus} A \lessapprox B}$ *(*QuickSub *Subtyping for Records)*

$$\text{QS-RCD}\quad \frac{\{l_i\ ^{i\in 1\cdots n}\} \subseteq \{k_j\ ^{j\in 1\cdots m}\} \qquad \forall l_i\ ^{i\in 1\cdots n}\ \exists k_j\ ^{j\in 1\cdots m}\ k_j = l_i \wedge \Psi \vdash_{\oplus} A_j \lessapprox_i B_i}{\Psi \vdash_{\oplus} \{k_j : A_j\ ^{j\in 1\cdots m}\} \lessapprox' \{l_i : B_i\ ^{i\in 1\cdots n}\}}$$

$$\text{where}\ \lessapprox' = \begin{cases} \lessapprox_1 \bullet \ldots \bullet \lessapprox_n, & \text{if } \{l_i\ ^{i\in 1\cdots n}\} = \{k_j\ ^{j\in 1\cdots m}\} \\ \lessapprox_1 \bullet \ldots \bullet \lessapprox_n \bullet <, & \text{if } \{l_i\ ^{i\in 1\cdots n}\} \subsetneq \{k_j\ ^{j\in 1\cdots m}\} \end{cases}$$

Fig. 6. The QuickSub subtyping rules for record types.

In many cases, this rule can be directly added to an existing set of inference rules for subtyping without modifications to the existing rules, which is the case for the nominal unfolding rules in Zhou et al. [2022b]. However, this does not work for other variants of iso-recursive subtyping rules, such as the weakly positive subtyping rules and the Amber rules, as well as the rules for the QuickSub algorithm. The addition of record types makes the subtyping relation non-antisymmetric.

Our solution. We follow our previous design and extend the subtyping rules to handle record types directly, as shown in Figure 6. Similarly to the standard approach to record subtyping (rule S-RCD), in rule QS-RCD, we also check whether the fields of the two record types are in a set inclusion relation, and then we compute the subtyping results of all the corresponding fields as $\lessapprox_i$. All the results are then combined with the $\bullet$ operator, which is the same composition operator in rule QS-ARROW. When interpreted algorithmically, the second premise of the rule can be implemented by traversing all the fields of $\{l_i : B_i\ ^{i\in 1\cdots m}\}$ and looking up the corresponding field in $\{k_j : A_j\ ^{j\in 1\cdots n}\}$. Assuming a proper data structure for record types, such as a hash table, or a sorted list, the cost of record traversal should be linear to the number of fields in the record type so that the overall complexity of QuickSub$^{\{\}}$ remains the same as QuickSub.

One key design here is that we also distinguish strict label inclusion from equivalent label sets. When the two record types have the same set of labels, all the results can be combined directly, so that when all the record fields have equivalent types, the two record types should be equivalent. This is not the case when there are labels that only appear in the left type, since they prevent the two record types from being equivalent. Therefore, the record field subtyping results should be folded to a strict subtyping result $<$.

Unlike QuickSub, there is no equivalence result for QuickSub$^{\{\}}$ with respect to weakly positive rules and the Amber rules. This is because the extension to non-antisymmetric relations for weakly positive rules is not studied in the literature. One way to bridge this gap is to have a direct equivalence proof for QuickSub$^{\{\}}$ with respect to nominal unfolding rules, which have been extended with record types [Zhou et al. 2022b]. We leave this as future work.

4.2 A Calculus with QuickSub$^{\{\}}$

Despite the lack of an equivalence result for QuickSub$^{\{\}}$ with respect to weakly positive rules and the Amber rules, we can still show that QuickSub$^{\{\}}$ is type sound, since the type soundness proof we have developed in Section 3.3 does not rely on other subtyping rules. We present a calculus with QuickSub$^{\{\}}$ in Figure 7, which extends the calculus in Figure 5 with expressions. The expression syntax is extended with record expressions and record projections, as indicated by the gray color, with standard typing and reduction rules.

We prove the unfolding lemma and transitivity theorem for QuickSub$^{\{\}}$ in the same way as we did for QuickSub. One technical detail in the proof of the unfolding lemma is that the generalized

$$\begin{array}{llll} \text{Expressions} & e & ::= & x \mid \mathsf{i} \mid e_1\ e_2 \mid \lambda x : A.\ e \mid \mathsf{unfold}\ [A]\ e \mid \mathsf{fold}\ [A]\ e \mid \{l_i = e_i\ ^{i\in 1\cdots n}\} \mid e.l \\ \text{Values} & v & ::= & \mathsf{i} \mid \lambda x : A.\ e \mid \mathsf{fold}\ [A]\ v \mid \{l_i = v_i\ ^{i\in 1\cdots n}\} \end{array}$$

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

$$\textsc{typing-proj}\ \frac{\Gamma \vdash e : \{l_i : A_i\ ^{i\in 1\cdots n}\}}{\Gamma \vdash e.l_i : A_i} \qquad \textsc{typing-rcd}\ \frac{\text{for each } i \quad \Gamma \vdash e_i : A_i}{\Gamma \vdash \{l_i = e_i\ ^{i\in 1\cdots n}\} : \{l_i : A_i\ ^{i\in 1\cdots n}\}}$$

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

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

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

Fig. 7. Extension to record expressions.

unfolding lemma requires a well bound condition, which makes use of the weakly positive restriction. To make the proof work for $\mathtt{QuickSub}^{\{\}}$, we extend the syntactic reflexivity of $\mu\beta.\ A$ in rule Posvar-recself to a more general equivalence relation $\mu\beta.\ A \approx \mu\beta.\ B$, considering record permutation. The two key lemmas for establishing type soundness, the unfolding lemma and transitivity, are:

Theorem 4.1 (Unfolding lemma for $\mathtt{QuickSub}^{\{\}}$).
If $\cdot \vdash_+ \mu\alpha.\ A \lessapprox \mu\alpha.\ B$, then $\cdot \vdash_+ A[\alpha \mapsto \mu\alpha.\ A] \lessapprox B[\alpha \mapsto \mu\alpha.\ B]$

Theorem 4.2 (Transitivity for $\mathtt{QuickSub}^{\{\}}$). If $\Psi \vdash_\oplus A \lessapprox_1 B$ and $\Psi \vdash_\oplus B \lessapprox_2 C$, then $\Psi \vdash_\oplus A\ (\lessapprox_1 \circ \lessapprox_2)\ C$.

With the unfolding lemma and transitivity theorem for $\mathtt{QuickSub}^{\{\}}$, we can show that the calculus in Figure 7 is type sound by standard progress and preservation lemmas.

5 Evaluation

We provide an OCaml implementation of `QuickSub`. The `QuickSub` rules in Figure 2 can be easily translated into a recursive function, as shown in Figure 8, by considering the subtyping results as outputs and the other components in the judgement $\Psi \vdash_\oplus A \lessapprox B$ as inputs. However, the set union and computation of the free variables in the last case of the function, if implemented naively, such as using the OCaml `Set` module, can be costly in some cases. Consider the following application of rule QS-recsqin:

$$\begin{array}{ll} & \mathsf{Sub}_{\alpha_1^+,\alpha_2^+,\ldots,\alpha_{n-1}^+}(\mu\alpha_n.\ \alpha_1 \to \ldots \to \alpha_n \to \mathsf{nat}, \mu\alpha_n.\ \alpha_1 \to \ldots \to \alpha_n \to \mathsf{nat}, +) = \approx_{\{\alpha_1,\alpha_2,\ldots,\alpha_{n-1}\}} \\ \text{since} & \mathsf{Sub}_{\alpha_1^+,\alpha_2^+,\ldots,\alpha_n^+}(\alpha_1 \to \ldots \to \alpha_n \to \mathsf{nat}, \alpha_1 \to \ldots \to \alpha_n \to \mathsf{nat}, +) = \approx_{\{\alpha_1,\alpha_2,\ldots,\alpha_n\}} \\ & \alpha_n \in \{\alpha_1, \ldots, \alpha_n\} \\ \text{so} & (\{\alpha_1, \alpha_2, \ldots, \alpha_n\} \cup \mathsf{FV}(\alpha_1 \to \ldots \to \alpha_n \to \mathsf{nat})) \setminus \{\alpha_n\} = \{\alpha_1, \alpha_2, \ldots, \alpha_{n-1}\} \end{array}$$

With functional `Set` operations, several meta-operations are required for this step: (1) determining the membership of α_n in the set takes $O(\log n)$ time, (2) computing the free variables of the type $\alpha_1 \to \ldots \to \alpha_n \to \mathsf{nat}$ takes $O(m)$ time, where m is the size of the type, (3) computing the union of the sets takes $O(n)$ time, and (4) removing α_n from the set takes $O(\log n)$ time.

$$
\begin{array}{lcll}
\mathsf{Sub}_\Psi(\mathsf{nat}, \mathsf{nat}, \oplus) & = & \approx_\emptyset & \\
\mathsf{Sub}_\Psi(\top, \top, \oplus) & = & \approx_\emptyset & \\
\mathsf{Sub}_\Psi(A, \top, \oplus) & = & < & (\text{if } A \neq \top) \\
\mathsf{Sub}_\Psi(\alpha, \alpha, \oplus) & = & \approx_\emptyset & (\text{if } \alpha^{\oplus} \in \Psi) \\
\mathsf{Sub}_\Psi(\alpha, \alpha, \oplus) & = & \approx_{\{\alpha\}} & (\text{if } \alpha^{\overline{\oplus}} \in \Psi) \\
\mathsf{Sub}_\Psi(A_1 \to A_2, B_1 \to B_2, \oplus) & = & \mathsf{Sub}_\Psi(A_2, A_1, \overline{\oplus}) \bullet \mathsf{Sub}_\Psi(B_1, B_2, \oplus) & \\
\mathsf{Sub}_\Psi(\mu\alpha.\, A_1, \mu\alpha.\, A_2, \oplus) & = & < & (\text{if } \mathsf{Sub}_{\Psi,\alpha^{\oplus}}(A_1, A_2, \oplus) = <) \\
\mathsf{Sub}_\Psi(\mu\alpha.\, A_1, \mu\alpha.\, A_2, \oplus) & = & \approx_S & (\text{if } \mathsf{Sub}_{\Psi,\alpha^{\oplus}}(A_1, A_2, \oplus) = \approx_S \text{ and } \alpha \notin S) \\
\mathsf{Sub}_\Psi(\mu\alpha.\, A_1, \mu\alpha.\, A_2, \oplus) & = & \approx_{(S \cup \mathsf{FV}(A_1)) \setminus \{\alpha\}} & (\text{if } \mathsf{Sub}_{\Psi,\alpha^{\oplus}}(A_1, A_2, \oplus) = \approx_S \text{ and } \alpha \in S) \\
\text{otherwise, } \mathsf{Sub}_\Psi(A, B, \oplus) \text{ fails} & & &
\end{array}
$$

Fig. 8. Functional implementation of `QuickSub`

To address this bottleneck, in our implementation we adopt a more efficient version SubImp of `QuickSub`, that carries two extra parameters, S_{ev} and S_{fv}, both of which are imperative boolean arrays that map all variables in the types to a boolean value. $S_{\text{ev}}[\alpha]$ indicates whether the variable α is in the equality variable set, while S_{fv} tracks the free variables on the fly. In the optimized subtyping function, for the QS-RECSQIN rule, $\mathsf{SubImp}_\Psi(\mu\alpha.\, A_1, \mu\alpha.\, A_2, \oplus, S_{\text{ev}}, S_{\text{fv}})$ now performs the following operations:

(1) Before making the recursive call on the inner body, set $S_{\text{fv}}[\alpha]$ to `true`, indicating that α is considered as a free variable in the inner body, which takes $O(1)$ time.
(2) After the recursive call, determine the membership of α in the set, which takes $O(1)$ time.
(3) Traverse S_{fv} and update S_{ev} for the union operation $S \cup \mathsf{FV}(A_1)$, which takes $O(n)$ time.
(4) Set $S_{\text{ev}}[\alpha]$ to `false` to discard α from the equality variable set, which takes $O(1)$ time.
(5) Update S_{fv} to remove α, which takes $O(1)$ time.

With this optimization, the only costly operation is the set union operation, which is $O(n)$, while the rest of the operations are $O(1)$, a large improvement over the functional implementation. Readers may refer to the OCaml code for the full implementation details.

In the rest of this section, we evaluate the imperative implementation of `QuickSub` on a set of benchmarks to evaluate its performance and scalability. For the sake of evaluation we add a new base type real and new type constructs, namely sum types ($A + B$) and product types ($A \times B$), to the implementation so that useful benchmarks can be generated. The subtyping rules are standard:

$$\frac{}{\mathsf{real} \leq \mathsf{real}}\ \text{S-REAL} \qquad \frac{}{\mathsf{nat} \leq \mathsf{real}}\ \text{S-NUM} \qquad \frac{A_1 \leq A_2 \quad B_1 \leq B_2}{A_1 + B_1 \leq A_2 + B_2}\ \text{S-SUM} \qquad \frac{A_1 \leq A_2 \quad B_1 \leq B_2}{A_1 \times B_1 \leq A_2 \times B_2}\ \text{S-PROD}$$

Note that whether real can be a supertype of nat remains debatable [Harper 2016], but we use it here for benchmarking purposes. The $\mathsf{nat} \leq \mathsf{real}$ subtyping is not integral to the main system, and can be replaced with dummy records (e.g., $\{l_{\text{Real}} : \mathsf{nat}, l_{\text{Nat}} : \mathsf{nat}\} \leq \{l_{\text{Real}} : \mathsf{nat}\}$) if needed.

We compare the performance of `QuickSub` with several algorithms in the literature, including:

- The slightly optimized implementation of the nominal unfolding rules for subtyping iso-recursive types by Zhou et al. [2022b], which avoids substitutions in positive positions:

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

  The polarized form of substitution $[\alpha \mapsto A]^{-}\, B$, only performs substitutions at negative occurrences of type variables. Zhou et al. [2022b, Section 2.7] discussed that special case of covariant subtyping, and mentioned that it would behave equivalently to rule SN-REC.

No.	QuickSub	Amber 1985	Complete 2017	Nominal 2022b	Equi 2002	Description	$\lvert S\rvert_{\max}$
1	**0.0045**	1.7230	2.0541	5.6194	42.0146	Disprove negative $<$	1
2	0.0079	**0.0004**	1.9483	6.3181	41.6360	Prove negative $\approx$ (simple)	1
3	**0.0085**	7.3775	3.7602	12.6697	Timeout	Prove positive $<$ (simple)	0
4	**0.0221**	5.7502	3.4782	91.0706	Timeout	Disprove positive $<$	0
5	0.0054	**0.0006**	3.8383	22.2383	Timeout	Prove positive $\approx$	0
6	**0.0038**	0.1829	1.2995	0.6027	Timeout	Prove mixed tests	1
7	**0.0082**	5.7185	3.5229	30.0276	Timeout	Prove positive $<$ (nested)	0
8	0.0817	**0.0057**	3.8423	Timeout	Timeout	Prove negative $\approx$ (worst)	500

Table 1. Time (seconds) taken for benchmarks with depth 5000 for (1) to (7) and 500 for (8). Timeout means taking more than 100 seconds or causing a stack overflow. $\lvert S\rvert_{\max}$ denotes the maximum size of the equality variable set during execution.

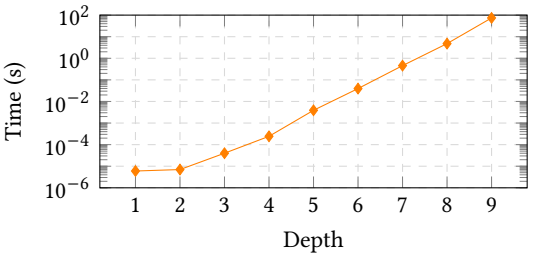
- The naive implementation of the Amber rules for subtyping iso-recursive types by Cardelli [1985]. When recursive types are encountered, the algorithm will first try rule Amber-self and then rule Amber-rec when the reflexivity check fails.
- The efficient implementation by Ligatti et al. [2017] for subtyping iso-recursive types, with the ability to relate between recursive types and unfoldings. We remove the $\bot$ type from the algorithm as it is not considered in other algorithms. The original implementation was written as an ML program, and we rewrite it in OCaml for comparison.
- The efficiently subtyping *equi-recursive types* algorithm described by Gapeyev et al. [2002, Figure 2]. The algorithm is considered efficient in that it only requires at most $O(n^2)$ recursive calls by remembering a seen set of recursive subtypes throughout the computation.

All these algorithms are implemented in OCaml using the same named representation of bindings for types as Ligatti et al.. For each test, we measure the execution time on a MacBook Pro with a 2 GHz Intel Core i5 processor and 16 GB of memory. The data are collected by averaging the execution time over 10 runs, excluding the maximum and minimum values, to reduce the impact of system noise.

5.1 Simple Recursive Types

We first test `QuickSub` without record types to evaluate the algorithm's asymptotic complexity and efficiency across different scenarios, including the worst case scenario (case 8). The benchmarks for testing the algorithm are sets of recursive types that follow certain patterns with different levels of recursive depth or type size. As our focus is on the performance of the algorithm, instead of relying on randomly generated types or property based testing tools such as QuickCheck [Claessen and Hughes 2000], we manually write 8 pattern generators for testing the algorithms. The patterns essentially generalize the various patterns discussed in Section 2.1. For details on how the benchmarks are generated, please refer to the appendix. To give a general idea of how each algorithm performs in different scenarios, we first evaluate `QuickSub` and other algorithms on a fixed large depth of recursive types for 8 different patterns, namely 500 for the worst case pattern and 5000 for others. As we will show later, the size of types in worst case pattern will grow quadratic with the depth, so we choose a smaller depth for the worst case pattern to avoid a stack overflow. The results are shown in Table 1. In each row, the algorithm that takes the least time is highlighted in bold.

The results in Table 1 show that `QuickSub` generally performs the best across most scenarios. Notably, it outperforms other algorithms in 5 out of the 8 cases, with the Amber rules performing



Depth	Width 100	1000	2000
1	0.02144	1.10701	4.73034
10	0.19642	34.11267	Timeout
100	53.19658	Timeout	Timeout
200	Timeout	Timeout	Timeout

Fig. 9. Equi-recursive subtyping algorithm evaluation. Left: Time (seconds measured in log scale) taken for the worst-case pattern as the recursive depth increases. Right: Time (seconds) taken for subtyping recursive record types with varying recursive depths and record widths.

better in those cases where equivalent subtyping is checked. This is not a surprise, as our Amber rule implementation always tries the reflexivity rule first. For these cases the performance of Amber rules can vary if the non-deterministic choice order between the rules for recursive types changes in the implementation. Nonetheless, in these cases `QuickSub` is still competitive with the Amber rules and outperforms others by a significant margin.

Substitution overhead. In equi-recursive subtyping and the nominal unfolding rules, the substitution operation becomes a significant overhead when the number of recursive variables increases. For equi-recursive subtyping, although Gapeyev et al. [2002] showed that the number of recursive calls is at most $O(n^2)$, the substitution operation is still frequently performed when recursive types are unfolded. So, the time complexity of the algorithm in practice, when measured by the number of meta-operations, can be significantly larger than $O(n^2)$. To illustrate this point, we conduct a small experiment on the worst case pattern (case 8) with depth ranging from 1 to 10, as shown in the left plot of Figure 9. The results show that as the depth increases, the time taken by the equi-recursive subtyping algorithm grows exponentially, and a depth of 10 recursive variables already leads to a timeout.

For the nominal unfolding rules, substitution leads to a timeout in many patterns. The complete algorithm implementation by Ligatti et al. [2017] avoids this with an unroll table, but it remains unknown whether this design can be adapted to other subtyping rules. Both `QuickSub` and the Amber rules are not affected by the substitution overhead due to their algorithm design.

Worst case. It is worth noting that Table 1 also records the maximum size of the equality variable set $|S|_{\max}$ during the execution of `QuickSub`. Since the equality variable set operation is the main bottleneck of `QuickSub`, the size of this set can be a good indicator of the scenario where the algorithm performs the worst. For cases (1) to (6), the size is constant 0 or 1, since they follow a pattern where inner recursive types do not refer to outer recursive variables, as illustrated in Figure 10a. Nested recursive types are not problematic when they appear positively. Case (7) captures this pattern, which is illustrated in Figure 10b. When variables occur negatively and the subtyping result is equality $\approx$, all of the variables need to be added to the equality variable set. We test such a scenario in case (8), which basically compares a reflexive relation for types with a pattern shown in Figure 10c. The results show that `QuickSub` is still able to handle such a worst case scenario. Note that to achieve this, `QuickSub` needs to be optimized using the imperative boolean array technique, as discussed earlier. In contrast, the functional version of `QuickSub`, which uses the standard OCaml `Set` module for equality variable sets, takes 4.78 seconds for case (8) in Table 1.

Overall, `QuickSub` remains consistently efficient, demonstrating strong performance across various recursive type patterns.

```
class A {
  foo(Nat) : A
  ...
  class B {
    bar(Real) : B
}}
```

(a) Simple Recursive Objects

```
class A {
  foo(Nat) : A
  ...
  class B {
    bar(Real) : A
}}
```

(b) Nested Recursive Objects

$$\mu\alpha_1.\ \alpha_1 \to (\mu\alpha_2.\ \alpha_1 \to \alpha_2 \to (\mu\alpha_3.\ \alpha_1 \to \alpha_2 \to \alpha_3 \to (\mu\alpha_4.\ \ldots\ \ldots)))$$

(c) Worst Case Pattern

Fig. 10. Illustrations of worst case pattern and corresponding objects.

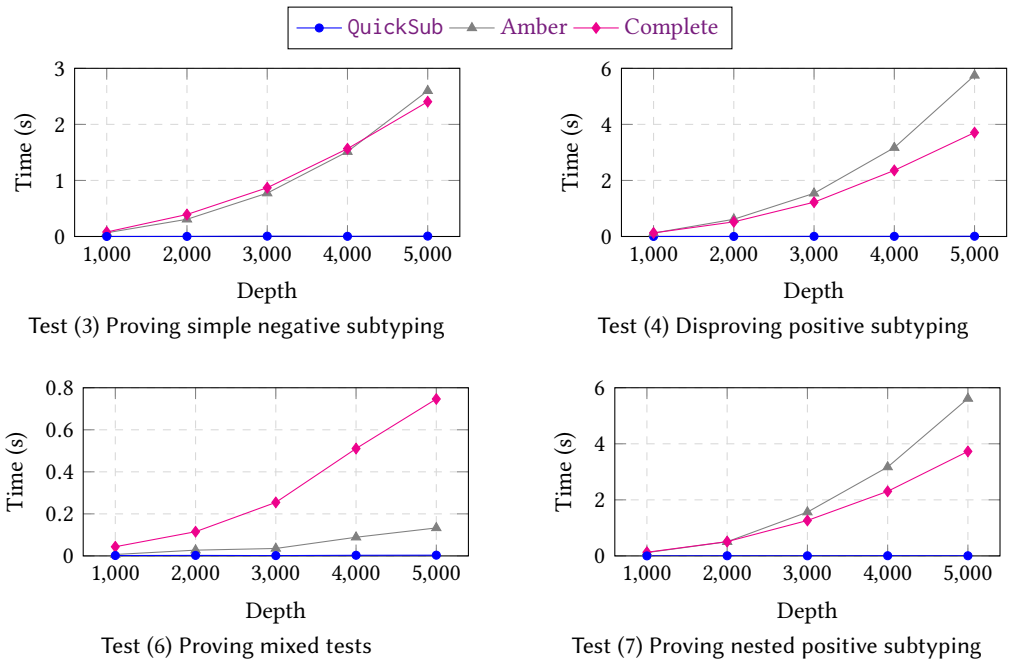


Fig. 11. Comparison of different works across multiple tests

Time complexity, empirically. Next, to reveal the time complexity of QuickSub and other algorithms, we fix the pattern and vary the depth of recursive types. We plot the results in Figure 11 for testing cases (3), (4), (6) and (7), which are all non-trivial cases (compared to reflexivity cases) that are commonly encountered in practice.

The results in Figure 11 show that QuickSub demonstrates a *linear complexity* trend across all tested scenarios. We do not present the results for the nominal unfolding rules or the equi-recursive algorithm as they time out or take a long time in all these selected cases. In all these cases, the performance of QuickSub is consistently better than other algorithms, with the execution time increasing *linearly* as the depth of recursive types increases, while other algorithms exhibit a steeper linear or even quadratic growth in execution time.

5.2 Subtyping Record Types

Next, we focus on scenarios frequently encountered in practical programming, where record types are used to encode objects or data structures. To evaluate how well the algorithms manage such

Test No.	QuickSub$^{\{\}}$	Amber	Equi	Complete	Description
1	**0.0124**	0.0822	Timeout	3.5369	Disprove positive subtyping
2	**0.0609**	0.0758	Timeout	0.8942	Disprove negative subtyping
3	**0.1051**	4.1899	Timeout	5.7472	Prove positive subtyping
4	**0.1147**	4.2414	Timeout	0.9660	Prove negative subtyping with $\top$ types

Table 2. Runtime results (seconds) for subtyping record types (depth = 100, width = 1000). Timeout means taking more than 100 seconds or causing a stack overflow.

structures, we conducted tests on benchmarks that contain record types extending in both width and recursive depths. The detailed methodology for generating these benchmarks is discussed in the appendix. These benchmarks are designed to simulate realistic conditions, reflecting common patterns as also noted in related work on optimizing iso-recursive subtyping in practical settings like WebAssembly [Rossberg 2023], who observed that even with shallow recursive depths, nested record types can cause significant performance overhead for subtyping algorithms. Here, recursive record types typically involve a large number of fields with a relatively shallow recursive depth.

Evaluation Setup. We have already discussed extending QuickSub to QuickSub$^{\{\}}$ for supporting record types in Section 4. As for the other algorithms, we extend the nominal unfolding rules [Zhou et al. 2022b] and Ligatti et al. [2017]'s rules with an extra case for pointwisely comparing the fields of the record types. The extension to the Amber rules is more involved, as the algorithm relies on a reflexivity check to handle subtyping between recursive types. We extend the syntactic reflexivity check to a more generalized check that allows permutations of the fields in the record types, which leads to a slight cost in performance.

Overall Performance. We first fix the depth of the recursive types to 100 and the width of the record types to 1000 and test QuickSub$^{\{\}}$ against other algorithms over four scenarios. The results are shown in Table 2. Our evaluations reveal that QuickSub$^{\{\}}$ consistently performs best.

Scalability of QuickSub$^{\{\}}$. We next extend our testing to benchmarks with varying widths and depths, focusing particularly on case (3) in Table 2 as it represents the most typical and practical scenario. We omit the results for the nominal unfolding rules and present the results for the equi-recursive algorithm in the right table of Figure 9 alone as they take significantly longer to complete compared to other algorithms or time out. The results, illustrated in Figure 12, show minimal differences in runtime among all algorithms when the depth is 1, reflecting a computational complexity of $O(m)$, where m denotes the type size. However, as the recursive depth increases, QuickSub$^{\{\}}$ starts to show significant advantages. For example, at a depth of 10, QuickSub$^{\{\}}$ already outperforms other algorithms when the record type has a certain number of fields. The margin of improvement becomes more pronounced as the depth increases, as can be seen by the increasing time scale in the four subplots. In practical programming tasks, where the definition of one object often relies on another, it is common to have nested recursive structures with a large number of fields to be compared, as we have shown in Figure 10. Our analysis shows that in such cases, QuickSub$^{\{\}}$ will demonstrate notable efficiency and scalability advantages over other algorithms.

6 Related Work

Iso-recursive subtyping. We have reviewed the most related work on subtyping iso-recursive types in the overview, such as the iso-recursive Amber rules [Cardelli 1985], the nominal unfolding rules [Zhou et al. 2022b], and the complete subtyping rules [Ligatti et al. 2017]. While for this paper we have focused on efficiency and have shown that QuickSub has significant advantages in

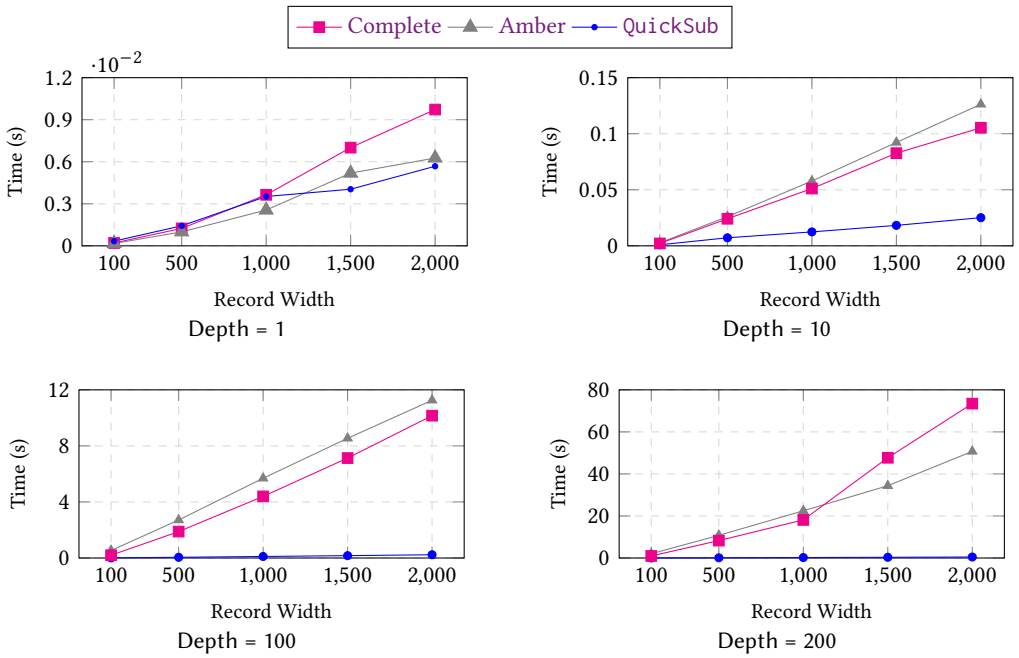


Fig. 12. Runtime results (seconds) for different algorithms with varying benchmark sizes in depth and width.

terms of performance, there are some other considerations that we have not discussed. For instance, an advantage of the nominal unfolding rules, compared to all other iso-recursive formulations (including QuickSub) is the *modularity* of the rules and proofs. That is, the nominal rules do not require changes to pre-existing rules (such as subtyping for functions) and they do not also require changes to proof cases that are not related to recursive types. QuickSub does not share this advantage and it does require changes to other rules, and cases that are not related to recursive types in the proofs. Nonetheless we have shown that QuickSub still allows us to develop the relevant metatheory without relying on other formulations of iso-recursive subtyping.

There are other alternative formulations for iso-recursive subtyping as well. Hofmann and Pierce [1995] introduced a subtyping relation that limits recursive subtyping to covariant types only. In other words, only functions with same input types can be related by subtyping, which makes their rules more restrictive than the Amber rules. Hosoya et al. [1998] restricted the recursive types to be top-level datatype declarations only, and formulated a subtyping relation that deals with mutually recursive declarations. However, this approach moves away from the standard conventional operational semantics and subtyping rules for recursive types as the rules rely on a special list of subtyping assumptions. Recently, Rossberg [2023] developed a calculus for higher-order declared subtyping with a standard formulation of iso-recursive types and subtyping, which effectively handles mutually recursive types without falling back to encodings of quadratic size as in the classic Amber rules. They observe that in practice, especially for languages more focused on nominal typing, the full power of equi-recursive subtyping or even iso-recursive subtyping is not needed. By restricting possible supertypes of recursive types to only types that have been declared to be in a subtyping relation, they can check the subtyping of recursive types only once when declaring the recursive types. In contrast, with general equi- or iso-recursive types, the recursive subtyping check needs to be done whenever needed. Compared to the line of work on declared

subtyping, we focus on the latter case, where the full power of iso-recursive subtyping is available, which could be helpful in languages with some form of structural typing.

Implementing iso-recursive subtyping. Some variants of iso-recursive subtyping already come with an algorithm for checking subtyping with efficiency in mind. For instance, both the complete subtyping rules [Ligatti et al. 2017] and the mutually iso-recursive subtyping rules [Rossberg 2023] have efficient algorithms. However, Ligatti et al. only presented their algorithm as an ML program and argued informally about the correspondence to his proposed complete rules (see also Section 2.1). In contrast, we formalize our algorithm as a set of inference rules and have formal results for the correctness of the algorithm in Section 3.2. For Amber-style subtyping, there are various algorithmic variants [Zhou et al. 2022b]. However, these variants were not designed for efficiency, but to help with theoretical issues. For example, the weakly positive rules that we discussed in Section 3.2, still rely on built-in reflexivity rule PosRes-self to ensure that the subtyping relation is reflexive for negative recursive types. Thus, a non-deterministic choice of the rules has to be made, requiring backtracking in an algorithm. Cardelli [1993] considered a recursive subtyping algorithm that is claimed to be equivalent to the iso-recursive Amber rules. As we discussed in Section 2.1, a formal proof of equivalence is missing, and due to the "tie" computation, and the way it checks equivalence by running twice the subtyping algorithm, the algorithm itself is not as efficient as `QuickSub`.

Equi-recursive types. Equi-recursive types, where a recursive type is considered as an infinite tree and equal to its unfoldings, also have a long history in the literature. They are widely used in various calculi and programming languages. Amadio and Cardelli [1993] laid the groundwork for equi-recursive subtyping, later refined by Brandt and Henglein [1998]; Gapeyev et al. [2002]; Kozen et al. [1993]. Equi-recursive types are used in session types [Castagna et al. 2009; Chen et al. 2014; Gay and Hole 2005; Gay and Vasconcelos 2010], gradual typing [Siek and Tobin-Hochstadt 2016], Scala's Dependent object types (DOT) calculus [Amin et al. 2016; Rompf and Amin 2016], and so on. There have also been efforts to adapt equi-recursive subtyping algorithms to more advanced type systems, where the general equi-recursive subtyping problem is impractical or undecidable. A recent effort in this direction was made by DeYoung et al. [2024], who have shown that equi-recursive subtyping with parametric polymorphism and recursively defined type constructors is undecidable. However it is still possible to obtain a sound (but incomplete) algorithm that works well in practice.

The relation between iso-recursive and equi-recursive types has also been studied in the literature [Abadi and Fiore 1996; Patrignani et al. 2021; Urzyczyn 1995; Zhou et al. 2024]. Notably, Abadi and Fiore argued that by inserting coercion functions, equi-recursive types can be encoded as iso-recursive types and vice versa. Thus, the two formulations have the same expressive power. Patrignani et al. further proved that the transformation from iso-recursive types to equi-recursive types, by erasing the unfold and fold operators, is fully abstract with respect to contextual equivalence. Abadi and Fiore's encoding of equi-recursive types via iso-recursive types has an important drawback: the coercion functions employed in the encoding are computationally relevant. Zhou et al. presented a new formulation of iso-recursive types, called full iso-recursive types, that enables encoding all equi-recursive programs without computational overhead. This is achieved by generalizing fold and unfold operations to computationally irrelevant casts, that can replace the coercions in Abadi and Fiore's work. Their results also extend to subtyping, which previous works have not considered. The subtyping rules used in their $\lambda_{Fi}^{\mu<:}$ calculus are based on the iso-recursive Amber rules, which means that `QuickSub` can be directly applied to their calculus and obtain a more efficient version of full iso-recursive subtyping.

Additionally, it is important to identify the different expressiveness results in the literature and their implications. Specifically, in terms of the expressive power of the subtyping relation, we have the iso-recursive Amber rules, which are less expressive than the complete rules by Ligatti et al. [2017], which in turn are less expressive than equi-recursive subtyping. However, the results by Abadi and Fiore [1996]; Zhou et al. [2024] show that by having more annotations (casts) or coercions in the term language, we can bridge this expressive power gap. Furthermore, Zhou et al. show that this can be done without any runtime performance cost with their full iso-recursive types. As implied by Zhou et al. and Amadio and Cardelli [1993], equi-recursive subtyping to some extent mixes the coinductive, infinite-tree view of recursive type equality with the inductive, finite-tree view of recursive subtyping. The two can be decomposed if needed. For example, let $A = \mathsf{nat} \to (\mu\alpha.\ \top \to \alpha)$ and $B = \mu\alpha.\ \mathsf{nat} \to \top \to \alpha$, then the equi-recursive subtyping $A \leq_e B$ is decomposed as follows:

$$A =_e \mathsf{nat} \to \top \to (\mu\alpha.\ \top \to \top \to \alpha) \leq \mathsf{nat} \to \top \to (\mu\alpha.\ \mathsf{nat} \to \top \to \alpha) =_e B$$

where $=_e$ denotes the coinductive equi-recursive equality relation and $\leq$ is an Amber-style iso-recursive subtyping relation. In such scenarios, the iso-recursive subtyping part can be replaced by any equivalent set of rules, such as our `QuickSub` rules, and the equi-recursive expressive power can be recovered by other means, such as casts or coercions.

Mechanized proofs on recursive types. Amber-style iso-recursive subtyping rules were only recently mechanized in proof assistants by Zhou et al. [2020, 2022b]. They formalize the Amber rules, nominal unfolding rules, and the weakly positive rules and prove several properties and equivalence results between them in Coq. Our work aligns closely with these efforts, by formalizing the equivalence proof of `QuickSub` to their rules and the type soundness proof in Coq. There are a few works on mechanizing other variants of recursive subtyping. Appel and Felty [2000]; Backes et al. [2014] formalized subtyping relations for iso-recursive types in Coq, focusing on positive subtyping. Amin and Rompf [2017]'s formalization of DOT involves a special form of equi-recursive types. Patrignani et al. [2021] formalized three calculi in Coq: a simply typed lambda calculus extended with iso-recursive types, equi-recursive types, and term-level fixpoints and showed several contextual equivalence results between them. Zhou et al. [2024] formalized their full iso-recursive types in Coq and adopted Zhou et al. [2022b]'s formalization for subtyping iso-recursive types.

7 Conclusion

In this paper, we introduce `QuickSub`, a novel efficient algorithm for iso-recursive subtyping. We show its correctness by proving its equivalence to the well-known Amber rules. We also show a direct type soundness proof for a calculus with `QuickSub`. Our OCaml implementation, accompanied by empirical evaluations, shows that `QuickSub` significantly outperforms existing algorithms, particularly in practical scenarios involving positive subtyping and nested recursive types. We also extend `QuickSub` to handle record types, broadening its applicability. Future work includes extending `QuickSub` to incorporate additional features and providing a direct equivalence proof for $\texttt{QuickSub}^{\{\}}$ with existing systems, such as the nominal unfolding rules. Furthermore, we plan to leverage full iso-recursive types [Zhou et al. 2024] to encode equi-recursive subtyping, and enhance the algorithm's expressiveness and efficiency in more complex type systems.

Data Availability Statement

The `QuickSub` formalization, implementation and evaluation examples are openly available in Zenodo [Zhou and Oliveira 2024] and https://github.com/ltzone/QuickSub.

A Generation of Benchmarks

We use the same named representation for the types in the benchmarks for all the algorithms in the evaluation. We assume that all the types tested have distinct names for each type variable, which is the same assumption made in [Ligatti et al. 2017].

For the patterns tested in Table 1, we generate the types as follows, where n is the depth of the type:

(1) Disproving negative strict subtyping: $\mu\alpha_1.\ \mathsf{nat} \to (\mu\alpha_2.\mathsf{nat} \to \cdots (\mu\alpha_n.\ \alpha_n \to \mathsf{nat})) \not\leq \mu\alpha_1.\ \mathsf{nat} \to (\mu\alpha_2.\mathsf{nat} \to \cdots (\mu\alpha_n.\ \alpha_n \to \mathsf{real}))$
(2) Proving negative equivalent subtyping with one variable at each level of recursive body: $\mu\alpha_1.\ \mathsf{nat} \to (\mu\alpha_2.\mathsf{nat} \to \cdots (\mu\alpha_n.\ \alpha_n \to \mathsf{nat})) \approx \mu\alpha_1.\ \mathsf{nat} \to (\mu\alpha_2.\mathsf{nat} \to \cdots (\mu\alpha_n.\ \alpha_n \to \mathsf{nat}))$
(3) Proving positive strict subtyping with one variable at the innermost position: $\mu\alpha_1.\ \mathsf{real} \to (\mu\alpha_2.\ \mathsf{real} \to \cdots (\mu\alpha_n.\ \mathsf{real} \to \alpha_n)) < \mu\alpha_1.\ \mathsf{nat} \to (\mu\alpha_2.\ \mathsf{nat} \to \cdots (\mu\alpha_n.\ \mathsf{nat} \to \alpha_n))$
(4) Disproving positive strict subtyping with multiple variables at the innermost position: $\mu\alpha_1.\ \mathsf{nat} \to (\mu\alpha_2.\ \mathsf{nat} \to \cdots (\mu\alpha_n.\ \mathsf{nat} \to \alpha_1 \times \ldots \times \alpha_n)) \not< \mu\alpha_1.\ \mathsf{nat} \to (\mu\alpha_2.\ \mathsf{nat} \to \cdots (\mu\alpha_n.\ \mathsf{real} \to \alpha_1 \times \ldots \times \alpha_n))$
(5) Proving positive equivalent subtyping with multiple variables at the innermost position: $\mu\alpha_1.\ \mathsf{real} \to (\mu\alpha_2.\ \mathsf{real} \to \cdots (\mu\alpha_n.\ \mathsf{real} \to \alpha_1 \times \ldots \times \alpha_n)) \approx \mu\alpha_1.\ \mathsf{real} \to (\mu\alpha_2.\ \mathsf{real} \to \cdots (\mu\alpha_n.\ \mathsf{real} \to \alpha_1 \times \ldots \times \alpha_n))$
(6) Proving mixed tests: generate 10 pairs of types that are randomly selected from the above patterns and compose them using product types in a pairwise way.
(7) Proving positive strict subtyping with multiple variables at the innermost position: $\mu\alpha_1.\ \mathsf{real} \to (\mu\alpha_2.\ \mathsf{real} \to \cdots (\mu\alpha_n.\ \mathsf{real} \to \alpha_1 \times \ldots \times \alpha_n)) < \mu\alpha_1.\ \mathsf{nat} \to (\mu\alpha_2.\ \mathsf{nat} \to \cdots (\mu\alpha_n.\ \mathsf{nat} \to \alpha_1 \times \ldots \times \alpha_n))$
(8) (The worst case scenario for `QuickSub`) Proving negative equivalent subtyping with multiple variables at the each level of recursive body: testing a type of the form described in Figure 10c with itself.

We use OCaml `Map` module to represent the fields of the record types in the benchmarks. For the patterns in Table 2, we generate the types as follows:

(1) Disproving positive subtyping:

$$
\begin{array}{l}
\mu\alpha_1.\ \{ \\
\quad l_1 : \mathsf{real}, \ldots, l_m : \mathsf{real} \\
\quad l_1' : \mathsf{real} \to \alpha_1, \ldots, l_m : \mathsf{real} \to \alpha_1 \\
\quad l_\mu : \mu\alpha_2.\ \{ \\
\qquad l_1 : \mathsf{real}, \ldots, l_m : \mathsf{real} \\
\qquad l_1' : \mathsf{real} \to \alpha_2, \ldots, l_m : \mathsf{real} \to \alpha_2 \\
\qquad l_\mu : \mu\alpha_3.\ \{\ldots \mu\alpha_n.\{\ldots\}\ \ldots\} \\
\quad \} \\
\}
\end{array}
\quad \not< \quad
\begin{array}{l}
\mu\alpha_1.\ \{ \\
\quad l_1 : \mathsf{nat}, \ldots, l_m : \mathsf{nat} \\
\quad l_1' : \mathsf{nat} \to \alpha_1, \ldots, l_m : \mathsf{nat} \to \alpha_1 \\
\quad l_\mu : \mu\alpha_2.\ \{ \\
\qquad l_1 : \mathsf{nat}, \ldots, l_m : \mathsf{nat} \\
\qquad l_1' : \mathsf{nat} \to \alpha_2, \ldots, l_m : \mathsf{nat} \to \alpha_2 \\
\qquad l_\mu : \mu\alpha_3.\ \{\ldots \mu\alpha_n.\{\ldots\}\ \ldots\} \\
\quad \} \\
\}
\end{array}
$$

(2) Disproving negative subtyping:

$$\begin{array}{l}\mu\alpha_1.\ \{\\ \quad l_1 : \text{real}, \ldots, l_m : \text{real}\\ \quad l'_1 : \alpha_1 \to \text{nat}, \ldots, l_m : \alpha_1 \to \text{nat}\\ \quad l_\mu : \mu\alpha_2.\ \{\\ \quad\quad l_1 : \text{real}, \ldots, l_m : \text{real}\\ \quad\quad l'_1 : \alpha_2 \to \text{nat}, \ldots, l_m : \alpha_2 \to \text{nat}\\ \quad\quad l_\mu : \mu\alpha_3.\ \{\ldots \mu\alpha_n.\{\ldots\}\ \ldots\}\\ \quad\}\\ \}\end{array} \not< \begin{array}{l}\mu\alpha_1.\ \{\\ \quad l_1 : \text{real}, \ldots, l_m : \text{real}\\ \quad l'_1 : \alpha_1 \to \text{real}, \ldots, l_m : \alpha_1 \to \text{real}\\ \quad l_\mu : \mu\alpha_2.\ \{\\ \quad\quad l_1 : \text{real}, \ldots, l_m : \text{real}\\ \quad\quad l'_1 : \alpha_2 \to \text{real}, \ldots, l_m : \alpha_2 \to \text{real}\\ \quad\quad l_\mu : \mu\alpha_3.\ \{\ldots \mu\alpha_n.\{\ldots\}\ \ldots\}\\ \quad\}\\ \}\end{array}$$

(3) Proving positive subtyping:

$$\begin{array}{l}\mu\alpha_1.\ \{\\ \quad l_1 : \text{nat}, \ldots, l_m : \text{nat}\\ \quad l'_1 : \text{real} \to \alpha_1, \ldots, l_m : \text{real} \to \alpha_1\\ \quad l_\mu : \mu\alpha_2.\ \{\\ \quad\quad l_1 : \text{nat}, \ldots, l_m : \text{nat}\\ \quad\quad l'_1 : \text{real} \to \alpha_2, \ldots, l_m : \text{real} \to \alpha_2\\ \quad\quad l_\mu : \mu\alpha_3.\ \{\ldots \mu\alpha_n.\{\ldots\}\ \ldots\}\\ \quad\}\\ \}\end{array} < \begin{array}{l}\mu\alpha_1.\ \{\\ \quad l_1 : \text{real}, \ldots, l_m : \text{real}\\ \quad l'_1 : \text{nat} \to \alpha_1, \ldots, l_m : \text{nat} \to \alpha_1\\ \quad l_\mu : \mu\alpha_2.\ \{\\ \quad\quad l_1 : \text{real}, \ldots, l_m : \text{real}\\ \quad\quad l'_1 : \text{nat} \to \alpha_2, \ldots, l_m : \text{nat} \to \alpha_2\\ \quad\quad l_\mu : \mu\alpha_3.\ \{\ldots \mu\alpha_n.\{\ldots\}\ \ldots\}\\ \quad\}\\ \}\end{array}$$

(4) Proving negative subtyping with $\top$ types:

$$\begin{array}{l}\mu\alpha_1.\ \{\\ \quad l_1 : \text{nat}, \ldots, l_m : \text{nat}\\ \quad l'_1 : \top \to \text{nat}, \ldots, l_m : \top \to \text{nat}\\ \quad l_\mu : \mu\alpha_2.\ \{\\ \quad\quad l_1 : \text{nat}, \ldots, l_m : \text{nat}\\ \quad\quad l'_1 : \top \to \text{nat}, \ldots, l_m : \top \to \text{nat}\\ \quad\quad l_\mu : \mu\alpha_3.\ \{\ldots \mu\alpha_n.\{\ldots\}\ \ldots\}\\ \quad\}\\ \}\end{array} < \begin{array}{l}\mu\alpha_1.\ \{\\ \quad l_1 : \text{real}, \ldots, l_m : \text{real}\\ \quad l'_1 : \alpha_1 \to \text{real}, \ldots, l_m : \alpha_1 \to \text{real}\\ \quad l_\mu : \mu\alpha_2.\ \{\\ \quad\quad l_1 : \text{real}, \ldots, l_m : \text{real}\\ \quad\quad l'_1 : \alpha_2 \to \text{real}, \ldots, l_m : \alpha_2 \to \text{real}\\ \quad\quad l_\mu : \mu\alpha_3.\ \{\ldots \mu\alpha_n.\{\ldots\}\ \ldots\}\\ \quad\}\\ \}\end{array}$$

Acknowledgments

We are grateful to the anonymous reviewers for their valuable comments. This work has been sponsored by Hong Kong Research Grant Council project number 17209821.

References

Martin Abadi and Luca Cardelli. 1996. *A theory of objects.* Springer Science & Business Media. https://doi.org/10.1007/978-1-4419-8598-9

Martin Abadi and Marcelo P Fiore. 1996. Syntactic considerations on recursive types. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science.* IEEE, 242–252. https://doi.org/10.1109/LICS.1996.561324

Roberto M Amadio and Luca Cardelli. 1993. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15, 4 (1993), 575–631. https://doi.org/10.1145/155183.155231

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The essence of dependent object types. *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday* (2016), 249–272. https://doi.org/10.1007/978-3-319-30936-1_14

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. https://doi.org/10.1145/3009837.3009866

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.

Michael Backes, Cătălin Hriţcu, and Matteo Maffei. 2014. Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations. *J. Comput. Secur.* 22, 2 (mar 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. https://doi.org/10.1145/1890028.1890031

Michael Brandt and Fritz Henglein. 1998. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae* 33, 4 (1998), 309–338. https://doi.org/10.3233/FI-1998-33401

Luca Cardelli. 1985. Amber, Combinators and Functional Programming Languages. *Proc. of the 13th Summer School of the LITP, Le Val D'Ajol, Vosges (France)* (1985).

Luca Cardelli. 1993. *An implementation of $F_{<:}$*. Digital Equipment Corporation Systems Research Center.

Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *Comput. Surveys* 17 (Dec. 1985), 471–522. https://doi.org/10.1145/6041.6042

Felice Cardone and Mario Coppo. 1991. Type inference with recursive types: Syntax and semantics. *Information and Computation* 92, 1 (1991), 48–80. https://doi.org/10.1016/0890-5401(91)90020-3

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. https://doi.org/10.1145/1599410.1599437

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. https://doi.org/10.1145/2643135.2643138

Ravi Chugh. 2015. IsoLATE: A type system for self-recursion. In *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 24*. Springer, 257–282. https://doi.org/10.1007/978-3-662-46669-8_11

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 268–279. https://doi.org/10.1145/351240.351266

Dario Colazzo and Giorgio Ghelli. 2005. Subtyping recursion and parametric polymorphism in kernel fun. *Information and Computation* 198, 2 (2005), 71–147. https://doi.org/10.1016/j.ic.2004.11.003

Mario Coppo. 1985. A completeness theorem for recursively defined types. In *International Colloquium on Automata, Languages, and Programming*. Springer, 120–129. https://doi.org/10.1007/BFb0015737

The Coq Development Team. 2024. *The Coq Proof Assistant*. https://doi.org/10.5281/zenodo.11551307

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*. 50–63. https://doi.org/10.1145/301618.301641

Nils Anders Danielsson and Thorsten Altenkirch. 2010. Subtyping, declaratively: An exercise in mixed induction and coinduction. In *Mathematics of Program Construction: 10th International Conference, MPC 2010, Québec City, Canada, June 21-23, 2010. Proceedings 10*. Springer, 100–118. https://doi.org/10.1007/978-3-642-13321-3_8

Nicolaas Govert De Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes mathematicae (proceedings)*, Vol. 75. Elsevier, 381–392.

Henry DeYoung, Andreia Mordido, Frank Pfenning, and Ankush Das. 2024. Parametric Subtyping for Structural Parametric Polymorphism. 8, POPL (2024).

Derek Dreyer. 2005. *Understanding and Evolving the ML Module System*. Ph. D. Dissertation. School of Computer Science, Carnegie Mellon University.

Derek R. Dreyer, Robert Harper, and Karl Krary. 2001. *Toward a Practical Type Theory for Recursive Modules*. Technical Report CMU-CS-01-112. School of Computer Science, Carnegie Mellon University.

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. https://doi.org/10.1145/586088.586093

Vladimir Gapeyev, Michael Y Levin, and Benjamin C Pierce. 2002. Recursive subtyping revealed. *Journal of Functional Programming* 12, 6 (2002), 511–548. https://doi.org/10.1017/S0956796802004318

Simon Gay and Malcolm Hole. 2005. Subtyping for session types in the pi calculus. *Acta Informatica* 42 (2005), 191–225. https://doi.org/10.1007/s00236-005-0177-z

Simon J Gay and Vasco T Vasconcelos. 2010. Linear type theory for asynchronous session types. *Journal of Functional Programming* 20, 1 (2010), 19–50. https://doi.org/10.1017/S0956796809990268

Giorgio Ghelli. 1993. Recursive types are not conservative over F≤. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 146–162. https://doi.org/10.1007/BFb0037104

Robert Harper. 2016. *Practical foundations for programming languages*. Cambridge University Press. https://doi.org/10.1017/CBO9781316576892

Martin Hofmann and Benjamin Pierce. 1995. Positive subtyping. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 186–197.

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).

Alan Jeffrey. 2001. A symbolic labelled transition system for coinductive subtyping of $F_{\mu\leq}$ types. In *2001 IEEE Conference on Logic and Computer Science (LICS 2001)*, Vol. 323.

Dexter Kozen, Jens Palsberg, and Michael I Schwartzbach. 1993. Efficient recursive subtyping. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 419–428. https://doi.org/10.1145/158511.158700

Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin. 2015. A theory of tagged objects. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2021. *The OCaml System, Release 4.12, Documentation and User's Manual*. https://ocaml.org/manual/4.12/index.html Accessed: 2024-07-06.

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. https://doi.org/10.1145/2994596

James H Morris. 1968. Lambda calculus models of programming languages. (1968).

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph. D. Dissertation. Chalmers University of Technology.

Marco Patrignani, Eric Mark Martin, and Dominique Devriese. 2021. On the semantic expressiveness of recursive types. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434302

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

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. https://doi.org/10.1145/2983990.2984008

Andreas Rossberg. 2023. Mutually Iso-Recursive Subtyping. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 347–373. https://doi.org/10.1145/3622809

Claudio V. Russo. 2001. Recursive Structures for Standard ML. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, Benjamin C. Pierce (Ed.). ACM, 50–61.

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: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Springer, 388–410. https://doi.org/10.1007/978-3-319-30936-1_21

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. *ACM SIGPLAN Notices* 46, 9 (2011), 266–278. https://doi.org/10.1145/2034574.2034811

Pawel Urzyczyn. 1995. Positive recursive type assignment. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, 382–391.

Litao Zhou and Bruno C. d. S. Oliveira. 2024. *QuickSub: Efficient Iso-Recursive Subtyping (Artifact)*. https://doi.org/10.5281/zenodo.13906402

Litao Zhou, Qianyong Wan, and Bruno C. d. S. Oliveira. 2024. Full Iso-Recursive Types. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 278 (Oct. 2024), 30 pages. https://doi.org/10.1145/3689718

Litao Zhou, Yaoda Zhou, and Bruno C d S Oliveira. 2023. Recursive Subtyping for All. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 1396–1425. https://doi.org/10.1145/3571241

Yaoda Zhou, Bruno C d S Oliveira, and Andong Fan. 2022a. A Calculus with Recursive Types, Record Concatenation and Subtyping. In *Asian Symposium on Programming Languages and Systems*. Springer, 175–195. https://doi.org/10.1007/978-3-031-21037-2_9

Yaoda Zhou, Bruno C d S Oliveira, and Jinxu Zhao. 2020. Revisiting iso-recursive subtyping. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28. https://doi.org/10.1145/3428216

Yaoda Zhou, Jinxu Zhao, and Bruno CDS Oliveira. 2022b. Revisiting Iso-recursive subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 44, 4 (2022), 1–54. https://doi.org/10.1145/3549537

Received 2024-07-10; accepted 2024-11-07