Recursive Subtyping for All

LITAO ZHOU* and YAODA ZHOU*, The University of Hong Kong, China
BRUNO C. D. S. OLIVEIRA, The University of Hong Kong, China

Recursive types and bounded quantification are prominent features in many modern programming languages, such as Java, C#, Scala or TypeScript. Unfortunately, the interaction between recursive types, bounded quantification and subtyping has shown to be problematic in the past. Consequently, defining a simple foundational calculus that combines those features and has desirable properties, such as *decidability*, *transitivity* of subtyping, *conservativity* and a sound and complete algorithmic formulation has been a long time challenge.

This paper presents an extension of kernel $F_{\leq}$, called $F^{\mu}_{\leq}$, with iso-recursive types. $F_{\leq}$ is a well-known polymorphic calculus with bounded quantification. In $F^{\mu}_{\leq}$ we add iso-recursive types, and correspondingly extend the subtyping relation with iso-recursive subtyping using the recently proposed nominal unfolding rules. We also add two smaller extensions to $F_{\leq}$. The first one is a generalization of the kernel $F_{\leq}$ rule for bounded quantification that accepts *equivalent* rather than *equal* bounds. The second extension is the use of so-called *structural* folding/unfolding rules, inspired by the structural unfolding rule proposed by Abadi, Cardelli, and Viswanathan [1996]. The structural rules add expressive power to the more conventional folding/unfolding rules in the literature, and they enable additional applications. We present several results, including: type soundness; transitivity and decidability of subtyping; the conservativity of $F^{\mu}_{\leq}$ over $F_{\leq}$; and a sound and complete algorithmic formulation of $F^{\mu}_{\leq}$. Moreover, we study an extension of $F^{\mu}_{\leq}$, called $F^{\mu}_{\leq\geq}$, which includes *lower bounded quantification* in addition to the conventional (upper) bounded quantification of $F_{\leq}$. All the results in this paper have been formalized in the Coq theorem prover.

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

Additional Key Words and Phrases: Iso-Recursive Subtyping, Bounded Polymorphism, Object Encodings

ACM Reference Format:
Litao Zhou, Yaoda Zhou, and Bruno C. d. S. Oliveira. 2023. Recursive Subtyping for All. *Proc. ACM Program. Lang.* 7, POPL, Article 48 (January 2023), 30 pages. https://doi.org/10.1145/3571241

1 INTRODUCTION

Recursive types and bounded quantification are two prominent features in many modern programming languages, such as Java, C#, Scala or TypeScript. Bounded quantification was introduced by Cardelli and Wegner [1985] in the Fun language, and has been widely studied [Cardelli et al. 1994; Curien and Ghelli 1992; Pierce 1994]. Bounded quantification addresses the interaction between parametric polymorphism and subtyping, allowing polymorphic variables to have subtyping bounds. Recursive types are needed in practically all programming languages to model recursive data structures (such as lists or trees), or recursive object types in Object-Oriented Programming (OOP) languages to encode *binary methods* [Bruce et al. 1995]. For adding recursive types to a

*Both authors contributed equally to this work.

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

2475-1421/2023/1-ART48
https://doi.org/10.1145/3571241

language with subtyping, it is desirable to have *recursive subtyping* between recursive types. The first rules for recursive subtyping, due to Cardelli [1985], are the well-known Amber rules. Recursive subtyping has been studied in two different forms: *equi-recursive* subtyping [Amadio and Cardelli 1993; Brandt and Henglein 1997; Gapeyev et al. 2003], and *iso-recursive* subtyping [Bengtson et al. 2011; Ligatti et al. 2017; Zhou et al. 2020, 2022]. In equi-recursive subtyping recursive types and their unfoldings are considered to be equal. In contrast, in iso-recursive subtyping they are only isomorphic and explicit fold/unfold operators are necessary to witness the isomorphism.

From the mid-80s and throughout the 90s there was a lot of work on establishing the type-theoretic foundations for OOP. Both recursive subtyping, as well as bounded quantification played a major part on this effort. The two features were perceived to be important to model objects in some forms of object encodings. At that time the key ideas around $F_{\leq}$ [Cardelli et al. 1994; Cardelli and Wegner 1985; Curien and Ghelli 1992], which is a polymorphic calculus with bounded quantification (but no recursive types) were reasonably well understood due to the early work on the Fun language by Cardelli and Wegner. Therefore $F_{\leq}$-like calculi were being used in foundational work on OOP. Some landmark papers on the foundations of OOP, which established important results such as the distinction between inheritance and subtyping [Cook et al. 1989], *f-bounded quantification* [Canning et al. 1989], or encodings of objects [Abadi et al. 1996; Bruce et al. 1999; Cook et al. 1989], essentially assumed some $F_{\leq}$ variant with recursive types. Typically, recursive subtyping was supported via the Amber rules. However, extensions of $F_{\leq}$ with recursive types had still not been developed and formally studied when many of those works were published.

After the first formalization of $F_{\leq}$ [Curien and Ghelli 1992], Ghelli [1993] questioned this state-of-affairs, which implicitly assumed that the extension of $F_{\leq}$ with recursive types was straightforward. He conducted the first formal study for such an extension, and showed a wide range of negative results. Most importantly, he showed that equi-recursive types are not conservative over full $F_{\leq}$. In other words, adding equi-recursive types to full $F_{\leq}$ changes the expressive power of the subtyping relation, *even when the types being compared do not involve any recursive types.* The simple addition of equi-recursive types allows well-formed, but invalid subtyping statements in $F_{\leq}$ to be valid in an extension with recursive types. Ghelli also shows that applying equi-recursive types to full $F_{\leq}$ invalidates transitivity elimination: we cannot drop the transitivity rule without losing expressive power. In addition, while subtyping in full $F_{\leq}$ is undecidable [Pierce 1994], the change in expressive power reopens questions about the decidability or undecidability of the system.

Even if we choose the weaker form of bounded quantification present in Fun language and kernel $F_{\leq}$, the natural extension of Amadio and Cardelli [1993]'s algorithm to kernel $F_{\leq}$ is incomplete [Colazzo and Ghelli 2005]. Nevertheless, instead of Amadio and Cardelli's *meet 2 times* rules, Colazzo and Ghelli gave an alternative *meet 3 times* algorithm, accompanied by a very challenging correctness proof, showing that the subtyping relation is transitive and complete, but did not prove conservativity. Based on an earlier draft from Colazzo and Ghelli, Jeffrey [2001] extended the system and proved it correct and complete. By transferring the polar bisimulations [Sangiorgi and Milner 1992] technique from concurrency theory, Jeffrey's system is more general than Colazzo and Ghelli's, but it is only partially decidable. It is decidable for kernel $F_{\leq}^{equi}$, but for full $F_{\leq}^{equi}$, only when the algorithm terminates it returns the correct answer, but it may not terminate. Furthermore, although more powerful, Jeffrey's full $F_{\leq}^{equi}$ is not conservative over $F_{\leq}$ either.

Table 1 summarizes the results of previous work on extending $F_{\leq}$ with recursive types. Note that, in the table, the *Type System* row simply means whether the typing relation of the $F_{\leq}$ extension with recursive types has been studied/presented in the paper. For properties such as type soundness, decidability or conservativity, there is a corresponding entry in the table, which states whether the property was proved or not. *Modularity* here means whether the original rules and definitions of

Table 1. Comparison among different works.

	$F^{equi}_{\leq}$ 1993	Kernel $F^{equi}_{\leq}$ 2005	$F^{equi}_{\leq}$ 2001	Kernel $F^{equi}_{\leq}$ 2001	$F^{iso}_{\leq}$ 1996	$F^{\mu}_{\leq}$ present work
Transitivity	×	✓	✓	✓	built-in	✓
Decidability		✓	✓̸	✓		✓
Conservativity	×		×			✓
Type System					✓	✓
Algorithmic Type System						✓
Type Soundness						✓
Modularity	×	×	×	×		✓
Mechanized Proofs	×	×	×	×	×	✓

A × symbol denotes a negative result (the property or feature does not hold). A ✓ denotes a positive result, while ✓̸ denotes a partial result (such as semi-decidability). Whitespace denotes that the property/feature has not been studied or it is unknown.

$F_{\leq}$ are the same or they need to be modified. The proofs in all the 4 systems with equi-recursive types are complex because of the strong recursion. Adding equi-recursive subtyping requires major changes in existing definitions, rules and proofs compared to $F_{\leq}$, making most of the existing metatheory on $F_{\leq}$ not reusable. No prior work has proved the conservativity of $F_{\leq}$ with equi-recursive types. This result is likely to be hard to prove because of the numerous non-modular changes in $F_{\leq}$ induced by the introduction of equi-recursive subtyping. Furthermore, in those works the full type systems are not provided. Perhaps motivated by the technical challenges and negative results posed by equi-recursive types, some researchers set their sights on iso-recursive types. In their work on object encodings, Abadi et al. [1996] proposed the $F_{<:\mu}$ calculus, which supports bounded universal types, bounded existential types and iso-recursive types via the Amber rules. However, reflexivity and transitivity are built-in, so the system is not algorithmic. Furthermore, while they presented the typing, subtyping and reduction rules, they have not proved any properties, including type soundness or the conservativity over full $F_{\leq}$. One potential reason for the absence of technical results is that the iso-recursive Amber rules are hard to work with formally [Backes et al. 2014; Ligatti et al. 2017; Zhou et al. 2020, 2022]: it is difficult to prove results such as transitivity, or define sound and complete algorithmic formulations.

This paper presents an extension of kernel $F_{\leq}$, called $F^{\mu}_{\leq}$, with iso-recursive types. In $F^{\mu}_{\leq}$ we add iso-recursive subtyping using the recently proposed nominal unfolding rules [Zhou et al. 2022]. The nominal unfolding rules have been formally proved to be type sound, and shown to have the same expressive power as the well-known iso-recursive Amber rules [Cardelli 1985]. Moreover, the nominal unfolding rules address the difficulties of working formally with the (iso-recursive) Amber rules. With the nominal unfolding rules, proving transitivity and other properties is easy, also enabling developing algorithmic formulations of subtyping instead. Furthermore, a nice property of the nominal unfolding rules is that they are modular, allowing an existing calculus to be extended with recursive types without major impact on existing definitions and proofs. In other words they allow reusing most existing metatheory and definitions that existed before the addition of iso-recursive types. Our work shows that the nominal unfolding rules proposed by Zhou et al. can be integrated modularly into $F_{\leq}$, while retaining desirable properties. In particular, we prove, for the first time, the conservativity of an extension of $F_{\leq}$ with recursive types over the original $F_{\leq}$.

We also add two smaller extensions to $F_{\leq}$. The first one is a generalization of the kernel $F_{\leq}$ rule for bounded quantification that accepts *equivalent* rather than *equal* bounds. The second extension is the use of so-called *structural* folding/unfolding rules, inspired by the structural unfolding rule proposed by Abadi et al. [1996]. The structural rules add expressive power to the more conventional

folding/unfolding rules in the literature, and they enable additional applications. In particular, we illustrate how the structural rules play an important role to model encodings of objects, as well as encodings of algebraic datatypes with subtyping.

We present several results, including: *type soundness*; *transitivity* and *decidability* of subtyping; the *conservativity* of $F^{\mu}_{\leq}$ over $F_{\leq}$; and a sound and complete algorithmic formulation of $F^{\mu}_{\leq}$. Moreover, we also present an extension of $F^{\mu}_{\leq}$, called $F^{\mu}_{\leq\geq}$, which has a bottom type and *lower bounded quantification* in addition to the conventional (upper) bounded quantification of $F_{\leq}$. As we show, lower bounded quantification is particularly interesting to model the subtyping of algebraic datatypes. All the results in this paper have been formalized in the Coq theorem prover.

In summary the contributions of this paper are:

- **$F^{\mu}_{\leq}$: an extension of kernel $F_{\leq}$ with iso-recursive types and subtyping.** We prove several important properties for $F^{\mu}_{\leq}$, including: *type soundness*; *transitivity* and *decidability* of subtyping; and the *unfolding lemma*.
- **The conservativity of $F^{\mu}_{\leq}$ over $F_{\leq}$.** Conservativity is an expected, but non-trivial property, that has eluded past work on the combination of bounded quantification and recursive types. We show that an extension of kernel $F_{\leq}$ with iso-recursive subtyping based on the nominal unfolding rules is conservative over kernel $F_{\leq}$.
- **Type soundness for the structural folding/unfolding rules.** We present the first formal type soundness proof for the structural unfolding rule, and we also present a new structural folding rule, together with its type soundness.
- **Decidability for a generalization of the kernel $F_{\leq}$ rule for bounded quantification,** which compares the bounds for equivalence. A key property is that equivalent types have equal sizes, enabling the decidability proof.
- **An extension of $F^{\mu}_{\leq}$ with both upper and lower bounded quantification:** We present an extended calculus, called $F^{\mu}_{\leq\geq}$, with both top and bottom types, and both upper and lower bounded quantification, and illustrate its applicability to encodings of datatypes with subtyping.
- **Coq formalization:** We have formalized all the calculi and proofs in this paper in Coq, and made the formalization available online: https://github.com/juda/Recursive-Subtyping-for-All

2 OVERVIEW

This section provides an overview of our work. We first briefly review basic concepts and some applications. Then we show our key ideas and results.

2.1 Bounded Quantification and Recursive Subtyping

Bounded Quantification. Bounded quantification allows types to be abstracted by type variables with a subtyping constraint (or bound). The standard calculus with bounded quantification, $F_{\leq}$ [Cardelli et al. 1994; Cardelli and Wegner 1985; Curien and Ghelli 1992], has two common variants when it comes to subtyping universal types. The full $F_{\leq}$ variant [Cardelli et al. 1994; Curien and Ghelli 1992] compares bounded quantifiers with the following rule:

$$\frac{\Gamma \vdash A_2 \leq A_1 \qquad \Gamma,\ \alpha \leq A_2 \vdash B \leq C}{\Gamma \vdash \forall(\alpha \leq A_1).\ B \leq \forall(\alpha \leq A_2).\ C}\ \text{S-FULLALL}$$

The most significant characteristic of full $F_{\leq}$ is that it allows two bounded quantifiers to be contravariant on their bound types A_1 and A_2 when being compared. However, the rich expressivity of full $F_{\leq}$ results in an undecidable subtyping relation [Pierce 1994], which is undesirable. In addition, as Ghelli [1993] demonstrates, the rule S-FULLALL may even prevent conservative extensions of $F_{\leq}$ in the presence of additional features.

There are several ways to restrict bounded quantification to a fragment with decidable subtyping, such as removing top types, or assuming no bounds when comparing type abstraction bodies [Castagna and Pierce 1994]. Among those the most widely used variant is the kernel $F_{\leq}$ calculus. In kernel $F_{\leq}$ bounded quantifiers can only be subtypes when their bound types are identical [Cardelli and Wegner 1985], which is stated in the rule S-KERNELALL.

$$\textsc{S-kernelall}\ \frac{\Gamma \vdash A \qquad \Gamma,\ \alpha \leq A \vdash B \leq C}{\Gamma \vdash \forall(\alpha \leq A).\ B \leq \forall(\alpha \leq A).\ C} \qquad \textsc{S-equivall}\ \frac{\Gamma \vdash A_1 \leq A_2 \qquad \Gamma \vdash A_2 \leq A_1 \qquad \Gamma,\ \alpha \leq A_2 \vdash B \leq C}{\Gamma \vdash \forall(\alpha \leq A_1).\ B \leq \forall(\alpha \leq A_2).\ C}$$

In the rest of the paper, we will focus on kernel $F_{\leq}$, in order to achieve decidable subtyping with iso-recursive subtyping. However, differently from kernel $F_{\leq}$, we will generalize the rule S-KERNELALL to a rule S-EQUIVALL that accepts equivalent bounds instead. The main motivation for using rule S-EQUIVALL is to enable more subtyping involving records. While typically kernel $F_{\leq}$ is presented without records, in this paper we include records in the calculus and we wish to consider types such as $\{x : \mathsf{nat}, y : \mathsf{nat}\}$ and $\{y : \mathsf{nat}, x : \mathsf{nat}\}$, to be equivalent (despite being syntactically different). Note that, while in plain $F_{\leq}$ the subtyping relation is antisymmetric [Baldan et al. 1999] (i.e. if two types are equivalent then they must be equal), the addition of records breaks antisymmetry since there are equivalent types that are not equal. The rule S-EQUIVALL is more general than the kernel rule with identical bounds, but retains decidability, as we shall see in §4.2.

Recursive Types. Recursive types $\mu\alpha.\ A$, can be traced back to Morris [1969]. There are two basic approaches to recursive types: *equi-recursive types* and *iso-recursive types*. The essential difference between them is how they consider the relationship between a recursive type $\mu\alpha.\ A$ and its unfoldings $[\alpha \mapsto \mu\alpha.\ A]\ A$. In *equi-recursive* types, a recursive type is equal to its unfolding. That is $\mu\alpha.\ A = [\alpha \mapsto \mu\alpha.\ A]\ A$. In other words, recursive types and their unfoldings are interchangeable in all contexts. In *iso-recursive* types, a recursive type and its one-step unfolding are not equal but only isomorphic. To convert between $\mu\alpha.\ A$ and $[\alpha \mapsto \mu\alpha.\ A]\ A$ back and forth we need explicit unfold and fold operators. A fold expression constructs a recursive type, while an unfold expression opens a recursive type, as rule TYPING-FOLD and rule TYPING-UNFOLD illustrate:

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

Recursive Subtyping. Subtyping between recursive types has been studied for many years [Amadio and Cardelli 1993; Cardelli 1985; Ligatti et al. 2017]. The most widely used subtyping rules for recursive types are the Amber rules [Cardelli 1985], which consist of three rules: rule S-AMBER, rule S-ASSMP and rule S-REFL.

$$\textsc{S-amber}\ \frac{\Gamma,\ \alpha \leq \beta \vdash A \leq B}{\Gamma \vdash \mu\alpha.\ A \leq \mu\beta.\ B} \qquad \textsc{S-assmp}\ \frac{\alpha \leq \beta \in \Gamma}{\Gamma \vdash \alpha \leq \beta} \qquad \textsc{S-refl}\ \frac{}{\Gamma \vdash A \leq A}$$

The Amber rules are simple, but their metatheory is troublesome. For example, transitivity is hard to prove [Bengtson et al. 2011; Zhou et al. 2020, 2022]. Furthermore, due to the reliance on the reflexivity rule (rule S-REFL), the Amber rules are problematic for subtyping relations that are not antisymmetric [Ligatti et al. 2017]. Recently, Zhou et al. [2020, 2022] proposed a new specification for iso-recursive subtyping and some equivalent algorithmic variants. For this paper we use one of those algorithmic variants, called the *nominal unfolding rules* [Zhou et al. 2022]. The main reason to choose the nominal unfolding rules is that are easy to work with formally and Zhou et al. have a full Coq development, including proofs of decidability, that we reuse and extend. The nominal unfolding rules are:

$$\frac{\Gamma,\ \alpha \vdash [\alpha \mapsto A^{\alpha}]\ A \leq [\alpha \mapsto B^{\alpha}]\ B}{\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B}\ \textsc{S-nominal} \qquad \frac{\Gamma \vdash A \leq B}{\Gamma \vdash A^{\alpha} \leq B^{\alpha}}\ \textsc{S-label}$$

The key intuition for the nominal unfolding rules is that, in order to correctly deal with subtyping for *contravariant* occurrences of recursive type variables, we must unfold the recursive bodies twice to check the validity of the subtyping statement. For instance, the subtyping statement $\mu\alpha.\ \alpha \to \mathsf{nat} \leq \mu\alpha.\ \alpha \to \top$ should be rejected. If we unroll the recursive types twice, we get the following *invalid* subtyping statement:

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

which requires both that $\mathsf{nat} \leq \top$ (which is true) and $\top \leq \mathsf{nat}$, which is false and is the reason why the subtyping statement should fail. Note that unrolling this example one time only results in $(\mu\alpha.\ \alpha \to \mathsf{nat}) \to \mathsf{nat} \leq (\mu\alpha.\ \alpha \to \top) \to \top$, which does not expose $\top \leq \mathsf{nat}$ and is the reason why unrolling 1-time only is not enough. The rule S-nominal mimics this double-unrolling process, but we obtain the following subtyping statement instead:

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

where the recursive types are replaced by α (note that $\alpha \leq \alpha$ trivially holds). This statement is sufficient to trigger enough subtyping comparisons to validate the correctness of the subtyping statement. In particular, it also requires checking that $\top \leq \mathsf{nat}$, which is false and will reject the subtyping statement. By using recursive variables instead of the recursive types in the (double) unfolding process we obtain a terminating procedure. The nominal unfolding rules use *labelled types* A^{α}, to ensure that we only compare types that arise from unfolding substitutions with related unfolded types. Labelled types are a syntactic devise used to prevent accepting subtyping statements such as $\mu\alpha.\ \mathsf{nat} \to \alpha \leq \mu\alpha.\ \mathsf{nat} \to \mathsf{nat} \to \top$, which would be unsound in an iso-recursive formulation. They also provide a distinct nominal identity to the recursive types being compared, so that they cannot be compared with other unrelated recursive types. Note that, in the rule S-nominal, we have reused the recursive variable name for the label α. However, we used the red color to distinguish the label and type variable names. Labels α should be the same in both substitutions, and distinct from any other labels and bound variables used elsewhere. Since in the paper presentation we use a nominal approach to represent binders, the label α should be interpreted as some unique freshly generated name[1]. Zhou et al. proved that the nominal unfolding rules are type sound, and have the same expressive power as the iso-recursive Amber rules. Moreover, unlike the Amber rules, they are easy to work with formally and they can be easily used in subtyping relations that are not antisymmetric.

2.2 Applications of Bounded Quantification and Recursive Types

We now turn to applications of bounded quantification and recursive types. In particular the classic application for both features is encodings of objects [Bruce et al. 1999]. In addition, we also show that the two features are useful to model encodings of algebraic datatypes with subtyping.

Object Encodings. A simple and well-known typed encoding of objects is the recursive records encoding [Bruce et al. 1999; Canning et al. 1989; Cook et al. 1989]. In this encoding the idea is that object types are encoded as recursive record types, and objects are encoded as records. We will use a simplified form of the encoding, where we do not deal with self-references. But self-references

[1]In our Coq formalization we use a locally nameless representation [Aydemir et al. 2008], which distinguishes free and bound variables naturally. With a locally nameless representation we can reuse the free variable name α for the label α.

could be dealt with in standard ways. For example, we can define a type Point:

$$\text{Point} \triangleq \mu\,\text{pnt}.\{\text{x} : \text{Int},\ \text{y} : \text{Int},\ \text{move} : \text{Int} \to \text{Int} \to \text{pnt}\}$$

which consists of its coordinates and a move function. We use a recursive type because move should return an updated point. To implement Point we define some auxiliary functions:

```
function getX(p : Point) = (unfold [Point] p).x
function getY(p : Point) = (unfold [Point] p).y
function moveTo(p : Point, x : Int, y : Int) = (unfold [Point] p).move x y
```

then a constructor mkPoint can be defined as:

```
function mkPoint(x1 : Int, y1 : Int) = fold [Point] { x=x1, y=y1, move = λx2 y2. mkPoint(x2, y2) }
```

Note that the auxiliary functions above would not be needed in a source language, since a source language would treat p.x as syntactic sugar for (unfold [Point] p).x. Similarly, the source language would automatically insert a fold in the object constructor. In other words, in a source language with iso-recursive subtyping the fold's and unfold's do not need to be explicitly written and are automatically inserted by the compiler. For instance, this is what Abadi et al. [1996]'s translation of a language with objects into an iso-recursive extension of $F_{\leq}$ does.

With subtyping, we can develop subtypes of Point, such as:

$$\text{ColorPoint} \triangleq \mu\,\text{pnt}.\{\text{x} : \text{Int},\ \text{y} : \text{Int},\ \text{move} : \text{Int} \to \text{Int} \to \text{pnt},\ \text{color} : \text{String}\}$$
$$\text{EqPoint} \triangleq \mu\,\text{pnt}.\{\text{x} : \text{Int},\ \text{y} : \text{Int},\ \text{move} : \text{Int} \to \text{Int} \to \text{pnt},\ \text{eq} : \text{pnt} \to \text{Bool}\}$$

Now we wish to translate the coordinates by one unit for a point, but we do not want to write such a translation function for all subclasses of Point. This is achieved with a polymorphic function:

```
function translate [P ≤ Point] (p : P) = (unfold [Point] p).move (getX p + 1) (getY p + 1)
```

The type of this translate function is $\forall(\text{P} \leq \text{Point}).\ \text{P} \to \text{Point}$, which is obtained from the following typing derivation (some parts omitted):

$$\dots\ \dfrac{\dfrac{\dfrac{\text{P} \leq \text{Point},\ \text{p} : \text{P} \vdash \text{p} : \text{P} \qquad \text{P} \leq \text{Point},\ \text{p} : \text{P} \vdash \text{P} \leq \text{Point}}{\text{P} \leq \text{Point},\ \text{p} : \text{P} \vdash \text{p} : \text{Point}}\ \textsc{typing-sub}}{\text{P} \leq \text{Point},\ \text{p} : \text{P} \vdash (\text{unfold}\ [\text{Point}]\ \text{p}) : \left\{\begin{array}{l}\text{x} : \text{Int},\ \text{y} : \text{Int},\\ \text{move} : \text{Int} \to \text{Int} \to \text{Point}\end{array}\right\}}\ \textsc{typing-unfold}}{\vdash \text{translate} : \forall(\text{P} \leq \text{Point}).\ \text{P} \to \text{Point}}$$

However, this type is unsatisfying because it loses type precision: it returns a Point instead of a P. The type that we want instead is:

$$\forall(\text{P} \leq \text{Point}).\ \text{P} \to \text{P}$$

Unfortunately, we cannot obtain this more general type with only bounded quantification, and the usual unfolding rule TYPING-UNFOLD. In the rule TYPING-UNFOLD, the unfold annotation *must* be a recursive type. However, if we wish to return P, then we should use unfold with the annotation P, which is not a recursive type, but a type variable. Some advanced techniques, such as *f-bounded quantification* [Canning et al. 1989], address this issue. $F^{\mu}_{\leq}$ uses a less intrusive approach to obtain the desired typing for translate via the structural unfolding rules, which we discuss in §2.3.

Encoding positive f-bounded quantification. Fortunately, with the structural rules, we can use a type variable as an annotation for unfold. This enables us to encode forms of f-bounded quantification with positive occurrences of recursive variables, which is the case for Point. We can change the unfold annotation in translate from the recursive type Point to its subtype, the type variable P:

```
function translate [P ≤ Point] (p : P) = (unfold [P] p).move (getX p + 1) (getY p + 1)
```

In §2.3 we discuss the typing of this program via the structural unfolding rule in detail. After this change the type of translate is $\forall(P \leq \text{Point}).\ P \to P$. Then we can apply translate to Point or any of its subtypes, without losing static precision. Thus, if we call translate [EqPoint] (mkEqPoint 0 0), then we obtain an EqPoint object at $(1, 1)$. Note also that here mkEqPoint is a constructor for objects with type EqPoint, which contain a binary method [Bruce et al. 1995] eq.

$$\text{function mkEqPoint}(x_1 : \text{Int},\ y_1 : \text{Int}) = \text{fold [EqPoint]}$$
$$\{\ x = x_1, y = y_1, \text{move} = \lambda x_2\ y_2.\ \text{mkEqPoint}(x_2, y_2), \text{eq} = \lambda p.\ (\text{getX p} == x_1) \wedge (\text{getY p} == y_1)\ \}$$

Encodings of Algebraic Datatypes with Subtyping. It is well-known that in the polymorphic lambda calculus (System F) [Girard 1972; Reynolds 1974], we can use Church [1932] encodings to encode algebraic datatypes [Böhm and Berarducci 1985]. However, Church encodings make it hard to encode some operations, or worst they prevent encoding certain operations with the correct time complexity. A well-known example [Church 1932] is the encoding of the predecessor function on natural numbers, which is linear with Church encodings instead of being constant time.

An alternative encoding of datatypes in the untyped lambda calculus, which avoids the issues of Church encodings, is due to Scott [1962]. Unfortunately, Scott encodings cannot be encoded in plain System F. However, the addition of recursive types to a polymorphic lambda calculus allows a typed encoding for Scott encodings [Parigot 1992]. Moreover, in the presence of subtyping, we can also encode algebraic datatypes with subtyping, enabling certain forms of reuse that are not possible without subtyping. Oliveira [2009] has shown this assuming a $F_{\leq}$-like language with recursive types and records, but he has not formalized such a language. Here we revisit Oliveira's example. A similar encoding for datatypes can be achieved in $F^{\mu}_{\leq}$. For example, one may define a datatype Exp_1 for mathematical expressions, with numeric, addition, and subtraction constructors:

$$\text{data}\ \ \text{Exp}_1 = \text{Num Int} \mid \text{Add Exp}_1\ \text{Exp}_1 \mid \text{Sub Exp}_1\ \text{Exp}_1$$

The encoding in $F^{\mu}_{\leq}$ of this datatype can be defined as follows:

$$\text{Exp}_1 \triangleq \mu E.\ \forall A.\ \{\text{num} : \text{Int} \to A,\ \text{add} : E \to E \to A,\ \text{sub} : E \to E \to A\} \to A$$

If we unfold the recursive type, this encoding is a polymorphic higher order function that takes a record with three fields (num, add and sub) as input. Each field corresponds to a constructor in the datatype definition. This encoding is particularly useful for case analysis, since the polymorphic function essentially encodes case analysis directly. To write a function that performs case analysis on this datatype, one can unfold the recursive type, instantiate A with the result type, and then provide a record that maps each case to its implementation function that takes the constructor components as input and returns a result of type A. For example, given an expression e with type Exp_1, a case analysis-based evaluation function can be written as:

$$\text{function eval}\ (e : \text{Exp}_1) = (\text{unfold}\ [\text{Exp}_1]\ e)\ [\text{Int}]$$
$$\{\ \text{num} = \lambda n.\ n,\ \ \text{add} = \lambda e_1\ e_2.\ (\text{eval}\ e_1 + \text{eval}\ e_2),\ \ \text{sub} = \lambda e_1\ e_2.\ (\text{eval}\ e_1 - \text{eval}\ e_2)\ \}$$

where we use $[\ldots]$ to represent type instantiation. Here Exp_1 is instantiated with the evaluation result type Int. A record of three functions is supplied to implement case analysis. The num field implements a function that returns the integer n of the Num constructor directly, while the functions in add and sub fields perform the evaluation process recursively. To construct concrete instances of the datatype, each constructor also comes with a corresponding encoding in the calculus:

$$\text{function}\ \ \text{Num}_1\ (n\text{: Int}) = \text{fold}\ \ [\text{Exp}_1]\ (\Lambda\ A.\ \lambda\ e.\ (e.\text{num}\ n))$$
$$\text{function}\ \ \text{Add}_1\ (e_1 : \text{Exp}_1, e_2 : \text{Exp}_1) = \text{fold}\ \ [\text{Exp}_1]\ (\Lambda\ A.\ \lambda\ e.\ (e.\text{add}\ e_1\ e_2))$$
$$\text{function}\ \ \text{Sub}_1\ (e_1 : \text{Exp}_1, e_2 : \text{Exp}_1) = \text{fold}\ \ [\text{Exp}_1]\ (\Lambda\ A.\ \lambda\ e.\ (e.\text{sub}\ e_1\ e_2))$$

One can easily check, using rule TYPING-FOLD, that the result type of each constructor encoding becomes Exp_1 after a recursive type folding. Therefore, in this encoding, the use of constructors and case analysis functions is natural: one can construct the expression 1 + 2 directly with

the encoded constructors as $\mathsf{Add}_1\ (\mathsf{Num}_1\ 1)\ (\mathsf{Num}_1\ 2)$, and get its evaluation result by calling eval $(\mathsf{Add}_1\ (\mathsf{Num}_1\ 1)\ (\mathsf{Num}_1\ 2))$.

Subtyping between datatypes. Now consider a larger datatype Exp_2, which extends the Exp_1 datatype with a new constructor Neg, for denoting negative numbers.

```
data Exp₂ = Num Int | Add Exp₂ Exp₂ | Sub Exp₂ Exp₂ | Neg Exp₂
```

This datatype is encoded in $F^{\mu}_{\leq}$ as:

$$\mathsf{Exp}_2 \triangleq \mu\mathsf{E}.\ \forall\mathsf{A}.\ \{\mathsf{num} : \mathsf{Int} \to \mathsf{A},\ \mathsf{add} : \mathsf{E} \to \mathsf{E} \to \mathsf{A},\ \mathsf{sub} : \mathsf{E} \to \mathsf{E} \to \mathsf{A}, \mathsf{neg} : \mathsf{E} \to \mathsf{A}\} \to \mathsf{A}$$

The datatype Exp_2 differs from Exp_1 only in the new constructor. However, other constructors are just the same. To reduce code duplication, polymorphism on datatype constructors is desirable. Note that Exp_2 has more constructors than Exp_1, so it should be safe to coerce Exp_1 expressions into Exp_2 expressions, i.e. $\mathsf{Exp}_1 \leq \mathsf{Exp}_2$. Therefore, we would like the constructor for Add to have the following type, so that both Exp_1 and Exp_2 can use this constructor:

$$\mathsf{Add}_\forall : \forall(\mathsf{E} \geq \mathsf{Exp}_1).\ \mathsf{E} \to \mathsf{E} \to \mathsf{E}$$

There are two problems here. Firstly, similarly to the issue that we have faced in the translate function, we would like to use a type variable in the fold's of the constructors. This way we can make the constructors polymorphic. Secondly, as evidenced by the desired type for Add, we need *lower bounded quantification*, but in $F^{\mu}_{\leq}$ (and $F_{\leq}$) we only have upper bounded quantification.

Polymorphic constructors with lower bounded quantification. For applications such as encodings of algebraic datatypes, the dual form of bounded quantification (lower bounded quantification) seems to be more useful. Thus we have an extended system, called $F^{\mu}_{\leq\geq}$, that also supports lower bounded quantification. Polymorphic datatype constructors become typeable with the structural folding rule. For example, we can encode the polymorphic Add constructor as:

```
function Add∀ [E ≥ Exp₁] (e₁ : E, e₂ : E) = fold [E] (Λ A. λ e. (e.add e₁ e₂))
```

Other polymorphic constructors such as $\mathsf{Num}_\forall$ and $\mathsf{Sub}_\forall$ can be encoded similarly, enabling more useful programming patterns. For example, if we want to implement a compiler that uses Exp_1 as its core language, but also want to support richer datatype constructors in a source language like Exp_2 does, we would like to be able to reduce code duplication across the two similar languages. For instance, if we define a pretty printer function for Exp_2

```
function print (e: Exp₂) = (unfold [Exp₂] e) [string] {
  num = λ n. ( int_to_string n ), add = λ e₁ e₂. ((print e₁) ++ "+" ++ ( print e₂)),
  sub = λ e₁ e₂. ((print e₁) ++ "−" ++ ( print e₂)), neg = λ e. ("−" ++ ( print e ))}
```

we can use this function to print Exp_1 expressions as well: all the constructors in Exp_1 are also in Exp_2 and have their pretty printing methods defined in the above function. With subtyping between algebraic datatypes, it holds that $\mathsf{Exp}_1 \leq \mathsf{Exp}_2$, so it is safe in our encodings to apply this print function to values of type Exp_1, without another pretty printing function for Exp_1.

Suppose also that we wish to implement a simple desugaring function that transforms Exp_2 into Exp_1, by transforming negative numbers $-n$ into subtractions $0 - n$. This function should do case analysis on Exp_2 and use *only* the constructors in Exp_1 to produce the result, i.e. it should have a type $\mathsf{Exp}_2 \to \mathsf{Exp}_1$. The following code, with polymorphic constructors, has the desired typing:

```
function desugar (e: Exp₂) = (unfold [Exp₂] e) [Exp₁] {
  num = λ n. Num∀ [Exp₁] n,
  add = λ e₁ e₂. Add∀ [Exp₁] (desugar e₁) (desugar e₂),
  sub = λ e₁ e₂. Sub∀ [Exp₁] (desugar e₁) (desugar e₂),
  neg = λ e. Sub∀ [Exp₁] (Num∀ [Exp₁] 0) (desugar e)}
```

In contrast, in many practical programming languages this task either involves code duplication or loss of type precision. In a typical functional language, we can define both Exp_1 and Exp_2 and also obtain precise static typing guarantees for the desugar function. But this comes at the cost of duplication, since the constructors for the two datatypes are different, and many operations, such as pretty printing need to be essentially duplicated. In $F^{\mu}_{\leq\geq}$, in addition to polymorphic constructors, we would just need to define the pretty printer for Exp_2, and that function would also work for Exp_1. Alternatively, one could define only Exp_2 and type desugar with the imprecise type $\mathsf{Exp}_2 \to \mathsf{Exp}_2$, which does not statically guarantee that the Neg constructor has been removed. This solution avoids the duplication at the cost of static typing guarantees. In $F^{\mu}_{\leq\geq}$ we do not need such compromise: we can avoid code duplication and preserve the static typing guarantees.

2.3 Key Ideas and Results

As Table 1 shows, no previous calculi with bounded quantification and recursive types are fully satisfactory in all dimensions. Equi-recursive types are quite problematic, since they can change the expressive power of the subtyping relation in unexpected ways. More importantly, adding equi-recursive subtyping to $F_{\leq}$ requires novel algorithms, and the extension is non-modular, requiring several changes to existing definitions and proofs.

Kernel $F_{\leq}$ with iso-recursive types. Our type system directly combines kernel $F_{\leq}$ and the nominal unfolding rules together. The addition of the nominal unfolding rules has almost no effect in the original proofs in kernel $F_{\leq}$. That is the proofs for important lemmas, such as transitivity, are nearly the same as those in kernel $F_{\leq}$, except that we need a new case to deal with recursive types. Thus proofs that have been very hard in the past, such as transitivity, are very simple in $F^{\mu}_{\leq}$.

The more challenging aspect in the metatheory of $F^{\mu}_{\leq}$ lies in the *unfolding lemma*:

$$\Gamma \vdash \mu\alpha.\, A \leq \mu\alpha.\, B \quad \Rightarrow \quad \Gamma \vdash [\alpha \mapsto \mu\alpha.\, A]\, A \leq [\alpha \mapsto \mu\alpha.\, B]\, B$$

which reveals an important property for iso-recursive types: if two iso-recursive types are subtypes, then their one-step unfoldings are also subtypes. To prove the unfolding lemma, a generalized lemma is needed [Zhou et al. 2022]. In $F^{\mu}_{\leq}$, we show that the previous generalized approach is insufficient, due to bounded quantification. Therefore, a more general lemma is proposed.

Another challenge is decidability. Although both kernel $F_{\leq}$ and the nominal unfolding rules (for simple calculi) have been independently proved decidable, their decidability proofs use very different measures. A natural combination is problematic, thus we need a new approach.

After overcoming those challenges, as Table 1 shows, $F^{\mu}_{\leq}$ performs well in various dimensions: it is transitive, decidable, conservative and modular. Furthermore, there is a simple, sound and complete algorithmic type system to enable implementations, and to provide important help in the proofs of results such as conservativity of typing.

Structural folding and unfolding rules. In our work, instead of standard rules for fold/unfold expressions, we use *structural rules*:

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

The key point about the structural rules is that the annotations are generalized to be a *subtype/supertype* of a recursive type, instead of exactly a recursive type. In particular, this generalization enables annotating fold/unfold with a bounded type variable, which is a subtype/supertype of a recursive type. This is forbidden in the traditional rules. In the rule TYPING-SUNFOLD, it is worthwhile to mention that when we have $A \leq \mu\alpha.\, B$ where α appears negatively in B, then there are very limited choices to what A can be. Essentially it can be $\mu\alpha.\, B$ itself and little else. In other words, negative

recursive types have very restricted subtyping, which is why the structural unfolding rule can be type safe. Note also that, since the structural unfolding rules provide almost no flexibility for negative recursive subtyping, they are insufficient to fully express f-bounded quantification for negative recursive types.

The structural unfolding rule was presented by Abadi et al. [1996] for supporting *structural update* in the object calculus that was being encoded into $F_{\leq}$ with iso-recursive types. In their work, the structural unfolding rule is presented with an informal explanation. We provide structural rules for both expressions, together with the formalization of the type soundness for both rules. With the structural unfolding rule we can, for instance, obtain the desired typing for the translate function.

$$\dots \frac{\dots \quad \dfrac{\mathsf{P} \leq \mathsf{Point},\ \mathsf{p} : \mathsf{P} \vdash \mathsf{p} : \mathsf{P} \qquad \mathsf{P} \leq \mathsf{Point},\ \mathsf{p} : \mathsf{P} \vdash \mathsf{P} \leq \mathsf{Point}}{\mathsf{P} \leq \mathsf{Point},\ \mathsf{p} : \mathsf{P} \vdash (\mathsf{unfold}\ [\mathsf{P}]\ \mathsf{p}) : \left\{ \begin{array}{l} \mathsf{x} : \mathsf{Int},\ \mathsf{y} : \mathsf{Int}, \\ \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \mathsf{P} \end{array} \right\}}\ \textsc{typing-sunfold}}{\vdash \mathsf{translate} : \forall(\mathsf{P} \leq \mathsf{Point}).\ \mathsf{P} \to \mathsf{P}}$$

Readers can compare this derivation to the one in §2.2, where the conventional unfolding rule and the subsumption rule are used instead. The use of rule TYPING-SUNFOLD enables us to give a more precise type for the translate function.

Lower bounded quantification and $F^{\mu}_{\leq\geq}$. We have also formalized an extension of $F^{\mu}_{\leq}$ with both upper and lower bounded quantification, called $F^{\mu}_{\leq\geq}$. All the same results that are proved for $F^{\mu}_{\leq}$ are also proved for $F^{\mu}_{\leq\geq}$, including transitivity, decidability and type soundness. The structural folding rules become more useful in $F^{\mu}_{\leq\geq}$. With lower bounded quantification and the structural folding rules we can get the correct typing for the polymorphic Add constructor:

$$\dots \frac{\dots \quad \dfrac{\dfrac{}{\mathsf{E} \geq \mathsf{Exp}_1,\ \mathsf{e}_1 : \mathsf{E},\ \mathsf{e}_2 : \mathsf{E} \vdash \Lambda\mathsf{A}.\ \lambda\mathsf{e}.(\mathsf{e.add}\ \mathsf{e}_1\ \mathsf{e}_2) : \forall\mathsf{A}. \left\{ \begin{array}{l} \mathsf{num} : \mathsf{Int} \to \mathsf{A}, \\ \mathsf{add} : \mathsf{E} \to \mathsf{E} \to \mathsf{A}, \\ \mathsf{sub} : \mathsf{E} \to \mathsf{E} \to \mathsf{A} \end{array} \right\} \to \mathsf{A}}\ \textsc{typing-sfold}}{\mathsf{E} \geq \mathsf{Exp}_1,\ \mathsf{e}_1 : \mathsf{E},\ \mathsf{e}_2 : \mathsf{E} \vdash \mathsf{fold}\ [\mathsf{E}]\ (\Lambda\mathsf{A}.\ \lambda\mathsf{e}.\ (\mathsf{e.add}\ \mathsf{e}_1\ \mathsf{e}_2)) : \mathsf{E}}}{\vdash \mathsf{Add}_{\forall} : \forall(\mathsf{E} \geq \mathsf{Exp}_1).\ \mathsf{E} \to \mathsf{E} \to \mathsf{E}}$$

3 BOUNDED QUANTIFICATION WITH ISO-RECURSIVE TYPES

This section introduces a full calculus, called $F^{\mu}_{\leq}$, with bounded quantification, records and recursive types. $F^{\mu}_{\leq}$ is an extension of kernel $F_{\leq}$ [Cardelli and Wegner 1985] with iso-recursive types.

3.1 Syntax and Well-Formedness

Syntax and Well-Formedness. The syntax of types and contexts for $F^{\mu}_{\leq}$ is shown below.

Types	$A, B, \dots$	$::=$	$\mathsf{nat} \mid \top \mid A_1 \to A_2 \mid \alpha \mid \mu\alpha.\ A \mid A^{\alpha} \mid \forall(\alpha \leq A).\ B \mid \{l_i : A_i\ ^{i \in 1 \cdots n}\}$
Expressions	e	$::=$	$x \mid \mathsf{i} \mid e_1\ e_2 \mid \lambda x : A.\ e \mid e\ A \mid \Lambda(\alpha \leq A).\ e$
			$\mid \mathsf{unfold}\ [A]\ e \mid \mathsf{fold}\ [A]\ e \mid \{l_i = e_i\ ^{i \in 1 \cdots n}\} \mid e.l$
Values	v	$::=$	$\mathsf{i} \mid \lambda x : A.\ e \mid \mathsf{fold}\ [A]\ v \mid \Lambda(\alpha \leq A).\ e \mid \{l_i = v_i\ ^{i \in 1 \cdots n}\}$
Contexts	Γ	$::=$	$\cdot \mid \Gamma, \alpha \leq A \mid \Gamma, x : A$

Meta-variables A, B, C, D range over types. Types consist of: natural numbers (nat), the top type ($\top$), function types ($A \to B$), type variables (α), recursive types ($\mu\alpha.\ A$), labelled types (A^{α}), universal types ($\forall(\alpha \leq A).\ B$), and record types ($\{l_i : A_i\ ^{i \in 1 \cdots n}\}$). Labelled types are types that are annotated with a label. They enable distinguishing between otherwise structurally compatible types (equal types or subtypes). That is if the two types being compared have different labels or one of the types is unlabelled, then the two types will not be related, even when (ignoring the labels) they would be

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

$$\textsc{wft-nat}\ \frac{}{\Gamma \vdash \mathsf{nat}} \qquad \textsc{wft-Top}\ \frac{}{\Gamma \vdash \top} \qquad \textsc{wft-var}\ \frac{\alpha \leq A \in \Gamma}{\Gamma \vdash \alpha} \qquad \textsc{wft-all}\ \frac{\Gamma \vdash A \qquad \Gamma,\ \alpha \leq A \vdash B}{\Gamma \vdash \forall(\alpha \leq A).\ B} \qquad \textsc{wft-arrow}\ \frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \to A_2}$$

$$\textsc{wft-rec}\ \frac{\Gamma,\ \alpha \leq \top \vdash A}{\Gamma \vdash \mu\alpha.\ A} \qquad \textsc{wft-label}\ \frac{\Gamma \vdash A}{\Gamma \vdash A^{\alpha}} \qquad \textsc{wft-rcd}\ \frac{\Gamma \vdash A_i \qquad \text{for each } i}{\Gamma \vdash \{l_i : A_i\ ^{i \in 1 \cdots n}\}}$$

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

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

$$\textsc{S-rec}\ \frac{\Gamma,\ \alpha \leq \top \vdash [\alpha \mapsto A^{\alpha}]\ A \leq [\alpha \mapsto B^{\alpha}]\ B}{\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B} \qquad \textsc{S-vartrans}\ \frac{\alpha \leq B \in \Gamma \qquad \Gamma \vdash B \leq A}{\Gamma \vdash \alpha \leq A}$$

$$\textsc{S-equivall}\ \frac{\Gamma \vdash A_1 \leq A_2 \qquad \Gamma \vdash A_2 \leq A_1 \qquad \Gamma,\ \alpha \leq A_2 \vdash B \leq C}{\Gamma \vdash \forall(\alpha \leq A_1).\ B \leq \forall(\alpha \leq A_2).\ C} \qquad \textsc{S-label}\ \frac{\Gamma \vdash A \leq B}{\Gamma \vdash A^{\alpha} \leq B^{\alpha}}$$

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

Fig. 1. Well-formedness and subtyping rules.

structurally compatible types. They are used for dealing with subtyping of iso-recursive types as part of the nominal unfolding approach. Expressions, denoted by meta-variable e, include: term variables (x), natural numbers (i), applications ($e_1\ e_2$), abstractions ($\lambda x : A.\ e$), type applications ($e\ A$), type abstractions ($\Lambda(\alpha \leq A).\ e$), fold expressions (fold $[A]\ e$), unfold expressions (unfold $[A]\ e$), records ($\{l_i = e_i\ ^{i \in 1 \cdots n}\}$) and record selection ($e.l$). Among them, natural numbers, abstractions and type abstractions are values. Fold expressions and records can be values if their inner expressions are also values. The context is used to store type variables with their bounds, and term variables with their types. Note that it is unnecessary to distinguish recursive variables and universal variables.

The definition of a well-formed environment $\vdash \Gamma$ is standard, ensuring that all variables in the environment are distinct and all types in the environment are well-formed. A type is well-formed if all of its free variables are in the context. The well-formedness rules for types are shown in Figure 1.

3.2 Subtyping

The bottom of Figure 1 shows the subtyping judgement. Our subtyping rules are mostly standard. The rules essentially include the rules of the algorithmic version of kernel $F_{\leq}$ [Cardelli et al. 1994; Cardelli and Wegner 1985], but the rule for bounded quantification is generalized. The rules S-VAR and S-VARTRANS are standard $F_{\leq}$ rules. Note that since we do not distinguish universal and recursive variables, those rules apply also to recursive type variables. The rule for function types (rule S-ARROW) is contravariant on the input types and covariant on the output types.

Subtyping bounded quantification. The rule for bounded quantification is interesting, stating that two universal types are subtypes if their bounds are equivalent (i.e. they are subtypes of each other) and the bodies are subtypes. Note that rule S-EQUIVALL is more general than rule S-KERNELALL since the latter one requires the bounds are equal. The reason to have the more general rule using equivalent bounds is that, for records, we wish to accept subtyping statements such as:

$$\forall(\alpha \leq \{x : \mathsf{nat}, y : \mathsf{nat}\}).\ \alpha \to \alpha \leq \forall(\alpha \leq \{y : \mathsf{nat}, x : \mathsf{nat}\}).\ \alpha \to \alpha$$

where the bounds can be syntactically different, but equivalent types. In the presence of records or other features (such as intersection and union types [Barbanera et al. 1995; Coppo et al. 1981; Pottinger 1980]) we can have such equivalent, but not syntactically equal types. Therefore, we should generalize the rule for bounded quantification do deal with those cases. This generalization to equivalent bounds retains decidable subtyping just as kernel $F_{\leq}$ as we shall see in §4.2.

Subtyping recursive types. For dealing with iso-recursive subtyping we employ the recent nominal unfolding rules [Zhou et al. 2022], which have equivalent expressive power to the well-known (iso-recursive) Amber rules [Cardelli 1985]. The nominal unfolding rules have been discussed in §2.1. The reason to choose the nominal unfolding rules is that they enable us to prove important metatheoretical results, such as transitivity and develop an algorithmic formulation of subtyping.

We extend the rule S-NOMINAL to the rule S-REC in $F^{\mu}_{\leq}$, by bounding recursive variables with $\top$ when they are introduced into the context. Therefore, recursive variables are also treated as universal variables, and we do not need to adjust the form of contexts in $F_{\leq}$ for $F^{\mu}_{\leq}$. Apart from this, no other changes are necessary, making the addition of recursive types mostly non-invasive. Consequently, the proofs of narrowing, reflexivity and transitivity are the same as the original one for $F_{\leq}$, except for the new cases dealing with recursive types and minor adjustments to the rule of bounded quantification due to the generalization to equivalent bounds.

Lemma 3.1 (Narrowing). If $\Gamma_1 \vdash C \leq C'$ and $\Gamma_1, \alpha \leq C', \Gamma_2 \vdash A \leq B$ then $\Gamma_1, \alpha \leq C, \Gamma_2 \vdash A \leq B$.

Theorem 3.2 (Reflexivity). If $\vdash \Gamma$ and $\Gamma \vdash A$ then $\Gamma \vdash A \leq A$.

Theorem 3.3 (Transitivity). If $\Gamma \vdash A \leq B$ and $\Gamma \vdash B \leq C$ then $\Gamma \vdash A \leq C$.

The unfolding lemma. Another important lemma is the *unfolding lemma*, which reveals that: if two recursive types are subtypes, then their unfoldings are also subtypes. The unfolding lemma is important for proving preservation in a system with iso-recursive subtyping. A key difficulty in the formalization of $F^{\mu}_{\leq}$ is proving the unfolding lemma which, due to the presence of bounded quantification, requires a different proof technique compared to the proofs by Zhou et al. [2022]. We discuss the proof of the unfolding lemma in §4.1.

Lemma 3.4 (Unfolding Lemma). If $\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B$ then $\Gamma \vdash [\alpha \mapsto \mu\alpha.\ A]\ A \leq [\alpha \mapsto \mu\alpha.\ B]\ B$.

3.3 Type Soundness

Figure 2 shows the typing rules and reduction rules. Most rules are standard except for the typing rule for unfold and fold. For these two expressions we use structural rules instead (rule TYPING-SUNFOLD and rule TYPING-SFOLD), as we explained in §2.3.

Structural unfolding lemma. Since the typing rules that we adopt for fold/unfold expressions are the structural rules, which generalize the conventional rules, we need a more general form for the unfolding lemma. The generalization of the lemma is necessary for the type preservation proof with the structural folding/unfolding rules. We call the new lemma the *structural unfolding lemma*:

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

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

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

$$\textsc{typing-app}\ \frac{\Gamma \vdash e_1 : A_1 \to A_2 \qquad \Gamma \vdash e_2 : A_1}{\Gamma \vdash e_1\ e_2 : A_2} \qquad \textsc{typing-proj}\ \frac{\Gamma \vdash e : \{l_i : A_i\ ^{i \in 1 \cdots n}\}}{\Gamma \vdash e.l_i : A_i} \qquad \textsc{typing-tabs}\ \frac{\Gamma,\ \alpha \leq A \vdash e : B}{\Gamma \vdash \Lambda(\alpha \leq A).\ e : \forall(\alpha \leq A).\ B}$$

$$\textsc{typing-tapp}\ \frac{\Gamma \vdash e : \forall(\alpha \leq B_1).\ B_2 \qquad \Gamma \vdash A \leq B_1}{\Gamma \vdash e\ A : [\alpha \mapsto A]\ B_2} \qquad \textsc{typing-rcd}\ \frac{\text{for each } i \quad \Gamma \vdash e_i : A_i}{\Gamma \vdash \{l_i = e_i\ ^{i \in 1 \cdots n}\} : \{l_i : A_i\ ^{i \in 1 \cdots n}\}}$$

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

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

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

$$\textsc{step-tapp}\ \frac{e_1 \hookrightarrow e_2}{e_1\ A \hookrightarrow e_2\ A} \qquad \textsc{step-tabs}\ \frac{}{(\Lambda(\alpha \leq A).\ e)\ B \hookrightarrow [\alpha \mapsto B]\ e} \qquad \textsc{step-proj}\ \frac{e \hookrightarrow e'}{e.l_j \hookrightarrow e'.l_j} \qquad \textsc{step-projrcd}\ \frac{}{\{l_i = v_i\ ^{i \in 1 \cdots n}\}.l_j \hookrightarrow v_j}$$

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

Fig. 2. Typing and Reduction Rules

Lemma 3.5 (Structural unfolding lemma). If $\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ C \leq \mu\alpha.\ D \leq \mu\alpha.\ B$ then $\Gamma \vdash [\alpha \mapsto \mu\alpha.\ C]\ A \leq [\alpha \mapsto \mu\alpha.\ D]\ B$.

In this lemma, in the one-step unfolding the recursive types substituted in the bodies are, respectively, a supertype and a subtype of $\mu\alpha.\ A$ and $\mu\alpha.\ B$. In contrast, in the unfolding lemma proposed by Zhou et al., the recursive types that get substituted in the bodies are the same. As §4.1 will discuss, both forms of the unfolding lemma can be proved using a more general lemma.

Type Soundness. To see how the structural unfolding lemma is used in the proof of type preservation, let us consider an expression $\mathsf{unfold}[D'](\mathsf{fold}[C']\ e)$. Starting from a closed expression, both C' and D' must be recursive types, thus we assume that C' is $\mu\alpha.\ C$ and D' is $\mu\alpha.\ D$,

$\boxed{\Gamma \vdash A \Uparrow B}$ *(Upper Exposure)*

$$\textsc{XA-promote}\ \frac{\alpha \leq A \in \Gamma \qquad \Gamma \vdash A \Uparrow B}{\Gamma \vdash \alpha \Uparrow B} \qquad \textsc{XA-up}\ \frac{A \text{ is not type variable}}{\Gamma \vdash A \Uparrow A}$$

$\boxed{\Gamma \vdash A \Downarrow B}$ *(Lower Exposure)*

$$\textsc{XA-top}\ \frac{}{\Gamma \vdash \top \Downarrow \mu\alpha.\, \top} \qquad \textsc{XA-down}\ \frac{A \text{ is not type variable or } \top}{\Gamma \vdash A \Downarrow A}$$

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

$$\textsc{atyp-app}\ \frac{\Gamma \vdash_a e_1 : A \qquad \Gamma \vdash A \Uparrow A_1 \to A_2 \qquad \Gamma \vdash_a e_2 : B \qquad \Gamma \vdash B \leq A_1}{\Gamma \vdash_a e_1\ e_2 : A_2}$$

$$\textsc{atyp-tapp}\ \frac{\Gamma \vdash_a e : B \qquad \Gamma \vdash B \Uparrow \forall(\alpha \leq B_1).\, B_2 \qquad \Gamma \vdash A \leq B_1}{\Gamma \vdash_a e\ A : [\alpha \mapsto A]\ B_2}$$

$$\textsc{atyp-sunfold}\ \frac{\Gamma \vdash_a e : A \qquad \Gamma \vdash B \Uparrow \mu\alpha.\, C \qquad \Gamma \vdash A \leq B}{\Gamma \vdash_a \mathsf{unfold}\ [B]\ e : [\alpha \mapsto B]\ C}$$

$$\textsc{atyp-sfold}\ \frac{\Gamma \vdash_a e : A \qquad \Gamma \vdash C \Downarrow \mu\alpha.\, B \qquad \Gamma \vdash A \leq [\alpha \mapsto C]\ B \qquad \Gamma \vdash C}{\Gamma \vdash_a \mathsf{fold}\ [C]\ e : C}$$

Fig. 3. Algorithmic Typing.

$$\textsc{typing-sunfold}\ \frac{\textsc{typing-sub}\ \dfrac{\textsc{typing-sfold}\ \dfrac{\Gamma \vdash e : [\alpha \mapsto C']\ A \qquad \Gamma \vdash \mu\alpha.\, A \leq C'}{\Gamma \vdash \mathsf{fold}[C']\ e : C'} \qquad \Gamma \vdash C' \leq D'}{\Gamma \vdash \mathsf{fold}[C']\ e : D'} \qquad \Gamma \vdash D' \leq \mu\alpha.\, B}{\Gamma \vdash \mathsf{unfold}[D'](\mathsf{fold}[C']\ e) : [\alpha \mapsto D']\ B}$$

The type of $\mathsf{unfold}[D'](\mathsf{fold}[C']\ e)$ becomes $[\alpha \mapsto \mu\alpha.\, D]\ B$, and it should be a subtype of $[\alpha \mapsto \mu\alpha.\, C]\ A$, which is the type of reduction result e.

The other parts of the type soundness proof are standard, thus we have:

Theorem 3.6 (Preservation). If $\vdash e : A$ and $e \hookrightarrow e'$ then $\vdash e' : A$.

Theorem 3.7 (Progress). If $\vdash e : A$ then e is a value or exists e', $e \hookrightarrow e'$.

3.4 Algorithmic Typing

The rules that we have presented in Figure 2 are declarative. The conclusion of the subsumption rule overlaps with all other rules, making it non-trivial to derive an implementation from the rules.

Figure 3 shows the algorithmic rules for typing. We only present new rules and rules that differ from Figure 2. Compared with the declarative typing rules, the subsumption rule (TYPING-SUB) is removed. Besides, application (TYPING-APP), type application (TYPING-TAPP), structural folding (TYPING-SFOLD) and structural unfolding (TYPING-SUNFOLD) rules are replaced by rules ATYP-APP,

ATYP-TAPP, ATYP-SFOLD and ATYP-SUNFOLD, respectively. In the algorithmic typing rules we take the standard approach of distributing subtyping checks in appropriate places in the other rules, thus eliminating the need for the subsumption rule.

One interesting point is the two exposure relations $\Uparrow$ and $\Downarrow$ in $F^{\mu}_{\leq}$. In $F_{\leq}$, there is only the *upper exposure* function ($\Gamma \vdash A \Uparrow B$), which is used to find the least non-variable upper bound for a variable in the context [Pierce 2002]. Thus the upper exposure function plays an important role for finding the minimal type with the algorithmic typing rules. To make our rules more general, we additionally define the *lower exposure* function ($\Gamma \vdash A \Downarrow B$) to find the greatest non-variable subtype B for A. For $F^{\mu}_{\leq}$, lower exposure only helps to find the correct shape for the recursive type body to be folded in rule ATYP-SFOLD by mapping $\top$ to $\mu\alpha.\ \top$. The lower exposure function will be more useful when we have lower bounded variables in the system, as we will see in §5.

The algorithmic rules are equivalent (sound and complete) with respect to the declarative rules:

Theorem 3.8 (Soundness of the algorithmic rules). If $\Gamma \vdash_a e : A$ then $\Gamma \vdash e : A$.

Theorem 3.9 (Completeness of the algorithmic rules). If $\Gamma \vdash e : A$ then there exists a B such that $\Gamma \vdash_a e : B$ and $\Gamma \vdash B \leq A$.

Theorem 3.9 implies that our algorithm can always find a minimal type, which is an important property for $F^{\mu}_{\leq}$.

4 METATHEORY OF $F^{\mu}_{\leq}$

In this section we discuss the most interesting and difficult aspects of the metatheory of $F^{\mu}_{\leq}$ in more detail. We cover three key properties: the *unfolding lemma*, *decidability* of subtyping and the *conservativity* of $F^{\mu}_{\leq}$ over the original $F_{\leq}$. The interaction between iso-recursive types and bounded quantification requires significant changes in the proofs of the unfolding lemma and decidability. In addition, conservativity cannot be proved using a declarative formulation of $F^{\mu}_{\leq}$, and we need to employ the algorithmic formulation instead.

4.1 Unfolding Lemma

The unfolding lemma (Lemma 3.4) is a core lemma for the metatheory of a calculus with iso-recursive subtyping. Though the statement of the unfolding lemma is quite simple and intuitive to understand, the lemma cannot be proved directly. We will firstly review the previous approach to prove the unfolding lemma, which does not account for bounded quantification, and then show how to transfer this approach to a system with bounded quantification.

The previous approach for proving the unfolding lemma. Because the premise of the unfolding lemma is a subtyping relation between two recursive types $\mu\alpha.\ A \leq \mu\alpha.\ B$, a direct induction on such subtyping relation is problematic. Thus, we need to prove a generalized lemma first. In Zhou et al.'s work that lemma has the following form:

Lemma 4.1 (The generalized unfolding lemma in Zhou et al. [2022]). If $\Gamma_1, \alpha, \Gamma_2, \vdash A \leq B$ and $\Gamma_1 \vdash \mu\alpha.\ C \leq \mu\alpha.\ D$ then

(1) $\Gamma_1, \alpha, \Gamma_2 \vdash [\alpha \mapsto C^{\alpha}]\ A \leq [\alpha \mapsto D^{\alpha}]\ B$ implies $\Gamma_1, \Gamma_2 \vdash [\alpha \mapsto \mu\alpha.\ C]\ A \leq [\alpha \mapsto \mu\alpha.\ D]\ B$;
(2) $\Gamma_1, \alpha, \Gamma_2 \vdash [\alpha \mapsto D^{\alpha}]\ A \leq [\alpha \mapsto C^{\alpha}]\ B$ implies $\Gamma_1, \Gamma_2 \vdash [\alpha \mapsto \mu\alpha.\ D]\ A \leq [\alpha \mapsto \mu\alpha.\ C]\ B$.

Due to the tricky interaction between rule S-VAR and rule S-ARROW, in the generalized unfolding lemma we need two mutual dependent lemmas: one is used for covariant cases (1) and the other one is used for contravariant cases (2). The proof for this lemma proceeds by induction on $\Gamma_1, \alpha, \Gamma_2, \vdash A \leq B$. In the inductive proof we need to switch between covariance and contravariance. In particular, in

the rule S-ARROW case for functions, we need an induction hypothesis that arises from conclusion (2) to prove the contravariant premise $\Gamma \vdash B_1 \leq A_1$ relating the input types of the function.

For the generalized unfolding lemma in $F^{\mu}_{\leq}$, Lemma 4.1 is unfortunately not general enough. In a setting with bounded quantification, Γ_2 may contain bounds with the type variable α, and those variables are not being substituted in Lemma 4.1. Let us consider the effect of adding rule S-VARTRANS. In the conclusion of Lemma 4.1, the variable α is substituted by another type and tracked in the context Γ_2. In the premises of rule S-VARTRANS, we need to find the upper bound of variable α in the contexts Γ_1 and Γ_2. With those two observations, in our new attempt, the context also needs to do substitutions. Thus, a natural attempt to solve this problem is to reformulate the lemma into the following form (for the covariant case (1)):

Proposition 4.2 (A first attempt at the generalized unfolding lemma). If $\Gamma_1,\ \alpha \leq \top,\ \Gamma_2 \vdash A \leq B$ and $\Gamma_1 \vdash \mu\alpha.\ C \leq \mu\alpha.\ D$ then $\Gamma_1,\ \alpha \leq \top,\ \Gamma_2[\alpha \mapsto ?^{\alpha}] \vdash [\alpha \mapsto C^{\alpha}]\ A \leq [\alpha \mapsto D^{\alpha}]\ B$ implies $\Gamma_1,\ \Gamma_2[\alpha \mapsto \mu\alpha.\ ?] \vdash [\alpha \mapsto \mu\alpha.\ C]\ A \leq [\alpha \mapsto \mu\alpha.\ D]\ B$.

Here, the syntax $\Gamma[\alpha \mapsto S]$ denotes that all the occurrences of α in context Γ will be replaced by a specified type S. However, we do not know yet what type should be filled in the hole ? in Proposition 4.2, so we leave the hole there for now. Although we omit conclusion (2) in Proposition 4.2, similar reasoning applies to that conclusion.

Let us now consider the effect of adding the rule S-EQUIVALL: Assume that $A := \forall(\beta \leq U_1).\ T_1$ and $B := \forall(\beta \leq U_2).\ T_2$. The goal would look like:

$$\Gamma_1, \Gamma_2[\alpha \mapsto \mu\alpha.\ ?] \vdash [\alpha \mapsto \mu\alpha.\ C]\ \forall(\beta \leq U_1).\ T_1 \leq [\alpha \mapsto \mu\alpha.\ D]\ \forall(\beta \leq U_2).\ T_2$$

After simplification and applying rule S-EQUIVALL, one of the goals becomes:

$$\Gamma_1, \Gamma_2[\alpha \mapsto \mu\alpha.\ ?],\ \beta \leq [\alpha \mapsto \mu\alpha.\ D]\ U_2 \vdash [\alpha \mapsto \mu\alpha.\ C]\ T_1 \leq [\alpha \mapsto \mu\alpha.\ D]\ T_2$$

From the above, we would expect that the hole ? is filled with D because all the substitutions in the context must be the same in order to apply induction hypothesis. Thus, a second attempt at the generalized unfolding lemma looks like:

Proposition 4.3 (The second attempt at the generalized unfolding lemma (1)). If $\Gamma_1,\ \alpha \leq \top,\ \Gamma_2 \vdash A \leq B$ and $\Gamma_1 \vdash \mu\alpha.\ C \leq \mu\alpha.\ D$ then $\Gamma_1,\ \alpha \leq \top,\ \Gamma_2[\alpha \mapsto D^{\alpha}] \vdash [\alpha \mapsto C^{\alpha}]\ A \leq [\alpha \mapsto D^{\alpha}]\ B$ implies $\Gamma_1,\ \Gamma_2[\alpha \mapsto \mu\alpha.\ D] \vdash [\alpha \mapsto \mu\alpha.\ C]\ A \leq [\alpha \mapsto \mu\alpha.\ D]\ B$.

Proposition 4.3 deals with rule S-VARTRANS and S-EQUIVALL successfully. However, the function case, which is correctly proven in Lemma 4.1, will break. Consider $A := A_1 \to A_2$ and $B := B_1 \to B_2$, and apply rule S-ARROW. We need to prove two subgoals:

(1) $\Gamma_1, \Gamma_2[\alpha \mapsto \mu\alpha.\ D] \vdash [\alpha \mapsto \mu\alpha.\ C]\ A_2 \leq [\alpha \mapsto \mu\alpha.\ D]\ B_2$;
(2) $\Gamma_1, \Gamma_2[\alpha \mapsto \mu\alpha.\ D] \vdash [\alpha \mapsto \mu\alpha.\ D]\ B_1 \leq [\alpha \mapsto \mu\alpha.\ C]\ A_1$.

Note that we do not have any induction hypothesis for proving subgoal (2) because occurrences of α in Γ_2 have been substituted by $\mu\alpha.\ D$, but we expect the α's to have been replaced by $\mu\alpha.\ C$ for applying the induction hypothesis. Even if we add the second conclusion back to the Proposition 4.3, we still have problems. For conclusion (2), the type used for the substitution in the context should be same as the type used for the substitution in the right-hand side of the subtyping. If we fill the hole ? with C in Proposition 4.2, the subgoal (1) in case S-ARROW will get stuck for a similar reason. Therefore, the type in the hole ? cannot be the type C or D.

In summary, in the previous approach by Zhou et al., without bounded quantification, only the interaction of covariance/contravariance between types has to be considered. In contrast, with bounded quantification, the interaction of covariance/contravariance among contexts and types also needs to be considered. Our generalization should be able to deal with all the complications arising from rule S-VAR, rule S-VARTRANS, rule S-ARROW and rule S-EQUIVALL.

The generalized unfolding lemma for $F^{\mu}_{\leq}$. Surprisingly, the generalization is relatively straightforward. For solving the issue that we mention above, instead of picking type C or type D, we pick an intermediate type S between C and D. We need the following auxiliary lemma:

Lemma 4.4. If $\Gamma,\ \alpha \leq \top \vdash [\alpha \mapsto A^{\alpha}]\ A \leq [\alpha \mapsto B^{\alpha}]\ B$ then $\Gamma,\ \alpha \leq \top \vdash A \leq B$.

The generalized unfolding lemma for $F^{\mu}_{\leq}$ is:

Lemma 4.5 (The generalized unfolding lemma for $F^{\mu}_{\leq}$). If (1) $\Gamma_1,\ \alpha \leq \top,\ \Gamma_2 \vdash A \leq B$, (2) $\Gamma_1 \vdash \mu\alpha.\ C \leq \mu\alpha.\ S$ and (3) $\Gamma_1 \vdash \mu\alpha.\ S \leq \mu\alpha.\ D$ then

(1) $\Gamma_1,\ \alpha \leq \top,\ \Gamma_2[\alpha \mapsto S^{\alpha}] \vdash [\alpha \mapsto C^{\alpha}]\ A \leq [\alpha \mapsto D^{\alpha}]\ B$ implies $\Gamma_1,\ \Gamma_2[\alpha \mapsto \mu\alpha.\ S] \vdash [\alpha \mapsto \mu\alpha.\ C]\ A \leq [\alpha \mapsto \mu\alpha.\ D]\ B$;
(2) $\Gamma_1,\ \alpha \leq \top,\ \Gamma_2[\alpha \mapsto S^{\alpha}] \vdash [\alpha \mapsto D^{\alpha}]\ A \leq [\alpha \mapsto C^{\alpha}]\ B$ implies $\Gamma_1,\ \Gamma_2[\alpha \mapsto \mu\alpha.\ S] \vdash [\alpha \mapsto \mu\alpha.\ D]\ A \leq [\alpha \mapsto \mu\alpha.\ C]\ B$.

By applying Lemma 4.5 with $\Gamma_1 = \Gamma$, $\Gamma_2 = \cdot$, $S = A$, $C = A$ and $D = B$, we prove the unfolding lemma (Lemma 3.4). The premises can be obtained by inversion on $\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B$, and then applying Lemma 4.4. Next we give an overview of the key points in the proof of Lemma 4.5 for the four tricky cases. Assume that we do induction on premise (1).

- Rule S-VAR: This case mostly follows the previous approach. The interesting point is that we cannot obtain $\mu\alpha.\ C \leq \mu\alpha.\ D$ directly from the conditions raised by the goals, but have one extra step via transitivity (Theorem 3.3).
- Rule S-VARTRANS: For this case we need to consider two subcases. If $A = \alpha$ then from the context $\alpha \leq \top$ we can easily know that B is $\top$ too. Otherwise, we apply the induction hypothesis directly.
- Rule S-ARROW: This case is surprisingly easy: we can follow the previous approach for dealing with the contravariant case for subtyping input types of functions, and apply the induction hypothesis derived from conclusion (2) directly. Note that the key point of avoiding the issue we discussed before is that, by picking an intermediate type S, we decouple the substitution in the context and in the subtyping relation for the function case. That is, the substitution with type $\mu\alpha.\ S$ is invariant in both subgoals, independent of the substitution in the subtyping relation.
- Rule S-EQUIVALL: This case is the most interesting one. Assume $A = \forall(\beta \leq U_1).\ T_1$ and $B = \forall(\beta \leq U_2).\ T_2$. Let us consider how to prove the goal (1). Doing inversion on condition arising from the goal, we obtain that $[\alpha \mapsto C^{\alpha}]\ U_1$ and $[\alpha \mapsto D^{\alpha}]\ U_2$ are equivalent. Meanwhile, we know U_1 and U_2 that are equivalent. There are two possibilities: either C and D are equivalent or α is not in type U_1 nor U_2. For the latter case, we can rewrite the substituted types in the contexts for aligning the contexts, then solve the issue we discussed above. The former case is quite subtle: since S lies in the middle of C and D, the three types are all equivalent. In other words, we can change the substituted types arbitrarily and get the equivalent types and contexts. The critical point is that, although the substitution in the context is indeed affected by the substitution in the supertype, since the bounds are equivalent, the type S will converge into the types C and D.

As a final note, we cannot apply this technique to full $F_{\leq}$. In full $F_{\leq}$, in the context, the type C and D will swap due to the contravariance. How to prove the unfolding lemma for full $F_{\leq}$ is not clear now, and we leave this as the future work.

4.2 Decidability

Decidability is one of the important properties of $F^{\mu}_{\leq}$. We first start by reviewing the approaches to prove decidability in kernel $F_{\leq}$, and nominal unfoldings, and then describe our approach to prove decidability. These two previous approaches to prove decidability employ different measures, which creates a challenge for proving the decidability of $F^{\mu}_{\leq}$.

Decidability of kernel $F_{\leq}$. It is well-known that bounded quantification for full $F_{\leq}$ is undecidable [Pierce 1994]. However, for kernel $F_{\leq}$, identical bounds make the system decidable. A common practice is to define a *weight* function to compute the size of a type [Pierce 2002]:

$$\begin{array}{lcl} weight_{\Gamma}(\top) & = & 1 \\ weight_{\Gamma_1,\alpha\leq A,\Gamma_2}(\alpha) & = & 1 + weight_{\Gamma_1}(A) \\ weight_{\Gamma}(\forall(\alpha \leq A).\ B) & = & 1 + weight_{\Gamma,\alpha\leq A}(B) \\ weight_{\Gamma}(A \to B) & = & 1 + weight_{\Gamma}(A) + weight_{\Gamma}(B) \end{array}$$

For universal types, we store its bound into a context Γ, and when we meet the universal variable, we retrieve its bound from the context and compute the size recursively. Since the size of a conclusion is always greater than any premise, this measure can be used to show that the subtyping algorithm in kernel $F_{\leq}$ terminates for all inputs.

Decidability of nominal unfoldings. The nominal unfolding rule in simple calculi with subtyping is also decidable [Zhou et al. 2022]. Compared with kernel $F_{\leq}$, the decidability proof of nominal unfoldings is trickier. Based on the substitution of the type body, after every unfolding, the size of types will increase. Thus a straightforward induction on the size of types does not work. Zhou et al. choose a size measure based on an over-approximation of the height of the fully unfolded tree. Concretely, they define a *height* function:

$$\begin{array}{lcl} height_{\Psi}(\top) & = & 0 \\ height_{\Psi}(\alpha) & = & \Psi(\alpha) \text{ if } \alpha \in \Psi \text{ else } 0 \\ height_{\Psi}(A \to B) & = & 1 + \max(height_{\Psi}(A), height_{\Psi}(B)) \\ height_{\Psi}(\mu\alpha.\ A) & = & 1 + \text{let } i = height_{\Psi,\alpha\mapsto 0}(A) \text{ in } height_{\Psi,\alpha\mapsto i+1}(A) \end{array}$$

The size measure of a type A is defined as $height(A)$ where the context is empty. In contrast to kernel $F_{\leq}$, the context here is used to store the size of the corresponding recursive variables. Zhou et al. proved that such *height* measure will precisely decrease by one for every nominal unfolding.

Decidability of $F^{\mu}_{\leq}$. Now consider how to combine these two approaches together. We wish to extend the measure of nominal unfoldings with the measure of kernel $F_{\leq}$ non-invasively. The easiest thing to do is to switch the maximum function to addition for the function case in the measure of nominal unfoldings, which simply widens the over-approximation. Then we consider the major differences between the two approaches. There are three main challenges:

- **The measures for variables are inconsistent in the two approaches**: In the *height* function, the type variable case is a base case, while in the *weight* function we will continue the computation for a variable by getting its bound from the contexts. In $F^{\mu}_{\leq}$, a recursive variable is regarded as a universal variable, so the new formulation should reflect those two distinct situations.
- **The purposes of contexts are inconsistent in two approaches**: The context of the *height* function is used to store the pre-computed size of the recursive type body so that measures of recursive variables are counted in their nominal unfolded form. The context of the *weight* function is a straightforward bookkeeping of universal bounds. In later computation, these bounds will be retrieved and their measure will be computed to serve as the measure of a universal variable. This is to ensure that the premises in rule S-VARTRANS have a smaller measure of types than the conclusion does. We need to treat both designs carefully if we want a unified context.
- **The measure information is lost in the bounded quantification case**: Recall that we employ the rule S-EQUIVALL instead of the standard rule S-KERNELALL for $F_{\leq}$. Since we impose equivalent bounds for kernel $F_{\leq}$, for the subtyping relation $\Gamma \vdash \forall(\alpha \leq A_1).\ B_1 \leq \forall(\alpha \leq A_2).\ B_2$, the measure would consist of the measures of type A_1, A_2, B_1 and B_2. However, the measure for the premise $\Gamma, \alpha \leq A_2 \vdash B_1 \leq B_2$ will lose the measure of type A_1 because we do not store it.

We first show the measure used for the decidability of $F^{\mu}_{\leq}$, and then discuss how it addresses the concerns above. The measure is relatively simple and based on the approach from Zhou et al. [2022]. Similarly, we define a context $\Psi := \cdot \mid \Psi, \alpha \mapsto i$ which is used to store the measures of (both universal and recursive) variables during the measure computation, where i represents a natural number. Then, a measure function $size_{\Psi}(A)$, defined on types, is:

$$
\begin{array}{lcl}
size_{\Psi}(\mathsf{nat}) & = & 1 \\
size_{\Psi}(\top) & = & 1 \\
size_{\Psi}(A \to B) & = & 1 + size_{\Psi}(A) + size_{\Psi}(B) \\
size_{\Psi}(A^{\alpha}) & = & 1 + size_{\Psi}(A) \\
size_{\Psi}(\alpha) & = & 1 + \begin{cases} \Psi(\alpha) & \alpha \in \Psi \\ 0 & \alpha \notin \Psi \end{cases} \\
size_{\Psi}(\forall(\alpha \leq A).\ B) & = & \text{let } i := size_{\Psi}(A) \text{ in } 1 + i + size_{\Psi, \alpha \mapsto i}(B) \\
size_{\Psi}(\mu\alpha.\ A) & = & \text{let } i := size_{\Psi, \alpha \mapsto 1}(A) \text{ in } 1 + size_{\Psi, \alpha \mapsto i}(A)
\end{array}
$$

The formulation of the *size* function is very similar to the *height* function. We have an extra rule for universal types, and slightly adjust the variable and recursive cases. The measure of universal types is the sum of the measure of the bound and the measure of the body. For variables, one is added when they are retrieved. Accordingly, we do not need to add one when storing the *size* of recursive variables into the context. For atomic constructs, we follow the *weight* function and measure them as 1.

We solve the first challenge in a straightforward way: there is no need to distinguish between recursive and universal variables. The fact that all recursive variables in the context are bounded by a top type whose measure is simply one fits our needs naturally.

As for the second concern, despite the different purposes of contexts, the key ideas of measuring types in kernel $F_{\leq}$ and nominal unfoldings are the same: they both relate the measure of a variable to its true meaning, either its unfolded form or its bound size. A slight modification is made based on the definition of *weight*. In the *weight* function, for a universal variable, its bound is first retrieved and then the measure is computed. To align with the "pre-computation" mechanism of measuring nominal unfoldings ($i := size_{\Psi, \alpha \mapsto 1}(A)$), we also pre-compute the measure of the bound ($i := size_{\Psi}(A)$) in the *size* function, so that we retrieve the measure instead of the type bound from the context. In a well-formed type, variables are guaranteed to be unique, so we can use a single context Ψ to store the measures for both recursive variables and universal variables.

A subtler issue lies on variables in the initial subtyping context. When measuring nominal unfoldings, the context in a subtyping relation is simply a list of variables, without any bound information, so variables that occur freely can be counted as 0. In contrast, now the subtyping context stores the bound information, and the measures of bounds play a role in deciding the subtyping relation. To address this issue, we need to make sure that the bound information is pre-computed in the measure function. We transform a subtyping context into an environment containing measures Ψ, which tracks universal variables. In our decidability proof statement (Lemma 4.6), Ψ is computed from the subtyping context Γ by an evaluation function $eval : \Gamma \hookrightarrow \Psi$, defined as:

$$
\begin{array}{lcl}
eval(\cdot) & = & \cdot \\
eval(\Gamma',\ x : A) & = & eval(\Gamma') \\
eval(\Gamma',\ \alpha \leq A) & = & \text{let } \Psi' = eval(\Gamma') \text{ in } \Psi', \alpha \mapsto size_{\Psi'}(A)
\end{array}
$$

With both *eval* and *size* we can then state the decidability theorem:

Lemma 4.6. If $size_{eval(\Gamma)}(A) + size_{eval(\Gamma)}(B) \leq k$ then $\Gamma \vdash A \leq B$ is either true or not.

Theorem 4.7 (Decidability). $\Gamma \vdash A \leq B$ is decidable.

As for the third concern, note that in $F_{\leq}$, the subtyping relation is antisymmetric [Baldan et al. 1999]. Adding recursive types does not change the property of antisymmetry. However, the addition of records makes the subtyping relation not antisymmetric: two record types may be syntactically different. The lack of antisymmetry poses a challenge for our decidability proof, in particular for rule S-EQUIVALL. However, the fields of two equivalent records must be a permutation of each other. Therefore, the measures of two equivalent record types are the same. As a result, the measure of two equivalent bounds A_1 and A_2 is equal, as Lemma 4.8 describes. The measure information of type A_1 can therefore be reconstructed from type A_2, addressing the final concern with decidability.

Lemma 4.8. If $\Gamma \vdash A \leq B$ and $\Gamma \vdash B \leq A$ then $size_{eval(\Gamma)}(A) = size_{eval(\Gamma)}(B)$.

4.3 Conservativity

One important feature of $F^{\mu}_{\leq}$ is that it is conservative over kernel $F_{\leq}$. Conservativity means that equivalent $F_{\leq}$ judgments in $F^{\mu}_{\leq}$ should behave in the same way as in $F_{\leq}$. For instance, if a subtyping statement is valid in $F_{\leq}$, then it should also be valid in $F^{\mu}_{\leq}$. Dually, if a subtyping statement over $F_{\leq}$-types is invalid in $F_{\leq}$, then it should also be invalid in $F^{\mu}_{\leq}$. In some calculi, including extensions of $F_{\leq}$ with *equi*-recursive types [Ghelli 1993], conservativity is lost after the addition of new features.

For avoiding ambiguity, let $\vdash_F \Gamma$, $\Gamma \vdash_F A$, $\Gamma \vdash_F A \leq B$, $\vdash_F e$ and $\Gamma \vdash_F e : A$ represent the well-formedness of environment, well-formedness of types, subtyping, well-formedness of expressions and the typing relation, respectively, in kernel $F_{\leq}$. Note that all these judgments are essentially subsets of the judgments introduced in §3, except that the rules involving records and recursive types are removed, and that the rule S-EQUIVALL is replaced with the rule S-KERNELALL.

Conservativity of subtyping. Our conservativity result for subtyping is relatively easy to establish:

Lemma 4.9 (Conservativity for subtyping). If $\vdash_F \Gamma$, $\Gamma \vdash_F A$, and $\Gamma \vdash_F B$ then $\Gamma \vdash_F A \leq B$ if and only if $\Gamma \vdash A \leq B$.

Here the well-formedness conditions ensure that Γ, A and B must be respectively a valid $F_{\leq}$ environment, and valid $F_{\leq}$ types. That is they cannot contain recursive types (or record types). Therefore, the lemma states that for environments and types without recursive types, the two subtyping relations (for $F_{\leq}$ and $F^{\mu}_{\leq}$) are equivalent, accepting the same statements. The only hurdle is that to establish the correspondence between rule S-EQUIVALL and the rule S-KERNELALL in kernel $F_{\leq}$, we need the antisymmetry property for kernel $F_{\leq}$ [Baldan et al. 1999].

Conservativity of typing. It is straightforward to obtain part of the conservativity result from a typing relation in $F_{\leq}$ to a typing relation in $F^{\mu}_{\leq}$. As for the reverse direction, the situation is more complicated. If we want to derive $\Gamma \vdash_F e : A$ from $\Gamma \vdash e : A$, when doing induction, for the subsumption case (rule TYPING-SUB), we need to guess an intermediate type. However, we do not know if it involves recursive types or not. Consider the following example:

$$\textsc{typing-sub}\ \frac{\textsc{typing-sub}\ \dfrac{\vdash \lambda x.\, x : \top \to \top \qquad \vdash \top \to \top \leq (\mu\alpha.\, \top) \to \top}{\vdash \lambda x.\, x : (\mu\alpha.\, \top) \to \top} \qquad \vdash (\mu\alpha.\, \top) \to \top \leq \top}{\vdash \lambda x.\, x : \top}$$

Although $\vdash \lambda x.\, x : \top$ do not involve recursive types, the typing subderivations can contain recursive types. As a result, the induction hypothesis cannot be applied.

This problem can be addressed by employing the algorithmic formulation of $F^{\mu}_{\leq}$, shown in §3.4. With algorithmic typing, we can have more precise information about the types of an expression, since algorithmic typing always gives the minimum type. Therefore, it can be proved that for

expressions that do not use fold/unfold constructors, their minimum types do not contain recursive types as well. Conservativity for algorithmic typing is proved as follows:

Lemma 4.10. If $\vdash_F \Gamma$, $\Gamma \vdash_F A$ and $\vdash_F e$ then $\Gamma \vdash_a e : A$ implies $\Gamma \vdash_F e : A$.

Now, given a typing relation $\Gamma \vdash e : A$ in $F^{\mu}_{\leq}$, we first use the minimum typing property (Theorem 3.9) to obtain its minimum type B such that $\Gamma \vdash_a e : B$ and $\Gamma \vdash B \leq A$. Applying Lemma 4.10 and Lemma 4.9, we complete the conservativity proof for the declarative version of $F^{\mu}_{\leq}$.

Theorem 4.11 (Conservativity). If $\vdash_F \Gamma$, $\Gamma \vdash_F A$ and $\vdash_F e$ then $\Gamma \vdash_F e : A$ if and only if $\Gamma \vdash e : A$.

5 A CALCULUS WITH LOWER AND UPPER BOUNDED QUANTIFICATION

In this section we introduce an extension of $F^{\mu}_{\leq}$, called $F^{\mu}_{\leq\geq}$, with lower bounded quantification and a bottom type. While upper bounded quantification has received a lot of attention in previous research, lower bounded quantification for an $F_{\leq}$-like language is much less explored, though it appears in a few works [Amin and Rompf 2017; Oliveira et al. 2020]. We follow the same approach as Oliveira et al. [2020], where their $F_{\leq}$ extension allows type variables to have either a lower bound or an upper bound, but not both bounds at once. As discussed in §2.2, this extension enables further applications, such as a form of extensible encodings of datatypes. We have proved all the same results for $F^{\mu}_{\leq\geq}$ that were proved for $F^{\mu}_{\leq}$, including type soundness, decidability, transitivity and conservativity over $F_{\leq}$.

5.1 The $F^{\mu}_{\leq\geq}$ Calculus

The syntax of types, expressions, values and contexts for the extended $F^{\mu}_{\leq\geq}$ calculus is shown below. The main novelties are that bottom types and lower bounded quantification are introduced. The syntactic additions are highlighted in gray.

Types	$A, B, \ldots$	::=	$\mathsf{nat} \mid \top \mid \bot \mid A_1 \to A_2 \mid \alpha \mid \mu\alpha.\, A \mid A^{\alpha}$
			$\mid \forall(\alpha \leq A).\, B \mid \forall(\alpha \geq A).\, B \mid \{l_i : A_i\ ^{i \in 1 \cdots n}\}$
Expressions	e	::=	$x \mid \mathsf{i} \mid e_1\ e_2 \mid \lambda x : A.\, e \mid e\ A \mid \Lambda(\alpha \leq A).\, e \mid \Lambda(\alpha \geq A).\, e$
			$\mid \mathsf{unfold}\ [A]\ e \mid \mathsf{fold}\ [A]\ e \mid \{l_i = e_i\ ^{i \in 1 \cdots n}\} \mid e.l$
Values	v	::=	$\mathsf{i} \mid \lambda x : A.\, e \mid \mathsf{fold}\ [A]\ v \mid \Lambda(\alpha \leq A).\, e \mid \Lambda(\alpha \geq A).\, e$
			$\mid \{l_i = v_i\ ^{i \in 1 \cdots n}\}$
Contexts	Γ	::=	$\cdot \mid \Gamma, \alpha \leq A \mid \Gamma, \alpha \geq A \mid \Gamma, x : A$

Subtyping, typing and reduction. Similarly to §3.1, the well-formedness for the additional bottom types and universal types with lower bounds are standard. Figure 4 shows the additional rules of $F^{\mu}_{\leq\geq}$ with respect to $F^{\mu}_{\leq}$ for subtyping, typing and reduction. Compared with $F^{\mu}_{\leq}$, we add rule S-BOT, S-VARTRANSLB and S-EQUIVALLLB, which are the dual forms of rule S-TOP, S-VARTRANS and S-EQUIVALL, respectively. The rule TYPING-TAPPLB and TYPING-TABSLB are the dual forms of rule TYPING-TAPP and TYPING-TABS for typing, respectively. The rule STEP-TABSLB rule is the dual form of rule STEP-TABS for reduction.

The new subtyping relation is reflexive and transitive:

Theorem 5.1 (Reflexivity for $F^{\mu}_{\leq\geq}$). If $\vdash \Gamma$ and $\Gamma \vdash A$ then $\Gamma \vdash A \leq A$.

Theorem 5.2 (Transitivity for $F^{\mu}_{\leq\geq}$). If $\Gamma \vdash A \leq B$ and $\Gamma \vdash B \leq C$ then $\Gamma \vdash A \leq C$.

Structural folding and lower bounded quantification. The structural folding rule TYPING-SFOLD on recursive types has already been shown for $F^{\mu}_{\leq}$. Note that this rule is not strictly necessary for $F^{\mu}_{\leq}$, because a recursive type can only be a subtype of another recursive type or the $\top$ type. Thus

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

$$\textsc{S-bot}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash \bot \leq A} \qquad \textsc{S-vartranslb}\ \frac{\alpha \geq B \in \Gamma \qquad \Gamma \vdash A \leq B}{\Gamma \vdash A \leq \alpha}$$

$$\textsc{S-equivalllb}\ \frac{\Gamma \vdash A_1 \leq A_2 \qquad \Gamma \vdash A_2 \leq A_1 \qquad \Gamma,\ \alpha \geq A_2 \vdash B \leq C}{\Gamma \vdash \forall(\alpha \geq A_1).\ B \leq \forall(\alpha \geq A_2).\ C}$$

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

$$\textsc{typing-tapplb}\ \frac{\Gamma \vdash e : \forall(\alpha \geq B_1).\ B_2 \qquad \Gamma \vdash B_1 \leq A}{\Gamma \vdash e\ A : [\alpha \mapsto A]\ B_2} \qquad \textsc{typing-tabslb}\ \frac{\Gamma,\ \alpha \geq A \vdash e : B}{\Gamma \vdash \Lambda(\alpha \geq A).\ e : \forall(\alpha \geq A).\ B}$$

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

$$\textsc{step-tabslb}\ \frac{}{(\Lambda(\alpha \geq A).\ e)\ B \hookrightarrow [\alpha \mapsto B]\ e}$$

Fig. 4. Additional subtyping, typing and reduction rules for $F^{\mu}_{\leq\geq}$ with respect to $F^{\mu}_{\leq}$.

the effect of structural folding in $F^{\mu}_{\leq}$, can be subsumed by other subtyping/typing rules. Perhaps for this reason, Abadi et al. [1996] have only considered a structural unfolding rule. However, in $F^{\mu}_{\leq\geq}$, a recursive type can also be a subtype of a type variable. In this case, the structural folding rule can give the desired typings to the $\mathsf{Add}_\forall$ constructors of the Exp_1 and Exp_2 datatypes that we have presented in §2.2, while the standard folding rule cannot. The rule TYPING-SFOLD has the same form in $F^{\mu}_{\leq\geq}$ as in $F^{\mu}_{\leq}$. Therefore, we believe that the structural folding rule that we have proposed, together with the structural unfolding lemma in the metatheory, is general.

Type soundness. Our type soundness proof for $F^{\mu}_{\leq\geq}$ is standard, thus we have:

Theorem 5.3 (Preservation for $F^{\mu}_{\leq\geq}$). If $\vdash e : A$ and $e \hookrightarrow e'$ then $\vdash e' : A$.

Theorem 5.4 (Progress for $F^{\mu}_{\leq\geq}$). If $\vdash e : A$ then e is a value or exists $e', e \hookrightarrow e'$.

5.2 Metatheory of $F^{\mu}_{\leq\geq}$

The addition of lower bounded quantification and the bottom type creates some difficulties in the metatheory of $F^{\mu}_{\leq\geq}$. The proof strategies for the unfolding lemma, decidability and conservativity, as we showed in the §4 for $F^{\mu}_{\leq}$, require some adjustments. In the following, we describe how to overcome the difficulties. We refer readers to the extended version of the paper for more details.

Unfolding Lemma. A type system that simultaneously allows introducing lower and upper bounded type variables will break the proof for the unfolding lemma shown in §4.1. The interaction of lower bounds and upper bounds invalidates the following inversion lemma for rule S-VARTRANS, which has been used to prove the unfolding lemma:

Lemma 5.5. If $\alpha \leq A \in \Gamma, \Gamma \vdash \alpha \leq B$, and $\alpha \neq B$ then $\Gamma \vdash A \leq B$.

This lemma holds when the bounds in the context can only have one direction. However, when we have both kinds of bounds in the context, a counter-example can be found as follows:

$$x \leq \top, y \geq x \vdash x \leq y \quad \not\Longrightarrow \quad x \leq \top, y \geq x \vdash \top \leq y$$

For avoiding using this inversion lemma in the proof of unfolding lemma, we need to refine the generalized unfolding lemma. In Lemma 4.5, there is more than one subtyping statement on the premises related to type A and B, and we do induction on the premise (1). In the refined generalized unfolding lemma, we integrate those subtyping statements into one statement, thus the premise (1) can be induced implicitly. Meanwhile, the context is also refined to be more general. The concrete generalized unfolding lemma for $F^{\mu}_{\leq\geq}$ is shown in the extended version of the paper. With these changes we prove:

Lemma 5.6 (Unfolding lemma for $F^{\mu}_{\leq\geq}$). If $\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B$ then $\Gamma \vdash [\alpha \mapsto \mu\alpha.\ A]\ A \leq [\alpha \mapsto \mu\alpha.\ B]\ B$.

Decidability. The interaction between bottom types and rule S-EQUIVALL breaks the measure-based decidability proof in §4.2. The bottom type in $F^{\mu}_{\leq}$ brings a new form of equivalent types: when $\alpha \leq \bot \in \Gamma$, one can derive that $\Gamma \vdash \alpha \leq \bot$ and $\Gamma \vdash \bot \leq \alpha$, as observed by Pierce [1997]. Simply extending the measure function with $size_{\Psi}(\bot) = 1$ will not work. For type variables, the measure function will recursively look up its bound in the context, and add one to the measure of its bound, making a variable equivalent to $\bot$ to *have a larger measure* than $\bot$. Therefore, replacing two equivalent types into the abstracted type body may not produce the same measures. We can construct derivations of rule S-EQUIVALL that have a larger measure in the premise than that of the conclusion, which makes the decidability proof fail with the current measure.

We solve this issue by replacing all the type variables who are a subtype of bottom by bottom. Every time that we compute the size for a variable bounded by an upper bound, we firstly recursively check whether it is a synonym for bottom, and return the size of bottom if it is. This can be implemented by an extension to the measure context Ψ. Dually, when we compute the size for a variable bounded by a lower bound, we recursively check if it is the supertype of top. With this change the measures work, and we can prove decidability.

Theorem 5.7 (Decidability for $F^{\mu}_{\leq\geq}$). $F^{\mu}_{\leq\geq}$ is decidable.

Conservativity. The proof of conservativity for $F^{\mu}_{\leq\geq}$ follows the same pattern as the proof for $F^{\mu}_{\leq}$. To prove conservativity of typing, we need the help of the algorithmic typing rules to obtain the minimum type of an $F_{\leq}$ term. However, the introduction of bottom types requires us to add several new algorithmic typing rules, since in the declarative system, one can always use the subsumption rule to transform a term with type $\bot$ to any function type or universal type, and apply it to any argument, as also observed by Pierce [1997]. We also develop a similar treatment for recursive types. Moreover, the meanings of the two exposure functions also need to be refined. For example, the upper exposure function ($\Uparrow$) is now used to find the least non-*upper-bounded*-variable upper bound in the context, so it will return the variable itself if the variable is lower bounded. For lower exposure, we also need a dual form of the rule XA-PROMOTE, which finds the non-variable lower bound for a variable:

$$\frac{\alpha \geq A \in \Gamma \qquad \Gamma \vdash A \Downarrow B}{\Gamma \vdash \alpha \Downarrow B}\ \text{XA-DOWNPROMOTE}$$

The complete set of $F^{\mu}_{\leq\geq}$ algorithmic typing rules can be found in the extended version of the paper.

Theorem 5.8 (Conservativity for $F^{\mu}_{\leq\geq}$). If $\vdash_F \Gamma, \Gamma \vdash_F A$ and $\vdash_F e$ then $\Gamma \vdash_F e : A$ if and only if $\Gamma \vdash e : A$.

6 RELATED WORK

Bounded Quantification and Recursive Types. Bounded quantification was firstly introduced by Cardelli and Wegner [1985] in the language Fun, where their kernel Fun calculus corresponds

to the kernel version of $F_{\leq}$. The full variant of $F_{\leq}$ was introduced by Curien and Ghelli [1992] and Cardelli et al. [1994], where the subtyping for bounds are contravariant. Although full $F_{\leq}$ is powerful, subtyping is proved to be undecidable [Pierce 1994]. As discussed in §1 there are several attempts to add recursive types to $F_{\leq}$, such as the work by Ghelli [1993], Colazzo and Ghelli [2005] and Jeffrey [2001]. Unfortunately, as Table 1 shows, such combinations are not painless, and even the successful combinations require the significant changes for the subtyping rules.

Ghelli illustrates how the combination of equi-recursive subtyping and full $F_{\leq}$ changes the expressiveness power of the subtyping relation with a simple example:

$$\begin{array}{lcllcl} B & \equiv & \forall\alpha.\ -(\forall(\alpha' \leq \alpha).\ -\alpha) & A' & \equiv & \forall(\beta \leq B).\ \forall(\beta' \leq \beta).\ -\beta \\ A & \equiv & \forall(\beta \leq B).\ \beta & R & \equiv & \forall(\beta \leq B).\ \mu X.\ \forall(\beta' \leq X).\ -X \end{array}$$

where $-A$ stands for $A \to \top$ and $\forall\alpha.\ A$ is the abbreviation of $\forall(\alpha \leq \top).\ A$. In full $F_{\leq}$, $A \leq A'$ does not hold. However, such system allows $A \leq R$ and $R \leq A'$ to be derived, which by transitivity, allows us to conclude that $A \leq A'$. In Colazzo and Ghelli's work, there is no independent universal type, and the shape of recursive types is either $\mu\alpha.\forall(x \leq A).\ B$ or $\mu\alpha.\ A \to B$. The recursive variables and universal variables are distinct, resulting in changes in environments and subtyping rules. For example, the subtyping environment is defined as $\Pi := \cdot \mid \Pi, (x, y) \leq (A, B) \mid \Pi, (\alpha = A,\ \beta = B)$, and the rule S-VARTRANS rule of $F_{\leq}$ is changed to:

$$\frac{(x, y) \leq (A', B') \in \Pi \qquad \forall\alpha', B \neq \alpha' \qquad B \neq \top \quad B \neq y \quad \Pi \vdash A' \leq B}{\Pi \vdash x \leq B}$$

The algorithm proposed by Jeffrey is also complex, and requires major changes. Both recursive variables and the subtyping algorithm are labelled polarly, and the implementation of α-conversion is not discussed. In contrast, our subtyping rules do not change the contexts, the types are not restricted, and most importantly, we do not have to change the rules in the original $F_{\leq}$. This has the benefit that we can largely reuse the existing metatheory of the original $F_{\leq}$, and it also enables our conservativity result. While it is plausible that Jeffrey's or Colazzo and Ghelli's work for the kernel $F_{\leq}$ extensions with recursive types are conservative, this has not been proved. Furthermore, such proof is likely to be non-trivial because of the major changes introduced by equi-recursive subtyping.

There are many other extensions to $F_{\leq}$. Bounded existentials are also studied by Cardelli and Wegner [1985]. Existential types can be encoded by universal types, thus we can obtain a form of bounded existentials for free in $F_{\leq}$ [Cardelli and Wegner 1985]. Another important extension is f-bounded quantification, firstly proposed by Canning et al. [1989], then studied by Baldan et al. [1999] in terms of the basic theory. In f-bounded quantification, the bounded variables are allowed to appear in the bound. For example, a bound of the form $\forall(\alpha \leq F[\alpha]).\ B$, where F is a type-level function applied to α, is allowed in f-bounded quantification. We can encode polymorphic binary methods [Bruce et al. 1995] and methods that have recursive types in their signatures with f-bounded quantification. For the example that we have showed in §2.2, the bound in the translate function is not Point but would have a form $F[\alpha] = \{x : \mathsf{Int},\ y : \mathsf{Int},\ \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}$. Therefore, with f-bounded quantification the translate function could have the type:

$$\forall(\alpha \leq \{x : \mathsf{Int},\ y : \mathsf{Int},\ \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}).\ \alpha \to \alpha$$

Then the α can instantiate to Point or subtypes of Point, because $\mathsf{Point} \leq F[\mathsf{Point}]$. Note that for subtyping statements such as $\alpha \leq \{x : \mathsf{Int},\ y : \mathsf{Int},\ \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}$ to hold, they must be interpreted using equi-recursive subtyping, since the f-bounds are normally records, and an iso-recursive type cannot be the subtype of a record. This approach is appealing because it can even deal with binary methods, where recursive types appear in negative positions. For example,

with f-bounded quantification we can model bounds such as $\alpha \leq \{x : \mathsf{Int},\ \mathsf{eq} : \alpha \to \mathsf{Bool}\}$, and still have the expected subtyping relations.

Whereas we show that with the structural unfolding rule we can model positive cases of f-bounded quantification (such as translate) in $F^{\mu}_{\leq}$, we can only model a restricted form of negative f-bounded quantification. For instance in $F^{\mu}_{\leq}$ we can have the bound $\alpha \leq \mu\ \mathsf{P}.\ \{x : \mathsf{Int},\ \mathsf{eq} : \mathsf{P} \to \mathsf{Bool}\}$ and we can instantiate α with P (where $P = \mu\ \mathsf{P}.\ \{x : \mathsf{Int},\ \mathsf{eq} : \mathsf{P} \to \mathsf{Bool}\}$). However, we would not be able to instantiate α with some types that have extra fields, such as $\mu\ \mathsf{P'}.\ \{x : \mathsf{Int},\ y : \mathsf{Int}, \mathsf{eq} : \mathsf{P'} \to \mathsf{Bool}\}$ for example. In contrast, f-bounded quantification allows such forms of instantiation. Nevertheless, given the overlap between some of the applications of iso-recursive types in $F^{\mu}_{\leq}$ and f-bounded quantification, we believe that it is worthwhile to investigate whether f-bounded quantification can be avoided to deal with general binary methods.

Dependent Object Types. The interest in languages with bounded quantification and recursive types has been reignited recently in the research community with the introduction of the dependent object types (DOT) [Rompf and Amin 2016] calculus. DOT is the foundation of Scala 3 [EPFL 2021], and also employs a generalized form of bounded quantification and recursive types. The generalized form of bounded quantification subsumes both upper and lower bounded quantification, which are present in $F^{\mu}_{\leq\geq}$. DOT supports both upper and lower bounds at the same time for path selection. In addition, DOT also supports path-dependent types [Amin et al. 2014] and intersection types [Barbanera et al. 1995; Coppo et al. 1981; Pottinger 1980]. DOT can encode full $F_{\leq}$ [Rompf and Amin 2016] and has been shown to be undecidable [Hu and Lhoták 2020; Mackay et al. 2020]. One of the limitations for DOT is that transitivity elimination is not possible [Rompf and Amin 2016], and even the decidable fragments of DOT lack transitivity [Hu and Lhoták 2020; Mackay et al. 2020]. The research on DOT has been intimately related to $F_{\leq}$. For instance, Amin and Rompf [2017] explain many of the features of DOT by incrementally extending $F_{\leq}$. In addition, there have been various attempts to prove the undecidability of DOT by a reduction to the undecidability problem in $F_{\leq}$. Although, as Hu and Lhoták [2020] observed, DOT is not conservative over $F_{\leq}$. Thus an undecidability result for DOT cannot be proved by reduction to the undecidability of full $F_{\leq}$. While $F^{\mu}_{\leq\geq}$ does not have all the features of DOT, our results can potentially help in research in that area, where the decidable fragments of DOT lack important properties such as transitivity. In addition $F^{\mu}_{\leq\geq}$ preserves the conservativity over $F_{\leq}$, while DOT does not.

Object Encodings. Recursive records can encode objects [Bruce et al. 1999; Canning et al. 1989; Cook et al. 1989]. Alternatively, existential types can also be used to encode objects [Pierce and Turner 1994], or they can be employed together with recursive types [Bruce 1994]. Pierce and Turner's object encoding is notable in that it requires only $F_{\leq}$, and does not employ recursive types. The *ORBE* encoding, presented by Abadi et al. [1996] consists of recursive types, bounded existential quantification, records, and the structural unfolding rule. In their work, an interface is encoded as:

$$ORBE(I) \triangleq \mu\alpha.\ \exists(\beta \leq \alpha).\ \beta \times (\beta \to I(\beta))$$

As Bruce et al. observe, the ORBE encoding requires full $F_{\leq}$ for the bounded quantification subtyping rule. When we try to compare two bounds, the type variable will be substituted with the existential types, which may result in bounds that are not equivalent. The overview paper by Bruce et al. [1999] makes a detailed comparison among different object encodings. Except for the *ORBE* encoding, $F^{\mu}_{\leq}$ could serve as a target for the existing object encodings. However, no complete formalization of $F_{\leq}$ with recursive types with desirable properties (such as type soundness and conservativity) existed at the time. Our work helps to further validate such encodings by providing a complete formalization of $F_{\leq}$ with recursive types, together with various desirable properties.

Algebraic Datatypes and Subtyping. Algebraic datatypes are a fundamental feature in modern functional programming languages, such as Haskell [Haskell Development Team 1990] and Ocaml [INRIA 1987]. However, such languages do not support subtyping between datatypes. Hosoya et al. [1998] discussed the interaction between mutually recursive datatypes and subtyping. They presented two variants of $F_{\leq}$ extending $F_{\leq}$ with user-defined datatype declarations. The first variant has user-defined subtyping declarations between datatypes, and can be viewed as having a form of nominal subtyping. The second variant allows structural subtyping among the datatypes. One advantage of employing user-defined datatypes is that it is simple to deal with formally, and that it allows mutually recursive datatype definitions easily. However, they do not support conventional recursive types of the form $\mu\alpha.\ A$ as we do in $F^{\mu}_{\leq}$. Moreover, they do not consider lower bounded quantification which, as argued in §2.2, seems to be quite useful in a system targeting algebraic datatypes. There has been some work integrating ML datatypes and OO classes [Bourdoncle and Merz 1997; Millstein et al. 2004]. In the implementation of hierarchical extensible datatypes, methods are simulated via functions with dynamic dispatch. Those works are focused on the design of intermediate languages that have complex constructs such as classes or datatypes. In contrast, we develop foundational calculi, where more complex constructs can be encoded. Finally, Poll [1998] investigated the categorical semantics of datatypes with subtyping and a limited form of inheritance on datatypes, improving our understanding on the relation between categorical datatypes and object types.

Oliveira [2009] showed encodings of algebraic datatypes with subtyping assuming a variant of $F_{\leq}$ extended with records, recursive types and higher kinds. He showed that adding subtyping to datatypes allows solving the Expression Problem [Wadler 1998]. However, as we mentioned in §2.2, he did not formalize the $F_{\leq}$ extension, although he showed a translation of the encoding into Scala. Moreover, his encoding is more complex than ours because he employs upper bounded quantification with higher kinds. In §2.2, we showed that first-order lower bounded quantification in $F^{\mu}_{\leq\geq}$, together with the structural folding rule enables such encodings. Like for encodings of objects, our work is helpful to further validate such encodings formally.

7 CONCLUSION

Recursive types and bounded quantification are important in many programming languages. However, although those features are well-studied independently, their interaction has posed a long term challenge. Our $F^{\mu}_{\leq}$ calculus illustrates how to integrate iso-recursive types and kernel $F_{\leq}$. We obtain a transitive and decidable subtyping relation, while the full calculus is shown to be conservative over $F_{\leq}$ and is proven to be type-sound. $F^{\mu}_{\leq}$ and $F^{\mu}_{\leq\geq}$ could serve as a theoretic foundation for object encodings and encodings of algebraic datatypes with subtyping. With the renewed interest on recursive types and bounded quantification due to the DOT calculus, we believe that our work is also helpful to find calculi with most features in DOT, while retaining desirable properties, such as decidability and transitivity of subtyping, or even conservativity over $F_{\leq}$. Investigating extensions of $F^{\mu}_{\leq}$ with some of the features of DOT is a clear avenue for future work. In addition, studying the combination of iso-recursive subtyping and full $F_{\leq}$, could be useful for modelling the *ORBE* encoding [Abadi et al. 1996].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. This work has been sponsored by Hong Kong Research Grant Council projects number 17209519, 17209520 and 17209821.

REFERENCES

Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. 1996. An interpretation of objects and object types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 396–409. https://doi.org/10.1145/237721.237809

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 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/10.1145/3093333.3009866

Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of path-dependent types. *Acm Sigplan Notices* 49, 10 (2014), 233–249. https://doi.org/10.1145/2714064.2660216

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering formal metatheory. *Acm sigplan notices* 43, 1 (2008), 3–15. https://doi.org/10.1145/1328897.1328443

Michael Backes, Cătălin Hriţcu, and Matteo Maffei. 2014. Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations. *Journal of Computer Security* 22, 2 (2014), 301–353. https://doi.org/10.3233/JCS-130493

Paolo Baldan, Giorgio Ghelli, and Alessandra Raffaeta. 1999. Basic theory of F-bounded quantification. *Information and Computation* 153, 2 (1999), 173–237. https://doi.org/10.1006/inco.1999.2802

Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. 1995. Intersection and Union Types: Syntax and Semantics. *Information and Computation* 119, 2 (June 1995), 202–230. https://doi.org/10.1006/inco.1995.1086

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

Corrado Böhm and Alessandro Berarducci. 1985. Automatic synthesis of typed Lambda-programs on term algebras. *Theoretical Computer Science* 39, 2-3 (1985).

François Bourdoncle and Stephan Merz. 1997. Type checking higher-order polymorphic multi-methods. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 302–315. https://doi.org/10.1145/263699.263743

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

Kim Bruce, Luca Cardelli, Giuseppe Castagna, Hopkins Objects Group, Gary T Leavens, and Benjamin C. Pierce. 1995. On binary methods. *Theory and Practice of Object Systems* 1, 3 (1995), 221–242. https://doi.org/10.1002/j.1096-9942.1995.tb00019.x

Kim B Bruce. 1994. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming* 4, 2 (1994), 127–206. https://doi.org/10.1017/S0956796800001039

Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. 1999. Comparing Object Encodings. *Information and Computation* 155, 1. https://doi.org/10.1007/BFb0014561

Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. 1989. F-Bounded Polymorphism for Object-Oriented Programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture* (Imperial College, London, United Kingdom) *(FPCA 1989)*. 8 pages. https://doi.org/10.1145/99370.99392

Luca Cardelli. 1985. Amber. In *LITP Spring School on Theoretical Computer Science*. Springer, 21–47. https://doi.org/10.1007/3-540-17184-3_38

Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. 1994. An extension of system F with subtyping. *Information and computation* 109, 1-2 (1994), 4–56. https://doi.org/10.1006/inco.1994.1013

Luca Cardelli and Peter Wegner. 1985. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)* 17, 4 (1985), 471–523. https://doi.org/10.1145/6041.6042

Giuseppe Castagna and Benjamin C. Pierce. 1994. Decidable Bounded Quantification. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 151–162. https://doi.org/10.1145/174675.177844

Alonzo Church. 1932. A set of postulates for the foundation of logic. 33, 2 (1932), 346–366. https://doi.org/10.2307/1968702

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

William R. Cook, Walter Hill, and Peter S. Canning. 1989. Inheritance is Not Subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '90)*. Association for Computing Machinery. https://doi.org/10.1145/96709.96721

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 1981. Functional characters of solvable terms. *Mathematical Logic Quarterly* 27, 2-6 (1981), 45–58. https://doi.org/10.1002/malq.19810270205

Pierre-Louis Curien and Giorgio Ghelli. 1992. Coherence of subsumption, minimum typing and type-checking in F≤. *Mathematical structures in computer science* 2, 1 (1992), 55–91.

EPFL. 2021. *Scala 3*. https://www.scala-lang.org/

Vladimir Gapeyev, Michael Levin, and Benjamin C. Pierce. 2003. Recursive Subtyping Revealed. *Journal of Functional Programming* 12, 6 (2003), 511–548. https://doi.org/10.1145/357766.351261 Preliminary version in *International Conference on Functional Programming (ICFP)*, 2000. Also appears as Chapter 21 of *Types and Programming Languages* by Benjamin C. Pierce (MIT Press, 2002).

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

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et elimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état. Université de Paris 7.

Haskell Development Team. 1990. *Haskell*. https://www.haskell.org/

Haruo Hosoya, Benjamin C. Pierce, and David N. Turner. 1998. Datatypes and subtyping. *Unpublished manuscript* (1998).

Jason ZS Hu and Ondřej Lhoták. 2020. Undecidability of D<: and its decidable fragments. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 1–30. https://doi.org/10.1145/3371077

INRIA. 1987. *OCaml*. https://ocaml.org/

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)*. 323.

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

Julian Mackay, Alex Potanin, Jonathan Aldrich, and Lindsay Groves. 2020. Decidable subtyping for path dependent types. *Proc. ACM Program. Lang.* 4, POPL (2020), 66:1–66:27. https://doi.org/10.1145/3371134

Todd Millstein, Colin Bleckner, and Craig Chambers. 2004. Modular typechecking for hierarchically extensible datatypes and functions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 26, 5 (2004), 836–889. https://doi.org/10.1145/583852.581489

James Hiram Jr Morris. 1969. *Lambda-calculus models of programming languages*. Ph.D. Dissertation. Massachusetts Institute of Technology.

Bruno C. d. S. Oliveira. 2009. Modular visitor components. In *European Conference on Object-Oriented Programming*. Springer, 269–293. https://doi.org/10.1007/978-3-642-03013-0_13

Bruno C. d. S. Oliveira, Shaobo Cui, and Baber Rehman. 2020. The Duality of Subtyping. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs)*, Robert Hirschfeld and Tobias Pape (Eds.). https://doi.org/10.4230/LIPIcs.ECOOP.2020.29

Michel Parigot. 1992. Recursive programming with proofs. *Theoretical Computer Science* 94, 2 (1992), 335–356. https://doi.org/10.1016/0304-3975(92)90042-E

Benjamin C. Pierce. 1994. Bounded quantification is undecidable. *Information and Computation* 112, 1 (1994), 131–165. https://doi.org/10.1006/inco.1994.1055

Benjamin C Pierce. 1997. *Bounded quantification with bottom*. Technical Report. Citeseer.

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

Benjamin C Pierce and David N Turner. 1994. Simple type-theoretic foundations for object-oriented programming. *Journal of functional programming* 4, 2 (1994), 207–247. https://doi.org/10.1017/S0956796800001040

Erik Poll. 1998. Subtyping and Inheritance for Categorical Datatypes: Preliminary Report (Type Theory and its Applications to Computer Systems). *Kyoto University Research Information Repository* 1023 (1998), 112–125.

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

John C. Reynolds. 1974. Towards a theory of type structure. In *Colloque sur la Programmation*. Springer, 408–425. https://doi.org/10.1007/3-540-06859-7_148

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

Davide Sangiorgi and Robin Milner. 1992. The Problem of "Weak Bisimulation up to". In *CONCUR*, Vol. 630. 32–46. https://doi.org/10.1007/BFb0084781

Dana Scott. 1962. A system of functional abstraction. (1962). Lectures delivered at University of California, Berkeley, California, USA, 1962/63.

Philip Wadler. 1998. The Expression Problem. (1998). discussion on the Java Genericity mailing list.

Yaoda Zhou, Bruno C. d. S. Oliveira, and Jinxu Zhao. 2020. Revisiting Iso-Recursive Subtyping. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). https://doi.org/10.1145/3549537

Yaoda Zhou, Jinxu Zhao, and Bruno C. d. S. Oliveira. 2022. Revisiting Iso-Recursive Subtyping. *ACM Transactions on Programming Languages and Systems* 44, 4, Article 24 (2022). https://doi.org/10.1145/3549537

Received 2022-07-07; accepted 2022-11-07