JFP, 55 pages, 2022. © Cambridge University Press 2022
doi:10.1017/xxxxx

Recursive Subtyping for All

LITAO ZHOU[1] YAODA ZHOU[1] QIANYONG WAN BRUNO C. D. S. OLIVEIRA

The University of Hong Kong
(e-mail: {ltzhou,ydzhou,qywan,bruno}@cs.hku.hk)

Abstract

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 shows how to extend $F_{\leq}$ with iso-recursive types in a new calculus called $F_{\leq}^{\mu}$. $F_{\leq}$ is a well-known polymorphic calculus with bounded quantification. In $F_{\leq}^{\mu}$ we add iso-recursive types, and correspondingly extend the subtyping relation with iso-recursive subtyping using the recently proposed nominal unfolding rules. In addition we use so-called *structural* folding/unfolding rules for typing iso-recursive expressions, 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; the conservativity of $F_{\leq}^{\mu}$ over $F_{\leq}$; and a sound and complete algorithmic formulation of $F_{\leq}^{\mu}$. We study two variants of $F_{\leq}^{\mu}$. The first one uses an extension of the kernel $F_{\leq}$ (a well-known decidable variant of $F_{\leq}$). This extension accepts *equivalent* rather than *equal* bounds and is shown to preserve decidable subtyping. The second variant employs the full $F_{\leq}$ rule for bounded quantification and has undecidable subtyping. Moreover, we also study an extension of the kernel version of $F_{\leq}^{\mu}$, called $F_{\leq\geq}^{\mu\wedge}$, with a form of intersection types and *lower bounded quantification*. All the properties from the kernel version of $F_{\leq}^{\mu}$ are preserved in $F_{\leq\geq}^{\mu\wedge}$. All the results in this paper have been formalized in the Coq theorem prover.

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 (Curien and Ghelli, 1992; Cardelli et al., 1994; 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 language with subtyping, it is desirable to have *recursive subtyping* between recursive types. The first rules for recursive

[1] BOTH AUTHORS CONTRIBUTED EQUALLY TO THIS WORK.

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, 1998; Gapeyev et al., 2003), and *iso-recursive* subtyping (Ligatti et al., 2017; Bengtson et al., 2011; 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 $F_{\leq}$ (Curien and Ghelli, 1992; Cardelli and Wegner, 1985; Cardelli et al., 1994), 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 (1985). 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 (Cook et al., 1989; Abadi et al., 1996; Bruce et al., 1999), 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 (1993) 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 reopened questions about the decidability or undecidability of the system.

Even if we choose the weaker form of bounded quantification present in the 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). In kernel $F_{\leq}$, only universal quantifiers with equal bounds are allowed to be in a subtyping relation. This more restrictive formulation of bounded quantification is known to be decidable. However, complications still arise after adding equi-recursive types to kernel $F_{\leq}$. Instead of Amadio and Cardelli (1993)'s *meet 2 times* rules, Colazzo and Ghelli (2005) 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 (2005), Jeffrey (2001) extended the system and proved it correct and

Table 1: Comparison among different works.

	Ghelli (1993)	Colazzo and Ghelli (2005)	Jeffrey (2001)		Abadi et al. (1996)	This Work	
Kernel/Full $F_{\leq}$	full	kernel	full	kernel	full	kernel	full
Equi-/Iso-Recursive	equi	equi	equi	equi	iso	iso	iso
Transitivity	×	✓	✓	✓	built-in	✓	✓
Decidability		✓	✓̸	✓		✓	×
Conservativity	×		×			✓	✓
Type System					✓	✓	✓
Algorithmic Typing						✓	✓
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.

complete. By transferring the polar bisimulations (Sangiorgi and Milner, 1992) technique from concurrency theory, Jeffrey (2001)'s system is more general than Colazzo and Ghelli's, but it is only partially decidable. It is decidable for kernel $F_{\leq}$ with equi-recursive types, but for full $F_{\leq}$ with equi-recursive types, only when the algorithm terminates it returns the correct answer, but it may not terminate. Furthermore, although being more powerful, Jeffrey (2001)'s full system is not conservative over full $F_{\leq}$ 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 $F_{\leq}$ 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, as can be seen from the literature. 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 kernel $F_{\leq}$ 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.

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 (Ligatti et al., 2017; Backes et al., 2014; Zhou et al., 2020, 2022): it is difficult to prove results such as transitivity, or define sound and complete algorithmic formulations.

This paper shows how to extend $F_{\leq}$ with iso-recursive types in a calculus called $F_{\leq}^{\mu}$. In $F_{\leq}^{\mu}$ 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. (2022) can be integrated modularly into $F_{\leq}$ subtyping rules, 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 $F_{\leq}$.

In $F_{\leq}^{\mu}$ we use the so-called *structural* folding/unfolding rules for typing expressions with recursive types, 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 in modeling encodings of objects, as well as encodings of algebraic datatypes with subtyping.

We study two variants of $F_{\leq}^{\mu}$. The first one has a generalization of the kernel $F_{\leq}$ rule for bounded quantification that accepts *equivalent* rather than equal bounds. The second variant uses the rule of full $F_{\leq}$ for bounded quantification. We will refer to the first variant as kernel $F_{\leq}^{\mu}$, and to the second variant as full $F_{\leq}^{\mu}$. We present several results, including: *type soundness*; *transitivity* and *(un)decidability* of subtyping; the *conservativity* of $F_{\leq}^{\mu}$ over $F_{\leq}$; and a sound and complete algorithmic formulation of $F_{\leq}^{\mu}$. The kernel $F_{\leq}^{\mu}$ variant is proved to have decidable subtyping, whereas the full $F_{\leq}^{\mu}$ variant has undecidable subtyping. We also present an extension of $F_{\leq}^{\mu}$, called $F_{\leq\geq}^{\mu\wedge}$, which has a bottom type, intersection types, and *lower bounded quantification* in addition to the conventional (upper) bounded quantification of $F_{\leq}$. As we show, lower bounded quantification is useful to model the subtyping of algebraic datatypes. Intersection types are used to encode record types, similarly to how the Dependent Object Calculus (DOT) (Rompf and Amin, 2016) encodes object types. All the results in this paper have been formalized in the Coq theorem prover.

In summary the contributions of this paper are:

- **$F_{\leq}^{\mu}$: extending $F_{\leq}$ with iso-recursive types.** We have two variants of $F_{\leq}^{\mu}$: kernel $F_{\leq}^{\mu}$ as the extension of kernel $F_{\leq}$ with iso-recursive subtyping, and full $F_{\leq}^{\mu}$ as the extension of full $F_{\leq}$ with iso-recursive subtyping. We prove several properties for $F_{\leq}^{\mu}$, including: type soundness; transitivity of subtyping; decidability of subtyping of kernel $F_{\leq}^{\mu}$; undecidability of subtyping of full $F_{\leq}^{\mu}$; and the unfolding lemma, a key property to ensure type soundness.

- **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 $F^{\mu}_{\leq}$ is conservative over $F_{\leq}$.
- **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 kernel $F^{\mu}_{\leq}$.** We show that kernel $F^{\mu}_{\leq}$ is decidable. The measure needed for decidability is non-trivial because there are significant differences in the measures for kernel $F_{\leq}$ and nominal unfoldings. We show how to develop a new measure that can account for both features at once. In addition, due to our generalization of the kernel rule to allow equivalent bounds, a key property for decidability is that equivalent types have equal sizes.
- **An extension of $F^{\mu}_{\leq}$ with intersection types, both upper and lower bounded quantification:** We present an extended calculus, called $F^{\mu\wedge}_{\leq\geq}$, with a form of intersection types, 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 at `https://github.com/juda/Recursive-Subtyping-for-All/tree/main/JFP`.

Differences to conference version. This article is a substantial enhancement of the conference paper (Zhou et al., 2023). It introduces three major improvements over the original conference paper. The first improvement is the extension of our results to the full $F^{\mu}_{\leq}$ calculus, along with proofs demonstrating its type soundness and its conservativity over $F_{\leq}$. The initial conference version only addressed the addition of iso-recursive types into kernel $F_{\leq}$ and left the extension to the full variant as an unresolved issue. The second improvement is a further generalization of the unfolding lemma that is capable of dealing with full $F_{\leq}$, intersection types and all the other extensions in this paper. The unfolding lemma is a central lemma in the metatheory of iso-recursive subtyping, and it is also where the main challenge in the metatheory lies. In the conference version, the generalized unfolding lemma was not able to deal with full $F_{\leq}$. Our new generalization addresses this issue, and it is shown to be general and applicable to a variety of extensions. The final improvement involves the combination of several important features within the system $F^{\mu\wedge}_{\leq\geq}$, and a much more detailed overview of $F^{\mu\wedge}_{\leq\geq}$. Unlike the conference version, which did not include intersection types, the updated $F^{\mu\wedge}_{\leq\geq}$ can model objects using structural folding/unfolding rules, intersection types and single-field record types. This alternative way to model objects is inspired by, and aligns closely with, the encoding of objects in the DOT calculus.

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 and Wegner, 1985; Curien and Ghelli, 1992; Cardelli et al., 1994), has two common variants when it comes to subtyping universal types. The full $F_{\leq}$ variant (Curien and Ghelli, 1992; Cardelli et al., 1994) compares bounded quantifiers with the following rule:

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

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 expressive power 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.

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

In our paper, we will show how iso-recursive subtyping can be integrated with both kernel and full variants of $F_{\leq}$. However, for the kernel variant, 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 §4.3.

Recursive Types. Recursive types $\mu\alpha.A$, can be traced back to Morris (1968). 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 unfolding $[\alpha \mapsto \mu\alpha.A]\ A$. 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. Equi-recursive types also allow for more general equalities than unfoldings. For example, the types $\mu\alpha.\mathsf{int} \to \alpha$ and

$\mu\alpha.\,\mathsf{int} \to \mathsf{int} \to \alpha$ are considered equivalent in the equi-recursive setting, since they have the same infinite unfolding (Amadio and Cardelli, 1993).

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 $[\alpha \mapsto \mu\alpha.\,A]\ A$ 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:

$$\text{TYPING-UNFOLD}\quad \frac{\Gamma \vdash e : \mu\alpha.A}{\Gamma \vdash \mathsf{unfold}\ [\mu\alpha.A]\ e : [\alpha \mapsto \mu\alpha.A]\ A} \qquad \text{TYPING-FOLD}\quad \frac{\Gamma \vdash e : [\alpha \mapsto \mu\alpha.A]\ A \qquad \Gamma \vdash \mu\alpha.A}{\Gamma \vdash \mathsf{fold}\ [\mu\alpha.A]\ e : \mu\alpha.A}$$

Despite being less convenient, iso-recursive types are known to have the same expressive power as equi-recursive types (Abadi and Fiore, 1996; Zhou et al., 2024). We will focus next on iso-recursive types.

Recursive Subtyping. Subtyping between recursive types has been studied for many years (Cardelli, 1985; Amadio and Cardelli, 1993; Ligatti et al., 2017). The most widely used subtyping rules for recursive types are the Amber rules, first introduced in 1985 by Cardelli (1985) in a manuscript describing the Amber language (Cardelli, 1985). The iso-recursive Amber rules deal with recursive subtyping with three rules: rule S-AMBER, rule S-ASSMP and rule S-REFL.

$$\text{S-AMBER}\quad \frac{\Gamma,\ \alpha \le \beta \vdash A \le B}{\Gamma \vdash \mu\alpha.A \le \mu\beta.B} \qquad \text{S-ASSMP}\quad \frac{\alpha \le \beta \in \Gamma}{\Gamma \vdash \alpha \le \beta} \qquad \text{S-REFL}\quad \frac{}{\Gamma \vdash A \le A}$$

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 (Zhou and Oliveira, 2025). 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 they are easy to work with formally: indeed, Zhou et al. (2022) have a full Coq development, including proofs of decidability, that we will reuse and extend.

Nominal Unfolding Rules. The nominal unfolding rules provide a formal mechanism for handling iso-recursive subtyping. These rules are designed to address the challenges posed by contravariant occurrences of recursive type variables. For recursive types it is expected that if two recursive types $\mu\alpha.\,A$ and $\mu\alpha.\,B$ are subtypes, then their unfolding $[\alpha \mapsto \mu\alpha.\,A]\ A$ and $[\alpha \mapsto \mu\alpha.\,B]\ B$ should also be subtypes. This property can be tricky to achieve with contravariant occurrences of recursive variables. The Amber rules deal with this issue by remembering pairs of recursive variables as subtyping assumptions, as can be seen in rule S-AMBER. In contrast, the nominal unfolding rules unfold the recursive body twice to ensure the correctness of the subtyping relation.

For example, consider the subtyping statement $\mu\alpha.\alpha \to \mathsf{nat} \leq \mu\alpha.\alpha \to \top$. If we unfold the recursive types twice, we obtain:

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

This statement requires both $\mathsf{nat} \leq \top$ (which is true) and $\top \leq \mathsf{nat}$ (which is false), thus correctly rejecting the subtyping statement. Unfolding once would not expose the invalid $\top \leq \mathsf{nat}$ comparison.

The nominal unfolding rules simulate this double-unfolding process by replacing recursive types with labeled types (A^{α}):

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

In rule S-NOMINAL, every time two recursive types are compared, a fresh label α is used to label to the unfolded parts. Labeled types can only be compared to other labeled types with the same label, which ensures that they arise from the same recursive type, as shown in rule S-LABEL. The bound type variable α in the recursive body becomes free variable after unfolding[1]. For instance, to compare $\mu\alpha.\alpha \to \mathsf{nat}$ and $\mu\alpha.\alpha \to \top$, the subtyping statement becomes:

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

The one-time unfolding is captured by the labels, since if we ignore the body of the labeled types, $\alpha \to \mathsf{nat}$ and $\alpha \to \top$ are compared. On the other hand, when ignoring the labels, the double unfolding statement is obtained, which exposes the invalid $\top \leq \mathsf{nat}$ comparison. The key design in the nominal unfolding rules is to use label as a syntactic device to ensure that recursive types are compared correctly. Without labels providing distinct identities to recursive types, unsound subtyping statements such as $\mu\alpha.\mathsf{nat} \to \alpha \leq \mu\alpha.\mathsf{nat} \to \mathsf{nat} \to \top$, which unfolds to $\mathsf{nat} \to \mathsf{nat} \to \alpha \leq \mathsf{nat} \to \mathsf{nat} \to \top$, may be accepted.

The nominal unfolding rules are formally proven to be type sound and have the same expressive power as the iso-recursive Amber rules (Zhou et al., 2022). They are also easier to work with formally, enabling the development of sound and complete algorithmic formulations of subtyping. Additionally, these rules are modular, allowing the extension of existing calculi with iso-recursive types without significant changes to existing definitions and proofs.

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.

[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 fresh label α. In the paper we use the named representation for better readability, so type variables α and label variables α are distinguished by color. In a black-and-white printout, these label variables can be identified by noting that they only occur as superscripts in labeled types A^{α}.

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[2]. For example, we can define a type Point:

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

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:

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

Now, suppose we wish to translate the coordinates by one unit for a point, but we do not want to write such a translation function for each subclass of Point. As a first attempt, 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 $\forall(\mathsf{P} \leq \mathsf{Point}).\ \mathsf{P} \to \mathsf{Point}$, which is obtained from the following typing derivation (some parts omitted):

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

[2] We will use a simplified form of the encoding that does not deal with self-references, which allows programmers to refer to the object itself in method implementations using the self parameter. But self-references could be dealt with in standard ways (Canning et al., 1989).

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

$$\forall(\mathsf{P} \le \mathsf{Point}).\, \mathsf{P} \to \mathsf{P}$$

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; Baldan et al., 1999), address this issue. In F-bounded quantification, the bounded variables are allowed to appear in the bound, and universal types take the form $\forall(\alpha \le F[\alpha]).\,B$, where F is a type-level function applied to the bound variable α. For the example above, the bound in the translate function is no longer the closed recursive type Point but would have the 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:

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

Then the α can be instantiated to Point or subtypes of Point, since $\mathsf{Point} \le F[\mathsf{Point}]$. Note that to satisfy the F-bounded constraints $\alpha \le F[\alpha]$, the subtyping statements must be interpreted in an equi-recursive setting. $F^{\mu}_{\le}$ uses a less intrusive approach to achieve the same effect for typing the translate function, without requiring recursive bounds or equi-recursive types. This is achieved by using the structural unfolding rule (Abadi et al., 1996), which we will 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 will discuss the typing of this program via the structural unfolding rule in detail. After this change the type of translate is $\forall(\mathsf{P} \le \mathsf{Point}).\, \mathsf{P} \to \mathsf{P}$. 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 $(1, 1)$. Here mkEqPoint is a constructor for objects with type EqPoint, which contain a binary method (Bruce et al., 1995) eq:

```
function mkEqPoint(x₁ : Int, y₁ : Int) = fold [EqPoint] {
  x = x₁,
  y = y₁,
  move = λx₂ y₂. mkEqPoint(x₂, y₂),
  eq = λp. (getX p == x₁) ∧ (getY p == y₁)
}
```

Encoding objects with bounded existentials. Recursive types are not the only way to encode objects. Another common encoding is to use bounded existentials (Cardelli and Wegner, 1985). Existential types can be used to encode objects (Pierce and Turner, 1994), or they can be employed together with recursive types (Bruce, 1994). Since the intentional behavior of existential types can be encoded by universal types, we can obtain a form of bounded existentials for free in $F_{\leq}$ (Cardelli and Wegner, 1985):

$$\begin{array}{rcl} \exists(\alpha \leq A).B & \triangleq & \forall(\beta \leq \top).\,(\forall(\alpha \leq A).\,B \to \beta \to \beta) \\ \mathsf{pack}\ [C, e]\ \mathsf{as}\ (\exists(\alpha \leq A).B) & \triangleq & \Lambda(\beta \leq \top).\ \lambda(f : \forall(\alpha \leq A).\,\alpha \to \beta).f\ C\ e \\ \mathsf{unpack}\ e_1\ \mathsf{as}\ [\alpha, x]\ \mathsf{in}\ e_2 & \triangleq & e_1\ C\ (\Lambda(\alpha \leq A).\ \lambda(x : B).e_2) \\ & & \text{where } e_1 : \exists(\alpha \leq A).B \text{ and } e_2 : C \end{array} \tag{2.1}$$

Abadi et al. (1996) presented an encoding of objects using a combination of recursive types and bounded existential quantification, called the *ORBE* encoding. In their work, an interface $I(\alpha)$ is defined as a record of type-level functions, each having a self variable α argument ($\{l_i : I_i(\alpha)^{i \in 1 \ldots n}\}$). For example, the interface for the Point object is:

$$I_{\mathsf{Point}}(\alpha) \triangleq \{\mathsf{x} : \mathsf{Int},\ \mathsf{y} : \mathsf{Int},\ \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}$$

The general *ORBE* encoding for an interface $I(\alpha)$ is:

$$ORBE(I) \triangleq \mu\alpha.\ \exists(\beta \leq \alpha).\ \left\{ \begin{array}{l} \mathsf{self} : \beta, \\ {\mathsf{l}_i^{\mathsf{sel}} : \beta \to I_i(\beta)}^{i \in 1,\ldots,n}, \\ {\mathsf{l}_i^{\mathsf{upd}} : (\beta \to I_i(\beta)) \to \beta}^{i \in 1,\ldots,n} \end{array} \right\}$$

The bounded existential quantification ($\exists(\beta \leq \alpha)$) is used to indicate that the true type of an object can be a subtype of the object type α. Intuitively, it allows the object implementation to contain more fields, such as private variables, than the interface specifies. The field self is the object itself with all its methods including private ones so that the listed methods can access the object's private fields. Through fields l_i^{sel}, users of the object can access the object's public methods. The fields l_i^{upd} are optional in the encoding. They allow users to update method l_i by taking a new function of self and returning a new object with the updated method, which is a feature not supported in many other object encodings. For example, the Point object from above can be encoded with the *ORBE* encoding as follows:

$$\mathsf{Point}_{ORBE} \triangleq \mu\ \mathsf{pnt}.\ \exists(\beta \leq \mathsf{pnt}).\ \left\{ \begin{array}{ll} \mathsf{self} : \beta, & \\ \mathsf{x}^{\mathsf{sel}} : \beta \to \mathsf{Int}, & \mathsf{x}^{\mathsf{upd}} : (\beta \to \mathsf{Int}) \to \beta, \\ \mathsf{y}^{\mathsf{sel}} : \beta \to \mathsf{Int}, & \mathsf{y}^{\mathsf{upd}} : (\beta \to \mathsf{Int}) \to \beta, \\ \mathsf{move}^{\mathsf{sel}} : \beta \to \mathsf{Int} \to \mathsf{Int} \to \beta, & \\ \mathsf{move}^{\mathsf{upd}} : (\beta \to \mathsf{Int} \to \mathsf{Int} \to \beta) \to \beta & \end{array} \right\}$$

We can implement the Point object with the *ORBE* encoding as follows:

```
function mkPointORBE(x₁: Int, y₁: Int) = fold [Point_ORBE] (
  pack [Point_ORBE, {
     self  = mkPointORBE(x₁, y₁),
    x^sel = λ (self': Point_ORBE). x₁,
    y^sel = λ (self': Point_ORBE). y₁,
    move^sel = λ (self': Point_ORBE) x₂ y₂. mkPointORBE(x₁ + x₂, y₁ + y₂),
    . . . }] as ∃(β ≤ Point_ORBE). {. . .})
```

Method calls are encoded by unfoldings of iso-recursive types and unpacking of bounded existentials. For example, accessing the x field of a Point object can be implemented as:

```
function  getXORBE(p: Point_ORBE) =
   (unpack (unfold  [Point_ORBE] p) as [α, o] in o.x^sel(o.self) )
```

We omit the encodings for the method update fields in the mkPointORBE function for brevity, but they can also be written using $F^{\mu}_{\leq}$. More details about the encoding can be found in the original work (Abadi et al., 1996). As we can see, the *ORBE* encoding requires both recursive types and bounded existentials. By rewriting all bounded existentials into universal quantification using (2.1), we are able to write all the programs and types in the *ORBE* encoding presented above in our $F^{\mu}_{\leq}$ calculus. Therefore, $F^{\mu}_{\leq}$ can serve as a target language for the *ORBE* encoding.

When it comes to subtyping, as Bruce et al. (1999) observe, the *ORBE* encoding requires full $F_{\leq}$ for the bounded quantification subtyping rule. Consider the encoding for the object ColorPoint, which has more fields than Point. $\mathsf{ColorPoint}_{ORBE}$ should extend the record in Point_{ORBE} with $\mathsf{color}^{\mathsf{sel}}$ and $\mathsf{color}^{\mathsf{upd}}$ fields. When we try to compare the two encodings, we see that the bounds in ($\beta \leq \mathsf{pnt}$) for the two types are not the same – the recursive variable pnt in $\mathsf{ColorPoint}_{ORBE}$ stands for more fields than in Point_{ORBE}. As a result, contravariant subtyping is needed for comparing the bound in the existential type, which in turn requires full $F_{\leq}$ instead of kernel $F_{\leq}$. Therefore, we also study the full $F^{\mu}_{\leq}$ calculus in this paper, in order to support subtyping between objects in the *ORBE* encoding.

Encodings of Algebraic Datatypes with Subtyping. It is well-known that, in the polymorphic lambda calculus (System F) (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 worse 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 that captures intentional behavior of datatypes in the untyped lambda calculus and avoids the issues of Church encodings, is due to Scott (1962). Unfortunately, Scott encodings cannot be encoded in plain System F. The addition of recursive types to a polymorphic lambda calculus allows a typed Scott encoding (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 constant, addition, and subtraction constructors:

```
data  Exp1 = Num Int | Add Exp1 Exp1 | Sub Exp1 Exp1
```

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

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

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 an 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:

```
function eval (e : Exp₁) = (unfold [Exp₁] e) [Int] {
  num = λn. n,
  add = λe₁ e₂. (eval e₁ + eval e₂),
  sub = λe₁ e₂. (eval e₁ − eval e₂)
}
```

where we use [. . .] 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:

```
function Num₁ (n: Int) = fold [Exp₁] (Λ A. λ e. (e.num n))
function Add₁ (e₁ : Exp₁, e₂ : Exp₁) = fold [Exp₁] (Λ A. λ e. (e.add e₁ e₂))
function Sub₁ (e₁ : Exp₁, e₂ : Exp₁) = fold [Exp₁] (Λ A. λ e. (e.sub e₁ e₂))
```

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 Add_1 (Num_1 1) (Num_1 2), and get its evaluation result by calling eval (Add_1 (Num_1 1) (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:

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

The datatype Exp_2 differs from Exp_1 only in the new constructor: the other constructors are just the same. To reduce code duplication, it is desired that the constructor functions such as Add_1 can be polymorphic and used for both datatypes. 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 $F^{\mu}_{\leq}$ encoding for the Add constructor to have the following type, so that both encodings of Exp_1 and Exp_2 can use this constructor function:

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

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 $F^{\mu\wedge}_{\leq\geq}$, 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.

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 $\mathsf{Exp}_2 \longrightarrow \mathsf{Exp}_1$. 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 $F^{\mu\wedge}_{\leq\geq}$, 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, in a typical functional language one could define only Exp_2 and type desugar with the imprecise type $\mathsf{Exp}_2 \longrightarrow \mathsf{Exp}_2$, 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 $F^{\mu\wedge}_{\leq\geq}$ we do not need such compromises: 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. In particular, equi-recursive types are 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 on 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, an even 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, we show that kernel $F^{\mu}_{\leq}$ 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.

Full $F_{\leq}$ with iso-recursive types. We have also integrated full $F_{\leq}$ with iso-recursive subtyping. The most significant challenge compared to the kernel variant is proving the unfolding lemma. As we will discuss in §4.3, the method used to prove the generalized unfolding lemma for the kernel variant does not apply to the full variant due to the contravariance of the bounds. Therefore, a yet more sophisticated adaptation of the generalized unfolding lemma is required. Additionally, we establish several other properties for full $F^{\mu}_{\leq}$, such as type soundness, conservativity, and undecidability.

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. 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 unfold and fold 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.

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

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

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

Records as intersection types. It is well known that multi-field records can be encoded using intersection types and single field records (Reynolds, 1988; Dunfield, 2012). In $F^{\mu\wedge}_{\leq\geq}$

we follow this alternative approach to type record expressions. The record type $\{\mathsf{x} : \mathsf{nat}, \mathsf{y} : \mathsf{nat}\}$ in $F^{\mu}_{\leq}$ is now simply syntactic sugar in $F^{\mu\wedge}_{\leq\geq}$ for the intersection type $\{\mathsf{x} : \mathsf{nat}\}\&\{\mathsf{y} : \mathsf{nat}\}$. To avoid records with duplicate labels being intersected, we restrict the labels in the intersection types to be disjoint. For instance, $\{x : \mathsf{nat}\} \,\&\, \{x : \mathsf{nat} \to \mathsf{nat}\}$ is not a valid type in $F^{\mu\wedge}_{\leq\geq}$. We will further discuss such design choice in §5. The combination of unrestricted intersection types and iso-recursive subtyping was studied in Zhou et al. (2022). Our work models a restricted form of intersection types. In $F^{\mu\wedge}_{\leq\geq}$, only types that are formed by intersecting single-field record types are considered, and the disjointness relation discussed in Zhou et al. (2022) is in turn simplified to a compatibility relation for checking well-formedness of intersection types. The intersection type operator & is commutative and associative in terms of type equivalence, so that record permutations are obtained for free.

The typing rules for record expressions and projections in $F^{\mu\wedge}_{\leq\geq}$ are shown below:

$$\textsc{typing-srcd}\quad \frac{{l_i}^{\,i\in 1\cdots n} \text{ are disjoint} \qquad \Gamma \vdash e_i : A_i \quad \forall i, 1 \leq i \leq n}{\Gamma \vdash \{l_i = e_i{}^{\,i\in 1\cdots n}\} : \{l_1 : A_1\} \,\&\, \ldots \,\&\, \{l_n : A_n\}} \qquad \textsc{typing-sproj}\quad \frac{\Gamma \vdash e : \{l : A\}}{\Gamma \vdash e.l : A}$$

With intersection types, record expressions are now typed using rule TYPING-SRCD. As a result, we no longer need a dedicated rule for subtyping multi-field record types, which has a complicated definition since it needs to decide the subset inclusion of record fields and check the subtyping relation for common fields. Instead, we can now rely on the subtyping relation for intersection types and a direct subtyping rule for subtyping types in single-field records. Moreover, record projections can be defined in terms of subtyping now, as rule TYPING-SPROJ shows. When projecting a field from a record expression e, we can simply check if the record type of e is a subtype of the expected record type.

The treatment of records in $F^{\mu\wedge}_{\leq\geq}$ aligns closely with the way Dependent Object Calculus (DOT) (Rompf and Amin, 2016) deals with object types. Rule TYPING-DOT-OBJECT shows the typing rule for object expressions in DOT. In DOT object expressions are a record with a self reference variable x bounded to the object itself, containing a list of labeled declarations $d_1 \ldots d_n$. When type checking objects, each declaration is checked on its own, and the intersection of all the declaration types forms the type of the object.

$$\textsc{typing-dot-object}\quad \frac{{d_i}^{\,i\in 1\cdots n} \text{ have disjoint labels} \qquad \Gamma,\ x : A_i \vdash d_i : A_i \quad \forall i, 1 \leq i \leq n}{\Gamma \vdash \{x \Rightarrow d_1 \ldots d_n\} : \{x \Rightarrow A_1 \,\&\, \ldots \,\&\, A_n\}}$$

$F^{\mu\wedge}_{\leq\geq}$ and DOT share the same idea of using intersection types to type record expressions or objects and both require the labels to be disjoint. Due to the use of path-dependent types (Amin et al., 2014) in DOT, the treatment of recursive types is different. In $F^{\mu\wedge}_{\leq\geq}$ we do not have a self reference variable in record expressions or types, and $F^{\mu\wedge}_{\leq\geq}$ lacks of some DOT features. On the other hand $F^{\mu\wedge}_{\leq\geq}$ has key properties, such as decidability, transitivity of subtyping, and being a conservative extension of $F_{\leq}$, which are partly missing in DOT. Despite these differences, we hope that $F^{\mu\wedge}_{\leq\geq}$ can provide insights into the design of DOT-like calculi with bounded quantification and recursive types and complement the existing work in terms of the design space.

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

Fig. 1: Syntax of $F_{\leq}^{\mu}$.

3 Bounded Quantification with Iso-Recursive Types

This section introduces a new calculus, called $F_{\leq}^{\mu}$, integrating bounded quantification, record types and recursive types. We show two variants of $F_{\leq}^{\mu}$. One is kernel $F_{\leq}^{\mu}$, by adopting the kernel rule for subtyping bounded quantification from kernel $F_{\leq}$ (Cardelli and Wegner, 1985). The other one is full $F_{\leq}^{\mu}$, which instead adopts the full rule for subtyping bounded quantification from full $F_{\leq}$ (Curien and Ghelli, 1992; Cardelli et al., 1994).

3.1 Kernel $F_{\leq}$ with Iso-recursive Subtyping

Firstly, we introduce how to combine kernel bounded quantification, multi-field records and iso-recursive subtyping in kernel $F_{\leq}^{\mu}$.

Syntax and Well-Formedness. The syntax of types and contexts for $F_{\leq}^{\mu}$ is shown in Figure 1. Meta-variables A, B, C, D range over types. Types consist of natural numbers (nat), the top type ($\top$), function types ($A \rightarrow B$), type variables (α), recursive types ($\mu\alpha.\, A$), labeled types (A^{α}), universal types ($\forall(\alpha \leq A).\, B$), and record types ($\{l_i : A_i\ ^{i \in 1 \cdots n}\}$). Labeled 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 unlabeled, then the two types will not be related, even when, ignoring the labels, they would be structurally compatible. Expressions, denoted by the 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 ($\mathsf{fold}\ [A]\ e$), unfold expressions ($\mathsf{unfold}\ [A]\ e$), 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 not necessary 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 at the top of Figure 2.

Subtyping for kernel $F_{\leq}^{\mu}$. The bottom of Figure 2 shows the subtyping judgement. Our subtyping rules are mostly standard. The rules essentially include the rules of the algorithmic

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

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

$$\textsc{wft-arrow}\ \frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \to A_2} \qquad \textsc{wft-rec}\ \frac{\Gamma,\ \alpha \leq \top \vdash A}{\Gamma \vdash \mu\alpha.A} \qquad \textsc{wft-label}\ \frac{\Gamma \vdash A}{\Gamma \vdash A^{\alpha}} \qquad \textsc{wft-rcd}\ \frac{\Gamma \vdash A_i \qquad \text{for each } i}{\Gamma \vdash \{l_i : A_i\ ^{i \in 1 \cdots n}\}}$$

$\boxed{\Gamma \vdash A \leq B}$ *(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. 2: Well-formedness and subtyping rules for kernel $F^{\mu}_{\leq}$.

version of kernel $F_{\leq}$ (Cardelli and Wegner, 1985; Cardelli et al., 1994), but the rule for bounded quantification is generalized. The rules S-var and S-vartrans are standard $F_{\leq}$ rules. 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. We have placed well-formedness checks on all the base cases of the subtyping rules, which ensures that the context and types in a subtyping relation are well-formed, as shown in Lemma 3.1. Note that for this regularity property to hold, in rule S-rcd we requires the left record type to be well-formed but not the right one, since the well-formedness of the right type can be derived from the subtyping relation on record fields, while there might be extra fields in the left record type that are not compared in the subtyping relation.

Lemma 3.1 (Regularity of subtyping). If $\Gamma \vdash A \leq B$, then the following well-formedness conditions hold: (1) $\vdash \Gamma$, (2) $\Gamma \vdash A$, and (3) $\Gamma \vdash B$.

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. Rule S-EQUIVALL is more general than rule S-KERNELALL since the latter requires the bounds to be 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 (Pottinger, 1980; Coppo et al., 1981; Barbanera et al., 1995), we can have such equivalent but not syntactically equal types. Therefore, we should generalize the rule for bounded quantification to deal with those cases. This generalization to equivalent bounds retains decidable subtyping just as kernel $F_{\leq}$ as we shall see in §4.3.

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 $F_{\leq}^{\mu}$, 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 $F_{\leq}^{\mu}$. 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. For those new cases, the proofs are all straightforward from the induction hypothesis.

Lemma 3.2 (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 3.3 (Reflexivity). If $\vdash \Gamma$ and $\Gamma \vdash A$ then $\Gamma \vdash A \leq A$.

Theorem 3.4 (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 type preservation in a calculus with iso-recursive subtyping. A key difficulty in the formalization of $F_{\leq}^{\mu}$ 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 3.5 (Unfolding Lemma). If $\Gamma \vdash \mu\alpha.\,A \le \mu\alpha.\,B$ then $\Gamma \vdash [\alpha \mapsto \mu\alpha.\,A]\ A \le [\alpha \mapsto \mu\alpha.\,B]\ B$.

3.2 Full $F_{\le}$ with Iso-recursive Subtyping

In full $F^{\mu}_{\le}$, which incorporates full $F_{\le}$ and iso-recursive types, the sole distinction from the kernel variant of $F^{\mu}_{\le}$ lies in permitting contravariant bounds, so we only present the differences between the two variants.

Syntax and subtyping. The syntax of the full $F^{\mu}_{\le}$ is identical to that of the kernel $F^{\mu}_{\le}$ (§3.1). As for the subtyping rules, rule S-EQUIVALL is replaced by rule S-FULLALL in full $F^{\mu}_{\le}$.

S-EQUIVALL

$$\frac{\Gamma \vdash A_1 \le A_2 \qquad \Gamma \vdash A_2 \le A_1 \qquad \Gamma,\ \alpha \le A_2 \vdash B \le C}{\Gamma \vdash \forall(\alpha \le A_1).\,B \le \forall(\alpha \le A_2).\,C}$$

S-FULLALL

$$\frac{\Gamma \vdash A_2 \le A_1 \qquad \Gamma,\ \alpha \le A_2 \vdash B \le C}{\Gamma \vdash \forall(\alpha \le A_1).\,B \le \forall(\alpha \le A_2).\,C}$$

The only distinction between these two rules lies in the variance of the bounds: rule S-FULLALL permits contravariance, allowing A_2 to be a subtype of A_1, whereas rule S-EQUIVALL demands A_2 to be equivalent to A_1. The change to the subtyping rules in full $F^{\mu}_{\le}$ does not impact many subtyping lemmas, such as reflexivity (Theorem 3.3) and transitivity (Theorem 3.4), which remain provable by reusing proof techniques from full $F_{\le}$. However, as we shall see in §4.1, the unfolding lemma (Lemma 3.5) needs a different proof technique due to the presence of contravariant bounds in full $F^{\mu}_{\le}$. In §4.1 we will present a new generalized unfolding lemma that can be proved in both kernel and full $F^{\mu}_{\le}$. With this new lemma, we can prove the unfolding lemma (Lemma 3.5) for full $F^{\mu}_{\le}$.

3.3 Typing, Reduction and Type Soundness

The two variants of the subtyping rules have no impact on proving type soundness. Therefore, the typing and reduction rules remain consistent across both variants. Figure 3 shows the typing rules and reduction rules. Most rules are standard except for the typing rules for unfold and fold. For these two expressions we use structural rules instead (rule TYPING-SUNFOLD and rule TYPING-SFOLD), as 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*:

Lemma 3.6 (Structural unfolding lemma). If $\Gamma \vdash \mu\alpha.\,A \le \mu\alpha.\,C \le \mu\alpha.\,D \le \mu\alpha.\,B$ then $\Gamma \vdash [\alpha \mapsto \mu\alpha.\,C]\ A \le [\alpha \mapsto \mu\alpha.\,D]\ B$.

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

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

TYPING-NAT
$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{i} : \mathsf{nat}}$$

TYPING-VAR
$$\frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x : A}$$

TYPING-SUB
$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash A \le B}{\Gamma \vdash e : B}$$

TYPING-ABS
$$\frac{\Gamma,\ x : A_1 \vdash e : A_2}{\Gamma \vdash \lambda x : A_1.\ e : A_1 \to A_2}$$

TYPING-APP
$$\frac{\Gamma \vdash e_1 : A_1 \to A_2 \qquad \Gamma \vdash e_2 : A_1}{\Gamma \vdash e_1\ e_2 : A_2}$$

TYPING-TABS
$$\frac{\Gamma,\ \alpha \le A \vdash e : B}{\Gamma \vdash \Lambda(\alpha \le A).\ e : \forall(\alpha \le A). B}$$

TYPING-TAPP
$$\frac{\Gamma \vdash e : \forall(\alpha \le B_1). B_2 \qquad \Gamma \vdash A \le B_1}{\Gamma \vdash e\ A : [\alpha \mapsto A]\ B_2}$$

TYPING-SFOLD
$$\frac{\Gamma \vdash e : [\alpha \mapsto B]\ A \qquad \Gamma \vdash \mu\alpha. A \le B}{\Gamma \vdash \mathsf{fold}\ [B]\ e : B}$$

TYPING-SUNFOLD
$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash A \le \mu\alpha. B}{\Gamma \vdash \mathsf{unfold}\ [A]\ e : [\alpha \mapsto A]\ B}$$

TYPING-RCD
$$\frac{l_i\ ^{i\in 1\cdots n} \text{ are disjoint} \qquad \Gamma \vdash e_i : A_i \quad \forall i, 1 \le i \le n}{\Gamma \vdash \{l_i = e_i\ ^{i\in 1\cdots n}\} : \{l_i : A_i\ ^{i\in 1\cdots n}\}}$$

TYPING-PROJ
$$\frac{\Gamma \vdash e : \{l_i : A_i\ ^{i\in 1\cdots n}\}}{\Gamma \vdash e.l_i : A_i}$$

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

STEP-BETA
$$\frac{}{(\lambda x : A.\ e_1)\ v_2 \hookrightarrow [x \mapsto v_2]\ e_1}$$

STEP-APPL
$$\frac{e_1 \hookrightarrow e_1'}{e_1\ e_2 \hookrightarrow e_1'\ e_2}$$

STEP-APPR
$$\frac{e_2 \hookrightarrow e_2'}{v_1\ e_2 \hookrightarrow v_1\ e_2'}$$

STEP-FLD
$$\frac{}{\mathsf{unfold}\ [A]\ (\mathsf{fold}\ [B]\ v) \hookrightarrow v}$$

STEP-UNFOLD
$$\frac{e \hookrightarrow e'}{\mathsf{unfold}\ [A]\ e \hookrightarrow \mathsf{unfold}\ [A]\ e'}$$

STEP-FOLD
$$\frac{e \hookrightarrow e'}{\mathsf{fold}\ [A]\ e \hookrightarrow \mathsf{fold}\ [A]\ e'}$$

STEP-TAPP
$$\frac{e_1 \hookrightarrow e_2}{e_1\ A \hookrightarrow e_2\ A}$$

STEP-TABS
$$\frac{}{(\Lambda(\alpha \le A).\ e)\ B \hookrightarrow [\alpha \mapsto B]\ e}$$

STEP-PROJ
$$\frac{e \hookrightarrow e'}{e.l_j \hookrightarrow e'.l_j}$$

STEP-PROJRCD
$$\frac{}{\{l_i = v_i\ ^{i\in 1\cdots n}\}.l_j \hookrightarrow v_j}$$

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. 3: Typing and Reduction Rules.

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

Fig. 4: Structural unfolding derivation.

lemma proposed by Zhou et al. (2022), the recursive types that get substituted in the bodies are the same. As §4.1 and §5.3 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 the typing derivation of an expression unfold $[D']$ (fold $[C']$ e) in Figure 4. 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 $\mu\alpha.D$. The type of unfold $[D']$ (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 3.7 (Preservation). If $\vdash e : A$ and $e \hookrightarrow e'$ then $\vdash e' : A$.

Theorem 3.8 (Progress). If $\vdash e : A$ then e is a value or $e \hookrightarrow e'$ for some e'.

3.4 Algorithmic Typing

The rules that we have presented in Figure 3 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 5 shows the algorithmic rules for typing. Compared with the declarative typing rules, the subsumption rule (TYPING-SUB) is removed. Also, the application (TYPING-APP), type application (TYPING-TAPP), structural folding (TYPING-SFOLD), structural unfolding (TYPING-SUNFOLD) and record projection (TYPING-PROJ) rules are replaced by rules ATYP-APP, ATYP-TAPP, ATYP-SFOLD ATYP-SUNFOLD and ATYP-PROJ, 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). Consider the term

$$(\Lambda(\alpha \leq \mathsf{nat} \to \mathsf{nat}).\ \lambda(y : \alpha).\ y\ 5) : \forall(\alpha \leq \mathsf{nat} \to \mathsf{nat}).\alpha \to \mathsf{nat}.$$

Without the upper exposure function in the rule ATYP-TAPP, the y in the function body would be typed as its minimal type α, which cannot be unified with a function type with the argument type nat. The exposure function finds the smallest type that matches the expected type, such as the function type for the argument y in the example above. Thus the upper exposure function plays an important role in finding the minimal type with the algorithmic typing rules. To make our rules more general, we additionally define the *lower*

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

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

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

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

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

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

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

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

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

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

$$\textsc{ATyp-rcd}\quad \frac{{l_i}^{\,i\in 1\cdots n} \text{ are disjoint} \qquad \Gamma \vdash_a e_i : A_i \quad \forall i, 1 \leq i \leq n}{\Gamma \vdash_a \{l_i = {e_i}^{\,i\in 1\cdots n}\} : \{l_i : {A_i}^{\,i\in 1\cdots n}\}}$$

$$\textsc{ATyp-proj}\quad \frac{\Gamma \vdash_a e : A \qquad \Gamma \vdash A \Uparrow \{l_i : {A_i}^{\,i\in 1\cdots n}\}}{\Gamma \vdash_a e.l_i : A_i}$$

Fig. 5: Algorithmic Typing.

exposure function ($\Gamma \vdash A \Downarrow B$) to find the greatest non-variable subtype B for A. For $F_{\leq}^{\mu}$, 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 $\mu\alpha.\top$, so that it is valid to type check expressions accepted by the structural rule TYPING-SFOLD like

$$(\mathsf{fold}[\top]\ 1) : \mu\alpha.\top$$

with the algorithmic typing rules. 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 3.9 (Soundness of the algorithmic rules). If $\Gamma \vdash_a e : A$ then $\Gamma \vdash e : A$.

Theorem 3.10 (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 3.10 implies that our algorithm can always find a minimal type, which is an important property for $F_{\leq}^{\mu}$.

It should be noted that there is no difference in terms of algorithmic typing rules for both variants of $F_{\leq}^{\mu}$, though for full $F_{\leq}^{\mu}$, the algorithm might not terminate, since subtyping is undecidable for full $F_{\leq}^{\mu}$.

4 Metatheory of $F_{\leq}^{\mu}$

In this section we discuss the most interesting and difficult aspects of the metatheory of $F_{\leq}^{\mu}$ in more detail. We cover three key properties: the *unfolding lemma* for $F_{\leq}^{\mu}$, the *conservativity* of $F_{\leq}^{\mu}$ over the original $F_{\leq}$ and *(un)decidability* of subtyping. The interaction between iso-recursive types and bounded quantification requires significant changes in the proofs of the unfolding lemma and decidability. In addition, we show how to prove the conservativity of $F_{\leq}^{\mu}$ over $F_{\leq}$ using the algorithmic formulation of $F_{\leq}^{\mu}$.

4.1 Unfolding Lemma

The unfolding lemma (Lemma 3.5) 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 first review previous approaches for proving the unfolding lemma, which do not account for bounded quantification or only apply to kernel $F_{\leq}^{\mu}$. Then we show how to generalize the unfolding lemma to address all the complications arising from the interaction of iso-recursive subtyping and bounded quantification, including the case of full $F_{\leq}^{\mu}$.

The generalized unfolding lemma for iso-recursive subtyping. We first review the unfolding lemma for the special case of iso-recursive subtyping without bounded quantification. Because the premise of the unfolding lemma is restricted to a subtyping relation between two recursive types $\mu\alpha.A \leq \mu\alpha.B$ instead of two generic types A and B, a direct induction on the premise is problematic, as it fails to provide a useful induction hypothesis for reasoning with nominal unfoldings like $[\alpha \mapsto A^{\alpha}]\ A$, where the type after substitution may not be a recursive type. In Zhou et al. (2022)'s work the unfolding lemma is generalized to the following form:

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

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

Due to the tricky interaction between rule S-VAR and rule S-ARROW, in the generalized unfolding lemma we need two mutually dependent lemmas: one for covariant cases (1) and the other for contravariant cases (2). The proof for Lemma 4.1 proceeds by induction on $\Gamma_1, \alpha, \Gamma_2, \vdash A \le B$. 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 $\Gamma \vdash B_1 \le A_1$ relating the input types of the function.

The generalized unfolding lemma for kernel $F^{\mu}_{\le}$. When bounded quantification is taken into account, Lemma 4.1 is unfortunately not general enough. The key difference is that now the contexts are no longer just a list of type variables, but also associate a type bound with each type variable. Moreover, the bounds are dependent on the type variables in the order they appear, so that in the context Γ_2, the bounds may contain the type variable α. Since rule S-VARTRANS may look up a bound in the context Γ_2, and compare it with the right-hand side of the subtyping relation, to apply the induction hypothesis, the bound type in the context should be equal to the substitution form $[\alpha \mapsto ?^{\alpha}]U$ as in the subtyping relation. To address this issue, Zhou et al. (2023) extends the unfolding lemma to the following form:

Lemma 4.2 (The generalized unfolding lemma for kernel $F^{\mu}_{\le}$ in Zhou et al. (2023)). If (1) $\Gamma_1,\ \alpha \le \top,\ \Gamma_2 \vdash A \le B$, (2) $\Gamma_1 \vdash \mu\alpha.C \le \mu\alpha.S$ and (3) $\Gamma_1 \vdash \mu\alpha.S \le \mu\alpha.D$ then

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

We explain a few key points in the proof of Lemma 4.2 together with some proof sketches below.

Firstly, *the context* Γ_2 *now comes with a substitution.* 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. With substitutions in the context Γ_2, in case S-VARTRANS and S-EQUIVALL, the premise from inversion can have the same form as the original premise and thus the induction hypothesis can be applied. For example, in case S-VARTRANS, assume that $A := \beta$, $B := B$, and Γ_2 contains a bound $\beta \le U$. We need to prove the following goal:

$$\Gamma_1, \Gamma_2[\alpha \mapsto \mu\alpha.S] \vdash \beta \le [\alpha \mapsto \mu\alpha.D]\ B$$

By analyzing the context we know that $\beta \le [\alpha \mapsto \mu\alpha.S]\ U \in \Gamma_2[\alpha \mapsto \mu\alpha.S]$, so we only need to show

$$\Gamma_1, \Gamma_2[\alpha \mapsto \mu\alpha.S] \vdash [\alpha \mapsto \mu\alpha.S]\ U \le [\alpha \mapsto \mu\alpha.D]\ B.$$

This can be proved by instantiating the induction hypothesis with $C := S$ and $D := D$. In this way, we overcome the issue with the substitutions in the context Γ_2.

Secondly, note that *the substituted type is neither C nor D, but an intermediate type S that lies between C and D*. This is to ensure that the induction hypothesis can be applied to both the contravariant and covariant subgoals in case S-ARROW. Otherwise, consider an alternative lemma where S is fixed to be D. In case S-ARROW, assume that $A := A_1 \to A_2$ and $B := B_1 \to B_2$. We need to prove two subgoals:

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

If one follows the original proof steps in Lemma 4.1, the induction hypothesis has to be instantiated with $C := D$ and $D := C$, so that the substitution matches with the two types in the subtyping relation for the contravariant subgoal (2). In that case, the substituted type in the context Γ_2 becomes $\mu\alpha.C$, which cannot be used to prove the subgoal (2). Therefore, in Lemma 4.2 an intermediate type S is introduced to decouple the substitution in the context and in the subtyping relation for the function case. In other words, the invariant substitution with type $\mu\alpha.S$ to the context makes the induction hypothesis applicable to both subgoals, regardless of the substitution in the subtyping relation.

Finally, *having the intermediate type S will not affect the inductive proof for case* S-EQUIVALL *in kernel $F^{\mu}_{\le}$*. We assume that $A := \forall(\beta \le A_1).A_2$ and $B := \forall(\beta \le B_1).B_2$. The goal would look like:

$$\Gamma_1, \Gamma_2[\alpha \mapsto \mu\alpha.S] \vdash [\alpha \mapsto \mu\alpha.C]\ \forall(\beta \le A_1).A_2 \le [\alpha \mapsto \mu\alpha.D]\ \forall(\beta \le B_1).B_2$$

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

$$\Gamma_1, \Gamma_2[\alpha \mapsto \mu\alpha.S],\ \beta \le [\alpha \mapsto \mu\alpha.D]\ B_1 \vdash [\alpha \mapsto \mu\alpha.C]\ A_2 \le [\alpha \mapsto \mu\alpha.D]\ B_2$$

To apply the induction hypothesis, we need to unify the new bound $\beta \le [\alpha \mapsto \mu\alpha.D]\ B_1$ and the existing context $\Gamma_2[\alpha \mapsto \mu\alpha.S]$ into the same substitution form. In other words, we need to show the following two environments are equivalent:

$$\Gamma_2[\alpha \mapsto \mu\alpha.S],\ \beta \le [\alpha \mapsto \mu\alpha.D]\ B_1 \equiv (\Gamma_2, \beta \le B_1)[\alpha \mapsto \mu\alpha.S] \tag{4.1}$$

This can be done by showing that $\mu\alpha.S$ is equivalent to $\mu\alpha.D$. To prove this, we rely on the fact that the bounds $[\alpha \mapsto C^{\alpha}]A_1$ and $[\alpha \mapsto D^{\alpha}]B_1$ are equivalent, and the following inversion lemma on substitution:

Lemma 4.3 (Substitution inversion). If A is equivalent to B, and $[\alpha \mapsto C^{\alpha}]\ A$ is equivalent to $[\alpha \mapsto D^{\alpha}]\ B$, then either C is equivalent to D or α is not in A nor B.

For the first case we get from Lemma 4.3, by the fact that S lies in the middle of C and D, we can show that all the three types $\mu\alpha.C$,$\mu\alpha.S$ and $\mu\alpha.D$ are equivalent. For the second case, since α is not in A_1 nor A_2, we can freely rewrite the substituted types to any types in the context. In either case, the rewriting in (4.1) can be achieved, so that the induction hypothesis can be applied to the subgoal. The critical point here is that, although the substitution form in the context is indeed affected by the new bound introduced by rule S-EQUIVALL, since kernel $F^{\mu}_{\leq}$ requires all pairs of the bounds in the subtyping relation to be equivalent, the type S will converge into the types C and D in the end.

The generalized unfolding lemma for kernel $F^{\mu}_{\leq\geq}$. In Zhou et al. (2023)'s work, an extension to kernel $F^{\mu}_{\leq}$ is proposed, called $F^{\mu}_{\leq\geq}$, which extends kernel $F^{\mu}_{\leq}$ with lower bounded quantification and bottom types. It is worth noting that these new features will break the proof of Lemma 4.2 we have discussed above. The interaction of lower bounds and upper bounds invalidates the following inversion lemma for rule S-VARTRANS, which has been used to prove Lemma 4.2:

Lemma 4.4. If $\alpha \leq A \in \Gamma$ and $\Gamma \vdash \alpha \leq B$, where $\alpha \neq B$, then $\Gamma \vdash A \leq B$.

In Lemma 4.2, there is more than one subtyping statement on the premises related to type A and B. During the proof we do induction on the premise (1), and use the inversion lemma to match the subderivation of $[\alpha \mapsto C^{\alpha}]\ A \leq [\alpha \mapsto D^{\alpha}]\ B$ with the induction hypothesis we get from the premise (1). Lemma 4.4 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:

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

To avoid using this inversion lemma in the proof of unfolding lemma, we need to refine the generalized unfolding lemma for kernel $F^{\mu}_{\leq}$:

Lemma 4.5 (The generalized unfolding lemma for $F^{\mu}_{\leq\geq}$ in Zhou et al. (2023)). If
1. $\Gamma_1,\ \alpha \leq \top,\ \Gamma_2 \vdash A$ and $\Gamma_1,\ \alpha \leq \top,\ \Gamma_2 \vdash B$;
2. $G \vdash [\alpha \mapsto C^{\alpha}]\ A \leq [\alpha \mapsto D^{\alpha}]\ B$;
3. G differs from $\Gamma_1,\ \alpha \leq \top,\ \Gamma_2[\alpha \mapsto S^{\alpha}]$ only in the components labeled by α, where S^{α} can be replaced by T^{α} that satisfies $\Gamma_2 \vdash \mu\alpha.S \leq \mu\alpha.T$ and $\Gamma_2 \vdash \mu\alpha.T \leq \mu\alpha.S$,

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

In the refined generalized unfolding lemma, we integrate the two subtyping statements into one statement, by having the premise (1) in Lemma 4.2 induced implicitly. This can be justified by the fact that the one-time unfolding is implicitly derived from the nominal unfolding:

Lemma 4.6 (Inversion on nominal unfoldings). If $\Gamma \vdash [\alpha \mapsto C^{\alpha}]\ A \leq [\alpha \mapsto D^{\alpha}]\ B$ then $\Gamma \vdash A \leq B$.

To accommodate this change, the context is also refined to be more general, by allowing the substitution type in the subtyping context of premise (2) to be any type T equivalent to S. We omit the detailed proof for this lemma, since it was done in the appendix of Zhou et al. (2023), and is no longer used in the generalized unfolding lemma for full $F^{\mu}_{\leq}$ and $F^{\mu\wedge}_{\leq\geq}$

we will present in this paper. However, the idea of using Lemma 4.6 to avoid the issue of using Lemma 4.4 still applies, as we will show next.

The generalized unfolding lemma for full $F^{\mu}_{\leq}$. It is not straightforward to generalize the unfolding lemma to full $F^{\mu}_{\leq}$. As we have seen in the last observation of Lemma 4.2, the proof relies on the fact that the bounds in the subtyping relation are equivalent, so that during the proof, the intermediate type $\mu\alpha.\,S$ can be rewritten to $\mu\alpha.\,C$ or $\mu\alpha.\,D$. However, in full $F^{\mu}_{\leq}$ we use rule S-FULLALL, which fails to maintain the equivalence of the bounds. We need to consider a new approach to generalize the unfolding lemma for full $F^{\mu}_{\leq}$.

If we revisit the use of the generalized substitution type S in the context Γ_2 in Lemma 4.2, we can see that, during the proof, in most cases we just pass the same type S around in the induction hypothesis. The exception is for the case of S-VARTRANS, where the type S is instantiated to C or D. This suggests that the issues with the unfolding lemma in full $F^{\mu}_{\leq}$ can be solved if we can find a different approach to the S-VARTRANS case and remove the intermediate type S from the generalized unfolding lemma. In that case, the requirement for the equivalent bounds can also be lifted.

To this end, we note that the introduction of the type S in Lemma 4.2 is essentially an over-generalization, since throughout the derivation of $[\alpha \mapsto C^{\alpha}]\ A \leq [\alpha \mapsto D^{\alpha}]\ B$, only the substitution C or D will be introduced into the context. Thus, the generalized unfolding lemma can focus only on the substitution of C or D. However, in full $F^{\mu}_{\leq}$, it can happen that we cannot find a uniform substitution type for the variable α in the context Γ_2. Due to the contravariant subtyping in rule S-ARROW, the substitutions can flip between the two sides of the subtyping relation, so both C and D can be introduced into the context. Therefore, we define the following notion of related contexts to characterize such contexts:

Definition 4.7 (Related contexts). Given a type variable α, an initial context Γ_0 and two types $\mu\alpha.\,C, \mu\alpha.\,D$, two contexts Γ and Γ_μ are related, written $\Gamma \cong \Gamma_\mu$, if they can be derived using the following rules:

$\boxed{\Gamma \cong \Gamma_\mu}$ *(Related contexts)*

EXTENV-BASE
$$\frac{\vdash \Gamma_0 \qquad \alpha \notin \text{dom}\ \Gamma_0}{\Gamma_0, \alpha \leq \top \cong \Gamma_0}$$

EXTENV-CONSC
$$\frac{\Gamma' \cong \Gamma'_\mu \qquad \beta \notin \text{dom}\ \Gamma_0}{\Gamma', \beta \leq [\alpha \mapsto C^{\alpha}]A \cong \Gamma'_\mu, \beta \leq [\alpha \mapsto \mu\alpha.\,C]A}$$

EXTENV-CONSD
$$\frac{\Gamma' \cong \Gamma'_\mu \qquad \beta \notin \text{dom}\ \Gamma_0}{\Gamma', \beta \leq [\alpha \mapsto D^{\alpha}]A \cong \Gamma'_\mu, \beta \leq [\alpha \mapsto \mu\alpha.\,D]A}$$

The definition of related contexts is parameterized by a type variable α and a subtyping relation $\Gamma_0 \vdash \mu\alpha.\,C \leq \mu\alpha.\ D$[3]. As we will see, the two contexts Γ and Γ_μ are essentially the subtyping contexts that will be used in the generalized unfolding lemma for the premise and

[3] We say that the related contexts are parameterized by the subtyping relation is just for the sake of convenience. They are actually parameterized by the components in the subtyping relation, i.e. the variable α, the shared context Γ_0, and the types C and D. Whether the subtyping relation $\Gamma_0 \vdash \mu\alpha.\,C \leq \mu\alpha.\,D$ holds does not matter, as we shall see shortly.

the conclusion respectively. They extend the context Γ_0 with new bindings that are either under the substitution C or D. Moreover, each pair of bindings in the two contexts should be matched in terms of the substitution type C or D and the base type A should be the same. For example, consider the following instance of the unfolding lemma we aim to prove:

$$\frac{\Gamma_1 \vdash [\alpha \mapsto C^{\alpha}](A_1 \to A_2) \le [\alpha \mapsto D^{\alpha}](B_1 \to B_2)}{\Gamma_2 \vdash [\alpha \mapsto \mu\alpha.C](A_1 \to A_2) \le [\alpha \mapsto \mu\alpha.D](B_1 \to B_2)}$$

$$\begin{array}{rl} \text{where} & \Gamma_1 = \alpha \le \top, \beta \le [\alpha \mapsto C^{\alpha}]T_1, \gamma \le [\alpha \mapsto D^{\alpha}]T_2 \\ & \Gamma_2 = \beta \le [\alpha \mapsto \mu\alpha.C]T_1, \gamma \le [\alpha \mapsto \mu\alpha.D]T_2 \\ \text{and} & A_1, A_2, B_1, B_2, T_1, T_2 \text{ can be any well-formed types.} \end{array}$$

By definition, Γ_1 and Γ_2 are related under the subtyping relation $\cdot \vdash \mu\alpha.C \le \mu\alpha.D$, i.e. $\Gamma_1 \cong \Gamma_2$. When proving the above goal, for the contravariant case of function types, we need to show that the following holds:

$$\frac{\Gamma_1 \vdash [\alpha \mapsto D^{\alpha}]B_1 \le [\alpha \mapsto C^{\alpha}]A_1}{\Gamma_2 \vdash [\alpha \mapsto \mu\alpha.D]B_1 \le [\alpha \mapsto \mu\alpha.C]A_1}$$

To prove this by induction, Γ_1 and Γ_2 should be related under the subtyping relation $\cdot \vdash \mu\alpha.D \le \mu\alpha.C$, that flips the order of $\mu\alpha.C$ and $\mu\alpha.D$ in the parameter of the related contexts. This is possible because the related contexts are defined to be symmetric in terms of the substitution types C and D. This flexibility allows us to prove the generalized unfolding lemma for full $F^{\mu}_{\le}$ without the need for the intermediate type S as in Lemma 4.2 to handle the contravariance.

With the notion of related contexts, we can prove inversion lemmas for looking up the bounds in the related contexts:

Lemma 4.8 (Inversion lemma for related contexts). If Γ and Γ_μ are related under the variable α, the shared context Γ_0, and the types $\mu\alpha.C$ and $\mu\alpha.D$, then for any type variable β, if $\beta \le U \in \Gamma$ and $\beta \ne \alpha$, one of the following holds:

1. there exists U', s.t. $U = [\alpha \mapsto C^{\alpha}]U'$ and $\beta \le [\alpha \mapsto \mu\alpha.C]U' \in \Gamma_\mu$, or
2. there exists U', s.t. $U = [\alpha \mapsto D^{\alpha}]U'$ and $\beta \le [\alpha \mapsto \mu\alpha.D]U' \in \Gamma_\mu$.

We can now state the generalized unfolding lemma for full $F^{\mu}_{\le}$:

Lemma 4.9 (The generalized unfolding lemma for full $F^{\mu}_{\le}$). If contexts Γ and Γ_μ are related under variable α and if $\Gamma_0 \vdash \mu\alpha.C \le \mu\alpha.D$, then

1. $\Gamma \vdash [\alpha \mapsto C^{\alpha}]A \le [\alpha \mapsto D^{\alpha}]B$ implies $\Gamma_\mu \vdash [\alpha \mapsto \mu\alpha.C]A \le [\alpha \mapsto \mu\alpha.D]B$
2. $\Gamma \vdash [\alpha \mapsto D^{\alpha}]A \le [\alpha \mapsto C^{\alpha}]B$ implies $\Gamma_\mu \vdash [\alpha \mapsto \mu\alpha.D]A \le [\alpha \mapsto \mu\alpha.C]B$
3. $\Gamma \vdash [\alpha \mapsto C^{\alpha}]A \le [\alpha \mapsto C^{\alpha}]B$ implies $\Gamma_\mu \vdash [\alpha \mapsto \mu\alpha.C]A \le [\alpha \mapsto \mu\alpha.C]B$
4. $\Gamma \vdash [\alpha \mapsto D^{\alpha}]A \le [\alpha \mapsto D^{\alpha}]B$ implies $\Gamma_\mu \vdash [\alpha \mapsto \mu\alpha.D]A \le [\alpha \mapsto \mu\alpha.D]B$

Compared to previous versions of the unfolding lemma, in addition to conclusion (1) and (2), which talk about the covariant and contravariant substitutions, now we add two more conclusions (3) and (4) for the case where the substitutions are the same as C or D. As we will show in the proof sketch below, these two additional conclusions generate useful induction hypotheses for the proof of rule S-VARTRANS, so that we no longer need to rely on the intermediate type S as we did in Lemma 4.2. Moreover, we follow the same approach as in Lemma 4.9 to drop the premise of $A \le B$ and avoid the use of inversion lemmas.

Proof We prove the four mutually dependent goals in the lemma by induction on the premise $\Gamma \vdash [\alpha \mapsto ?^{\alpha}]A \leq [\alpha \mapsto ?^{\alpha}]B$. We unroll the mutual induction hypothesis here and assume four separate induction hypotheses available in the proof for the sake of presentation. In the rest of the proof we will refer to them as IH(n) for the induction hypothesis generated by the proof goal (n). We show the interesting cases below:

- Rule S-VARTRANS: We show the proof goal (1) here. Assume $A = \beta$, where $\beta \leq U \in \Gamma$, and $\Gamma \vdash U \leq [\alpha \mapsto D^{\alpha}]B$, by Lemma 4.8 we get two cases:
  1. There exists U', $U = [\alpha \mapsto C^{\alpha}]U'$ and $\beta \leq [\alpha \mapsto \mu\alpha. C]U' \in \Gamma_{\mu}$. We need to show $\Gamma_{\mu} \vdash [\alpha \mapsto \mu\alpha. C]U' \leq [\alpha \mapsto \mu\alpha. D]B$. We can apply IH(1) directly.
  2. There exists U', $U = [\alpha \mapsto D^{\alpha}]U'$ and $\beta \leq [\alpha \mapsto \mu\alpha. D]U' \in \Gamma_{\mu}$. We need to show $\Gamma_{\mu} \vdash [\alpha \mapsto \mu\alpha. D]U' \leq [\alpha \mapsto \mu\alpha. D]B$. We can apply IH(4) directly. Note that in this case, the substituted type on both sides is D, which motivated us to state cases (3) and (4) in the lemma.

  The proof for the other cases is similar to this one, by first applying Lemma 4.8 to get the corresponding cases, and then choosing the induction hypothesis that applies to complete the proof.
- Rule S-ARROW: Assume $A = A_1 \to A_2$ and $B = B_1 \to B_2$, for proof goal (1), we need to prove two subgoals:
  1. $\Gamma_{\mu} \vdash [\alpha \mapsto \mu\alpha. D]B_1 \leq [\alpha \mapsto \mu\alpha. C]A_1$
  2. $\Gamma_{\mu} \vdash [\alpha \mapsto \mu\alpha. C]A_2 \leq [\alpha \mapsto \mu\alpha. D]B_2$

  The covariant subgoal (2) follows from IH(1) directly. In subgoal (1), due to the contravariant subtyping, the substituted types are flipped in the subtyping relation. Therefore we need to apply IH(2) to complete the proof. The proof of subgoal (2) is similar to the proof of subgoal (1), by applying IH(1) to the contravariant subgoal, and IH(2) to the covariant subgoal. For subgoals (3) and (4), the induction hypothesis can be applied directly since the substituted types are the same on both sides.
- Rule S-FULLALL: Assume $A = \forall(\beta \leq B_1). A_1$ and $B = \forall(\beta \leq B_2). A_2$. We show the proof of goal (1) here. In this case we need to prove two subgoals:
  1. $\Gamma_{\mu} \vdash [\alpha \mapsto \mu\alpha. D]B_2 \leq [\alpha \mapsto \mu\alpha. C]B_1$
  2. $\Gamma_{\mu}, \beta \leq [\alpha \mapsto \mu\alpha. D]B_2 \vdash [\alpha \mapsto \mu\alpha. C]A_1 \leq [\alpha \mapsto \mu\alpha. D]A_2$

  The proof of subgoal (1) is similar to the proof of subgoal (1) in the rule S-ARROW case, by applying IH(2) to the contravariant subgoal. Next, in order to apply IH(1) to the covariant subgoal (2), we need to show that the context $(\Gamma, \beta \leq [\alpha \mapsto D^{\alpha}]B_2)$ and $(\Gamma_{\mu}, \beta \leq [\alpha \mapsto \mu\alpha. D]B_2)$ are related, which follows from the definition of related contexts. Therefore, we can apply IH(1) to complete the proof. The proof of other goals is similar to the proof of goal (1). Thanks to our definition of related contexts, we do not need to worry about whether the substituted type is C or D when we add a new binding into the context in the proof of subgoal (2). ∎

To conclude, in this generalized unfolding lemma, we introduce two more conclusions for the case where the substitutions are the same as C or D, and we define related contexts to characterize the contexts that will be used in the proof. In this way, we prove the case of rule S-VARTRANS without the need for an intermediate type S, so that rule S-FULLALL can be handled as well. In fact, the proof technique we use here is quite general. We also redevelop a generalized unfolding lemma for kernel $F_{\leq}^{\mu}$ using the same approach as in full $F_{\leq}^{\mu}$. As we will see, this generalized unfolding lemma can also be used to prove $F^{\mu\wedge}_{\leq\geq}$, without the need for significant changes.

4.2 Conservativity

One important feature of $F^{\mu}_{\leq}$ is that it is conservative over $F_{\leq}$. Conservativity means that equivalent $F_{\leq}$ judgements 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.

To avoid ambiguity, we let $\vdash_F \Gamma$ be the well-formedness of environment, $\Gamma \vdash_F A$ be the well-formedness of types, $\Gamma \vdash_F A \leq B$ be the subtyping relation, $\vdash_F e$ be the well-formedness of expressions, and $\Gamma \vdash_F e : A$ be the typing relation in $F_{\leq}$, where the subscript F stands for the original $F_{\leq}$ calculus. All the definitions and rules for $F_{\leq}$ are essentially subsets of the corresponding definitions and rules for $F^{\mu}_{\leq}$ presented in §3, except that the rules involving records and recursive types are removed, and that in kernel $F_{\leq}$, the rule S-EQUIVALL is replaced with the rule S-KERNELALL. Note that the properties we will show below in this section are demonstrated in both variants of $F^{\mu}_{\leq}$. In other words, full $F^{\mu}_{\leq}$ is conservative over full $F_{\leq}$ and kernel $F^{\mu}_{\leq}$ is conservative over kernel $F_{\leq}$.

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

Lemma 4.10 (Conservativity for subtyping). If Γ, A, and B are well-formed in $F_{\leq}$, namely (1) $\vdash_F \Gamma$, (2) $\Gamma \vdash_F A$, and (3) $\Gamma \vdash_F B$, 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 proof of this lemma is straightforward, except for the case of rule S-EQUIVALL from $F^{\mu}_{\leq}$ to $F_{\leq}$, as the rule S-KERNELALL in $F_{\leq}$ requires the two types to be exactly the same instead of being equivalent. This is easy to fix, given that kernel $F_{\leq}$ subtyping is antisymmetric. This property was shown by Baldan et al. (1999) for a restricted form of F-bounded quantification, and we adapt their proof to our setting. The antisymmetry property is stated as follows:

Lemma 4.11 (Antisymmetry of kernel $F_{\leq}$ subtyping). If $\Gamma \vdash_F A \leq B$ and $\Gamma \vdash_F B \leq A$ in kernel $F_{\leq}$, then $A = B$.

Conservativity of typing. It is straightforward to obtain one direction 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:

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

Although the final judgement $\vdash \lambda x.\ x : \top$ does 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 $F_{\leq}^{\mu}$, 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 either. We state this property as the conservativity lemma for subtyping:

Lemma 4.12. If Γ, A, and e are well-formed in $F_{\leq}$, namely (1) $\vdash_F \Gamma$, (2) $\Gamma \vdash_F A$, and (3) $\vdash_F e$, then $\Gamma \vdash_a e : A$ implies $\Gamma \vdash_F e : A$.

Now, given a typing relation $\Gamma \vdash e : A$ in $F_{\leq}^{\mu}$, we first use the minimum typing property (Theorem 3.10) to obtain its minimum type B such that $\Gamma \vdash_a e : B$ and $\Gamma \vdash B \leq A$. Applying Lemma 4.12 and Lemma 4.10, we complete the conservativity proof for the declarative version of $F_{\leq}^{\mu}$.

Theorem 4.13 (Conservativity). If Γ, A, and e are well-formed in $F_{\leq}$, namely (1) $\vdash_F \Gamma$, (2) $\Gamma \vdash_F A$, and (3) $\vdash_F e$, then $\Gamma \vdash_F e : A$ if and only if $\Gamma \vdash e : A$.

4.3 Decidability

This section focuses on the decidability of kernel $F_{\leq}^{\mu}$. We first start by reviewing the approaches to proving decidability in kernel $F_{\leq}$, and in nominal unfoldings, and then describe our approach to prove decidability. These two previous approaches to proving decidability employ different measures, which creates a challenge for proving the decidability of kernel $F_{\leq}^{\mu}$.

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):

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

For a universal type, 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. (2022) choose a size measure based on an over-approximation of the height of the fully unfolded tree. Concretely, the height of a type A in a measure context Ψ ($\Psi := \cdot \mid \Psi, \alpha \mapsto i$, where i is a natural number) is defined as:

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

The size measure of a type A is defined as $\mathit{height}(A)$ 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. The key design in the *height* function is that the measure of a recursive type $\mu\alpha.\,A$ is computed by first setting the measure of the recursive variable to 0, and then computing the measure of the body, which achieves the effect of measuring the type body A considering α as a free variable. The computed measure is then incremented by one to account for the labeled type A^{α}, then $\mathit{height}(A)$ is computed again with the context updated for the recursive variable, so that intuitively the result of $\mathit{height}(\mu\alpha.\,A)$ measures the size of $[\alpha \mapsto A^{\alpha}]A$ plus one. Zhou et al. (2022) prove that such *height* measure work well with nominal unfolding rules, as the height of a type will precisely decrease by one for every nominal unfolding.

Decidability of kernel $F_{\leq}^{\mu}$. To combine these two approaches, we need to extend the measure of nominal unfoldings with the measure of kernel $F_{\leq}$ in a seamless manner. One easy fix to unify the two measures is to use the maximum function for both measures, as the nominal unfolding measure does. However, there are three remaining main challenges that we must address:

- **Inconsistent measures for variables**: In the *height* function, type variables are treated as base cases, whereas in the *weight* function, the computation continues by retrieving the variable's bound from the context. In kernel $F_{\leq}^{\mu}$, we do not distinguish between recursive and universal variables, so we need to find a unified way to measure variables.
- **Different purposes of contexts**: The context in the *weight* function straightforwardly keeps track of universal bounds, which are later retrieved to compute the measure of a universal variable. This ensures that the premises in rule S-VARTRANS have smaller type measures than the conclusion. However, this trick does not work for nominal unfoldings, as shown by the case $\mathit{height}_{\Psi}(\mu\alpha.\,A)$, where the context is extended with two different measures for the same variable at different points in the computation to simulate the nominal unfolding. This discrepancy complicates the unification of the two measures.
- **Loss of measure information with equivalent bounds**: We use rule S-EQUIVALL instead of the standard rule S-KERNELALL for $F_{\leq}$. Given the equivalent bounds in kernel $F_{\leq}$,

the measure for the subtyping relation $\Gamma \vdash \forall(\alpha \leq A_1).B_1 \leq \forall(\alpha \leq A_2).B_2$ includes the measures of A_1, A_2, B_1, and B_2. However, the measure for the premise $\Gamma, \alpha \leq A_2 \vdash B_1 \leq B_2$ loses the measure of A_1 because it is not stored.

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

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

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 what the variable will be substituted with in the context of the subtyping rule, either its unfolded form as a labeled type or its bound type. 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 arises with 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 the *height* function. 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 track universal variables. In our decidability proof statement (Lemma 4.14), Ψ is computed from the subtyping context Γ by an evaluation function $eval : \Gamma \hookrightarrow \Psi$, defined as:

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

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

Lemma 4.14. If $size_{eval(\Gamma)}(A) + size_{eval(\Gamma)}(B) \le k$ then there exists an algorithm that terminates and decides whether $\Gamma \vdash A \le B$.

Theorem 4.15 (Decidability of kernel $F^{\mu}_{\le}$ subtyping). $\Gamma \vdash A \le B$ is decidable in kernel $F^{\mu}_{\le}$.

Theorem 4.16 (Decidability of kernel $F^{\mu}_{\le}$ typing). $\Gamma \vdash e : A$ is decidable in kernel $F^{\mu}_{\le}$.

As for the third concern, note that in $F_{\le}$, 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 equivalent record types may be syntactically different. The lack of antisymmetry poses a challenge for our decidability proof, in particular for rule S-EQUIVALL. Nevertheless, for kernel $F^{\mu}_{\le}$ two equivalent records must have the same set of fields, and the two types for each field must be equivalent. Therefore, the measures of two equivalent record types remain the same. As a result, the measure of two equivalent bounds A_1 and A_2 is equal, as Lemma 4.17 describes. The measure information of type A_1 can therefore be reconstructed from type A_2, addressing the final concern with decidability.

Lemma 4.17. If $\Gamma \vdash A \le B$ and $\Gamma \vdash B \le A$ then $size_{eval(\Gamma)}(A) = size_{eval(\Gamma)}(B)$.

Undecidability of subtyping full $F^{\mu}_{\le}$. It is well known that the full $F_{\le}$ subtyping relation is undecidable (Pierce, 1994). Although the original formulation of full $F_{\le}$ includes the transitivity rule, it can be reformulated into a syntax-directed version (Curien and Ghelli, 1992) that eliminates the transitivity rule, as we have adopted in $F^{\mu}_{\le}$. The syntax-directed version of $F_{\le}$ naturally forms a subtype checking algorithm. However, for full $F_{\le}$, Ghelli (1993) demonstrated a non-terminating example for the subtyping algorithm. Furthermore, Pierce (1994) proved the undecidability of full $F_{\le}$ by encoding a Turing machine using full $F_{\le}$. Since we have shown in §4.2 that full $F^{\mu}_{\le}$ is conservative over the syntax-directed version of full $F_{\le}$, Ghelli (1993)'s counterexample for full $F_{\le}$ also applies to full $F^{\mu}_{\le}$. Therefore, the undecidability of full $F^{\mu}_{\le}$ is a corollary of the undecidability of full $F_{\le}$ and the conservativity of full $F^{\mu}_{\le}$ over full $F_{\le}$.

Theorem 4.18 (Undecidability of typing and subtyping for full $F^{\mu}_{\le}$). The subtyping relation $\Gamma \vdash A \le B$ and the typing relation $\Gamma \vdash e : A$ are undecidable in full $F^{\mu}_{\le}$.

5 A Calculus with Lower and Upper Bounded Quantification

In this section we introduce an extension of $F^{\mu}_{\le}$, called $F^{\mu\wedge}_{\le\ge}$, with lower bounded quantification, the bottom type, and an alternative formulation of record types in terms of *intersections of single field record types*. While upper bounded quantification has received a lot of attention in previous research, lower bounded quantification for an $F_{\le}$-like language is much less explored, though it appears in a few works (Oliveira et al., 2020; Amin and Rompf, 2017). We follow the same approach as Oliveira et al. (2020), whose $F_{\le}$ extension allows type variables to have either a lower bound or an upper bound, but not both bounds at once. We also introduce single-field record types and intersection types to replace record types in $F_{\le}^{\mu}$. Intersection types enable type level record extension and further applications resembling the treatment of object types in the DOT calculus (Rompf and Amin, 2016). As discussed in §2.2, our extensions in $F_{\le\ge}^{\mu\wedge}$ enable further applications, such as a form of extensible encodings of datatypes. We have proved all the same results for $F_{\le\ge}^{\mu\wedge}$ that were proved for kernel $F_{\le}^{\mu}$, including type soundness, decidability, transitivity and conservativity over $F_{\le}$.

5.1 The $F_{\le\ge}^{\mu\wedge}$ Calculus

The syntax of types, expressions, values and contexts for the extended $F_{\le\ge}^{\mu\wedge}$ calculus is shown below. The main novelties are that bottom types and lower bounded quantification are introduced. We also remove record types ($\{l_i : A_i\ ^{i\in 1\cdots n}\}$) from the syntax, and instead introduce intersection types and single-field record types. The syntactic differences are highlighted in gray.

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

Subtyping, typing and reduction. The well-formedness for the additional bottom types, single-field record types and universal types with lower bounds are standard, as shown in Figure 6. For intersection types, we only allow single-field record types, or intersections of record types with distinct labels to be well-formed. This can be characterized by a compatibility relation $A \# B$ between types. We make this simplified design choice to avoid the complexity of general unrestricted intersection types, which would cause trouble in the two key properties of the type system, namely the structural unfolding lemma (Lemma 5.11) and the decidability of subtyping (Theorem 5.14), as we will discuss in §5.3.

As for the subtyping rules, compared with $F_{\le}^{\mu}$, we add rules S-BOT, S-VARTRANSLB, and S-EQUIVALLLB for subtyping with bottom types and lower bounded quantification. The record subtyping rule S-RCD in $F_{\le}^{\mu}$ is now replaced by rule S-SRCD for subtyping single-field record types together with rules S-ANDLA, S-ANDLB, and S-ANDR for subtyping intersection types. Note that in the subtyping rule for intersection types, we also add the compatibility restriction in the premise, to ensure the regularity of the subtyping relation (Lemma 5.1).

Lemma 5.1 (Regularity of subtyping in $F_{\le\ge}^{\mu\wedge}$). If $\Gamma \vdash A \le B$ then the following well-formedness conditions hold: (1) $\vdash \Gamma$, (2) $\Gamma \vdash A$ and (3) $\Gamma \vdash B$.

$\boxed{A \# B}$ *(Compatible record types)*

$$\textsc{Comp-rcd}\ \frac{l_1 \neq l_2}{\{l_1 : A\} \# \{l_2 : B\}} \qquad \textsc{Comp-andL}\ \frac{A_1 \# B \qquad A_2 \# B}{A_1 \,\&\, A_2 \# B} \qquad \textsc{Comp-andR}\ \frac{A \# B_1 \qquad A \# B_2}{A \# B_1 \,\&\, B_2}$$

$\boxed{\Gamma \vdash A}$ *(Well-formedness of types)*

$$\textsc{wft-bot}\ \frac{}{\Gamma \vdash \bot} \qquad \textsc{wft-allLB}\ \frac{\Gamma \vdash A \qquad \Gamma,\ \alpha \geq A \vdash B}{\Gamma \vdash \forall(\alpha \geq A).B} \qquad \textsc{wft-and}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash B \qquad A \# B}{\Gamma \vdash A \,\&\, B} \qquad \textsc{wft-srcd}\ \frac{\Gamma \vdash A}{\Gamma \vdash \{l : A\}}$$

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

$$\textsc{S-bot}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash \bot \leq A} \qquad \textsc{S-srcd}\ \frac{\Gamma \vdash A \leq B}{\Gamma \vdash \{l : A\} \leq \{l : B\}} \qquad \textsc{S-vartransLB}\ \frac{\alpha \geq B \in \Gamma \qquad \Gamma \vdash A \leq B}{\Gamma \vdash A \leq \alpha}$$

$$\textsc{S-equivallLB}\ \frac{\Gamma \vdash A_1 \leq A_2 \qquad \Gamma \vdash A_2 \leq A_1 \qquad \Gamma,\ \alpha \geq A_2 \vdash B \leq C}{\Gamma \vdash \forall(\alpha \geq A_1).B \leq \forall(\alpha \geq A_2).C} \qquad \textsc{S-andLA}\ \frac{\Gamma \vdash B \qquad \Gamma \vdash A \leq C \qquad A \# B}{\Gamma \vdash A \,\&\, B \leq C}$$

$$\textsc{S-andLB}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash B \leq C \qquad A \# B}{\Gamma \vdash A \,\&\, B \leq C} \qquad \textsc{S-andR}\ \frac{\Gamma \vdash A \leq B \qquad \Gamma \vdash A \leq C \qquad B \# C}{\Gamma \vdash A \leq B \,\&\, C}$$

Fig. 6: Additional well-formedness and subtyping rules for $F^{\mu\wedge}_{\leq\geq}$ with respect to $F^{\mu}_{\leq}$.

The new subtyping relation is reflexive and transitive:

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

Theorem 5.3 (Transitivity for $F^{\mu\wedge}_{\leq\geq}$). If $\Gamma \vdash A \leq B$ and $\Gamma \vdash B \leq C$ then $\Gamma \vdash A \leq C$.

Figure 7 shows the changes of $F^{\mu\wedge}_{\leq\geq}$ with respect to $F^{\mu}_{\leq}$ in terms of typing and reduction. For lower bounded quantification, we add rules TYPING-TAPPLB and TYPING-TABSLB for typing and rule STEP-TABSLB for reduction, which are simply dual forms of rules TYPING-TAPP, TYPING-TABS, and STEP-TABS, respectively. For records, since the syntax of record expressions is unchanged, there are no further changes in the reduction rules. The typing rule for record projections is also simplified. Since record types are now represented by single-field record types, the projection of a record can be directly modeled by the subtyping relation. Rules TYPING-RCDNIL and TYPING-SRCD form the typing rules for record expressions.

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

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

$$\frac{\Gamma \vdash e : \forall(\alpha \geq B_1).B_2 \qquad \Gamma \vdash B_1 \leq A}{\Gamma \vdash e\, A : [\alpha \mapsto A]\ B_2}\ \text{TYPING-TAPPLB} \qquad \frac{\Gamma,\ \alpha \geq A \vdash e : B}{\Gamma \vdash \Lambda(\alpha \geq A).\ e : \forall(\alpha \geq A).B}\ \text{TYPING-TABSLB}$$

$$\frac{\Gamma \vdash e : \{l : A\}}{\Gamma \vdash e.l : A}\ \text{TYPING-SPROJ} \qquad \frac{}{\Gamma \vdash \{\,\} : \top}\ \text{TYPING-RCDNIL} \qquad \frac{l_i{}^{i\in 1\cdots n} \text{ are disjoint} \qquad \Gamma \vdash e_i : A_i \quad \forall i, 1 \leq i \leq n}{\Gamma \vdash \{l_i = e_i{}^{i\in 1\cdots n}\} : \{l_1 : A_1\}\ \&\ \ldots\ \&\ \{l_n : A_n\}}\ \text{TYPING-SRCD}$$

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

$$\frac{}{(\Lambda(\alpha \geq A).\ e)\ B \hookrightarrow [\alpha \mapsto B]\ e}\ \text{STEP-TABSLB}$$

Fig. 7: Additional typing and reduction rules for $F^{\mu\wedge}_{\leq\geq}$ with respect to $F^{\mu}_{\leq}$.

another recursive type or the $\top$ type. Thus 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 $F^{\mu\wedge}_{\leq\geq}$, 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 $F^{\mu\wedge}_{\leq\geq}$ 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 broadly applicable to various type system extensions to $F^{\mu}_{\leq}$.

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

Theorem 5.4 (Preservation for $F^{\mu\wedge}_{\leq\geq}$). If $\vdash e : A$ and $e \hookrightarrow e'$ then $\vdash e' : A$.

Theorem 5.5 (Progress for $F^{\mu\wedge}_{\leq\geq}$). If $\vdash e : A$ then e is a value or exists e', $e \hookrightarrow e'$.

5.2 Algorithmic typing

Similarly to $F^{\mu}_{\leq}$, we can define an algorithmic typing system for $F^{\mu\wedge}_{\leq\geq}$. We present the changes in the algorithmic typing rules for $F^{\mu\wedge}_{\leq\geq}$ in Figure 8. Rules ATYP-TABSLB and ATYP-TAPPLB are added to handle lower bounded quantification. Rules ATYP-RCDNIL and ATYP-SRCD replace the record typing rule TYPING-RCD in $F^{\mu}_{\leq}$. In addition to these standard changes, there are also a few special cases that need to be handled for $F^{\mu\wedge}_{\leq\geq}$.

Firstly, bottom types bring several extra cases to the algorithmic typing rules. 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). To ensure that the algorithmic typing rules are complete, we need to add rules ATYP-APPBOT and ATYP-TAPPBOT to handle these cases. We also develop a similar treatment for recursive types, as shown in rules ATYP-SUNFOLDBOT and ATYP-SFOLDTOP.

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

$$\textsc{atyp-tabslb}\quad \frac{\Gamma,\ \alpha \geq A \vdash_a e : B}{\Gamma \vdash_a \Lambda(\alpha \geq A).\ e : \forall(\alpha \geq A).B}$$

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

$$\textsc{atyp-sproj}\quad \frac{\Gamma \vdash_a e : A \qquad \Gamma \vdash A \Rightarrow_l B}{\Gamma \vdash_a e.l : B}$$

$$\textsc{atyp-rcdnil}\quad \frac{}{\Gamma \vdash_a \{\,\} : \top}$$

$$\textsc{atyp-srcd}\quad \frac{{l_i}^{\,i \in 1 \cdots n} \text{ are disjoint} \qquad \Gamma \vdash_a e_i : A_i \ \ \forall i, 1 \leq i \leq n}{\Gamma \vdash_a \{l_i = {e_i}^{\,i \in 1 \cdots n}\} : \{l_1 : A_1\}\ \&\ \ldots\ \&\ \{l_n : A_n\}}$$

$$\textsc{atyp-appbot}\quad \frac{\Gamma \vdash_a e_1 : A \qquad \Gamma \vdash A \Uparrow \bot \qquad \Gamma \vdash_a e_2 : A_2}{\Gamma \vdash_a e_1\ e_2 : \bot}$$

$$\textsc{atyp-tappbot}\quad \frac{\Gamma \vdash_a e : B \qquad \Gamma \vdash B \Uparrow \bot \qquad \Gamma \vdash A}{\Gamma \vdash_a e\, A : \bot}$$

$$\textsc{atyp-sunfoldbot}\quad \frac{\Gamma \vdash_a e : A \qquad \Gamma \vdash B \Uparrow \bot \qquad \Gamma \vdash A \leq B}{\Gamma \vdash_a \mathsf{unfold}\ [B]\ e : \bot}$$

$$\textsc{atyp-sfoldtop}\quad \frac{\Gamma \vdash_a e : A \qquad \Gamma \vdash C \Downarrow \top \qquad \Gamma \vdash C}{\Gamma \vdash_a \mathsf{fold}\ [C]\ e : \top}$$

Fig. 8: The additional algorithmic typing rules for $F^{\mu\wedge}_{\leq\geq}$.

Moreover, with two kinds of bounded quantification, 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 upper bound in the context that is *not an upper-bounded variable*, so it will return the variable itself if the variable is lower bounded. We redefine the exposure functions for $F^{\mu\wedge}_{\leq\geq}$ in Figure 9. For lower exposure, we also need a dual form of the rule XA-PROMOTE, which finds the greatest lower bound in the context that is *not a lower-bounded variable*, as shown in rule XA-DOWNPROMOTE.

Record exposure. Furthermore, for typing record projections, recall that in the declarative rule TYPING-SPROJ, the lookup of the field label l is implied by the implicit subtyping between the expression type and the single field record type for $\{l : A\}$. In the algorithmic system, we need to find a mechanism to find such A. Therefore we define a new exposure relation for record types in $F^{\mu\wedge}_{\leq\geq}$. The record exposure relation $\Gamma \vdash A \Rightarrow_l B$ indicates that from the type A we can lookup the field label l and get the type B. We show the full definition of the record exposure relation in Figure 9. Note that, in addition to single-field record types (rule XR-SRCD) and intersection types (rules XR-ANDA, XR-ANDB, and XR-ANDR), one can also lookup $\bot$ from $\bot$ (rule XR-BOT), as well as upper bounds from upper bounded variables (rule XR-PROMOTE). With the record exposure relation we define rule ATYP-SPROJ for record projections to replace rule ATYP-PROJ in $F^{\mu}_{\leq}$. The record exposure relation is sound and complete for the subtyping relation $A \leq \{l : B\}$.

Lemma 5.6 (Record exposure properties for $F^{\mu\wedge}_{\leq\geq}$).

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

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

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

$$\textsc{XA-downpromote}\quad \frac{\alpha \geq A \in \Gamma \qquad \Gamma \vdash A \Downarrow B}{\Gamma \vdash \alpha \Downarrow B} \qquad \textsc{XA-downinv}\quad \frac{\alpha \leq A \in \Gamma}{\Gamma \vdash \alpha \Downarrow \alpha} \qquad \textsc{XA-downward}\quad \frac{A \text{ is not a type variable}}{\Gamma \vdash A \Downarrow A}$$

$\boxed{\Gamma \vdash A \Rightarrow_l B}$ *(Record Exposure)*

$$\textsc{XR-promote}\quad \frac{\alpha \leq A \in \Gamma \qquad \Gamma \vdash A \Rightarrow_l B}{\Gamma \vdash \alpha \Rightarrow_l B} \qquad \textsc{XR-andA}\quad \frac{\Gamma \vdash A_1 \Rightarrow_l B}{\Gamma \vdash A_1 \,\&\, A_2 \Rightarrow_l B} \qquad \textsc{XR-andB}\quad \frac{\Gamma \vdash A_2 \Rightarrow_l B}{\Gamma \vdash A_1 \,\&\, A_2 \Rightarrow_l B}$$

$$\textsc{XR-srcd}\quad \frac{}{\Gamma \vdash \{l : A\} \Rightarrow_l A} \qquad \textsc{XR-bot}\quad \frac{}{\Gamma \vdash \bot \Rightarrow_l \bot}$$

Fig. 9: The new exposure functions for $F^{\mu\wedge}_{\leq\geq}$.

1. If $\Gamma \vdash A \Rightarrow_l B$ then $\Gamma \vdash A \leq \{l : B\}$.
2. If $\Gamma \vdash A \leq \{l : B\}$ then there exists C such that $\Gamma \vdash A \Rightarrow_l C$ and $\Gamma \vdash C \leq \{l : B\}$.

With these considerations in the algorithmic typing rules, we prove the soundness and completeness of the algorithmic typing system with respect to the declarative typing rules defined in Figure 7.

Theorem 5.7 (Soundness of the algorithmic rules for $F^{\mu\wedge}_{\leq\geq}$). If $\Gamma \vdash_a e : A$ then $\Gamma \vdash e : A$.

Theorem 5.8 (Completeness of the algorithmic rules for $F^{\mu\wedge}_{\leq\geq}$). If $\Gamma \vdash e : A$ then there exists B such that $\Gamma \vdash_a e : B$ and $\Gamma \vdash B \leq A$.

5.3 *Metatheory of* $F^{\mu\wedge}_{\leq\geq}$

The addition of lower bounded quantification, bottom types, and intersection types creates some difficulties in the metatheory of $F^{\mu\wedge}_{\leq\geq}$. In the following, we describe how to overcome the difficulties, by adjusting the proof techniques we have used for $F^{\mu}_{\leq}$.

Unfolding Lemma. As discussed in §4.1, in a type system that simultaneously allows introducing lower and upper bounded types, the inversion lemma for rule S-vartrans (Lemma 4.4) is not valid. This is exactly the case for $F^{\mu\wedge}_{\leq\geq}$. To resolve this issue, the unfolding lemmas should only state the subtyping relation between the nominal unfoldings $[\alpha \mapsto C^{\alpha}]A \leq [\alpha \mapsto D^{\alpha}]B$ and remove the one-step unfolding relation $A \leq B$ from the premise. Therefore, we use the same statement of the generalized unfolding lemma as in full $F^{\mu}_{\leq}$ (Lemma 4.9), under an extended version of related contexts (Definition 4.7) that takes lower bounded bindings into account. It turns out that, with only changes of the proof in cases S-EQUIVALL and S-EQUIVALLLB, the generalized unfolding lemma can be proved for $F^{\mu\wedge}_{\leq\geq}$ as well, which results in the following unfolding lemma for $F^{\mu\wedge}_{\leq\geq}$.

Lemma 5.9 (Unfolding lemma for $F^{\mu\wedge}_{\leq\geq}$). If $\Gamma \vdash \mu\alpha.\,A \leq \mu\alpha.\,B$, then $\Gamma \vdash [\alpha \mapsto \mu\alpha.\,A]\ A \leq [\alpha \mapsto \mu\alpha.\,B]\ B$.

To prove type soundness, we need to show the structural unfolding lemma. If we check the typing derivation with structural folding and unfolding illustrated in Figure 4 again in $F^{\mu\wedge}_{\leq\geq}$, we can see that by inversion on the subtyping relation $\cdot \vdash \mu\alpha.\,A \leq C'$ and $\cdot \vdash D' \leq \mu\alpha.\,B$, we can no longer guarantee that C' and D' are recursive types, since they can also be intersection types. To remedy this, the compatibility relation $A \,\#\, B$ is enforced by the well-formedness of intersection types, so that intersection types can only be formed from single-field record types, not by recursive types. We can prove the following lemma to derive a contradiction for the case of intersection types and recover the structural unfolding lemma as well as type soundness for $F^{\mu\wedge}_{\leq\geq}$.

Lemma 5.10. For any types A, B_1 and B_2, it cannot happen that $\mu\alpha.\,A \leq B_1 \,\&\, B_2$ or $B_1 \,\&\, B_2 \leq \mu\alpha.\,A$ in $F^{\mu\wedge}_{\leq\geq}$.

Lemma 5.11 (Structural unfolding lemma for $F^{\mu\wedge}_{\leq\geq}$). If $\Gamma \vdash \mu\alpha.\,A \leq \mu\alpha.\,C \leq \mu\alpha.\,D \leq \mu\alpha.\,B$ then $\Gamma \vdash [\alpha \mapsto \mu\alpha.\,C]\ A \leq [\alpha \mapsto \mu\alpha.\,D]\ B$.

Decidability. The interaction between bottom types and rule S-EQUIVALL breaks the measure-based decidability proof in §4.3. The bottom type in $F^{\mu\wedge}_{\leq\geq}$ 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. For example, consider the following subtyping derivation:

$$\frac{\alpha \leq \bot,\ \beta \leq \alpha \vdash \alpha \leq \beta \qquad \alpha \leq \bot,\ \beta \leq \alpha \vdash \beta \leq \alpha \qquad \alpha \leq \bot,\ \beta \leq \alpha,\ \gamma \leq \beta \vdash A \leq B}{\alpha \leq \bot,\ \beta \leq \alpha \vdash \forall(\gamma \leq \alpha).\,A \leq \forall(\gamma \leq \beta).\,B}$$

If we follow the measure function defined in §4.3, the measure for the third premise is:

$$size_{\alpha\mapsto 1,\ \beta\mapsto 2,\ \gamma\mapsto 3}(A) + size_{\alpha\mapsto 1,\ \beta\mapsto 2,\ \gamma\mapsto 3}(B)$$

while the measure for the goal is

$$\begin{aligned} & size_{\alpha\mapsto 1,\ \beta\mapsto 2}(\forall(\gamma \leq \alpha).\,A) + size_{\alpha\mapsto 1,\ \beta\mapsto 2}(\forall(\gamma \leq \beta).\,B) \\ \hookrightarrow\quad & size_{\alpha\mapsto 1,\ \beta\mapsto 2,\ \gamma\mapsto 2}(A) + size_{\alpha\mapsto 1,\ \beta\mapsto 2,\ \gamma\mapsto 3}(B) \end{aligned}$$

which can be less than the measure of the premise since γ is assigned a smaller measure in the goal. A similar issue arises when a variable is lower bounded by $\top$, making it equivalent to top types but with a different measure.

This issue can be resolved by replacing all the types whose supertype is $\bot$ with $\bot$, and all the types whose subtype is $\top$ with $\top$ before computing the measure. That way, the subtyping relation $\alpha \le \bot,\ \beta \le \alpha \vdash \forall(\gamma \le \alpha).\, A \le \forall(\gamma \le \beta).\, B$ becomes

$$\alpha \le \bot,\ \beta \le \bot \vdash \forall(\gamma \le \bot).\, A \le \forall(\gamma \le \bot).\, B$$

and the measure works again. This idea can be implemented by modifying the measure function to identify upper/lower bounded variables that are equivalent to bottom/top types, as can be seen in Figure 10. The new bindings $\alpha \mapsto \bot$ and $\alpha \mapsto \top$ are used to store the measure of variables or indicate them as upper bounded by $\bot$ or lower bounded by $\top$. Figure 10 shows the measures needed for the decidability of $F^{\mu\wedge}_{\le\ge}$. The primary measure function is $size_\Psi(A)$. The main changes are in the cases for bounded quantification where we now use *isTop* and *isBot* functions to detect whether the bounds are, respectively, equivalent to top or bottom. The *isTop* and *isBot* functions use the information in the measure context Ψ to check whether the bound type A is equivalent to $\top$ or $\bot$. If so, when the variable bounded by A is looked up in the context, it will have a measure of 1. The example above will now be resolved by the new measure function as follows:

$$\begin{array}{ll} & size_{\alpha\mapsto\bot,\ \beta\mapsto\bot}(\forall(\gamma \le \alpha).\, A) + size_{\alpha\mapsto\bot,\ \beta\mapsto\bot}(\forall(\gamma \le \beta).\, B) \\ \hookrightarrow & size_{\alpha\mapsto\bot,\ \beta\mapsto\bot,\ \gamma\mapsto\bot}(A) + size_{\alpha\mapsto\bot,\ \beta\mapsto\bot,\ \gamma\mapsto\bot}(B) \\ \multicolumn{2}{l}{\text{since } isBot_{\alpha\mapsto\bot,\ \beta\mapsto\bot}(\alpha) = \mathsf{true} \text{ and } isBot_{\alpha\mapsto\bot,\ \beta\mapsto\bot}(\beta) = \mathsf{true}} \end{array}$$

In this way, we retain the important property that equivalent types have the same measure.

Lemma 5.12. If $\Gamma \vdash A \le B$ and $\Gamma \vdash B \le A$ then $size_{eval(\Gamma)}(A) = size_{eval(\Gamma)}(B)$ in $F^{\mu\wedge}_{\le\ge}$.

Note that in the proof of Lemma 5.12, in the case of intersection types, we make use of the compatibility relation $A \# B$ to ensure that any equivalent types have the same measure. Without the compatibility restriction, the labels may be duplicated within the intersection type, which will lead to a different measure for equivalent types. By limiting compatible types to be single-field records and their intersections only, we also rule out the occurrence of $\top$ in intersection types, which will cause the same problem. With the new measure function, we can prove the decidability of subtyping and typing in $F^{\mu\wedge}_{\le\ge}$.

Theorem 5.13 (Decidability of $F^{\mu\wedge}_{\le\ge}$ subtyping). $\Gamma \vdash A \le B$ is decidable in $F^{\mu\wedge}_{\le\ge}$.

Theorem 5.14 (Decidability of $F^{\mu\wedge}_{\le\ge}$ typing). $\Gamma \vdash e : A$ is decidable in $F^{\mu\wedge}_{\le\ge}$.

Conservativity. The proof of conservativity for $F^{\mu\wedge}_{\le\ge}$ follows the same pattern as the proof for $F^{\mu}_{\le}$. To prove conservativity of typing, we need the help of the algorithmic typing rules to obtain the minimum type of an $F_\le$ term. We have defined the algorithmic typing rules for $F^{\mu\wedge}_{\le\ge}$ and proved the completeness of the algorithmic typing rules in §5.2. With the algorithmic typing rules, conservativity for $F^{\mu\wedge}_{\le\ge}$ is straightforward.

Theorem 5.15 (Conservativity for $F^{\mu\wedge}_{\le\ge}$). If Γ, A and e are well-formed in $F_\le$, namely (1) $\vdash_F \Gamma$ (2) $\Gamma \vdash_F A$ and (3) $\vdash_F e$, then $\Gamma \vdash_F e : A$ if and only if $\Gamma \vdash e : A$.

$$\Psi := \cdot \mid \Psi, \alpha \mapsto i \mid \Psi, \alpha \mapsto \bot \mid \Psi, \alpha \mapsto \top$$

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

$$\begin{array}{lcl}
isBot_\Psi(\bot) & = & \mathsf{true} \\
isBot_{\Psi,\alpha\mapsto\bot}(\alpha) & = & \mathsf{true} \\
isBot_{\Psi,\beta\mapsto\_}(\alpha) & = & isBot_\Psi(\alpha) \text{ if } \alpha \neq \beta \\
\text{otherwise } isBot_\Psi(A) & = & \mathsf{false}
\end{array}$$

$$\begin{array}{lcl}
isTop_\Psi(\top) & = & \mathsf{true} \\
isTop_{\Psi,\alpha\mapsto\top}(\alpha) & = & \mathsf{true} \\
isTop_{\Psi,\beta\mapsto\_}(\alpha) & = & isTop_\Psi(\alpha) \text{ if } \alpha \neq \beta \\
\text{otherwise } isTop_\Psi(A) & = & \mathsf{false}
\end{array}$$

Fig. 10: The measures for the decidability of $F^{\mu\wedge}_{\leq\geq}$.

6 Coq Proofs

We develop and verify our formalization in Coq 8.13 (The Coq Development Team, 2021), and use Metalib to formalize variables and binders using the locally nameless representation (Aydemir et al., 2008).

The Coq formalization is available online[4]. The directory "kernel_fsub_main" includes all definitions and proofs for kernel $F^{\mu}_{\leq}$ described in Section 3, while the directory "full_fsub_main" includes the full $F^{\mu}_{\leq}$ variant. Definition and proofs for $F^{\mu\wedge}_{\leq\geq}$ described in Section 5 are in the "kernel_fsub_ext" directory. Each directory can be checked independently, and the dependency of the proofs follows a sequential order in each directory.

[4] `https://github.com/juda/Recursive-Subtyping-for-All/tree/main/JFP`

Table 2: Paper-to-proofs correspondence guide for kernel $F^{\mu}_{\leq}$ (in kernel_fsub_main/ directory).

Definition	File	Name in Coq	Notation
Types (Figure 1)	Rules.v	typ	
Expressions (Figure 1)	Rules.v	exp	
Values (Figure 1)	Rules.v	value	
Contexts (Figure 1)	Rules.v	env	
Well-formed Type (Figure 2)	Rules.v	WF E A	$\Gamma \vdash A$
Subtyping (Figure 2)	Rules.v	sub E A B	$\Gamma \vdash A \leq B$
Typing (Figure 3)	Rules.v	typing E e A	$\Gamma \vdash e : A$
Reduction (Figure 3)	Rules.v	step e1 e2	$e_1 \hookrightarrow e_2$
Upper Exposure (Figure 5)	AlgoTyping.v	exposure E A B	$\Gamma \vdash A \Uparrow B$
Lower Exposure (Figure 5)	AlgoTyping.v	exposure2 E B A	$\Gamma \vdash A \Downarrow B$
Algorithmic Typing (Figure 5)	AlgoTyping.v	typing E e A	$\Gamma \vdash_a e : A$
Measure (§4.3)	Decidability.v	bindings_rec G E n A	$size_{\Psi}(A)$
Context Measure (§4.3)	Decidability.v	mk_benv E	$eval(\Gamma)$

Table 3: Paper-to-proofs correspondence guide for full $F^{\mu}_{\leq}$ (in full_fsub_main/ directory). Definitions that are the same as kernel $F^{\mu}_{\leq}$ are omitted.

Definition	File	Name in Coq	Notation
Subtyping (§3.2)	Rules.v	sub E A B	$\Gamma \vdash A \leq B$
Related contexts (Definition 4.7)	UnfoldingEquiv.v	sub_env_ext E X C D E1 E2	$\Gamma \cong \Gamma_{\mu}$

6.1 Definitions

All three systems share a similar structure for definitions: the files *Rules.v* contains the core definitions for the calculus, and *AlgoTyping.v* contains the algorithmic rules for typing. Tables 2, 3 and 4 shows the correspondence between the paper definitions and the Coq formalization. The formalization mainly follows the definitions in the paper except for some technical details. One difference to note is that throughout the paper, we use only substitution to represent unfolding of a recursive type, application of universal quantification and function abstraction. In the Coq proof, due to the use of the locally nameless representation, we also make use of the opening operation on pre-terms (Aydemir et al., 2008). We also merge several rules for exposure and typing record expressions in the paper, for readability.

6.2 Lemmas and Theorems

Tables 5, 6 and 7 show the correspondence of lemmas and theorems between the paper and the Coq formalization. We provide the file location and theorem name in Coq for each lemma and theorem in the paper, and include a brief description for each of them.

Table 4: Paper-to-proofs correspondence guide for $F^{\mu\wedge}_{\leq\geq}$ (in kernel_fsub_ext/ directory).

Definition	File	Name in Coq	Notation
Types (§5.1)	Rules.v	typ	
Expressions (§5.1)	Rules.v	exp	
Values (§5.1)	Rules.v	value	
Contexts (§5.1)	Rules.v	env	
Compatible types (Figure 6)	Rules.v	Compatible A B	$A \# B$
Well-formed Type (Figure 6)	Rules.v	WF E A	$\Gamma \vdash A$
Subtyping (Figure 6)	Rules.v	sub E A B	$\Gamma \vdash A \leq B$
Typing (Figure 7)	Rules.v	typing E e A	$\Gamma \vdash e : A$
Reduction (Figure 7)	Rules.v	step e1 e2	$e_1 \hookrightarrow e_2$
Algorithmic Typing (Figure 8)	AlgoTyping.v	typing E e A	$\Gamma \vdash_a e : A$
Upper Exposure (Figure 9)	AlgoTyping.v	exposure E A B	$\Gamma \vdash A \Uparrow B$
Lower Exposure (Figure 9)	AlgoTyping.v	exposure2 E B A	$\Gamma \vdash A \Downarrow B$
Record Exposure (Figure 9)	AlgoTyping.v	exposure_i E A l B	$\Gamma \vdash A \Rightarrow_l B$
Measure (Figure 10)	Decidability.v	bindings_rec G E n A	$size_\Psi(A)$

7 Related Work

Throughout the paper, we have already reviewed some of the closest related work in detail. In this section, we discuss other related work.

7.1 Bounded Quantification, Recursive Types and Object Encodings

Bounded quantification was first 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 is contravariant. Although full $F_\leq$ is powerful, subtyping 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 significant changes for the subtyping rules. Ghelli (1993) illustrates how the combination of equi-recursive subtyping and full $F_\leq$ significantly alters the expressiveness of the subtyping relation. Specifically, he shows that there exist such subtyping relations $A \not\leq A'$ that do not hold in full $F_\leq$, but are derivable when equi-recursive subtyping is added, by finding an intermediate type B which contains equi-recursive types such that $A \leq B$ and $B \leq A'$. In Colazzo and Ghelli (2005)'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:

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

Table 5: Descriptions for the proof scripts for kernel $F^{\mu}_{\leq}$ (in kernel_fsub_main/ directory).

Theorems	Description	Files	Name in Coq
Lemma 3.1	Regularity of subtyping	Reflexivity.v	sub_regular
Lemma 3.2	Narrowing	Transitivity.v	sub_narrowing
Theorem 3.3	Reflexivity	Reflexivity.v	Reflexivity
Theorem 3.4	Transitivity	Transitivity.v	sub_transitivity
Lemma 3.5	Unfolding lemma	Unfolding.v	unfolding_lemma
Lemma 3.6	Structural unfolding	Preservation.v	structural_unfolding _lemma_general
Theorem 3.7	Preservation	Preservation.v	preservation
Theorem 3.8	Progress	Progress.v	progress
Theorem 3.9	Algo-typing soundness	AlgoTyping.v	typing_algo_sound
Theorem 3.10	Algo-typing completeness	AlgoTyping.v	minimum_typing
Lemma 4.2	Generalized unfolding lemma for kernel $F^{\mu}_{\leq}$ in Zhou et al. (2023)	Unfolding.v	sub_generalize_ intensive
Lemma 4.3	Substitution inversion	Unfolding.v	subst_reverse_equiv
Lemma 4.10	Subtyping conservativity	Conservativity.v	sub_conserv
Lemma 4.11	Antisymmetry of kernel $F_{\leq}$ subtyping	Conservativity.v	sub_antisym
Lemma 4.12	Algo-typing conservativity	Conservativity.v	typing_algo_conserv
Theorem 4.13	Typing conservativity	Conservativity.v	typing_conserv
Theorem 4.15	Decidability of subtyping	Decidability.v	decidability
Theorem 4.16	Decidability of typing	DecidabilityTy.v	decidable_typing
Lemma 4.17	Equivalent measure	Decidability.v	equiv_measure

The algorithm proposed by Jeffrey (2001) is also complex, and requires major changes. Both recursive variables and the subtyping algorithm are labeled with polarity modes, 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 (2001)'s or Colazzo and Ghelli (2005)'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, denoted as $\forall(\alpha \leq F[\alpha]).B$. We can encode polymorphic binary methods (Bruce et al., 1995) and methods that have recursive types in their signatures with F-bounded quantification. However, as we discussed in §2.2, for

Table 6: Descriptions for the proof scripts for full $F_{\leq}^{\mu}$ (in full_fsub_main/ directory).

Theorems	Description	Files	Name in Coq
Lemma 3.1	Regularity of subtyping	Reflexivity.v	sub_regular
Lemma 3.2	Narrowing	Transitivity.v	sub_narrowing
Theorem 3.3	Reflexivity	Reflexivity.v	Reflexivity
Theorem 3.4	Transitivity	Transitivity.v	sub_transitivity
Lemma 3.5	Unfolding lemma	UnfoldingEquiv.v	unfolding_lemma
Lemma 3.6	Structural unfolding	Preservation.v	structural_unfolding _lemma_general
Theorem 3.7	Preservation	Preservation.v	preservation
Theorem 3.8	Progress	Progress.v	progress
Theorem 3.9	Algo-typing soundness	AlgoTyping.v	typing_algo_sound
Theorem 3.10	Minimum typing	AlgoTyping.v	minimum_typing
Lemma 4.8	Related context inversion	UnfoldingEquiv.v	sub_env_ext_sem
Lemma 4.9	Generalized unfolding lemma for full $F_{\leq}^{\mu}$	UnfoldingEquiv.v	sub_generalize_ intensive
Lemma 4.10	Subtyping conservativity	Conservativity.v	sub_conserv
Lemma 4.12	Algorithmic typing conservativity	Conservativity.v	typing_algo_conserv
Theorem 4.13	Typing conservativity	Conservativity.v	typing_conserv

subtyping statements to satisfy the bound $\alpha \leq F[\alpha]$, they must be interpreted using equi-recursive subtyping, as F-bounds are normally records, and an iso-recursive type cannot be the subtype of a record type. F-bounded quantification 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 $F_{\leq}^{\mu}$, we can only model a restricted form of negative F-bounded quantification. For instance in $F_{\leq}^{\mu}$ 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 $\mu\mathsf{P'}.\{x : \mathsf{Int},\ y : \mathsf{Int}, \mathsf{eq} : \mathsf{P'} \to \mathsf{Bool}\}$. In contrast, F-bounded quantification allows such forms of instantiation. Nevertheless, given the overlap between some of the applications of iso-recursive types in $F_{\leq}^{\mu}$ 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.

F-bounded quantification offers an elegant method for encoding objects that possess binary methods. When not seeking to fully encode binary methods, other object encodings are also available. Recursive records can encode objects (Bruce et al., 1999; Cook et al., 1989; Canning 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 (1994)'s object encoding using existential types is notable in that it requires only $F_{\leq}$, and does not employ recursive types. The *ORBE*

Table 7: Descriptions for the proof scripts for $F^{\mu\wedge}_{\leq\geq}$ (in kernel_fsub_ext/ directory).

Theorems	Description	Files	Name in Coq
Lemma 5.1	Regularity of subtyping	Reflexivity.v	sub_regular
Theorem 5.2	Reflexivity	Reflexivity.v	Reflexivity
Theorem 5.3	Transitivity	Transitivity.v	sub_transitivity
Theorem 5.4	Preservation	Preservation.v	preservation
Theorem 5.5	Progress	Progress.v	progress
Lemma 5.6	Record exposure	AlgoTyping.v	exposure_i_sound exposure_i_ex
Theorem 5.7	Algo-typing soundness	AlgoTyping.v	typing_algo_sound
Theorem 5.8	Algo-typing completeness	AlgoTyping.v	minimum_typing
Lemma 5.9	Unfolding lemma	UnfoldingEquiv.v	unfolding_lemma
Lemma 5.11	Structural Unfolding lemma	Preservation.v	structural_unfolding _lemma_general
Lemma 5.12	Equivalent measure	Decidability.v	equiv_measure
Theorem 5.13	Decidability of subtyping	Decidability.v	decidability
Theorem 5.14	Decidability of typing	DecidabilityTy.v	decidable_typing
Theorem 5.15	Conservativity	Conservativity.v	typing_conserv

encoding (Abadi et al., 1996), as we discussed in §2.2, consists of recursive types, bounded existential quantification, records, and the structural unfolding rule. As Bruce et al. (1999) 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. Our $F^{\mu}_{\leq}$ calculus could act as a target for all those existing object encodings discussed above. To our best knowledge, a complete formalization of $F_{\leq}$ with recursive types, featuring desirable properties such as type soundness and conservativity, was not available at the time. Our work contributes to the validation of these encodings by offering a complete formalization of $F_{\leq}$ with recursive types, along with several desirable properties.

7.2 Dependent Object Types

The renewed interest in languages featuring bounded quantification and recursive types has been reignited recently within the research community, following the introduction of the dependent object types (DOT) calculus (Rompf and Amin, 2016). DOT is now the foundation of Scala 3 (EPFL, 2021). 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 $F_{\leq}$. DOT implements a generalized form of bounded quantification along with recursive types. This generalized form encompasses both upper and lower bounded quantification. Furthermore, DOT facilitates path selection that simultaneously supports upper and lower bounds. Additionally, DOT incorporates intersection types (Pottinger, 1980; Coppo et al., 1981; Barbanera et al., 1995) for typing objects. A distinctive characteristic of DOT is its use of path-dependent types (Amin et al., 2014). With path-dependent types, the treatment of recursive types is different from our calculus $F^{\mu\wedge}_{\leq\geq}$. In DOT, recursive types are introduced using recursive self types: $\{z \Rightarrow T^z\}$. The variable z is a term variable. Thus a recursive self type provides a limited form of dependent types, modeling a dependently-typed fixpoint operator. In contrast, in $F^{\mu\wedge}_{\leq\geq}$, the variable α of a recursive type $\mu\alpha.A$ is a type variable. In $F^{\mu\wedge}_{\leq\geq}$, the type of objects is similar to that in DOT: we employ intersections of the types of all the fields, and we require that the labels are disjoint. If the object uses recursive types, then we use a fold around the term.

Previous attempts to prove the undecidability of DOT reduced the problem to the undecidability problem in $F_{\leq}$, relying on a translation from $F_{\leq}$ to types in DOT (Rompf and Amin, 2016). However, as Hu and Lhoták (2020) later observed, the translation is not conservative. For example, in $F_{\leq}$, $\top \to \top \leq \forall(\alpha \leq \top).\top$ is not a valid subtyping statement because function types and universal types are not comparable. However, after translating them into DOT, the statement becomes $\forall(\alpha : \top).\ \top \leq \forall(\alpha : \{\top..\top\}).\ \top$, in which $\{\top..\top\}$ indicates that α is both upper and lower bounded by $\top$. This statement is valid in DOT variants that allow full or equivalent subtyping for bounded quantification, which breaks the conservativity from $F_{\leq}$ to DOT. Nevertheless, Hu and Lhoták (2020) showed that the undecidability of DOT can be reduced to an undecidable fragment $F^{-}_{\leq}$ of full $F_{\leq}$, that excludes the function types, and proved that DOT is undecidable.

There are two notable decidable variants of DOT: the strong kernel $D_{<:}$ calculus from Hu and Lhoták (2020) and the Wyvern language by Mackay et al. (2020). These systems share features akin to $F^{\mu\wedge}_{\leq\geq}$, making comparisons worthwhile. As variants of DOT, both support path-dependent types. The decidable variant by Hu and Lhoták (2020) selects specific features from DOT, including upper and lower bounds for path selection, and an equivalent subtyping quantifier for $F_{\leq}$, but it lacks recursive and intersection types. To handle the complexity of path types in proof of decidability, they define an algorithmic version of the subtyping rules, called stare-at subtyping, prove its equivalence to the declarative rules, and use a simple measure to show that the algorithmic rules terminate. The system developed by Mackay et al. (2020) shares several similarities with $F^{\mu\wedge}_{\leq\geq}$, including the enforcement of comparable constraints on bounds and the integration of a restricted version of intersection types for typing objects. However, it distinguishes itself by using equi-recursive types for recursion.

Reflecting on the complexity inherent in full intersection types, Mackay et al. (2020) also adopt a restricted form of these types to refine recursive objects. To ensure decidability, their methodology employs a kernel variant of $F_{\leq}$. As for the proof of decidability, Mackay et al. (2020) define type graphs, a graphical representation of types and type declarations, along with the dependency information between them. They provide a general algorithm for checking type graph subtyping, and show that in the restricted system, all types are homomorphic to type graphs that obey the material/shape separation property, which ensures that the subtyping algorithm terminates. In contrast, the decidability proof for $F^{\mu\wedge}_{\leq\geq}$ does not rely on alternative subtyping rules or type representations, and is solely based on measures. Both decidable systems in DOT incorporate top and bottom types and have been demonstrated to be reflexive, similarly to $F^{\mu\wedge}_{\leq\geq}$. However, one of the limitations for DOT is that transitivity elimination is not possible (Rompf and Amin, 2016), and even the two decidable fragments of DOT lack transitivity (Hu and Lhoták, 2020; Mackay et al., 2020). In contrast, in $F^{\mu\wedge}_{\leq\geq}$ transitivity can be derived from the subtyping rules.

Table 8: Comparison $F^{\mu\wedge}_{\leq\geq}$ with DOT and its variants.

	$F^{\mu\wedge}_{\leq\geq}$	Rompf and Amin (2016)	Hu and Lhoták (2020)	Mackay et al. (2020)
Path-dependent Types	$\times$	✓	✓	✓
Bounded Quantification	$\alpha \leq A$ or $\alpha \geq A$	$A \leq \alpha \leq B$	$A \leq \alpha \leq B$	$\alpha \leq A$ or $\alpha \geq A$
Recursive Types	iso	equi	$\times$	equi
Intersection Types	limited	✓	$\times$	limited
Quantifier Subtyping	equiv $F_{\leq}$	full $F_{\leq}$	equiv $F_{\leq}$	kernel $F_{\leq}$
Conservativity	✓	$\times$	$\times$	$\times$
Reflexivity	✓	✓	✓	✓
Transitivity	✓	built-in	$\times$	$\times$
Decidability	measures	$\times$	algo rules	type graphs

While $F^{\mu\wedge}_{\leq\geq}$ 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 $F^{\mu\wedge}_{\leq\geq}$ preserves the conservativity over kernel $F_{\leq}$, while DOT does not. Table 8 presents a comparative analysis of the four calculi.

7.3 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.

More recently, Rossberg (2023) proposed another calculus with a similar idea of using declared subtyping for recursive types, aiming at providing better and more efficient support for mutually recursive datatypes in type-safe low-level languages like Wasm. In their work, recursive types take the form $\mu\langle\alpha_1 \leq A_1, \alpha_2 \leq A_2\rangle.\,B$, where the declared bound A_2 can refer to α_1 so that two mutually recursive types can be defined at once. This avoids the polynomial explosion of encoding mutual recursion using single recursion à la Bekić's Lemma (Bekić, 2005). However, to deal with type bounds in the μ-operator they need to employ higher-order subtyping (Pierce and Steffen, 1997).

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 for 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 $F^{\mu\wedge}_{\leq\geq}$, together with the structural folding rule enables such encodings. As for encodings of objects, our work is helpful to further validate such encodings formally.

8 Conclusion

Recursive types and bounded quantification play a significant role in various programming languages. While these features have been extensively studied individually, their combined interaction has remained a challenging problem for a long time. Our $F^{\mu}_{\leq}$ calculus demonstrates a method for integrating iso-recursive types with two variations of $F_{\leq}$. We achieve a transitive and decidable subtyping relation for the kernel variant, and both calculi maintain conservativity over $F_{\leq}$ and are type sound. $F^{\mu}_{\leq}$ and $F^{\mu\wedge}_{\leq\geq}$ could provide a theoretical basis for object encodings and subtyping in algebraic data types. In particular, the full $F^{\mu}_{\leq}$ we have studied in this paper provides a foundation for the *ORBE* object encoding (Abadi et al., 1996). Recently, there has been a renewed interest in recursive types and bounded quantification, sparked by the DOT calculus. Our research helps in identifying calculi that include most features found in DOT, while preserving properties such as subtyping's decidability and transitivity, or even conservativity over $F_{\leq}$. Exploring extensions of $F^{\mu}_{\leq}$ to include more features from DOT constitutes an interesting direction for future research.

Conflicts of Interest

None

References

Abadi, M., Cardelli, L. & Viswanathan, R. (1996) An interpretation of objects and object types. Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA. Association for Computing Machinery. pp. 396–409.

Abadi, M. & Fiore, M. P. (1996) Syntactic considerations on recursive types. Proceedings 11th Annual IEEE Symposium on Logic in Computer Science. IEEE. pp. 242–252.

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

Amin, N. & Rompf, T. (2017) Type soundness proofs with definitional interpreters. Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 666–679.

Amin, N., Rompf, T. & Odersky, M. (2014) Foundations of path-dependent types. Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. Portland Oregon USA. ACM. pp. 233–249.

Aydemir, B., Charguéraud, A., Pierce, B. C., Pollack, R. & Weirich, S. (2008) Engineering formal metatheory. *ACM SIGPLAN Notices*. **43**(1), 3–15.

Backes, M., Hriţcu, C. & Maffei, M. (2014) *Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations*. **22**(2), 301–353.

Baldan, P., Ghelli, G. & Raffaetà, A. (1999) Basic Theory of F-Bounded Quantification. *Information and Computation*. **153**(2), 173–237.

Barbanera, F., Dezani-Ciancaglini, M. & de'Liguoro, U. (1995) Intersection and union types: Syntax and semantics. *Information and Computation*. **119**(2), 202–230.

Bekić, H. (2005) Definable operations in general algebras, and the theory of automata and flowcharts. *Programming Languages and Their Definition: H. Bekič (1936–1982)*. pp. 30–55.

Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A. D. & Maffeis, S. (2011) Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. **33**(2), 1–45.

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

Bourdoncle, F. & Merz, S. (1997) Type checking higher-order polymorphic multi-methods. Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 302–315.

Brandt, M. & Henglein, F. (1998) Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*. **33**(4), 309–338.

Bruce, K., Cardelli, L., Castagna, G., Group, H. O., Leavens, G. T. & Pierce, B. C. (1995) On binary methods. *Theory and Practice of Object Systems*. **1**(3), 221–242.

Bruce, K. B. (1994) A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*. **4**(2), 127–206.

Bruce, K. B., Cardelli, L. & Pierce, B. C. (1999) Comparing Object Encodings. *Information and Computation*. **155**(1), 108–133.

Canning, P., Cook, W., Hill, W., Olthoff, W. & Mitchell, J. C. (1989) F-bounded polymorphism for object-oriented programming. Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture. Imperial College, London, United Kingdom.

Canning, P. S., Cook, W. R., Hill, W. L. & Olthoff, W. G. (1989) Interfaces for strongly-typed object-oriented programming. *ACM SIGPLAN Notices*. **24**(10), 457–467.

Cardelli, L. (1985) Amber, combinators and functional programming languages. Proc. of the 13th Summer School of the LITP, Le Val D'Ajol, Vosges (France).

Cardelli, L., Martini, S., Mitchell, J. & Scedrov, A. (1994) An Extension of System F with Subtyping. *Information and Computation*. **109**(1-2), 4–56.

Cardelli, L. & Wegner, P. (1985) On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*. **17**, 471–522.

Castagna, G. & Pierce, B. C. (1994) Decidable bounded quantification. Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA. Association for Computing Machinery. pp. 151–162.

Church, A. (1932) A set of postulates for the foundation of logic. *Annals of Mathematics*. **33**(2), 346–366.

Colazzo, D. & Ghelli, G. (2005) Subtyping recursion and parametric polymorphism in kernel Fun. *Information and Computation*. **198**(2), 71–147.

Cook, W. R., Hill, W. & Canning, P. S. (1989) Inheritance is not subtyping. Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Association for

Computing Machinery.

Coppo, M., Dezani-Ciancaglini, M. & Venneri, B. (1981) Functional characters of solvable terms. *Mathematical Logic Quarterly*. **27**(2-6), 45–58.

Curien, P.-L. & Ghelli, G. (1992) Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. *Mathematical structures in computer science*. **2**(1), 55–91.

Dunfield, J. (2012) Elaborating intersection and union types. *ACM SIGPLAN Notices*. **47**(9), 17–28.

EPFL. (2021) Scala 3. available at https://www.scala-lang.org/.

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

Ghelli, G. (1993) Recursive types are not conservative over F≤. International Conference on Typed Lambda Calculi and Applications. Springer. pp. 146–162.

Haskell Development Team. (1990) Haskell. available at https://www.haskell.org/.

Hosoya, H., Pierce, B. C., Turner, D. N. *et al.*. (1998) Datatypes and subtyping. *Unpublished manuscript*.

Hu, J. & Lhoták, O. (2020) Undecidability of D<: And Its Decidable Fragments. *Proceedings of the ACM on Programming Languages*. **4**(POPL), 1–30.

INRIA. (1987) OCaml. available at https://ocaml.org/.

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

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

Mackay, J., Potanin, A., Aldrich, J. & Groves, L. (2020) Decidable subtyping for path dependent types. *Proceedings of the ACM on Programming Languages*. **4**(POPL), 1–27.

Millstein, T., Bleckner, C. & Chambers, C. (2004) Modular typechecking for hierarchically extensible datatypes and functions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. **26**(5), 836–889.

Morris, J. H. (1968) *Lambda Calculus Models of Programming Languages*. Ph.D. thesis.

Oliveira, B. C. d. S. (2009) Modular visitor components. European Conference on Object-Oriented Programming. Springer. pp. 269–293.

Oliveira, B. C. d. S., Cui, S. & Rehman, B. (2020) The duality of subtyping. 34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference).

Parigot, M. (1992) Recursive programming with proofs. *Theoretical Computer Science*. **94**(2), 335–356.

Pierce, B. & Steffen, M. (1997) Higher-order subtyping. *Theoretical Computer Science*. **176**(1), 235–282.

Pierce, B. C. (1994) Bounded quantification is undecidable. *Information and Computation*. **112**(1), 131–165.

Pierce, B. C. (1997) Bounded quantification with bottom. CSCI Technical Report 492. Computer Science Department, Indiana University.

Pierce, B. C. (2002) *Types and Programming Languages*. MIT press.

Pierce, B. C. & Turner, D. N. (1994) Simple type-theoretic foundations for object-oriented programming. *Journal of functional programming*. **4**(2), 207–247.

Poll, E. (1998) Subtyping and inheritance for categorical datatypes: Preliminary report (type theory and its applications to computer systems). *Kyoto University Research Information Repository*. **1023**, 112–125.

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

Reynolds, J. C. (1974) Towards a theory of type structure. Programming Symposium: Proceedings, Colloque Sur La Programmation Paris, April 9–11, 1974. Springer. pp. 408–425.

Reynolds, J. C. (1988) *Preliminary Design of the Programming Language Forsythe*. Carnegie-Mellon University. Department of Computer Science.

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

Rossberg, A. (2023) Mutually iso-recursive subtyping. *Proceedings of the ACM on Programming Languages*. **7**(OOPSLA2), 347–373.

Sangiorgi, D. & Milner, R. (1992) The problem of “weak bisimulation up to”. CONCUR. pp. 32–46.

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

The Coq Development Team. (2021) Coq. v8.13, available at https://coq.inria.fr.

Wadler, P. (1998) The expression problem. discussion on the Java Genericity mailing list.

Zhou, L. & Oliveira, B. C. d. S. (2025) QuickSub: Efficient Iso-Recursive Subtyping. *Proceedings of the ACM on Programming Languages*. **9**(POPL).

Zhou, L., Wan, Q. & Oliveira, B. C. d. S. (2024) Full Iso-Recursive Types. *Proceedings of the ACM on Programming Languages*. **8**(OOPSLA2), 278:192–278:221.

Zhou, L., Zhou, Y. & Oliveira, B. C. d. S. (2023) Recursive subtyping for all. *Proceedings of the ACM on Programming Languages*. **7**(POPL), 1396–1425.

Zhou, Y., Oliveira, B. C. d. S. & Fan, A. (2022) A calculus with recursive types, record concatenation and subtyping. Asian Symposium on Programming Languages and Systems. Springer. pp. 175–195.

Zhou, Y., Oliveira, B. C. d. S. & Zhao, J. (2020) Revisiting iso-recursive subtyping. *Proceedings of the ACM on Programming Languages*. **4**(OOPSLA), 1–28.

Zhou, Y., Zhao, J. & Oliveira, B. C. d. S. (2022) Revisiting iso-recursive subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. **44**(4), 1–54.