Full Iso-Recursive Types

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

There are two well-known formulations of recursive types: *iso-recursive* and *equi-recursive* types. Abadi and Fiore [LICS 1996] have shown that iso- and equi-recursive types have the same expressive power. However, their encoding of equi-recursive types in terms of iso-recursive types requires explicit coercions. These coercions come with significant additional *computational overhead*, and complicate reasoning about the equivalence of the two formulations of recursive types.

This paper proposes a generalization of iso-recursive types called *full* iso-recursive types. Full iso-recursive types allow encoding all programs with equi-recursive types without computational overhead. Instead of explicit term coercions, all type transformations are captured by *computationally irrelevant* casts, which can be erased at runtime without affecting the semantics of the program. Consequently, reasoning about the equivalence between the two approaches can be greatly simplified. We present a calculus called λ^{μ}_{Fi}, which extends the simply typed lambda calculus (STLC) with full iso-recursive types. The λ^{μ}_{Fi} calculus is proved to be type sound, and shown to have the same expressive power as a calculus with equi-recursive types. We also extend our results to subtyping, and show that equi-recursive subtyping can be expressed in terms of iso-recursive subtyping with cast operators.

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

Additional Key Words and Phrases: Recursive types, Subtyping, Type system

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

1 Introduction

Recursive types are used in many programming languages to express recursive data structures, or recursive interfaces. There are two well-known formulations of recursive types: *iso-recursive* and *equi-recursive* types. With equi-recursive types [Morris 1968], a recursive type $\mu\alpha.\ A$ and its unfolding $A[\alpha \mapsto \mu\alpha.\ A]$ are equal, since they represent the same infinite tree [Amadio and Cardelli 1993]. With iso-recursive types, a recursive type is only isomorphic to its unfolding [Crary et al. 1999]. To witness the isomorphism, explicit fold and unfold operators are used.

Because both formulations provide alternative ways to model recursive types, the relationship between iso- and equi-recursive types has been a topic of study [Abadi and Fiore 1996; Patrignani et al. 2021; Urzyczyn 1995]. Understanding this relationship is important to answer questions such as whether the expressive power of the two formulations is the same or not. Urzyczyn proved that these two formulations have the same expressive power when the types considered are restricted to

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

ACM 2475-1421/2024/10-ART278
https://doi.org/10.1145/3689718

be positive. Abadi and Fiore extended Urzyczyn's result and showed that unrestricted formulations of iso- and equi-recursive types also have the same expressive power, leading to the well-known statement that "iso-recursive types have the same expressive power as equi-recursive types". In addition, Patrignani et al. showed that the translation from iso-recursive to equi-recursive types is fully abstract with respect to contextual equivalence.

However, the encoding proposed by Abadi and Fiore requires explicit coercions, which are interpreted as functions to be evaluated at runtime. Iso-recursive types can only encode equi-recursive types with significant additional *computational overhead*. Moreover, these explicit coercions cannot be easily erased and therefore complicate the reasoning about *behavioral equivalence*. To address the latter challenge, Abadi and Fiore defined an axiomatized program logic and showed that the iso-recursive term obtained by their encoding behaves in the same way as the original equi-recursive term in the logic. However, the soundness of their program logic is left as a conjecture, since they did not consider an operational semantics in their work. Thus, behavioral equivalence between programs written with equi-recursive and iso-recursive types lacks a complete proof in the literature. Without introducing explicit coercions, iso-recursive types are strictly weaker than equi-recursive types, since the infinite tree view of equi-recursive types equates more types than isomorphic unfoldings of recursive types.

This paper proposes a *generalization* of iso-recursive types called *full* iso-recursive types. Full iso-recursive types overcome the challenges of traditional iso-recursive types in achieving the typing expressiveness and behavioral equivalence seen in equi-recursive types. Instead of fold and unfold operators and explicit coercions, we use a more general notion of *computationally irrelevant cast operators* [Cretin 2014; Sulzmann et al. 2007], which allow transformations on any types that are equivalent in an equi-recursive setting. Full iso-recursive types can encode *all* programs with equi-recursive types *without* computational overhead, since casts can be erased at runtime without affecting the semantics of the program. Consequently, the semantic equivalence between programs written with equi-recursive and full iso-recursive types is also greatly simplified, and allows for a complete proof, compared to Abadi and Fiore's work.

We present a calculus called λ^{μ}_{Fi}, which extends the simply typed lambda calculus (STLC) with full iso-recursive types. The λ^{μ}_{Fi} calculus is proved to be type sound, and shown to have the same typing power as a calculus with equi-recursive types. To prove the latter result, we define a type-directed elaboration from the calculus with equi-recursive types to λ^{μ}_{Fi}, and an erasure function that removes all casts from full iso-recursive terms to obtain equi-recursive terms. Moreover, the termination and divergence behavior of programs is preserved under the elaboration and erasure operations. Therefore, λ^{μ}_{Fi} is sound and complete w.r.t. the calculus with equi-recursive types in terms of both typing and dynamic semantics. On the other hand, traditional iso-recursive types can be seen as a special case of full iso-recursive types. One can easily recover the traditional unfold and fold operators by using the corresponding cast operators accordingly. So all the results for iso-recursive types can be adapted to full iso-recursive types as well.

Our results extend to subtyping: equi-recursive subtyping can be expressed in terms of iso-recursive subtyping with cast operators. Although subtyping between equi-recursive types [Amadio and Cardelli 1993; Brandt and Henglein 1998; Gapeyev et al. 2002] and subtyping between iso-recursive types [Abadi and Cardelli 1996; Zhou et al. 2022] has been studied in depth, the relationship between the two approaches has been largely unexplored. We revisit Amadio and Cardelli [1993]'s seminal work on equi-recursive subtyping and observe that equi-recursive subtyping can be decomposed into a combination of equi-recursive equalities and iso-recursive subtyping. Since our cast operators can capture all the equi-recursive equalities, in the calculus $\lambda^{\mu<:}_{Fi}$ extended with subtyping, we can achieve a simple encoding of equi-recursive subtyping.

The implications of full iso-recursive types are two-fold: theoretical and practical. Theoretically, full iso-recursive types offer the expressive power of equi-recursive types but with a more manageable metatheory. This allows for future extensions with more advanced type system features. Complex type system features, such as type-level lambdas found in System F_ω [Cai et al. 2016], can lead to undecidable type equivalence relations with equi-recursive types. Iso-recursive types provide explicit control over folding and unfolding, avoiding issues with undecidability. This advantage is underscored in work on recursive modules [Crary et al. 1999], which adopted iso-recursive types to obtain more practical module systems with decidable type equivalence [Dreyer 2005; Dreyer et al. 2001; Russo 2001] in settings similar to System F_ω. Rossberg [2023] also pointed out that iso-recursive types enjoy efficient meta operations, such as type equivalence checking. Full iso-recursive types enhance these benefits by retaining the expressive power of equi-recursive types without adding runtime computational overhead. Beyond avoiding decidability limitations, (full) iso-recursive types also simplify other theoretical challenges seen in equi-recursive types, particularly in the presence of subtyping. For example, the integration of bounded quantification with equi-recursive subtyping introduces substantial complexity [Colazzo and Ghelli 2005; Ghelli 1993; Jeffrey 2001]. In contrast, with iso-recursive types these extensions are natural and straightforward [Zhou et al. 2023]. Since the subtyping rules we adopt in $\lambda^{\mu<:}_{Fi}$ are standard, previous results on iso-recursive subtyping can potentially be extended to full iso-recursive types too.

Full iso-recursive types also open the path for new applications. In the design of realistic compilers, it is common to have source languages that are lightweight in terms of type annotations; and target languages, which are used internally, that are heavy on annotations, but are simple to type-check [Crary 2000]. For instance, the GHC Haskell compiler works in this way: the source language (Haskell) has a lot of convenience via type inference, and no explicit casts are needed in source programs. A source program is then elaborated to a variant of System Fc [Sulzmann et al. 2007], which is a System F like language with explicit type annotations, type applications and also explicit casts. Our work enables designing source languages with equi-recursive types, which are elaborated to target languages with full iso-recursive types. Equi-recursive types offer convenience because they can avoid explicit folds and unfolds, but type-checking is complex. With full iso-recursive types we need to write explicit casts, but type-checking is simple. Thus we can have an architecture similar to GHC. In this scenario it is important that no computational overhead is introduced during the elaboration, which is why using standard iso-recursive types is not practical. In addition, source languages could also use full iso-recursive types directly, and explicit casts could be hidden into language constructs (such as constructors, method calls and/or pattern matching) or inferred by the type system to reduce the overhead of writing annotations. This is another way to use full iso-recursive types, which is similar to current applications of iso-recursive types.

The main contributions of this paper are as follows:

- **Full iso-recursive types:** a novel formulation of recursive types, which generalizes the traditional iso-recursive fold and unfold operators to cast operators.
- **The λ^{μ}_{Fi} calculus,** which extends the STLC with full iso-recursive types. We present a type system, a call-by-value operational semantics, and a type soundness proof.
- **Equivalence to equi-recursive types.** We show that λ^{μ}_{Fi} is equivalent to STLC extended with equi-recursive types in terms of typing and dynamic semantics.
- **Extension to subtyping.** We present $\lambda^{\mu<:}_{Fi}$, an extension of λ^{μ}_{Fi} with iso-recursive subtyping, and show the same metatheory results for $\lambda^{\mu<:}_{Fi}$, namely, type soundness, typing equivalence and behavioral equivalence to equi-recursive types with subtyping.
- **Coq formalization** and proofs for all the new metatheory results of full iso-recursive types, except for Theorem 5.5, which is adapted from the literature [Amadio and Cardelli 1993].

$\boxed{A \doteq B}$ *(Equi-recursive Equality)*

$$\textsc{Tyeq-contract}\ \dfrac{A[\alpha \mapsto B_1] \doteq B_1 \qquad A[\alpha \mapsto B_2] \doteq B_2 \qquad A \text{ is contractive in } \alpha}{B_1 \doteq B_2} \qquad \textsc{Tyeq-unfold}\ \dfrac{}{\mu\alpha.\ A \doteq A[\alpha \mapsto \mu\alpha.\ A]}$$

$$\textsc{Tyeq-mu-cong}\ \dfrac{A \doteq B}{\mu\alpha.\ A \doteq \mu\alpha.\ B} \qquad \textsc{Tyeq-trans}\ \dfrac{A \doteq B \qquad B \doteq C}{A \doteq C} \qquad \textsc{Tyeq-refl}\ \dfrac{}{A \doteq A} \qquad \textsc{Tyeq-symm}\ \dfrac{A \doteq B}{B \doteq A} \qquad \textsc{Tyeq-arr}\ \dfrac{A_1 \doteq A_2 \qquad B_1 \doteq B_2}{A_1 \to B_1 \doteq A_2 \to B_2}$$

Fig. 1. Amadio and Cardelli's equi-recursive type equality.

2 Overview

This section provides an overview of our work. We first briefly review the two main approaches to recursive types, namely iso-recursive types and equi-recursive types, and the relationship between the two approaches. Then we introduce our key ideas and results.

2.1 Equi-Recursive Types

Equi-recursive types treat recursive types and their unfoldings as equal. The advantage of equi-recursive types is that they are simple to use, since there is no need to insert explicit annotations in the term language to transform between equal types, as shown in rule Typ-eq.

$$\textsc{Typ-eq}\ \dfrac{\Gamma \vdash e : A \qquad A \doteq B}{\Gamma \vdash e : B}$$

The metatheory of equi-recursive types has been studied by Amadio and Cardelli [1993]. They proposed a tree model for specifying equality (or subtyping). In essence, two recursive types are equal (or subtypes) if their infinite unfoldings are equal (or subtypes). The tree model provides a clear and solid foundation for the interpretation of equi-recursive types.

Amadio and Cardelli also provided a rule-based axiomatization to compare equi-recursive types, as shown in Figure 1. They proved the soundness and completeness of the rules to the tree-based interpretation. For example, rule Tyeq-unfold states that a recursive type is equal to its unfolding, and rule Tyeq-mu-cong states that the equality is congruent with respect to the recursive type operator. Rule Tyeq-contract states that two types are equal if they are the fixpoints of the same type function $A[\alpha]$. Note that A needs to be contractive in α, i.e. either α is not free in A or A can be unfolded to a type of the form $A_1 \to A_2$. This is to prevent equating arbitrary types using non-contractive type functions, such as when A is α. Rule Tyeq-contract allows recursive types that have equal infinite unfoldings, but are not directly related by finite unfoldings, to be equal. For example, let $A[\alpha] = \texttt{Int} \to \texttt{Int} \to \alpha$, then $B_1 = \mu\alpha.\ \texttt{Int} \to \texttt{Int} \to \alpha$ and $B_2 = \mu\alpha.\ \texttt{Int} \to \alpha$ are equal according to rule Tyeq-contract:

$$\dfrac{\dfrac{}{B_1 \doteq A[\alpha \mapsto B_1]}\textsc{Tyeq-unfold} \qquad \dfrac{\cdots}{B_2 \doteq A[\alpha \mapsto B_2]} \qquad A \text{ is contractive in } \alpha}{\mu\alpha.\ \texttt{Int} \to \texttt{Int} \to \alpha \doteq \mu\alpha.\ \texttt{Int} \to \alpha}\textsc{Tyeq-contract}$$

Here, the missing derivation is:

$$\dfrac{\dfrac{\dfrac{}{\texttt{Int} \doteq \texttt{Int}}\textsc{Tyeq-refl} \qquad \dfrac{}{\mu\alpha.\ \texttt{Int} \to \alpha \doteq \texttt{Int} \to \mu\alpha.\ \texttt{Int} \to \alpha}\textsc{Tyeq-unfold}}{\texttt{Int} \to B_2 \doteq \texttt{Int} \to \texttt{Int} \to B_2}\textsc{Tyeq-arrow}}{B_2 \doteq A[\alpha \mapsto B_2]}\textsc{Tyeq-trans} \text{ and } \textsc{Tyeq-unfold}$$

$\boxed{H \vdash A \doteq B}$ *(Inductive Equi-recursive Equality)*

$$\frac{A = B \in H}{H \vdash A \doteq B}\ \textsc{Tye-assump} \qquad \frac{}{H \vdash A \doteq A}\ \textsc{Tye-refl} \qquad \frac{H \vdash A \doteq B \qquad H \vdash B \doteq C}{H \vdash A \doteq C}\ \textsc{Tye-trans} \qquad \frac{}{H \vdash \mu\alpha.\,A \doteq A[\alpha \mapsto \mu\alpha.\,A]}\ \textsc{Tye-unfold}$$

$$\frac{H \vdash A \doteq B}{H \vdash B \doteq A}\ \textsc{Tye-symm} \qquad \frac{H, A_1 \to B_1 = A_2 \to B_2 \vdash A_1 \doteq A_2 \qquad H, A_1 \to B_1 = A_2 \to B_2 \vdash B_1 \doteq B_2}{H \vdash A_1 \to B_1 \doteq A_2 \to B_2}\ \textsc{Tye-arrfix}$$

Fig. 2. Brandt and Henglein's inductively defined equi-recursive type equality.

Despite its equivalence to the tree model, Amadio and Cardelli's axiomatization is not easy to use in practice. The type function A in rule Tyeq-contract and the intermediate type B in rule Tyeq-trans are not directly derivable from the equality conclusion. Thus, Amadio and Cardelli's rule does not form an algorithm for equi-recursive equality, but rather it is a declarative specification. Later on, there have been a few alternative axiomatizations of equi-recursive types [Brandt and Henglein 1998; Danielsson and Altenkirch 2010; Gapeyev et al. 2002], which are all proved to be equivalent to the tree model. Among them, Brandt and Henglein proposed an inductively defined relation $H \vdash A \doteq B$ for equi-recursive type equality, shown in Figure 2. H is a list of type equality assumptions that is used to derive the equality $A \doteq B$. New equalities are added to H every time function types are compared, as shown in rule Tye-arrfix. Compared to rule Tyeq-contract, rule Tye-arrfix encodes the coinductive essence of equi-recursive types in a simpler way. Moreover, Brandt and Henglein showed a terminating algorithm for checking type equality using their rules. Therefore, we choose Brandt and Henglein's axiomatization as the basis for our work.

2.2 Iso-Recursive Types

Iso-recursive types [Crary et al. 1999] are a different approach that treats recursive types and their unfoldings as different, but isomorphic up to an unfold/fold operator. With iso-recursive types foldings and unfoldings of the recursive types must be explicitly triggered, and there is no typing rule Typ-eq to implicitly convert between equivalent types. Rule Typ-unfold and rule Typ-fold show the typing rules for unfolding and folding a term of recursive types. A fold expression constructs a recursive type, while an unfold expression opens a recursive type to its unfolding.

$$\frac{\Gamma \vdash e : \mu\alpha.\,A}{\Gamma \vdash \text{unfold}\,[\mu\alpha.\,A]\;e : A[\alpha \mapsto \mu\alpha.\,A]}\ \textsc{Typ-unfold} \qquad \frac{\Gamma \vdash e : A[\alpha \mapsto \mu\alpha.\,A]}{\Gamma \vdash \text{fold}\,[\mu\alpha.\,A]\;e : \mu\alpha.\,A}\ \textsc{Typ-fold}$$

One advantage of iso-recursive types, as we discussed in §1, is that they are easier to extend to more complex type systems, which may easily make the type equality relation undecidable or complicate the metatheory. Instead, iso-recursive types utilize explicit fold/unfold annotations to control type-level conversions, thus avoiding these issues. One disadvantage of iso-recursive types is their inconvenience in use due to the explicit fold and unfold operators. However, this disadvantage can be mitigated by hiding folding and unfolding under other language constructs, such as pattern matching, constructors or method calls [Crary et al. 1999; Harper and Stone 2000; Lee et al. 2015; Pierce 2002; Vanderwaart et al. 2003; Yang and Oliveira 2019; Zhou et al. 2022]. As we shall see in §2.3, a further disadvantage of iso-recursive types is that folding and unfolding alone is not enough to provide all of the expressive power of the type equality rules. In some cases, explicit, computationally relevant, term coercions are necessary.

2.3 Relating Iso-Recursive and Equi-Recursive Types

The relationship between iso-recursive types and equi-recursive types has been a subject of study in the literature [Abadi and Fiore 1996; Patrignani et al. 2021; Urzyczyn 1995]. This subsection reviews existing approaches to relate the two approaches and their issues.

Encoding iso-recursive types. Encoding of iso-recursive types in terms of equi-recursive types is straightforward, simply by erasing the fold and unfold operators [Abadi and Fiore 1996]. Since the rule Tyeq-unfold states that a recursive type is equal to its unfolding, it is easy to see that the encoding is type preserving. The encoding is also behavior preserving, since the reduction rules with fold and unfold operators will become no-ops when erased, as shown below:

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

Notice that in the process of reducing folded and unfolded expression e, we merely reduce e. The type A does not influence the reduction of e. Eventually, when e reaches a value v, an unfold cancels a fold and we simply obtain v. The two type annotations in rule Red-elim can be arbitrary. In a well typed term, A and B must be recursive types that are equal or in a subtyping relation, depending on the whether recursive subtyping is considered, but the reduction rule in its general form does not require this, and the type annotations do not affect the result v either. In other words, folding and unfolding are *computationally irrelevant*: they do not influence the runtime result, and can be erased, to avoid runtime costs. Moreover, Patrignani et al. [2021] proved that the erasure operation is fully abstract, i.e. two terms that cannot be distinguished by any program contexts in the iso-recursive setting are also indistinguishable in the equi-recursive setting.

Encoding equi-recursive types via fold and unfold. It takes more effort to encode equi-recursive types in terms of iso-recursive types. Since equi-recursive types treat recursive types and their unfoldings as equal, we need to insert explicit fold and unfold operators in the iso-recursive setting to transform between equal types. For example, let e be a function that keeps taking integer arguments and returning itself, which can be typed as a recursive type $\mu\alpha.\ \mathtt{Int} \to \alpha$. In an equi-recursive setting, $(e\ 1)$ can be typed as $\mu\alpha.\ \mathtt{Int} \to \alpha$, by using the rule Typ-eq and rule Tyeq-unfold to unfold the recursive type to $\mathtt{Int} \to (\mu\alpha.\ \mathtt{Int} \to \alpha)$ so that e can be applied to the argument 1. However, in the iso-recursive setting, we need to insert an unfold operator to make the transformation explicit, as shown in the following derivation:

$$\textsc{Typ-app}\ \dfrac{\textsc{Typ-unfold}\ \dfrac{\vdash e : \mu\alpha.\ \mathtt{Int} \to \alpha}{\vdash \mathsf{unfold}\ [\mu\alpha.\ \mathtt{Int} \to \alpha]\ e : \mathtt{Int} \to (\mu\alpha.\ \mathtt{Int} \to \alpha)} \qquad \vdash 1 : \mathtt{Int}}{\vdash (\mathsf{unfold}\ [\mu\alpha.\ \mathtt{Int} \to \alpha]\ e)\ 1 : \mu\alpha.\ \mathtt{Int} \to \alpha}$$

Fold/unfold is not enough: computationally relevant explicit coercions. The above example shows that, for some equi-recursive terms, inserting fold and unfold operators within the term language can achieve an encoding in terms of iso-recursive types. However, this is not always the case. Recall that $\mu\alpha.\ \mathtt{Int} \to \mathtt{Int} \to \alpha$ and $\mu\alpha.\ \mathtt{Int} \to \alpha$ are also equal in the equi-recursive setting, but they are not directly related by fold and unfold operators. The two types have different arities of function arguments in the recursive body, while fold and unfold operators only transform between different views of the same recursive structure. Thus, fold and unfold operations alone cannot achieve this transformation. More sophisticated operations are needed to fully encode equi-recursive equalities.

To address this issue, Abadi and Fiore [1996] proposed an approach to insert *explicit coercion functions*. They showed that, for any two equi-recursive types A and B considered to be equal following the derivation in Figure 1, there exists a coercion function $f : A \to B$ that can be applied to terms of type A to obtain terms of type B. With the coercion function, terms that are well typed by rule Typ-tyeq can now have an encoding in terms of iso-recursive types, possibly with the help of explicit coercion functions.

One issue is that the insertion of coercion functions affects the computational structure of the terms. For example, assume that e has a function type $\mathtt{Int} \to \mathtt{Int} \to (\mu\alpha.\ \mathtt{Int} \to \alpha)$. This type can be partially folded to $\mathtt{Int} \to (\mu\alpha.\ \mathtt{Int} \to \alpha)$. In an equi-recursive setting, due to the rule Typ-eq, the term e can be assigned the type $\mathtt{Int} \to (\mu\alpha.\ \mathtt{Int} \to \alpha)$ directly without any changes. In an iso-recursive setting, in addition to folding and unfolding, we need explicit coercions. The coercion function for this transformation is

$$\lambda(x : \mathtt{Int} \to \mathtt{Int} \to \mu\alpha.\ \mathtt{Int} \to \alpha).\ \lambda(y : \mathtt{Int}).\ \mathsf{fold}\ [\mu\alpha.\ \mathtt{Int} \to \alpha]\ (x\, y)$$

Now, applying the coercion function to e results in a term of type $\mathtt{Int} \to (\mu\alpha.\ \mathtt{Int} \to \alpha)$. Unfortunately, such explicit coercion functions are computationally relevant. They introduce an extra function application. Thus, an encoding of equi-recursive types in terms of iso-recursive types can introduce non-trivial computational overhead. The issue is particularly problematic because some coercions need to essentially be *recursive* functions, which is the case for $\mu\alpha.\ \mathtt{Int} \to \alpha \doteq \mu\alpha.\ \mathtt{Int} \to \mathtt{Int} \to \alpha$. Therefore, it is impractical to use such an encoding in a language implementation.

Issues with reasoning. Explicit coercions also bring new challenges in terms of reasoning, and in particular in proving the behavioral preservation of the encoding. Continuing with the previous example, if we transform this resulting term back to an equi-recursive setting, by erasing the fold and unfold operators, we will get a term:

$$(\lambda(x : \mathtt{Int} \to \mathtt{Int} \to \mu\alpha.\ \mathtt{Int} \to \alpha).\ \lambda(y : \mathtt{Int}).\ (x\, y))\, e \tag{1}$$

This term is equivalent to e under $\beta-$ and $\eta-$reduction, but it is not the same as e anymore. In more complicated cases, especially for derivations involving the use of rule Tyeq-contract, one needs to use complex coercion function combinators to achieve the encoding. In turn, this leads to a more significant change in the syntactic structure of the terms, making it difficult to reason about the behavior preservation of the encoding. Abadi and Fiore proved that the encoding is equivalent to the original term in an axiomatized program logic, but the soundness of the program logic is conjectured to be sound, and the authors did not consider an operational semantics. Thus, while it is expected that the behavioral equivalence result holds (assuming the conjecture and a suitable operational semantics), there is no complete proof in the literature for this result.

2.4 Subtyping

Equi-recursive subtyping. It is common to extend recursive types with subtyping. For equi-recursive types, Amadio and Cardelli proposed a set of rules, which rely on the equality relation in Figure 1. We show some selected rules below:

$$\frac{A \doteq B}{\Sigma \vdash A \leq B}\ \textsc{ACSub-eq} \qquad \frac{\Sigma \vdash B_1 \leq A_1 \qquad \Sigma \vdash A_2 \leq B_2}{\Sigma \vdash A_1 \to A_2 \leq B_1 \to B_2}\ \textsc{ACSub-arrow} \qquad \frac{\alpha \leq \beta \in \Sigma}{\Sigma \vdash \alpha \leq \beta}\ \textsc{ACSub-var} \qquad \frac{\Sigma, \alpha \leq \beta \vdash A \leq B}{\Sigma \vdash \mu\alpha.\ A \leq \mu\beta.\ B}\ \textsc{ACSub-rec}$$

Two types are in a subtyping relation if their infinite unfoldings are equal, or equivalently, if they can be proved by the Brandt and Henglein or Amadio and Cardelli's axiomatization, as shown in rule ACSub-eq. The subtyping relation is structural, as can be seen in rule ACSub-arrow. For dealing with recursive types, rule ACSub-rec states that two recursive types are in a subtyping relation if their recursive bodies are subtypes, when assuming that the recursive variable of the two types are in a subtyping relation. The subtyping rules are also referred to as the Amber rules, since rule ACSub-rec was adopted by the implementation of the Amber programming language [Cardelli

1985]. The Amber rules are proved to be sound and complete to the tree model interpretation of equi-recursive subtyping [Amadio and Cardelli 1993].

Iso-recursive subtyping. For iso-recursive types, one can replace the equi-recursive equality relation in rule ACSub-eq with the syntactic equality relation to obtain the iso-recursive style Amber rules. The iso-recursive Amber rules are well-known and widely used for subtyping iso-recursive types [Abadi and Cardelli 1996; Bengtson et al. 2011; Chugh 2015; Duggan 2002; Lee et al. 2015; Swamy et al. 2011]. However, the metatheory for the iso-recursive Amber rules has not been well studied until recently [Zhou et al. 2020, 2022]. Zhou et al. provided a new specification for iso-recursive subtyping and proved a number of metatheory results, including type soundness, transitivity of the subtyping relation, and equivalence to the iso-recursive Amber rules.

Unlike type equality, the relation between equi-recursive and iso-recursive subtyping has been less studied. One attempt that we are aware of is the work by Ligatti et al. [2017]. They provided an extension of the iso-recursive subtyping rules to allow for subtyping between recursive types and their unfoldings, but their rules cannot account for the full expressiveness of equi-recursive subtyping. For example, $\mu\alpha.\ \mathtt{Int} \to \alpha \leq \mu\alpha.\ \mathtt{Int} \to \mathtt{Int} \to \alpha$ is a valid subtyping relation in the equi-recursive Amber rules using rule ACSub-eq, but it is not derivable in Ligatti et al.'s rules.

2.5 Key Ideas and Results

As we have shown, encoding iso-recursive types with equi-recursive types is simple. As for the other direction, Abadi and Fiore showed that iso-recursive types can be encoded with equi-recursive types, which leads to a well-known statement that "iso-recursive types have the same expressive power as equi-recursive types". However, their encoding involves the insertion of explicit coercion functions, and lacks a complete proof of correctness. In our work, we present a novel approach to iso-recursive types, full iso-recursive types, which extends the unfold and fold operators to a more general form. We show that full iso-recursive types and equi-recursive types can be mutually encoded and the encoding preserves the semantic behavior. Compared to the previous work, the correctness proof of our encoding is straightforward and foundational, without relying on any a priori assumptions.

Type Casting. The key idea of our approach is the introduction of a type casting relation that generalizes the unfold and fold operators. Instead of allowing only the unfold and fold operators to transform between recursive types and their unfoldings, we allow terms of any type to be transformed to their equivalent type using the type casting relation. The rules Typ-unfold and Typ-fold are now replaced by the following rule:

$$\textsc{Typ-cast}\quad \frac{\Gamma \vdash e : A \qquad \cdot;\cdot \vdash A \hookrightarrow B : c}{\Gamma \vdash \mathsf{cast}\ [c]\ e : B}$$

The type casting relation $A \hookrightarrow B : c$ states that type A can be cast to type B using the casting operator c. Essentially, the type casting relation is an equivalent form of Brandt and Henglein's type equality relation, augmented with a casting operator c, a new syntactic construct to witness the proof of the type casting relation. There are a few different designs in our type casting relation compared to the type equality relation in Figure 2 though. For example, we remove the rule Tye-symm from the type casting relation, for reasons that we will discuss in §4.2, and proved that rule Tye-symm is admissible from the remaining rules. After resolving these issues, it is easy to encode the equi-recursive rule Typ-eq using the type casting relation in rule Typ-cast. For instance, the encoding in (1) can now be replaced by the following term:

$$\mathsf{cast}\ [\mathsf{id} \to \mathsf{fold}_{(\mu\alpha.\ \mathtt{Int}\to\alpha)}]\ e$$

Here id is the identity casting operator, and $\text{fold}_{\mu\alpha.\ \texttt{Int}\to\alpha}$ is the casting operator that witnesses the proof of a folding from $\texttt{Int} \to (\mu\alpha.\ \texttt{Int} \to \alpha)$ to $\mu\alpha.\ \texttt{Int} \to \alpha$. Thus, the full iso-recursive typing rules are equivalent to the equi-recursive typing rules. Moreover, the cast operators can be automatically inferred and inserted into the terms. Brandt and Henglein present an algorithm for finding equi-recursive subtyping derivations [Brandt and Henglein 1998, Figure 5], prove its termination, and argue that their results also hold for type equality derivations. Since the algorithm does not use the changed rules in our type casting relation, we can still use the same algorithm to automatically elaborate any terms written with equi-recursive types to λ^{μ}_{Fi}. We identify the equi- to full iso-recursive encoding as one of the key applications of our work, as already discussed in §1.

On the other hand, the unfolding and folding operators in standard iso-recursive types can be recovered from our type casting relation. For example, the term $(\text{cast}\ [\text{fold}_A]\ e)$ is essentially equivalent to $(\text{fold}\ [A]\ e)$, and the term $(\text{cast}\ [\text{unfold}_A]\ e)$ is equivalent to $(\text{unfold}\ [A]\ e)$ in terms of typing and dynamic semantics, as we will show in the following sections. Therefore, full iso-recursive types are a generalization of standard iso-recursive types.

Push Rules. The extended typing rules brings new challenges to the design of semantics and the proof of type soundness. With the casting operator, there are terms that are not simple unfoldings or foldings of recursive types: the operational semantics needs to be extended to handle these terms. For example, terms such as $(\text{cast}\ [\text{id} \to \text{fold}_{(\mu\alpha.\ \texttt{Int}\to\alpha)}]\ e)$, which have no analogous representation in calculi with standard iso-recursive types, need to be considered during reduction. To address this issue, we introduce a set of new reduction rules to handle casting operators:

$$\textsc{Red-cast}\ \frac{e \hookrightarrow e'}{\text{cast}\ [c]\ e \hookrightarrow \text{cast}\ [c]\ e'} \qquad \textsc{Red-cast-arr}\ \frac{}{(\text{cast}\ [c_1 \to c_2]\ v_1)\ v_2 \hookrightarrow \text{cast}\ [c_2]\ (v_1\ (\text{cast}\ [\neg c_1]\ v_2))} \qquad \textsc{Red-cast-id}\ \frac{}{\text{cast}\ [\text{id}]\ v \hookrightarrow v}$$

The reduction rules are designed in a call-by-value fashion, and we also define the cast of values of function types (i.e. $\text{cast}\ [c_1 \to c_2]\ v$) as values. The new reduction rules in our system are referred to as *push* rules, since they push the casting operators inside the terms to make the terms more reducible, as shown in rule Red-Cast-Arr. Our design is inspired by the homonymous push rules in the design of calculi with coercions [Cretin 2014; Sulzmann et al. 2007]. Note that the casting operator $\neg c_1$ computes the dual of the casting operator c_1, which is used to indicate the reverse transformation that c_1 represents (e.g. $\neg\ \text{fold}_{\mu\alpha.\ A} = \text{unfold}_{\mu\alpha.\ A}$, $\neg\ \text{id} = \text{id}$). Intuitively, the push rules are designed to ensure that the casting operators can be reduced to smaller constructs and applied to the correct types, so that the reduction steps are always type preserving. We will reason about the behavior of the push rules formally in §4.1. A running example of a reduction using the push rules is shown as follows:

$$\begin{array}{lll} & (\text{cast}\ [\text{id} \to \text{fold}_{(\mu\alpha.\ \texttt{Int}\to\alpha)}]\ v)\ 1 & \\ \hookrightarrow & \text{cast}\ [\text{fold}_{(\mu\alpha.\ \texttt{Int}\to\alpha)}]\ (v\ (\text{cast}\ [\text{id}]\ 1)) & (\textsc{Red-cast-arr}) \\ \hookrightarrow & \text{cast}\ [\text{fold}_{(\mu\alpha.\ \texttt{Int}\to\alpha)}]\ (v\ 1) & (\textsc{Red-cast-id}\ \text{and}\ \textsc{Red-cast}) \end{array}$$

A key result of our work is the type soundness of the full iso-recursive calculus, which is proved by showing that the push rules preserve the type of the terms and the type casting relation. This is one step beyond Brandt and Henglein's work, where a coercion typing rule similar to our casting rules was introduced, but no results about the dynamic semantics were studied. Another contribution of our work is that with full iso-recursive types, we retain the computational structure of the terms when encoding equi-recursive types. Erasing the casting operators from the terms will result in the original terms, which is not the case for the previous work [Abadi and Fiore 1996]. Our casts are *computationally irrelevant.* Unlike regular iso-recursive types, which require computationally relevant term coercions for some type conversions, no such coercions are needed in our approach. For example, all the reduction steps in the example above are erased to the original term ($v\ 1$). This round-tripping property simplifies the correctness reasoning of the encoding.

We show that all the terms that are well-typed in the equi-recursive setting can be encoded in the full iso-recursive setting. Furthermore, the encoding is behavior preserving, i.e. evaluating the encoded terms will result in a value that is equal to the value of the original equi-recursive term up to erasure. In this sense, we get back a fully verified statement that *"full iso-recursive types have the same expressive power as equi-recursive types"*.

Subtyping. Our results extend to subtyping. Our main observation is that the equi-recursive subtyping relation can be defined by a combination of equi-recursive equality and the iso-recursive subtyping relation [Abadi and Cardelli 1996; Cardelli 1985; Zhou et al. 2022], as shown below:

$$A \leq_e B \triangleq \exists C_1\, C_2.\ (A \doteq C_1) \land (C_1 \leq_i C_2) \land (C_2 \doteq B)$$

Here $\leq_e$ denotes an equi-recursive subtyping relation, and $\leq_i$ denotes an iso-recursive subtyping relation. This alternative definition of equi-recursive subtyping is implicitly implied from Amadio and Cardelli's work, but it is somewhat hidden behind their proofs and definitions. We reinterpret their proofs and definitions to highlight that this alternative definition is equivalent to existing equi-recursive subtyping definitions in the literature. This alternative definition of equi-recursive subtyping is important because we can reuse the existing type casting relation in the full iso-recursive setting with subtyping. For example, given an equi-recursive term e that has the type A with $A \leq_e B$, we can encode $e : B$ in the full iso-recursive setting as $((\mathsf{cast}\ [c_2]\ (\mathsf{cast}\ [c_1]\ e')) : B)$, in which c_1 and c_2 are casts encoding the equality relation $A \doteq C_1$ and $C_2 \doteq B$, and e' is the encoding of $(e : A)$. This term type checks with the iso-recursive subtyping relation.

Our encoding is still computationally irrelevant in the presence of subtyping. Thus, all the results – including type soundness, well-typed encoding, and behavior preservation – are also applicable to the system with subtyping. This is a significant improvement over previous work [Abadi and Fiore 1996], which has not studied the relationship between equi-recursive and iso-recursive subtyping.

3 A Calculus with Full Iso-Recursive Types

In this section we will introduce a calculus with full iso-recursive types, called λ^{μ}_{Fi}. Our calculus is based on the simply typed lambda calculus extended with recursive types and type cast operators.

3.1 Syntax and Well-Formedness

The syntax of λ^{μ}_{Fi} is shown at the top of Figure 3.

Types. Meta-variables A, B range over types. Types include base types ($\mathtt{Int}$), function types ($A_1 \to A_2$), type variables (α), and recursive types ($\mu\alpha.\ A$).

Expressions. Meta-variables e range over expressions. Most of the expressions are standard, including: variables (x), integers (n), applications ($e_1\ e_2$) and lambda abstractions ($\lambda x : A.\ e$). We also have a type cast operator ($\mathsf{cast}\ [c]\ e$) that transforms the type of the expression e to an equivalent type using the cast operator c. The cast operators c include cast variables (ι), the identity cast (id), the fold and unfold casts (fold_A and unfold_A), the arrow cast ($c_1 \to c_2$), the sequential cast ($c_1; c_2$), and the fixpoint cast ($\mathsf{fix}\ \iota.\ c$). We will define the type cast rules for these operators in §3.2.

Values. Meta-variables v range over values. Integers (n) and lambda abstractions ($\lambda x : A.\ e$), are considered as values, which are standard for a simply typed lambda calculus. In standard iso-recursive types, the folding of a value is considered a value, since it serves as the canonical form of a recursive type (e.g. $\mathsf{fold}\ [\mu\alpha.\ A]\ v$ has the type $\mu\alpha.\ A$) and there are no further reduction rules for reducing $\mathsf{fold}\ [A]\ v$. Therefore, in our calculus the corresponding encoding ($\mathsf{cast}\ [\mathsf{fold}_A]\ v$)

Types	A, B	$::=$	$\mathsf{Int} \mid A_1 \to A_2 \mid \alpha \mid \mu\alpha.\, A$
Expressions	e	$::=$	$x \mid \mathsf{n} \mid e_1\ e_2 \mid \lambda x : A.\, e \mid \mathsf{cast}\,[c]\,e$
Values	v	$::=$	$\mathsf{n} \mid \lambda x : A.\, e \mid \mathsf{cast}\,[\mathsf{fold}_A]\,v \mid \mathsf{cast}\,[c_1 \to c_2]\,v$
Cast Operators	c	$::=$	$\iota \mid \mathsf{id} \mid \mathsf{fold}_A \mid \mathsf{unfold}_A \mid c_1 \to c_2 \mid c_1; c_2 \mid \mathsf{fix}\,\iota.\, c$
Type Contexts	Δ	$::=$	$\cdot \mid \Delta, \alpha$
Typing Contexts	Γ	$::=$	$\cdot \mid \Gamma, x : A$
Type Cast Contexts	$\mathbb{E}$	$::=$	$\cdot \mid \mathbb{E}, \iota : A \hookrightarrow B$

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

$$\textsc{WFT-int}\ \frac{}{\Delta \vdash \mathsf{Int}} \qquad \textsc{WFT-var}\ \frac{\alpha \in \Delta}{\Delta \vdash \alpha} \qquad \textsc{WFT-arrow}\ \frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \to B} \qquad \textsc{WFT-rec}\ \frac{\Delta, \alpha \vdash A}{\Delta \vdash \mu\alpha.\, A}$$

Fig. 3. Syntax and type well-formedness of λ^{μ}_{Fi}.

is also considered a value. We also consider arrow casts of a value ($\mathsf{cast}\,[c_1 \to c_2]\,v$) to be values, since they cannot be reduced further without applying them to an argument.

Contexts and Well-formedness. Type contexts Δ track bound type variables α. A type is well-formed if all of its free variables are in the context. Well-formedness for types is standard, and shown at the bottom Figure 3. Typing contexts Γ track bound variables x with their types. A typing context is well-formed ($\vdash \Gamma$) if there are no duplicate variables and all the types are well-formed. We also define a type cast context $\mathbb{E}$ to keep track of the cast variables ι and the cast operators that they are associated with. This will be used in the type cast rules, defined in §3.2. For type variables and term variables, we assume the usual notions of free and bound variables, and the usual capture-avoiding substitution function, denoted by $A[\alpha \mapsto B]$, that replaces the free occurrences of variable α in A by B, while avoiding the capture of any bound variable in A. When needed, we assume that α-equivalence is applied at will to avoid the clashing of free variables.

3.2 Typing

The top of Figure 4 shows the typing rules for λ^{μ}_{Fi}. Most rules are standard except for the typing rule for type casting (rule Typ-cast). This rule replaces the standard folding and unfolding rules for iso-recursive types, as we explained in §2.5. Rule Typ-cast relies on the type casting rules shown at the bottom of Figure 4. In the type casting judgment $\Delta; \mathbb{E} \vdash A \hookrightarrow B : c$, Δ is the type context used to ensure that all the types in the cast derivation are well-formed. $\mathbb{E}$ tracks of the cast variables ι that appear in c and the cast operator that they are associated with. New cast variables are introduced when a fixpoint cast is encountered, as shown in rule Cast-fix, which gives us the ability to encode the coinductive reasoning in equi-recursive equalities.

For instance, the equality $\mu\alpha.\, \mathsf{Int} \to \mathsf{Int} \to \alpha \doteq \mu\alpha.\, \mathsf{Int} \to \alpha$ that we have seen in §2.1 can be encoded as the following type casting relation:

$$\mu\alpha.\, \mathsf{Int} \to \mathsf{Int} \to \alpha \hookrightarrow \mu\alpha.\, \mathsf{Int} \to \alpha : \begin{array}{l} \mathsf{unfold}_{\mu\alpha.\, \mathsf{Int} \to \mathsf{Int} \to \alpha}; (\mathsf{fix}\,\iota.\, \mathsf{id} \to ((\mathsf{id} \to \\ \qquad (\mathsf{unfold}_{\mu\alpha.\, \mathsf{Int} \to \mathsf{Int} \to \alpha}; \iota; \mathsf{fold}_{\mu\alpha.\, \mathsf{Int} \to \alpha}) \\ ); \mathsf{fold}_{\mu\alpha.\, \mathsf{Int} \to \alpha})); \mathsf{fold}_{\mu\alpha.\, \mathsf{Int} \to \alpha} \end{array}$$

We show the derivation of this type casting relation in Figure 5. Essentially the coinductive "cyclic reasoning" of $\mathsf{Int} \to \mathsf{Int} \to A \doteq \mathsf{Int} \to B$ in the derivation is captured by the fixpoint cast operator ($\mathsf{fix}\,\iota.\, \mathsf{id} \to c_1$), as rule Cast-fix shows. Note that due to the complexity of equi-recursive equalities, the cast operator can sometimes be quite lengthy, as in this example. But as we will

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

$$\textsc{Typ-int}\quad \frac{\vdash \Gamma}{\Gamma \vdash n : \mathsf{Int}} \qquad \textsc{Typ-var}\quad \frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Typ-abs}\quad \frac{\Gamma, x : A_1 \vdash e : A_2}{\Gamma \vdash \lambda x : A_1.\, e : A_1 \to A_2}$$

$$\textsc{Typ-app}\quad \frac{\Gamma \vdash e_1 : A_1 \to A_2 \qquad \Gamma \vdash e_2 : A_1}{\Gamma \vdash e_1\, e_2 : A_2} \qquad \textsc{Typ-cast}\quad \frac{\Gamma \vdash e : A \qquad \cdot;\cdot \vdash A \hookrightarrow B : c}{\Gamma \vdash \mathsf{cast}\,[c]\; e : B}$$

$\boxed{\Delta; \mathbb{E} \vdash A \hookrightarrow B : c}$ *(Type Casting)*

$$\textsc{Cast-id}\quad \frac{\Delta \vdash A \qquad \vdash \mathbb{E}}{\Delta; \mathbb{E} \vdash A \hookrightarrow A : \mathsf{id}} \qquad \textsc{Cast-arrow}\quad \frac{\Delta; \mathbb{E} \vdash A_1 \hookrightarrow A_2 : c_1 \qquad \Delta; \mathbb{E} \vdash B_1 \hookrightarrow B_2 : c_2}{\Delta; \mathbb{E} \vdash A_1 \to B_1 \hookrightarrow A_2 \to B_2 : c_1 \to c_2}$$

$$\textsc{Cast-unfold}\quad \frac{\Delta \vdash \mu\alpha.\, A \qquad \vdash \mathbb{E}}{\Delta; \mathbb{E} \vdash \mu\alpha.\, A \hookrightarrow A[\alpha \mapsto \mu\alpha.\, A] : \mathsf{unfold}_{\mu\alpha.\, A}} \qquad \textsc{Cast-fold}\quad \frac{\Delta \vdash \mu\alpha.\, A \qquad \vdash \mathbb{E}}{\Delta; \mathbb{E} \vdash A[\alpha \mapsto \mu\alpha.\, A] \hookrightarrow \mu\alpha.\, A : \mathsf{fold}_{\mu\alpha.\, A}}$$

$$\textsc{Cast-seq}\quad \frac{\Delta; \mathbb{E} \vdash A \hookrightarrow B : c_1 \qquad \Delta; \mathbb{E} \vdash B \hookrightarrow C : c_2}{\Delta; \mathbb{E} \vdash A \hookrightarrow C : c_1; c_2} \qquad \textsc{Cast-var}\quad \frac{\Delta \vdash A \qquad \Delta \vdash B \qquad \vdash \mathbb{E} \qquad \iota : A \hookrightarrow B \in \mathbb{E}}{\Delta; \mathbb{E} \vdash A \hookrightarrow B : \iota}$$

$$\textsc{Cast-fix}\quad \frac{\Delta; \mathbb{E}, \iota : A_1 \to B_1 \hookrightarrow A_2 \to B_2 \vdash A_1 \hookrightarrow A_2 : c_1 \qquad \Delta; \mathbb{E}, \iota : A_1 \to B_1 \hookrightarrow A_2 \to B_2 \vdash B_1 \hookrightarrow B_2 : c_2}{\Delta; \mathbb{E} \vdash A_1 \to B_1 \hookrightarrow A_2 \to B_2 : \mathsf{fix}\, \iota.\, (c_1 \to c_2)}$$

Fig. 4. Typing and type cast rules for λ^{μ}_{Fi}.

show, this is not a runtime burden because the cast operators can be erased during runtime without affecting the behavior of the program.

Our type casting rules, ignoring the cast variables and operators, are very similar to the type equality rules in Brandt and Henglein's axiomatization of type equality. Despite some subtle differences, which we will discuss in §4.2, our type casting rules are sound and complete with respect to their type equality rules.

Theorem 3.1 (Soundness and completeness of type casting). For any types A and B, $\cdot \vdash A \doteq B$ if and only if there exists a cast operator c such that $\cdot;\cdot \vdash A \hookrightarrow B : c$.

Equivalence to a calculus with equi-recursive typing. The only difference between the equi-recursive typing rules and λ^{μ}_{Fi}'s typing rules is replacing type casting in rule Typ-cast with a type equality relation. Figure 6 shows an alternative definition of the equi-recursive typing rules. The gray parts are used to generate a term in λ^{μ}_{Fi}, and can be ignored for understanding the equi-recursive typing rules. The standard equi-recursive type equality relation in rule Typ-eq is replaced by the type casting relation in rule ETyp-eq. Since the two relations are equivalent by Theorem 3.1, the typing rules in Figure 6 are equivalent to the standard equi-recursive typing rules.

$$\dfrac{\overset{\textsc{Cast-id}}{\mathbb{E} \vdash \mathtt{Int} \hookrightarrow \mathtt{Int} : \mathsf{id}} \qquad \dfrac{\dfrac{\iota : C \hookrightarrow D \in \mathbb{E}}{\mathbb{E} \vdash C \hookrightarrow D : \iota}\ \textsc{Cast-var}}{\mathbb{E} \vdash A \hookrightarrow B : \mathsf{unfold}_A; \iota; \mathsf{fold}_B}\ \textsc{Cast-unfold}, \textsc{Cast-seq} \text{ and } \textsc{Cast-fold}}{\dfrac{\mathbb{E} \vdash \mathtt{Int} \to A \hookrightarrow \mathtt{Int} \to B : \mathsf{id} \to \mathsf{unfold}_A; \iota; \mathsf{fold}_B}{\mathbb{E} \vdash \mathtt{Int} \to A \hookrightarrow B : (\mathsf{id} \to \mathsf{unfold}_A; \iota; \mathsf{fold}_B); \mathsf{fold}_B}\ \textsc{Cast-seq}, \textsc{Cast-fold}}\ \textsc{Cast-arrow}$$

$$\dfrac{\dfrac{\dfrac{\overset{\textsc{Cast-id}}{\mathbb{E} \vdash \mathtt{Int} \hookrightarrow \mathtt{Int} : \mathsf{id}} \qquad \dfrac{\cdots}{\mathbb{E} \vdash \mathtt{Int} \to A \hookrightarrow B : c_1}}{\mathbb{E} \vdash \mathtt{Int} \to \mathtt{Int} \to A \hookrightarrow \mathtt{Int} \to B : \mathsf{id} \to c_1}\ \textsc{Cast-arrow}}{\cdot \vdash \mathtt{Int} \to \mathtt{Int} \to A \hookrightarrow \mathtt{Int} \to B : \mathsf{fix}\,\iota.\ \mathsf{id} \to c_1}\ \textsc{Cast-fix}}{\cdot \vdash A \hookrightarrow B : c}\ \textsc{Cast-unfold}, \textsc{Cast-seq} \text{ and } \textsc{Cast-fold}$$

$$\begin{array}{ll}
A = \mu\alpha.\ \mathtt{Int} \to \mathtt{Int} \to \alpha & B = \mu\alpha.\ \mathtt{Int} \to \alpha \\
C = \mathtt{Int} \to \mathtt{Int} \to A & D = \mathtt{Int} \to B \\
c = \mathsf{unfold}_A; (\mathsf{fix}\,\iota.\ \mathsf{id} \to c_1); \mathsf{fold}_B & c_1 = (\mathsf{id} \to \mathsf{unfold}_A; \iota; \mathsf{fold}_B); \mathsf{fold}_B \\
\mathbb{E} = \iota : C \hookrightarrow D &
\end{array}$$

Fig. 5. A derivation of type casting with fixpoint casts (well-formedness conditions omitted).

$\boxed{\Gamma \vdash_e e : A \ \rhd e'}$ *(Equi-recursive typing and full iso-recursive elaboration)*

$$\textsc{ETyp-int}\ \dfrac{\vdash \Gamma}{\Gamma \vdash_e n : \mathtt{Int} \ \rhd n} \qquad \textsc{ETyp-var}\ \dfrac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash_e x : A \ \rhd x} \qquad \textsc{ETyp-abs}\ \dfrac{\Gamma, x : A_1 \vdash_e e : A_2 \ \rhd e'}{\Gamma \vdash_e \lambda x : A_1.\, e : A_1 \to A_2 \ \rhd \lambda x : A_1.\, e'}$$

$$\textsc{ETyp-app}\ \dfrac{\Gamma \vdash_e e_1 : A_1 \to A_2 \ \rhd e_1' \qquad \Gamma \vdash_e e_2 : A_1 \ \rhd e_2'}{\Gamma \vdash_e e_1\, e_2 : A_2 \ \rhd e_1'\, e_2'} \qquad \textsc{ETyp-eq}\ \dfrac{\Gamma \vdash_e e : A \ \rhd e' \qquad \cdot; \cdot \vdash A \hookrightarrow B : c}{\Gamma \vdash_e e : B \ \rhd \mathsf{cast}\,[c]\; e'}$$

Fig. 6. An equivalent equi-recursive typing system and elaboration rules to λ^{μ}_{Fi}.

Theorem 3.2 (Equivalence of alternative equi-recursive typing). For any expression e and type A, $\Gamma \vdash_e e : A$ in the standard equi-recursive typing with rule Typ-eq if and only if there exists a full iso-recursive term e' such that $\Gamma \vdash_e e : A \rhd e'$ using the rules in Figure 6.

Our alternative formulation of equi-recursive typing also provides a way to elaborate equi-recursive terms into full iso-recursive terms, as shown by the gray colored parts in Figure 6. The elaboration is type-directed, by inserting the appropriate casts where a type equality is needed following the typing derivation of the equi-recursive terms. The interesting point here is that, by replacing Brandt and Henglein's type equality relation with our type casting relation, we obtain a cast c, which can be viewed as evidence of the type transformation from type A into type B. Then, we use c in an explicit cast in the elaborated λ^{μ}_{Fi} term, which will trigger the type transformation in λ^{μ}_{Fi}. Every well-typed equi-recursive term can be elaborated into a full iso-recursive term, and every full iso-recursive term can be erased to an equi-recursive term that has the same type. It follows that the full iso-recursive typing rules are sound and complete with respect to equi-recursive types:

$\boxed{e \hookrightarrow e'}$ *(Reduction)*

$$\textsc{Red-beta}\ \frac{}{(\lambda x : A.\, e)\, v' \hookrightarrow e[x \mapsto v']} \qquad \textsc{Red-appl}\ \frac{e_1 \hookrightarrow e_1'}{e_1\, e_2 \hookrightarrow e_1'\, e_2} \qquad \textsc{Red-appr}\ \frac{e_2 \hookrightarrow e_2'}{v_1\, e_2 \hookrightarrow v_1\, e_2'} \qquad \textsc{Red-cast}\ \frac{e \hookrightarrow e'}{\mathsf{cast}\,[c]\, e \hookrightarrow \mathsf{cast}\,[c]\, e'}$$

$$\textsc{Red-cast-id}\ \frac{}{\mathsf{cast}\,[\mathsf{id}]\, v \hookrightarrow v} \qquad \textsc{Red-cast-arr}\ \frac{}{(\mathsf{cast}\,[c_1 \to c_2]\, v_1)\, v_2 \hookrightarrow \mathsf{cast}\,[c_2]\, (v_1\, (\mathsf{cast}\,[\neg c_1]\, v_2))}$$

$$\textsc{Red-cast-seq}\ \frac{}{\mathsf{cast}\,[c_1; c_2]\, v \hookrightarrow \mathsf{cast}\,[c_2]\, (\mathsf{cast}\,[c_1]\, v)} \qquad \textsc{Red-castelim}\ \frac{}{\mathsf{cast}\,[\mathsf{unfold}_A]\, (\mathsf{cast}\,[\mathsf{fold}_B]\, v) \hookrightarrow v}$$

$$\textsc{Red-cast-fix}\ \frac{}{\mathsf{cast}\,[\mathsf{fix}\,\iota.\, c]\, v \hookrightarrow \mathsf{cast}\,[c[\iota \mapsto \mathsf{fix}\,\iota.\, c]]\, v}$$

Fig. 7. Reduction rules.

Theorem 3.3 (Equi-recursive to full iso-recursive typing). For any expressions e, e' and type A, if $\Gamma \vdash_e e : A \rhd e'$ then $\Gamma \vdash e' : A$.

Theorem 3.4 (Full iso-recursive to equi-recursive typing). For any expressions e and type A, if $\Gamma \vdash e : A$ then $\Gamma \vdash_e |e| : A \rhd e$.

In the theorem above, the full iso-recursive expressions can be erased to the equi-recursive expressions by removing the casts. The erasure operation $|e|$ is defined as follows:

$$\begin{array}{llll} |\mathsf{n}| & = \mathsf{n} & |x| & = x \\ |e_1\, e_2| & = |e_1|\, |e_2| & |\lambda x : A.\, e| & = \lambda x : A.\, |e| \\ |\mathsf{cast}\,[c]\, e| & = |e| & & \end{array}$$

Moreover, our elaboration achieves the round-tripping property – elaborating an equi-recursive term into a full iso-recursive term and then erasing the casts will get back the original equi-recursive term. This is not the case for previous work in relating recursive types [Abadi and Fiore 1996], in which computationally relevant coercions are inserted as term-level functions and erasing the unfold/fold annotations do not recover the original term. The round-tripping property is crucial for a simple proof of the behavioral equivalence between the two systems, which we discuss next.

Theorem 3.5 (Round-tripping of the encoding). For any expression e, e' and type A, if $\Gamma \vdash_e e : A \rhd e'$, then $|e'| = e$.

3.3 Semantics

Figure 7 shows the reduction rules for λ^{μ}_{Fi}. In addition to the standard reduction rules for the simply typed lambda calculus, we add the reduction rules for the cast operators. Our reduction rules are call-by-value. The inner expressions of the cast operators are reduced first (rule Red-cast). Then, based on different cast operators, the cast operator is pushed into the expression in various ways. For identity casts, the cast operator is simply erased (rule Red-cast-id). Arrow casts are values, but when they are applied to an argument, the cast operator is pushed into the argument (rule Red-cast-arr). Note that the cast operator needs to be reversed when pushed into the function argument in order to ensure type preservation after the reduction. The reverse operation is defined by analyzing the structure of c as follows:

$$\begin{array}{llll} \neg\,\iota & = \iota & \neg\,\mathsf{id} & = \mathsf{id} \\ \neg\,\mathsf{fold}_A & = \mathsf{unfold}_A & \neg\,\mathsf{unfold}_A & = \mathsf{fold}_A \\ \neg\,(c_1 \rightarrow c_2) & = (\neg\,c_1) \rightarrow (\neg\,c_2) & \neg\,(c_1; c_2) & = (\neg\,c_2); (\neg\,c_1) \\ \neg\,(\mathsf{fix}\,\iota.\ c) & = \mathsf{fix}\,\iota.\ \neg\,c & & \end{array}$$

We will see why this reverse operation is necessary in §4.1. A single sequential cast is split into two separate casts (rule Red-cast-seq), so that the sub-components can be reduced independently. Fold casts are values, but can be eliminated by an outer unfold cast (rule Red-castelim). Thus, rule Red-castelim corresponds to the traditional fold/unfold cancellation rule used in calculi with conventional iso-recursive types. Finally, fixpoint casts are reduced by unrolling the fixpoint (rule Red-cast-fix). The addition of the push rules for the cast operators is necessary for the type soundness of λ^{μ}_{Fi}, since the cast rules are necessary to preserve types. λ^{μ}_{Fi} is type sound, proved with the usual preservation and progress theorems:

Theorem 3.6 (Progress). For any expression e and type A, if $\cdot \vdash e : A$ then either e is a value or there exists an expression e' such that $e \hookrightarrow e'$.

Theorem 3.7 (Preservation). For any expression e and type A, if $\cdot \vdash e : A$ and $e \hookrightarrow e'$ then $\cdot \vdash e' : A$.

A running example. To illustrate how the reduction rules work in the λ^{μ}_{Fi} calculus, we present a relatively complex example that involves fixpoint casts. Assume that we have a term e with the type $\mu\alpha.\ \mathtt{Int} \rightarrow \mathtt{Int} \rightarrow \alpha$, which repeatedly takes two integers at a time and returns itself. With term-level recursion constructors (which can be easily added to λ^{μ}_{Fi} or encoded by fold/unfold casts [Abadi and Fiore 1996]), denoted as rec $(x : A).\ e$, such a term can be defined as follows:

$$e = \mathsf{rec}\ (\mathit{self} : \mu\alpha.\ \mathtt{Int} \rightarrow \mathtt{Int} \rightarrow \alpha).\ \mathsf{cast}\ [\mathsf{fold}_{\mu\alpha.\ \mathtt{Int}\rightarrow\mathtt{Int}\rightarrow\alpha}]\ (\lambda(x : \mathtt{Int}).\ \lambda(y : \mathtt{Int}).\ \mathit{self})$$

Now we would like to apply this term to a single integer argument. Using the cast operator c that we have defined in Figure 5, we can write $(\mathsf{cast}\ [\mathsf{unfold}_{\mu\alpha.\ \mathtt{Int}\rightarrow\alpha}]\ (\mathsf{cast}\ [c]\ e))\ 1$. This expression first turns this term into a type $\mu\alpha.\ \mathtt{Int} \rightarrow \alpha$, and then applies the term after iso-recursive unfolding to an integer 1. In Figure 8 we try to evaluate this term. The reduction rules are applied step by step. The push rules Red-cast-seq and Red-cast-arr reduce the cast operator into more atomic forms, and the unfold/fold cast pairs are eventually eliminated by rule Red-castelim in various steps. Notably, the rule Red-cast-fix for fixpoint casts may introduce a larger cast operator after substitution during runtime. However, when combined with other push rules, the resulting term can be safely reduced. In our case, the term reduces to a value, since the result is a fold of arrow casts and belongs to the value syntax.

Equivalence to the equi-recursive dynamic semantics. As explained in §2.5, the reduction rules for the cast operators are computationally irrelevant. Therefore, they can be erased from expressions without affecting the behavior of the expressions. We can obtain the following theorem easily:

Theorem 3.8 (Full iso-recursive to equi-recursive behavioral preservation). For any expression e, if $e \hookrightarrow^* v$ then $|e| \hookrightarrow^*_e |v|$.

Here $\hookrightarrow_e$ is the reduction relation in the equi-recursive setting, which is basically defined by a subset of the reduction rules in Figure 7 (rules Red-beta, Red-appl, and Red-appr). We use $\hookrightarrow^*$ to denote the reflexive, transitive closure of the reduction relation. Theorem 3.8 can be further illustrated by the example in Figure 8. It can be seen that most of the steps related to the cast operators are simply for the sake of typing correctness, and all the cast operators in the reductions

$$\begin{array}{lll}
 & (\mathsf{cast}\,[\mathsf{unfold}_B]\,(\mathsf{cast}\,[c]\,\underline{e}))\;1 & \\
\hookrightarrow & (\mathsf{cast}\,[\mathsf{unfold}_B]\,(\mathsf{cast}\,[\underline{c}]\,(\mathsf{cast}\,[\mathsf{fold}_A]\,(\lambda(x\;y:\mathtt{Int}).\,e))))\;1 & \text{(unrolling rec)} \\
\hookrightarrow & (\mathsf{cast}\,[\mathsf{unfold}_B]\,(\mathsf{cast}\,[\underline{\mathsf{unfold}_A;}\,\mathsf{fix}\,\iota.\,\mathsf{id}\to c_1;\mathsf{fold}_B] & \\
 & \quad (\mathsf{cast}\,[\mathsf{fold}_A]\,(\lambda(x\;y:\mathtt{Int}).\,e))))\;1 & \text{(definition of } c\text{)} \\
\hookrightarrow & (\mathsf{cast}\,[\mathsf{unfold}_B]\,(\mathsf{cast}\,[(\mathsf{fix}\,\iota.\,\mathsf{id}\to c_1);\mathsf{fold}_B] & \\
 & \quad \underline{(\mathsf{cast}\,[\mathsf{unfold}_A]\,(\mathsf{cast}\,[\mathsf{fold}_A]}\,(\lambda(x\;y:\mathtt{Int}).\,e)))))\;1 & (\textsc{Red-cast-seq}) \\
\hookrightarrow & (\mathsf{cast}\,[\mathsf{unfold}_B]\,(\mathsf{cast}\,[(\mathsf{fix}\,\iota.\,\mathsf{id}\to c_1)\,\underline{;\mathsf{fold}_B}]\,(\lambda(x\;y:\mathtt{Int}).\,e)))\;1 & (\textsc{Red-castelim}) \\
\hookrightarrow & \underline{(\mathsf{cast}\,[\mathsf{unfold}_B]\,(\mathsf{cast}\,[\mathsf{fold}_B]} & \\
 & \quad (\mathsf{cast}\,[\mathsf{fix}\,\iota.\,\mathsf{id}\to c_1]\,(\lambda(x\;y:\mathtt{Int}).\,e))))\;1 & (\textsc{Red-cast-seq}) \\
\hookrightarrow & (\mathsf{cast}\,[\underline{\mathsf{fix}\,\iota.\,\mathsf{id}\to c_1}]\,(\lambda(x\;y:\mathtt{Int}).\,e))\;1 & (\textsc{Red-castelim}) \\
\hookrightarrow & (\mathsf{cast}\,[\mathsf{id}\to \underline{c_1[\iota\mapsto c]}]\,(\lambda(x\;y:\mathtt{Int}).\,e))\;1 & (\textsc{Red-cast-fix}) \\
\hookrightarrow & (\mathsf{cast}\,[\underline{\mathsf{id}\to((\mathsf{id}\to\mathsf{unfold}_A;c;\mathsf{fold}_B);\mathsf{fold}_B)}]\,(\lambda(x\;y:\mathtt{Int}).\,e))\;\underline{1} & \text{(substitution)} \\
\hookrightarrow & (\mathsf{cast}\,[(\mathsf{id}\to\mathsf{unfold}_A;c;\mathsf{fold}_B);\mathsf{fold}_B] & \\
 & \quad ((\lambda(x\;y:\mathtt{Int}).\,e)\;\underline{(\mathsf{cast}\,[\neg\,\mathsf{id}]\;1)})) & (\textsc{Red-cast-arr}) \\
\hookrightarrow & (\mathsf{cast}\,[(\mathsf{id}\to\mathsf{unfold}_A;c;\mathsf{fold}_B);\mathsf{fold}_B] & \\
 & \quad \underline{(\lambda(x\;y:\mathtt{Int}).\,e\;1)}) & (\textsc{Red-cast-id}) \\
\hookrightarrow & \mathsf{cast}\,[(\mathsf{id}\to\mathsf{unfold}_A;c;\mathsf{fold}_B)\underline{;\mathsf{fold}_B}]\,(\lambda(y:\mathtt{Int}).\,e) & (\textsc{Red-beta}) \\
\hookrightarrow & \mathsf{cast}\,[\mathsf{fold}_B]\,(\mathsf{cast}\,[\mathsf{id}\to\mathsf{unfold}_A;c;\mathsf{fold}_B]\,(\lambda(y:\mathtt{Int}).\,e)) & (\textsc{Red-cast-seq})
\end{array}$$

Fig. 8. Small step execution trace of a λ^{μ}_{Fi} program (assuming same abbreviations as in Figure 5).

can be erased without affecting the behavior of the program. If we erase the cast operators during runtime, the trace in Figure 8 simply boils down to

$$e\;1 \hookrightarrow (\lambda(x\;y:\mathtt{Int}).\,e)\;1 \hookrightarrow \lambda(y:\mathtt{Int}).\,e$$

The other direction of the behavioral preservation also holds, but only applies to well-typed expressions and relies on the elaboration process defined in Figure 6. The proof of this direction is also more involved, and we will detail it in §4.3. To summarize, the two systems are behaviorally equivalent, in terms of both termination and divergence behavior:

Theorem 3.9 (Behavioral equivalence). For any expression e, e' and type A, if $\cdot \vdash_e e : A \rhd e'$, then

(1) $e \hookrightarrow^*_e v$ if and only if there exists v' such that $e' \hookrightarrow^* v'$ and $|v'| = v$.
(2) e diverges if and only if e' diverges.

4 Metatheory of Full Iso-Recursive Types

In this section we discuss the key proof techniques and results in the metatheory of λ^{μ}_{Fi}. The metatheory covers three components: type soundness of λ^{μ}_{Fi} (Theorem 3.6 and 3.7), the typing equivalence between λ^{μ}_{Fi} and equi-recursive types (Theorem 3.2, 3.3 and 3.4), and the behavioral equivalence between λ^{μ}_{Fi} and equi-recursive types (Theorem 3.9).

4.1 Type Soundness

Progress. For λ^{μ}_{Fi} we need to ensure that the definition of value and the reduction rules. In particular, the push rules for type casting, are complementary to each other, i.e. a cast expression is either a value or can be further reduced. The definition of value has been discussed in §3.1. There are two canonical forms for a value with function types ($A_1 \to A_2$) in λ^{μ}_{Fi}: lambda abstractions ($\lambda x : A.\, e$) and arrow casts ($\mathsf{cast}\,[c_1 \to c_2]\,v$). Therefore in the progress proof for function

$$\begin{tikzcd}
v_1 : B_1 \arrow[r] \arrow[d, hook, shift left, "c_1"] & B_2 \arrow[d, hook, "c_2"] \\
v_2 : A_1 \arrow[u, hook, shift left, "\neg c_1"] & A_2
\end{tikzcd}$$

$$\dfrac{\dfrac{v_1 : B_1 \to B_2 \qquad \dfrac{B_1 \hookrightarrow A_1 : c_1 \qquad B_2 \hookrightarrow A_2 : c_2}{B_1 \to B_2 \hookrightarrow A_1 \to A_2 : c_1 \to c_2}\,\textsc{Cast-arrow}}{\mathsf{cast}\,[c_1 \to c_2]\; v_1 : A_1 \to A_2}\,\textsc{Typ-cast} \qquad v_2 : A_1}{(\mathsf{cast}\,[c_1 \to c_2]\; v_1)\; v_2 : A_2}\,\textsc{Typ-app}$$

$$\dfrac{\dfrac{v_1 : B_1 \to B_2 \qquad \dfrac{v_2 : A_1 \qquad \dfrac{B_1 \hookrightarrow A_1 : c_1}{A_1 \hookrightarrow B_1 : \neg c_1}\,\text{Lemma 4.1}}{\mathsf{cast}\,[\neg c_1]\; v_2 : B_1}\,\textsc{Typ-cast}}{v_1\; (\mathsf{cast}\,[\neg c_1]\; v_2) : B_2}\,\textsc{Typ-app} \qquad B_2 \hookrightarrow A_2 : c_2}{\mathsf{cast}\,[c_2]\; (v_1\; (\mathsf{cast}\,[\neg c_1]\; v_2)) : A_2}\,\textsc{Typ-cast}$$

Fig. 9. Typing derivation for the function push rule (empty environments omitted).

applications ($e_1\ e_2$), we need to consider one extra case when e_1 is an arrow cast. We push the cast operator further by rule Red-cast-arr as a reduction step in this case to complete the proof.

Preservation. The preservation proof is standard by first doing induction on the typing derivation $\cdot \vdash e : A$ and then induction on the reduction relation $e \hookrightarrow e'$. The interesting cases are when the reduction rule is a push rule. Most cases of the push rules are straightforward, by inversion on the type casting relation and then reconstructing the casting derivation for the reduced expression. Two tricky cases require extra care: the push rules for arrow cast (rule Red-cast-arr) and the fixpoint cast (rule Red-cast-fix).

In the Red-cast-arr case, we illustrate the typing derivation for the two terms in the reduction rule in Figure 9. We show the type casting relation (using the $\hookrightarrow$ arrow) between the types of the two principal terms v_1 and v_2 in the left part of the figure, and the detailed typing derivation steps for the terms before and after the reduction are shown in the right part. By inversion on the typing derivation of $\cdot \vdash (\mathsf{cast}\,[c_1 \to c_2]\, v_1)\ v_2 : A_2$ we know that v_1 has a function type $B_1 \to B_2$, v_2 has type A_1, and $B_1 \to B_2$ can be cast to $A_1 \to A_2$ by $c_1 \to c_2$, which in turn implies that $B_1 \hookrightarrow A_1 : c_1$ and $B_2 \hookrightarrow A_2 : c_2$. In order to show that the type of the reduced expression $(\mathsf{cast}\,[c_2]\ (v_1\ (\mathsf{cast}\,[\neg c_1]\, v_2)))$ is still A_2, we need to prove $A_1 \hookrightarrow B_1 : \neg c_1$. This goal can be achieved by Lemma 4.1 below, which is proved by induction on the type casting relation. The analysis of rule Red-cast-arr also shows that it is necessary to insert the reverse operation on the cast operator c_1 to ensure the preservation of λ^{μ}_{Fi}.

Lemma 4.1 (Reverse of casting). For any types A and B, and casting operators c, if $\cdot; \mathbb{E} \vdash A \hookrightarrow B : c$ then $\cdot; \neg\,\mathbb{E} \vdash B \hookrightarrow A : \neg\, c$, where $\neg\,\mathbb{E}$ reverses the direction of casting in $\mathbb{E}$ (i.e. $\iota : C \hookrightarrow D$ in $\mathbb{E}$ becomes $\iota : D \hookrightarrow C$ after reversing).

As for the Red-cast-fix case, the reduction rule unfolds the fixpoint cast $(\mathsf{fix}\,\iota.\ c)$ to $(c[\iota \mapsto (\mathsf{fix}\,\iota.\ c)])$. By inversion on the type casting relation $\cdot; \cdot \vdash A \hookrightarrow B : \mathsf{fix}\,\iota.\ c$, we know that

$$\cdot; \iota : A \hookrightarrow B \vdash A \hookrightarrow B : c \tag{2}$$

Essentially the cast operator $(\mathsf{fix}\,\iota.\ c)$ and its unrolling $(c[\iota \mapsto (\mathsf{fix}\,\iota.\ c)])$ should represent the same proof. The type casting judgement (2) can be interpreted as: if we know that there is a cast variable ι that can cast A to B, then we can cast A to B by c, using the cast variable ι. Since we already know that $\mathsf{fix}\,\iota.\ c$ can do the same job as ι in casting A to B, it should be safe to replace ι with $\mathsf{fix}\,\iota.\ c$ in the cast operator c, and show that $\cdot; \cdot \vdash A \hookrightarrow B : c[\iota \mapsto (\mathsf{fix}\,\iota.\ c)]$. This idea can be formalized by the following cast substitution lemma, which is proved by induction on the type casting relation of c_1.

Lemma 4.2 (Cast substitution). For any contexts $\Gamma, \mathbb{E}$, types A, B, C, D, cast operators c_1, c_2 and cast variable ι, if $\Gamma; \mathbb{E} \vdash A \hookrightarrow B : c_1$, and $\Gamma; \mathbb{E}, \iota : A \hookrightarrow B \vdash C \hookrightarrow D : c_2$ then $\Gamma; \mathbb{E} \vdash C \hookrightarrow D : c_2[\iota \mapsto c_1]$.

4.2 Typing Equivalence

As discussed in §3.2, the key to the typing equivalence between full iso-recursive types and equi-recursive types is to show our type casting rules are equivalent to Brandt and Henglein's type equality rules (Theorem 3.1). This section focuses on the proof of this theorem.

Most of our type casting rules, ignoring the cast variables and operators, are very similar to their type equality rules, so the proof for these cases is straightforward. For instance, the treatment of coinductive reasoning by introducing new premises for function types in our rule Cast-fix is exactly the same treatment as their rule Tye-arrfix. We discuss the only two differences below.

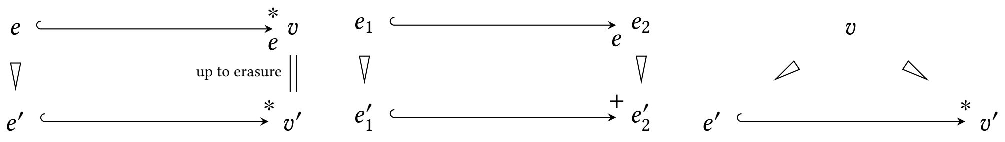
Arrow cast for type soundness. In addition to transforming function types with a fixpoint cast using rule Cast-fix as Brandt and Henglein did in rule Tye-arrfix, we also allow function types to be cast without a fixpoint cast as well, as shown in rule Cast-arr. This is a harmless extension, since one can always wrap an arrow cast with a dummy fixpoint, which does not use the fixpoint variable in the body. However, having this rule is essential to the type soundness of λ^{μ}_{Fi}. By rule Cast-fix, all the fixpoint casts in well-typed expressions are in the form of $\mathsf{fix}\, \iota.\ c_1 \to c_2$. During the reduction, we need to unroll those fixpoint casts using rule Red-cast-fix to a bare arrow cast in the form of $c'_1 \to c'_2$, which cannot be typed without the rule Cast-arr. In other words, while casts of arrows are values, casts of fixpoints are not values. Due to this difference we separate the two rules and prove that the extension does not affect the soundness and completeness of our type casting rules.

Removing the symmetry rule from equality. The other difference is that our rules do not include a symmetry rule for type casting. The main reason for this design choice is that we would like to have a simple treatment of reversing cast variables (i.e. $\neg\, \iota = \iota$ without considering contexts $\mathbb{E}$) while avoiding the overhead of reversing the whole environment ($\neg\, \mathbb{E}$) frequently in the typing derivation. In our current design, the reverse operation is only needed for cast operators in the reduction rule Red-cast-arr and absent in the type casting rules. Since the casts are erasable, this reverse operation will cause no runtime overhead. Alternative design choices, such as adding a symmetry operator in the type casting rules, or turning the function casting rule into a contravariant one, would all require reversing the whole environment for the sake of soundness.

$$\frac{\Delta; \mathbb{E} \vdash A \hookrightarrow B : c}{\Delta; \neg\, \mathbb{E} \vdash B \hookrightarrow A : \mathsf{sym}\ c}\ \textsc{AltCast-symm} \qquad \frac{\Delta; \neg\, \mathbb{E} \vdash A_2 \hookrightarrow A_1 : c_1}{\Delta; \mathbb{E} \vdash A_1 \hookrightarrow A_2 : c_1}\ \textsc{AltCast-arrow}$$

Although by setting $\mathbb{E}$ in Lemma 4.1 to be empty, we can show that symmetry is admissible in our type casting relation, when the initial environment is empty, there is still a missing step to prove the equivalence between our type casting rules and Brandt and Henglein's type equality rules. Note that changing the type casting context from $\neg\, \mathbb{E}$ in Lemma 4.1 to $\mathbb{E}$ as rule Tye-symm specifies will not work. In other words, rule Tye-symm, in its general form, with the same list of assumptions H in both the premise and conclusion, is not admissible in our type casting relation. The reason is that invalid assumptions may exist in the list H, which are not derivable by the type casting rules. For example, $\mathtt{Int} \doteq \mathtt{Int} \to \mathtt{Int} \vdash \mathtt{Int} \to \mathtt{Int} \doteq \mathtt{Int}$ is a valid judgement using rules Tye-symm and Tye-assump, but cannot be derived from our type casting rules.

Nevertheless we can still prove that our system is complete to Brandt and Henglein's equality, when the initial environment is empty. The idea is that starting from an empty assumption list, one can always replace the use of rule Tye-symm in the derivation with a complete derivation that redoes the proof goal in a symmetric way to obtain a derivation without using rule Tye-symm. The replacement is feasible since when the initial environment is empty, all the type equalities



(a) The "only if" direction. (b) Lemma 4.3. (c) Lemma 4.4.

Fig. 10. Illustration of the proof idea for behavioral equivalence.

introduced to the environment are guaranteed to be derivable from an empty assumption list by the type casting rules. Interested readers can refer to our Coq formalization for the details of the proof.

4.3 Behavioral Equivalence

To prove Theorem 3.9 it suffices to show one of the two propositions in the theorem. Since the type soundness of λ^{μ}_{Fi} ensures that a well-typed term does not get stuck – it can either diverge or reduce to a value, we only need to show the preservation of termination behavior and the preservation of divergence behavior can then be proved by contradiction. The "if" direction of the theorem (from full-iso recursive reduction to equi-recursive reduction) is easy, directly from the behavioral preservation property of the erasure function (Theorem 3.8). The "only if" direction (from equi-recursive reduction to full-iso recursive reduction) is more involved, and we illustrate the proof idea in Figure 10.

Basically we would like to show that if an equi-recursive term e can be reduced to v, then its corresponding full iso-recursive term e' can also be reduced to a value v', as illustrated in Figure 10a. To prove this, we first show that every step of equi-recursive reduction can be simulated by several steps of full iso-recursive reduction (Lemma 4.3), as shown in Figure 10b. Moreover, the encoding of an equi-recursive value might not be a value in the full iso-recursive setting, since the encoding result may contain casts that are not values. Therefore we also need to show that the encoding of an equi-recursive value can be further reduced to a value in the full iso-recursive setting, and still preserves the elaboration relation (Lemma 4.4), as shown in Figure 10c. By induction on the length of the equi-recursive reduction steps in Figure 10a, we complete the proof for Theorem 3.9.

Lemma 4.3 (Simulation of equi-recursive reduction). For any expressions e_1, e_1', e_2 and type A, if $\cdot \vdash_e e_1 : A \rhd e_1'$ and $e_1 \hookrightarrow_e e_2$, then there exists e_2' such that $e_1' \hookrightarrow^+ e_2'$ and $\cdot \vdash_e e_2 : A \rhd e_2'$.

Lemma 4.4 (Reductions of full iso-recursive terms from equi-recursive values). For any expression e and type A, if $\cdot \vdash_e v : A \rhd e'$, then there exists v' such that $e' \hookrightarrow^* v'$ and $\cdot \vdash_e v : A \rhd v'$.

During the proof we find the congruence lemmas for full iso-recursive reductions and the substitution lemma for the elaboration relation useful, as shown below.

Lemma 4.5 (Congruence lemma of full iso-recursive reduction).

(1) If $e_1 \hookrightarrow^* e_1'$, then $e_1\ e_2 \hookrightarrow^* e_1'\ e_2$.
(2) If $e_2 \hookrightarrow^* e_2'$, then $v_1\ e_2 \hookrightarrow^* v_1\ e_2'$.
(3) If $e \hookrightarrow^* e'$, then cast $[c]\ e \hookrightarrow^*$ cast $[c]\ e'$.

and the propositions above also hold for the transitive closure $\hookrightarrow^+$.

Lemma 4.6 (Substitution lemma for elaboration). For any typing context Γ, expressions e_1, e_1', e_2, e_2' and types A, B, if $\Gamma, x : A \vdash_e e_1 : B \rhd e_1'$ and $\Gamma \vdash_e e_2 : A \rhd e_2'$, then $\Gamma \vdash_e e_1[x \mapsto e_2] : B \rhd e_1'[x \mapsto e_2']$.

The proof of Lemma 4.4 is straightforward by induction on the typing derivation $\cdot \vdash_e v : A \rhd e'$ and using the congruence lemmas. In the rest of this section we focus on the proof of Lemma 4.3.

PROOF OF LEMMA 4.3. The proof of Lemma 4.3 is done by first induction on the equi-recursive reduction relation $e_1 \hookrightarrow_e e_2$, and then induction on the elaboration relation $\cdot \vdash_e e_1 : A \rhd e_1'$. Most of the cases are straightforward, by applying the induction hypothesis and using the congruence lemmas to construct the reduction steps.

The tricky case is when $e_1 \hookrightarrow_e e_2$ is a beta reduction (case RED-BETA). By inversion on the reduction relation, we know that e_1 is a function application ($e_1 = (\lambda x : A_1.e_0)\ v_1$) and e_2 is ($e_0[x \mapsto v_1]$). By induction on the elaboration relation $\cdot \vdash_e e_1 : A \rhd e_1'$, case ETYP-EQ can be proved using the induction hypothesis. We consider case ETYP-APP, where

$$\cdot \vdash \lambda x : A_1.e_0 : A_1 \to A \rhd e_3' \quad \text{and} \quad \cdot \vdash v_1 : A_1 \rhd e_4' \tag{3}$$

However, we do not know the exact form of e_3' and e_4', since many different casts can be inserted in the elaboration derivation using rule ETYP-EQ, and the current induction hypothesis cannot deal with this. To address this issue, Lemma 4.4 is used to show that regardless of the form of e_3' and e_4', they can always be further reduced to values. By applying Lemma 4.4 to the two terms in the tricky case (3) above, we also know that the value of evaluating e_3' preserves the function type, so it must be one of the two canonical forms: a lambda abstraction ($\lambda x : A_1.e_0'$) or an arrow cast ($\text{cast}\,[c_1 \to c_2]\,v_1'$), and then we can construct the reduction steps for ($e_3'\ e_4'$) as shown below:

$$\begin{array}{llcll}
 & e_3'\ e_4' & & & \\
\hookrightarrow^* & (\lambda x : A_1.e_0')\ e_4' & \text{or} & (\text{cast}\,[c_1 \to c_2]\,v_1')\ e_4' & (\text{Lemma 4.4 and 4.5(1)}) \\
\hookrightarrow^* & (\lambda x : A_1.e_0')\ v_2' & \text{or} & (\text{cast}\,[c_1 \to c_2]\,v_1')\ v_2' & (\text{Lemma 4.4 and 4.5(2)}) \\
\hookrightarrow & e_0'[x \mapsto v_2'] & \text{or} & (\text{cast}\,[c_2]\,(v_1'\ (\text{cast}\,[\neg c_1]\,v_2'))) & (\text{Rule RED-BETA or RED-CAST-ARR})
\end{array}$$

Now we are left to prove the second goal of this case, that is, the result of the reduction constructed above can be derived from the elaboration relation, i.e.

$$\cdot \vdash_e e_0[x \mapsto v_1] : A \rhd e_0'[x \mapsto v_2'] \quad \text{or} \quad \cdot \vdash_e e_0[x \mapsto v_1] : A \rhd (\text{cast}\,[c_2]\,(v_1'\ (\text{cast}\,[\neg c_1]\,v_2')))$$

The latter case for rule RED-CAST-ARR follows from the induction hypothesis. The first case for rule RED-BETA can be proved by the substitution lemma for the elaboration relation (Lemma 4.6). □

With Lemma 4.5, 4.4 and 4.6, we complete the proof of Lemma 4.3, and the behavioral preservation theorem (Theorem 3.9) follows. Compared to the behavioral equivalence proof in Abadi and Fiore's work, we show that it is much more straightforward to prove the behavioral equivalence between full iso-recursive types and equi-recursive types, and our proof for λ^{μ}_{Fi} is completely mechanized in Coq. The proof does not rely on any user-defined conjectures or axioms except *functional_extensionality_dep* and *proof_irrelevance* introduced by the Coq standard library, and *eq_rect_eq* and *JMeq_eq* as their corollaries, or those introduced by Metalib that we used to formalize variables and binders with the locally nameless representation [Aydemir et al. 2008].

5 Recursive Subtyping

In this section we show that our results in the previous sections can be extended to a calculus with subtyping called $\lambda^{\mu<:}_{Fi}$.

5.1 A Calculus with Subtyping

Adapting the results in §3 to a calculus with subtyping requires only a few changes. In terms of types, we add a top type ($\top$). Expressions and values remain the same.

$\boxed{\Sigma \vdash A \leq_{\oplus} B}$ *(Equi-recursive/Iso-recursive Subtyping)*

$$\textsc{Sub-top}\ \frac{}{\Sigma \vdash A \leq_{\oplus} \top} \qquad \textsc{Sub-int}\ \frac{}{\Sigma \vdash \mathsf{Int} \leq_i \mathsf{Int}} \qquad \textsc{Sub-eq}\ \frac{A \doteq B}{\Sigma \vdash A \leq_e B} \qquad \textsc{Sub-var}\ \frac{\alpha \leq \beta \in \Sigma}{\Sigma \vdash \alpha \leq_{\oplus} \beta} \qquad \textsc{Sub-self}\ \frac{}{\Sigma \vdash \mu\alpha.\ A \leq_i \mu\alpha.\ A}$$

$$\textsc{Sub-trans}\ \frac{\Sigma \vdash A \leq_e B \qquad \Sigma \vdash B \leq_e C}{\Sigma \vdash A \leq_e C} \qquad \textsc{Sub-arrow}\ \frac{\Sigma \vdash B_1 \leq_{\oplus} A_1 \qquad \Sigma \vdash A_2 \leq_{\oplus} B_2}{\Sigma \vdash A_1 \rightarrow A_2 \leq_{\oplus} B_1 \rightarrow B_2} \qquad \textsc{Sub-rec}\ \frac{\Sigma, \alpha \leq \beta \vdash A \leq_{\oplus} B}{\Sigma \vdash \mu\alpha.\ A \leq_{\oplus} \mu\beta.\ B}$$

Fig. 11. Amadio and Cardelli's equi-recursive and iso-recursive subtyping rules.

Subtyping. The equi-recursive and iso-recursive subtyping rules that we use in this section are based on the Amber rules [Amadio and Cardelli 1993], as shown in Figure 11. The subtyping rules use a special environment Σ, which tracks a set of pairs of type variables that are assumed in the subtyping relation, as explained in §2.4. We use $\leq_{\oplus}$ parameterized by a metavariable $\oplus \in \{i, e\}$ to denote the subtyping rules for both relations: i denotes iso-recursive subtyping, and e denotes equi-recursive subtyping. We use $\leq_e$ to denote the subtyping rules (rules Sub-eq and Sub-trans) that only apply to equi-recursive types, and $\leq_i$ to denote the subtyping rules (rules Sub-int and Sub-self) that only apply to iso-recursive types. Rule Sub-eq embeds the equi-recursive equality relation in Figure 1 into the subtyping relation, so is only present in equi-recursive subtyping. For the iso-recursive subtyping relation, we choose the variant of the Amber rules presented by Zhou et al. [2022], which replaces the built-in reflexivity with the more primitive rules Sub-int and Sub-self and removes transitivity rule Sub-trans from the original Amber rules. The different treatment of reflexivity between the two formulations are denoted by dedicated rules Sub-int and Sub-self for iso-recursive subtyping and rule Sub-eq for equi-recursive subtyping in Figure 11. Zhou et al. discussed the technical challenges of having reflexivity and transitivity built-in in the iso-recursive subtyping relation, and showed that they are admissible from the other rules. For the iso-recursive subtyping relation we strictly follow their formulation to maximally reuse their mechanized proof in our work.

Lemma 5.1 (Reflexivity of iso-recursive subtyping). If A is a closed type, then $\Sigma \vdash A \leq_i A$.

Lemma 5.2 (Transitivity of iso-recursive subtyping). If $\cdot \vdash A \leq_i B$ and $\cdot \vdash B \leq_i C$, then $\cdot \vdash A \leq_i C$.

In Lemma 5.1, the assumption that A is a closed type can be implied by the fact that the free variable sets of A and B in the subtyping relation $\Sigma \vdash A \leq_i B$ are disjoint, which is also a side condition in Amadio and Cardelli's equi-recursive Amber rules. As for the transitivity, Lemma 5.2 only holds when the environment is empty. Otherwise, one may also get into problematic subtyping relations, as discussed by Zhou et al.. For the reasons above, we choose to use the variant of iso-recursive Amber rules without built-in reflexivity and transitivity in this section.

Typing and Reduction. As for the typing rules, we extend the full iso-recursive type system in Figure 4 with rule Typ-sub. Accordingly, the elaboration rules defined in Figure 6 are also extended with rule ETyp-isub for iso-recursive subtyping. As we will discuss later, rules ETyp-isub and ETyp-eq can together be used to encode equi-recursive subtyping. Therefore in the equi-recursive typing relation (Figure 6 without the gray parts), instead of adding rule ETyp-isub on top of the existing rules, we replace rule ETyp-eq with the rule ETyp-bare-sub so that the typing relation faithfully reflects the conventional equi-recursive type system. There are no changes to the reduction rules.

$$\textsc{Typ-sub}\ \frac{\Gamma \vdash e : A \qquad \cdot \vdash A \leq_i B}{\Gamma \vdash e : B} \qquad \textsc{ETyp-isub}\ \frac{\Gamma \vdash_e e : A \ \rhd e' \qquad \cdot \vdash A \leq_i B}{\Gamma \vdash_e e : B \ \rhd e'} \qquad \textsc{ETyp-bare-sub}\ \frac{\Gamma \vdash_e e : A \qquad \cdot \vdash A \leq_e B}{\Gamma \vdash_e e : B}$$

5.2 Type Soundness

There are no significant technical challenges in extending the type soundness proof to $\lambda_{Fi}^{\mu<:}$. The only part that requires extra care is the preservation lemma, in which we need to show that rule Red-castelim preserves the typing in the presence of subtyping. Let us consider an expression cast $[\mathsf{unfold}_{\mu\alpha.\ A}]$ (cast $[\mathsf{fold}_{\mu\alpha.\ B}]\ v$). The derivation below shows the typing of this expression.

$$\textsc{Typ-cast}\ \frac{\textsc{Typ-sub}\ \dfrac{\textsc{Typ-cast}\ \dfrac{\cdot \vdash v : B[\alpha \mapsto \mu\alpha.\ B] \qquad \ldots}{\cdot \vdash \text{cast}\ [\mathsf{fold}_{\mu\alpha.\ B}]\ v : \mu\alpha.\ B} \qquad \cdot \vdash \mu\alpha.\ B \leq_i \mu\alpha.\ A}{\cdot \vdash \text{cast}\ [\mathsf{fold}_{\mu\alpha.\ B}]\ v : \mu\alpha.\ A} \qquad \ldots}{\cdot \vdash \text{cast}\ [\mathsf{unfold}_{\mu\alpha.\ A}]\ (\text{cast}\ [\mathsf{fold}_{\mu\alpha.\ B}]\ v) : A[\alpha \mapsto \mu\alpha.\ A]}$$

By inversion we know that after reduction using rule Red-castelim, the result v has the type $B[\alpha \mapsto \mu\alpha.\ B]$, and that $\mu\alpha.\ B \leq_i \mu\alpha.\ A$. The preservation proof goal for this case can be expressed as the following lemma:

Lemma 5.3 (Unfolding lemma). If $\cdot \vdash \mu\alpha.\ B \leq_i \mu\alpha.\ A$, then $\cdot \vdash B[\alpha \mapsto \mu\alpha.\ B] \leq_i A[\alpha \mapsto \mu\alpha.\ A]$.

The proof of this lemma has been shown by Zhou et al. [2022, Corollary 59]. They proposed an alternative formulation of the iso-recursive subtyping rules, which is equivalent to the iso-recursive Amber rules. Therefore, by adopting their results, we complete the type soundness proof for $\lambda_{Fi}^{\mu<:}$.

Theorem 5.4 (Type soundness of $\lambda_{Fi}^{\mu<:}$). For any term e and type A in $\lambda_{Fi}^{\mu<:}$,
(1) (Progress) if $\cdot \vdash e : A$ then either e is a value or there exists a term e' such that $e \hookrightarrow e'$.
(2) (Preservation) if $\cdot \vdash e : A$ and $e \hookrightarrow e'$ then $\cdot \vdash e' : A$.

5.3 Typing Equivalence

Similarly to λ_{Fi}^{μ} with equality, we can prove that $\lambda_{Fi}^{\mu<:}$ with iso-recursive subtyping is sound and complete with respect to a calculus with equi-recursive subtyping. The key idea is encoding the equi-recursive subtyping relation using a combination of equi-recursive equality and the iso-recursive subtyping relation, as explained in §2.5. The encoding can be justified by the following theorem:

Theorem 5.5 (Equi-recursive subtyping decomposition). $\cdot \vdash A \leq_e B$ if and only if there exist types C_1 and C_2 such that $A \doteq C_1$, $\cdot \vdash C_1 \leq_i C_2$, and $C_2 \doteq B$.

The soundness direction (i.e. the "if" direction) of this lemma is straightforward by rules Sub-eq and Sub-trans and the fact that $\leq_i$ is a sub-relation of $\leq_e$. The completeness direction can be derived from Amadio and Cardelli's proof of completeness with respect to the tree model for the equi-recursive Amber rules. They proved that for any types A, B that are in the tree model interpretation of the equi-recursive subtyping relation, one can find types C_1 and C_2 such that $A \doteq C_1$, $C_1 \leq_e C_2$, and $C_2 \doteq B$ hold [Amadio and Cardelli 1993, Lemma 5.4.1, Lemma 5.4.3]. Moreover, the derivation of $C_1 \leq_e C_2$ satisfies the *one-expansion property*, which means that in the derivation each recursive type is unfolded at most once, informally speaking. Although this result is expressed as an equi-recursive subtyping relation in their conclusion, we can rewrite all the occurrences of $C_1 \leq_e C_2$ with one-expansion property in their proof to an iso-recursive subtyping relation $C_1 \leq_i C_2$. Every application of rules Sub-eq and Sub-trans in their proofs can either be

$$\dfrac{\dfrac{\vdash \texttt{Int} \leq_e \texttt{Int} \qquad \dfrac{\begin{array}{c}\textsc{Tyeq-unfold}\\ \textsc{Tyeq-contract}\\ A_1 \doteq \top \to A_2\end{array} \qquad \dfrac{\vdash \top \leq_e \top \qquad \dfrac{\mathcal{P}_1}{\vdash A_2 \leq_e B}\,\textsc{Sub-rec}}{\vdash \top \to A_2 \leq_e \top \to B}\,\textsc{Sub-arrow}}{\vdash A_1 \leq_e \top \to B}\,\textsc{Sub-eq}\text{ and }\textsc{Sub-trans}}{\vdash A \leq_e \texttt{Int} \to \top \to B}\,\textsc{Sub-arrow} \qquad \begin{array}{c}\textsc{Tyeq-unfold}\\ \vdash \texttt{Int} \to \top \to B \doteq B\end{array}}{\vdash A \leq_e B}\,\begin{array}{l}\textsc{Sub-trans}\\ \text{and }\textsc{Sub-eq}\end{array}$$

$$\dfrac{\dfrac{\vdash \texttt{Int} \leq_i \texttt{Int} \qquad \dfrac{\begin{array}{c}\text{Lemma 5.1}\\ \vdash \top \to A_2 \leq_i \top \to A_2\end{array} \qquad \dfrac{\vdash \top \leq_i \top \qquad \dfrac{\mathcal{P}_2}{\vdash A_2 \leq_i B}\,\textsc{Sub-rec}}{\vdash \top \to A_2 \leq_i \top \to B}\,\textsc{Sub-arrow}}{\vdash \top \to A_2 \leq_i \top \to B}\,\text{Lemma 5.2}}{\vdash C \leq_i D}\,\textsc{Sub-arrow} \qquad \begin{array}{c}\text{Lemma 5.1}\\ \vdash D \leq_i D\end{array}}{\vdash C \leq_i D}\,\text{Lemma 5.2}$$

$A = \texttt{Int} \to (\mu\alpha.\, \top \to \alpha)$ $\qquad$ $B = \mu\alpha.\, \texttt{Int} \to \top \to \alpha$
$C = \texttt{Int} \to \top \to (\mu\alpha.\, \top \to \top \to \alpha)$ $\qquad$ $D = \texttt{Int} \to \top \to (\mu\alpha.\, \texttt{Int} \to \top \to \alpha)$
$A_1 = \mu\alpha.\, \top \to \alpha$ $\qquad$ $A_2 = \mu\alpha.\, \top \to \top \to \alpha$
$\mathcal{P}_1$ is the judgement $\alpha \leq \beta \vdash \top \to \top \to \alpha \leq_e \texttt{Int} \to \top \to \beta$
$\mathcal{P}_2$ is the judgement $\alpha \leq \beta \vdash \top \to \top \to \alpha \leq_i \texttt{Int} \to \top \to \beta$

Fig. 12. Illustration of decomposing an equi-recursive subtyping derivation.

replaced by rule Sub-rec, in which the recursive type body does not involve the type variable and unfolds to itself, or by rule Sub-self for two recursive types that are syntactically equal up to α-renaming. In other words, Amadio and Cardelli's proof of their Lemma 5.4.3 can be seen as a proof that the iso-recursive subtyping relation is complete with respect to the equi-recursive subtyping relation with the one-expansion property, because they never use the power of the rules Sub-eq and Sub-trans.

The idea of this decomposition can be illustrated by an example in Figure 12. The upper part of the figure shows a derivation by following Amadio and Cardelli's subtyping algorithm for equi-recursive subtyping. Note that there are two applications of rule Sub-eq in the derivation, one for expanding the type $\mu\alpha.\, \top \to \alpha$ to $\top \to \mu\alpha.\, \top \to \top \to \alpha$ and the other for expanding the type B to its unfolding $\texttt{Int} \to \top \to B$. Although rule Sub-eq is applied in the middle of the derivation, we can always lift these uses of rule Sub-eq to the top of the derivation, by replacing two types in the conclusion with their more expanded forms. The lower part of the figure shows such a derivation, in which we use C and D to denote (a simplified form of) the expanded types obtained from Amadio and Cardelli's proof. A key observation here is that the original structure of the derivation is preserved in the new derivation. To highlight this, we use a dummy application of the reflexivity and transitivity lemma to show the correspondence between the two derivations.

With the decomposition theorem, we can use the following rule to encode the equi-recursive subtyping relation:

ETyp-sub
$$\dfrac{\Gamma \vdash_e e : A \;\rhd e' \qquad \cdot;\cdot \vdash A \hookrightarrow C_1 : c_1 \qquad \cdot \vdash C_1 \leq_i C_2 \qquad \cdot;\cdot \vdash C_2 \hookrightarrow B : c_2}{\Gamma \vdash_e e : B \;\rhd \mathsf{cast}\,[c_2]\,(\mathsf{cast}\,[c_1]\,e')}$$

If one ignores the gray parts in the rule, rule ETYP-SUB is equivalent to rule ETYP-BARE-SUB. We can first apply Theorem 3.1 to rewrite our type casting rules to equi-recursive equalities and then use Theorem 5.5 to show the equivalence to the equi-recursive subtyping relation. On the other hand, rule ETYP-SUB can be derived from the primitive rules ETYP-EQ and ETYP-ISUB in $\lambda_{Fi}^{\mu<:}$. Therefore, we can conclude that $\lambda_{Fi}^{\mu<:}$ with iso-recursive subtyping is sound and complete with respect to a calculus with equi-recursive subtyping in terms of typing.

Unlike λ_{Fi}^{μ}, the cast operator in rule ETYP-SUB cannot be automatically generated, since Theorem 5.5 by Amadio and Cardelli is not done in a constructive way, and they choose regular equations, a different representation of recursive types, to complete the proof. Therefore, it is not easy to turn their results into an algorithm that generates the cast operator for equi-recursive subtyping, which we leave as future work. Nevertheless, $\lambda_{Fi}^{\mu<:}$ itself is still useful if one wants to work with iso-recursive types but expects expressive power similar to equi-recursive subtyping.

Theorem 5.6 (Typing equivalence for $\lambda_{Fi}^{\mu<:}$). For any expressions e, e' and type A,

(1) (Soundness) if $\Gamma \vdash e : A$ then $\Gamma \vdash_e |e| : A \rhd e$.
(2) (Completeness) if $\Gamma \vdash_e e : A \rhd e'$ then $\Gamma \vdash e' : A$.
(3) (Round-tripping) if $\Gamma \vdash_e e : A \rhd e'$, then $|e'| = e$.

5.4 Behavioral Equivalence

We also show that $\lambda_{Fi}^{\mu<:}$ with iso-recursive subtyping is sound and complete with respect to a calculus with equi-recursive subtyping in terms of dynamic semantics. Since there are no changes to the reduction rules, the proof of $\lambda_{Fi}^{\mu<:}$ to equi-recursive behavioral equivalence by erasure of cast operators remains the same as Theorem 3.8. The proof of the other direction comes almost for free as well. We simply follow the same steps as described in §4.3 and use the same lemmas and theorems to show the completeness of $\lambda_{Fi}^{\mu<:}$ in preserving the equi-recursive reductions, except that during the proof of Lemma 4.3, we may need to insert an application of rule ETYP-ISUB at certain points to prove that the encoding is well-typed. In terms of dynamic semantics, $\lambda_{Fi}^{\mu<:}$ is equivalent to a calculus with equi-recursive subtyping.

Theorem 5.7 (Behavioral equivalence of $\lambda_{Fi}^{\mu<:}$). For any expression e, e' and type A in $\lambda_{Fi}^{\mu<:}$, if $\cdot \vdash_e e : A \rhd e'$, then

(1) $e \hookrightarrow_e^* v$ if and only if there exists v' such that $e' \hookrightarrow^* v'$ and $|v'| = v$.
(2) e diverges if and only if e' diverges.

6 Related Work

Throughout the paper, we have discussed some of the closest related work in detail. This section covers additional related work.

Relating iso-recursive and equi-recursive types. Recursive types were first introduced by Morris, who presented equi-recursive types to model recursive definitions. Later on, iso-recursive types were introduced [Crary et al. 1999; Gunter 1992; Harper and Mitchell 1993]. The terms for these two types of recursive formulations were coined by Crary et al..

Both equi-recursive and iso-recursive types have been applied in various programming language areas. Equi-recursive types are used in several contexts, including: session types [Castagna et al. 2009; Chen et al. 2014; Gay and Hole 2005; Gay and Vasconcelos 2010], gradual typing [Siek and Tobin-Hochstadt 2016], and the foundation of Scala through Dependent object types (DOT) [Amin et al. 2016; Rompf and Amin 2016], among others. Iso-recursive types have also been utilized in different calculi and language designs due to their ease of use in type checking [Abadi and Cardelli 1996; Bengtson et al. 2011; Chugh 2015; Duggan 2002; Lee et al. 2015; Swamy et al. 2011].

In real-world programming languages, different languages have adopted different formulations. Iso-recursive types are used in languages like Standard ML [Vanderwaart et al. 2003], Haskell [Weirich et al. 2011], and OCaml [Dreyer et al. 2001], while equi-recursive types are used in languages like Modula-3 [Cardelli et al. 1989] and Scala [Odersky et al. 2004].

Urzyczyn [1995] studied the relationship between positive iso- and equi-recursive types, showing their equivalence in typing power. Closer to our work, Abadi and Fiore [1996] explored translating equi-recursive terms to iso-recursive terms using explicit coercion functions but did not address the operational semantics. Moreover, their behavioral equivalence argument relies on a program logic which was conjectured to be sound. As we have argued in §2.3, the use of explicit coercions has important drawbacks. Firstly, it adds significant computational overhead to the encoding, making the encoding impractical. Secondly, it introduces major complications to reasoning, and also prevents a round-tripping property. Thirdly, their coercion rules are based on a *declarative specification* of equi-recursive type equality by Amadio and Cardelli [1993], which does not have a known algorithm to generate coercions automatically. To see this, consider their coercion typing rule for rule Tyeq-contract:

$$\frac{s : A[\alpha \mapsto B_1] \doteq B_1 : t \qquad s' : A[\alpha \mapsto B_2] \doteq B_2 : t' \qquad A \text{ is contractive in } \alpha}{\mathsf{it}(t', s').\mathsf{coit}(t, s) : B_1 \doteq B_2 : \mathsf{it}(t, s).\mathsf{coit}(t', s')}$$

The judgement $s : A \doteq B : t$ extends Amadio and Cardelli's equi-recursive type equality with two coercion functions $s : A \to B$ and $t : B \to A$. For example, the first premise says that from equality $A[\alpha \mapsto B_1] \doteq B_1$ one can derive coercion functions $s : A[\alpha \mapsto B_1] \to B_1$ and $t : B_1 \to A[\alpha \mapsto B_1]$. They define special combinators $\mathsf{it}(\cdot,\cdot)$ and $\mathsf{coit}(\cdot,\cdot)$ to combine these coercion functions. Basically, the it combinator can be used to obtain a coercion function $\mathsf{it}(s, t) : \mu\alpha.\ A \to B_1$, while the coit combinator gives $\mathsf{coit}(t', s') : B_2 \to \mu\alpha.\ A$. Therefore their composition $\mathsf{it}(t, s).\mathsf{coit}(t', s')$ serves as a coercion function from B_2 to B_1. As can be seen from the rule, to obtain a well-typed coercion function from the rule Tyeq-contract, these complex combinators are necessary and cause a significant overhead. By using casts, we avoid all these issues with coercion functions, leading to an easier, and fully formalized, way to establish behavioral equivalence.

Recently, Patrignani et al. [2021] examined the contextual equivalence between iso- and equi-recursive types, providing a mechanized proof in Coq for fully abstract compilers. Their focus was on the compilation from iso-recursive to equi-recursive types. They proved that the translation from iso-recursive to equi-recursive types, by erasing unfold/fold operations, is fully abstract with respect to contextual equivalence. The work also covered the compiler from term-level fixpoints to equi-recursive types, but did not explore the translation from equi-recursive to iso-recursive types. In our work, we establish the bidirectional equivalence between full iso-recursive and equi-recursive types, taking into account both typing and operational semantics. Furthermore, in addition to type equality, we also study calculi with subtyping, which have not been covered in previous work studying the relationship between iso- and equi-recursive typing.

Subtyping recursive types. Amadio and Cardelli [1993] were the first to present a comprehensive formal study of subtyping with equi-recursive types. This work inspired further research that refined and simplified the original study [Brandt and Henglein 1998; Danielsson and Altenkirch 2010; Gapeyev et al. 2002; Komendantsky 2011]. In particular, Brandt and Henglein [1998] introduced a fixpoint rule for a coinductive relation within an inductive framework. Their rules give rise to a natural operational interpretation of proofs as coercions, as they indicated as future work in their paper. Our work is inspired by their work, and we formally present an operational interpretation of equi-recursive equalities in our paper. However, instead of using coercions to model the subtyping relation as they suggested, we use cast operators to model the equalities between equi-recursive types. Furthermore, we show that our computationally irrelevant cast operators simplify the metatheory and extend to subtyping as well.

Iso-recursive subtyping, notably through the Amber rules [Cardelli 1985], has long been used. The iso-recursive Amber rules, while easy to implement, are difficult to reason with formally. The only known direct proof for transitivity of subtyping for an algorithmic version of the Amber rules was given by Bengtson et al. [2011]. This proof relies on a complex inductive argument and was found difficult to formalize in theorem provers [Backes et al. 2014; Zhou et al. 2022]. Zhou et al., proposed alternative formulations of iso-recursive subtyping equivalent to the Amber rules and are also easier to reason with. Their work comes with a comprehensive formalization of the metatheory of iso-recursive subtyping. Our work is based on some of their findings. In particular we reuse their mechanized proof of the unfolding lemma to show the type soundness of iso-recursive subtyping, but instead apply it in a setting with full iso-recursive types. Thus, we extend their work to a more general setting, in terms of typing and operational semantics.

To address the complexities of iso-recursive subtyping, several alternative formulations of iso-recursive subtyping have been proposed. Hofmann and Pierce [1995] introduced a subtyping relation that limits recursive subtyping to covariant types only, making the rules more restrictive than the Amber rules. Ligatti et al. [2017] offered a broader subtyping relation for iso-recursive types, allowing a recursive type and its unfolded version to be considered subtypes of each other. This approach extends the iso-recursive Amber rules but is still not complete with respect to the equi-recursive subtyping, since it does not consider types not directly related by unfolding or folding as subtypes. Additionally, Rossberg [2023] developed a calculus for higher-order iso-recursive subtyping, to handle mutually recursive types more effectively.

Mechanizing recursive types. Danielsson and Altenkirch [2010]; Jones and Pearce [2016] formalized equi-recursive subtyping relations in Agda using a mixed coinduction and induction technique. Jones and Pearce presented a semantic interpretation of subtyping and proved that their semantic interpretation is sound with respect to an inductive interpretation of types, but they did not lift their results to cover function types. Instead, they focused on other constructs like product and sum types. Danielsson and Altenkirch are closer to our work since they also did not consider semantic interpretations, but formalized in Agda an alternative equi-recursive subtyping relation that allows an explicit transitivity rule to be included. They formally proved that this relation is equivalent to the tree model of subtyping as well as Brandt and Henglein's subtyping relation. In a similar vein, Komendantsky [2011] showed how to implement mixed coinduction and induction within Coq, formalizing rules that closely resemble those introduced by Danielsson and Altenkirch [2010]. They also validated their approach against Amadio and Cardelli's tree model of subtyping. Zhou et al. [2022] focused on formalizing Amber-style iso-recursive subtyping in Coq, adding to the understanding of iso-recursive subtyping. Patrignani et al. [2021], which we have discussed earlier, formalized three calculi in Coq: a simply typed lambda calculus extended with iso-recursive types, equi-recursive types, and term-level fixpoints. Their work is focused on the translation from iso-recursive to equi-recursive types, by erasing unfold/fold operations, and the translation from a calculus with term-level fixpoints to a calculus with equi-recursive types. All of our results are mechanized in Coq, with the exception of the decomposition lemma (Theorem 5.5). This lemma is implied from Amadio and Cardelli's work, but relies on a significant amount of technical machinery, which we have not formalized in Coq. Thus we assume it as an axiom in our Coq formalization.

Casts for type-level computation. In this paper, we employ explicit cast operators to represent the transformations between types related by equi-recursive equalities. Several studies [Cretin 2014; Gundry 2013; Kimmell et al. 2012; Sjöberg et al. 2012; Sjöberg and Weirich 2015; Stump et al. 2009; Sulzmann et al. 2007; Weirich et al. 2017; Yang and Oliveira 2019] have also used explicit casts for managed type-level computation. However, casts in those approaches primarily address type-level computations within contexts such as dependent types or type-level programming, rather than the operational interpretation of recursive type equalities. When considering the dynamic semantics of cast-like operations, there have been two major approaches. One approach is to use an elaboration semantics, used in works like [Sjöberg et al. 2012; Sjöberg and Weirich 2015; Stump et al. 2009], where the semantics are only defined for a cast-free language and the casts need to be erased before execution. Another approach is to use push rules as seen in [Sulzmann et al. 2007; Weirich et al. 2013, 2017; Yorgey et al. 2012], which is the approach that we adopt in our work. Our push rules designs resemble the ones used in the work of Sulzmann et al. [2007], where input arguments are applied a reversed cast, but our work directly creates a new cast expression for the concrete input expression while Sulzmann et al. rewrite the λ-term. Similar designs can also be seen in other lines of work, such as the blame calculus and gradual typing [Findler and Felleisen 2002; Siek and Taha 2007]. However, with full iso-recursive types we expect the casts to be erasable during runtime, which is not the case in gradual typing. Pure Iso-type Systems (PITS) [Yang and Oliveira 2019] provides a generalization of iso-recursive types with explicit casts, but their focus is on unifying the syntax of terms and types while retaining decidable type checking, instead of subsuming equi-recursive type casting as we do. Also, the form of casts is different from ours. For example, they do not have a fixpoint cast, to enable coinductive reasoning.

7 Conclusion

This paper proposes full iso-recursive types, a generalization of iso-recursive types that can be used to encode the full power of equi-recursive types. The key idea is to introduce a computationally irrelevant cast operator in the term language that captures all the equi-recursive type equalities. We present λ^{μ}_{Fi}, a calculus that extends simply typed lambda calculus with full iso-recursive types. λ^{μ}_{Fi} is proved to be type sound and has the same expressive power as a calculus with equi-recursive types, in terms of typing and dynamic semantics. Our results can also be extended to subtyping, by encoding equi-recursive subtyping using iso-recursive subtyping with cast operators.

As future work, we plan to extend λ^{μ}_{Fi} with other programming language features, such as polymorphism and intersection and union types. It is also interesting to see whether our results can scale to real world languages (e.g. Haskell). In particular, it would be interesting to employ full iso-recursive types in an internal target language with explicit cast operators for a source language using equi-recursive types.

Data-Availability Statement

The artifact that supports the paper is available on Zenodo [Zhou et al. 2024] and https://github.com/ltzone/Full-Iso-Recursive-Types.

Acknowledgments

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

References

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

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

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

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

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

Michael Backes, Cătălin Hriţcu, and Matteo Maffei. 2014. Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations. *J. Comput. Secur.* 22, 2 (mar 2014), 301–353.

Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D Gordon, and Sergio Maffeis. 2011. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 2 (2011), 1–45. https://doi.org/10.1145/1890028.1890031

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

Yufei Cai, Paolo G Giarrusso, and Klaus Ostermann. 2016. System F-omega with equirecursive types for datatype-generic programming. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 30–43.

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

Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. 1989. The Modula–3 type system. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 202–212. https://doi.org/10.1145/75277.75295

Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Luca Padovani. 2009. Foundations of session types. In *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*. 219–230. https://doi.org/10.1145/1599410.1599437

Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2014. On the preciseness of subtyping in session types. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*. 135–146. https://doi.org/10.1145/2643135.2643138

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

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

Karl Crary. 2000. Typed compilation of inclusive subtyping. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 68–81. https://doi.org/10.1145/351240.351247

Karl Crary, Robert Harper, and Sidd Puri. 1999. What is a recursive module?. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. 50–63. https://doi.org/10.1145/301618.301641

Julien Cretin. 2014. *Erasable coercions: a unified approach to type systems.* Ph. D. Dissertation. Université Paris-Diderot-Paris VII.

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

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

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

Dominic Duggan. 2002. Type-safe linking with recursive DLLs and shared libraries. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 24, 6 (2002), 711–804. https://doi.org/10.1145/586088.586093

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*. 48–59. https://doi.org/10.1145/581478.581484

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

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

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

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

Adam Michael Gundry. 2013. Type inference, Haskell and dependent types. (2013).

Carl A Gunter. 1992. *Semantics of programming languages: structures and techniques.* MIT press.

Robert Harper and John C Mitchell. 1993. On the type structure of Standard ML. *Acm transactions on programming languages and systems (TOPLAS)* 15, 2 (1993), 211–252. https://doi.org/10.1145/169701.169696

Robert Harper and Christopher Stone. 2000. A type-theoretic interpretation of Standard ML. (2000). https://doi.org/10.7551/mitpress/5641.003.0019

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

Alan Jeffrey. 2001. A symbolic labelled transition system for coinductive subtyping of F< types. In *2001 IEEE Conference on Logic and Computer Science (LICS 2001)*, Vol. 323.

Timothy Jones and David J Pearce. 2016. A mechanical soundness proof for subtyping over recursive types. In *Proceedings of the 18th Workshop on Formal Techniques for Java-like Programs.* 1–6. https://doi.org/10.1145/2955811.2955812

Garrin Kimmell, Aaron Stump, Harley D Eades III, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjöberg, Nathan Collins, and Ki Yung Ahn. 2012. Equational reasoning about programs with general recursion and call-by-value semantics. In *Proceedings of the sixth workshop on Programming languages meets program verification.* 15–26. https://doi.org/10.1145/2103776.2103780

Vladimir Komendantsky. 2011. Subtyping by folding an inductive relation into a coinductive one. In *International Symposium on Trends in Functional Programming.* Springer, 17–32.

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

Jay Ligatti, Jeremy Blackburn, and Michael Nachtigal. 2017. On subtyping-relation completeness, with an application to iso-recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 39, 1 (2017), 1–36. https://doi.org/10.1145/2994596

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

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. The Scala language specification.

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

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

Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications.* 624–641. https://doi.org/10.1145/2983990.2984008

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

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

Jeremy Siek and Walid Taha. 2007. Gradual typing for objects. In *European Conference on Object-Oriented Programming.* Springer, 2–27.

Jeremy G Siek and Sam Tobin-Hochstadt. 2016. The recursive union of some gradual types. In *A List of Successes that can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday.* Springer, 388–410. https://doi.org/10.1007/978-3-319-30936-1_21

Vilhelm Sjöberg, Chris Casinghino, Ki Yung Ahn, Nathan Collins, Harley D Eades III, Peng Fu, Garrin Kimmell, Tim Sheard, Aaron Stump, and Stephanie Weirich. 2012. Irrelevance, heterogeneous equality, and call-by-value dependent type systems. *arXiv preprint arXiv:1202.2923* (2012).

Vilhelm Sjöberg and Stephanie Weirich. 2015. Programming up to congruence. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 369–382. https://doi.org/10.1145/2676726.2676974

Aaron Stump, Morgan Deters, Adam Petcher, Todd Schiller, and Timothy Simpson. 2009. Verified programming in Guru. In *Proceedings of the 3rd workshop on Programming languages meets program verification.* 49–58. https://doi.org/10.1145/1481848.1481856

Martin Sulzmann, Manuel MT Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation.* 53–66. https://doi.org/10.1145/1190315.1190324

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

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

Joseph C Vanderwaart, Derek Dreyer, Leaf Petersen, Karl Crary, Robert Harper, and Perry Cheng. 2003. Typed compilation of recursive datatypes. *ACM SIGPLAN Notices* 38, 3 (2003), 98–108. https://doi.org/10.1145/640136.604187

Stephanie Weirich, Justin Hsu, and Richard A Eisenberg. 2013. System FC with explicit kind equality. *ACM SIGPLAN Notices* 48, 9 (2013), 275–286. https://doi.org/10.1145/2544174.2500599

Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A Eisenberg. 2017. A specification for dependent types in Haskell. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–29. https://doi.org/10.1145/3110275

Stephanie Weirich, Dimitrios Vytiniotis, Simon Peyton Jones, and Steve Zdancewic. 2011. Generative type abstraction and type-level computation. *ACM SIGPLAN Notices* 46, 1 (2011), 227–240. https://doi.org/10.1145/1925844.1926411

Yanpeng Yang and Bruno C. d. S. Oliveira. 2019. Pure iso-type systems. *Journal of Functional Programming* 29 (2019), e14. https://doi.org/10.1017/S0956796819000108

Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. 53–66. https://doi.org/10.1145/2103786.2103795

Litao Zhou, Qianyong Wan, and Bruno C. d. S. Oliveira. 2024. *Full Iso-recursive Types (Artifact).* https://doi.org/10.5281/zenodo.12669929

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

Yaoda Zhou, Bruno C. d. S. Oliveira, and Jinxu Zhao. 2020. Revisiting iso-recursive subtyping. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28. https://doi.org/10.1145/3428291

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

Received 2024-04-04; accepted 2024-08-18