System F-omega with Equirecursive Types for Datatype-generic Programming

Yufei Cai, Paolo G. Giarrusso, and Klaus Ostermann

University of Tübingen, Germany

December 8, 2015

Traversing an algebraic datatype by hand requires boilerplate code which duplicates the structure of the datatype. Datatype-generic programming (DGP) aims to eliminate such boilerplate code by decomposing algebraic datatypes into type constructor applications from which generic traversals can be synthesized. However, different traversals require different decompositions, which yield isomorphic but unequal types. This hinders the interoperability of different DGP techniques.

In this paper, we propose F^{μ}_{ω}, an extension of the higher-order polymorphic lambda calculus F_{ω} with records, variants, and equirecursive types. We prove the soundness of the type system, and show that type checking for first-order recursive types is decidable with a practical type checking algorithm. In our soundness proof we define type equality by interpreting types as infinitary λ-terms (in particular, Berarducci-trees). To decide type equality we β-normalize types, and then use an extension of equivalence checking for usual equirecursive types.

Thanks to equirecursive types, new decompositions for a datatype can be added modularly and still inter-operate with each other, allowing multiple DGP techniques to work together. We sketch how generic traversals can be synthesized, and apply these components to some examples.

Since the set of datatype decomposition becomes extensible, System F^{μ}_{ω} enables using DGP techniques incrementally, instead of planning for them up-front or doing invasive refactoring.

The main body of this document is a version of the paper of the same title in *Symposium on Principles of Programming Languages* 2016 [15]. The appendices are new. They supply the soundness and decidability proofs and go into greater detail about the generation of traversable functors.

Contents

1. Introduction

Programs operating on algebraic data types are often repetitive and fragile. Such programs typically depends on details of the data structure that are irrelevant to the purpose of the program, hence datatype definitions and recursion schemes are redundantly duplicated many times. Research on datatype-generic programming strives to abstract code duplicated across a data structure definition and its consumers into reusable form, hence separating the concerns of traversing the data structure recursively and of handling each case appropriately [26, 25].

But to this end, each technique for datatype-generic programming decomposes a datatype in a different way. Different decompositions do not inter-operate well because they create *incompatible* datatypes. For instance, we can refactor a consumer of algebraic data into a fold, after replacing the datatype T with the fixed point of a functor F, that is, $T_1 = \mu\ F$. Other techniques require different incompatible datatype refactorings, replacing T with a different T_2. In general, even if all these decompositions are isomorphic, that is, $T_1 \cong T \cong T_2$, a typechecker will not recognize them as equivalent and will prevent the programmer from making use of different decompositions at the same time. A programmer could manually define and use the isomorphisms between these datatypes, but this would be another elaborate and error-prone source of redundancy.

We argue that this problem can be fixed in a language which is on the one hand

powerful enough to express datatype-generic programming techniques—System F_ω—and on the other hand supports interoperability between different datatype decompositions by equirecursive types. Equirecursive types—as opposed to isorecursive types—aim to make many isomorphic datatypes equal. For instance, a recursive type $\mu\ F$ is equal to its unfolding $F\ (\mu\ F)$. Systems supporting equirecursive types have been studied, but they either lack known practical typechecking algorithms, or do not provide support for type constructors, which is required for datatype-generic programming. Hence, in this paper we fill this gap.

More specifically, we make the following contributions.

- We formally define System F^μ_ω, an extension of System F_ω with equirecursive datatypes (Sec. 4).
- We define and study the *coinductive equational theory* of F^μ_ω types, based on the theory of infinitary λ-calculus. Using this theory, we prove type soundness for F^μ_ω (Sec. 5.2).
- We show that $F^{\mu*}_\omega$, that is F^μ_ω restricted to first-order recursive types, enjoys decidable typechecking (Sec. 5.3) but is still expressive enough to support DGP (Sec. 2.2).
- To further support DGP, we automate the generation of traversal schemes from type constructors corresponding to traversable functors (Sec. 2.4 and Sec. 6).

The rest of the paper is structured as follows. Sec. 2 motivates F^μ_ω and gives a high-level overview. Sec. 3 discusses related work on DGP and equirecursive types. Sec. 4 formalizes the static semantics of F^μ_ω. Sec. 5 discusses the soundness of F^μ_ω and the decidability of typechecking in $F^{\mu*}_\omega$. Sec. 6 is about boilerplate generation. Sec. 7 lists future work. Sec. 8 concludes.

Material added in this extended version appears in appendices. We will point to it throughout the text.

2. Overview

In conventional functional programming with algebraic datatypes and pattern matching, functions that operate on algebraic data types are tightly coupled to the details of the datatype. For instance, consider a Haskell function to compute the free variables of a lambda term with integer literals.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Term} = \mathit{Lit}\ \mathit{Int} \quad\ \ | \ \mathit{Abs}\ \mathit{String}\ \mathit{Term} \\
&\qquad\qquad\ \ | \ \mathit{Var}\ \mathit{String} \mid \mathit{App}\ \mathit{Term}\ \mathit{Term} \\
&\mathit{fv} :: \mathit{Term} \to \mathit{Set}\ \mathit{String} \\
&\mathit{fv}\ (\mathit{Lit}\ n) \quad\ = \mathit{empty} \\
&\mathit{fv}\ (\mathit{Var}\ x) \quad\ = \mathit{singleton}\ x \\
&\mathit{fv}\ (\mathit{Abs}\ x\ t) \ = \mathit{delete}\ x\ (\mathit{fv}\ t) \\
&\mathit{fv}\ (\mathit{App}\ t_1\ t_2) = \mathit{union}\ (\mathit{fv}\ t_1)\ (\mathit{fv}\ t_2)
\end{aligned}
$$

The definition of *fv* combines the logic to compute free variables with the boilerplate to perform a traversal, and some more boilerplate to merge results collected across the traversal.

However, the traversal boilerplate can be derived from the datatype description: it is sufficient to rewrite the algebraic datatype *Term* as the least fixed point of its *pattern functor* *TermF*:

```
data TermF t = Lit' Int      | Abs' String t
             | Var' String | App' t t
type Term' = Fix TermF
newtype Fix f = Roll { unroll :: f (Fix f) }
```

Using *TermF*, one can now use standard DGP techniques to decouple the free variables algorithm from the structure of the datatype. One can mechanically and automatically derive a definition of the *fmap* function, and based on the *fmap* function one can define generic traversals such as catamorphisms (that abstract over structural recursion) or even a generic *traverse* function [39] with which one can, say, accumulate the contents of algebraic data (using any monoid for combination) in a highly generic way.

Term and *Term'* are obviously isomorphic, but not equal.

Decomposing *Term* into *Fix* and *TermF* is not the only option, though; many different decompositions are useful and sensible. For instance, consider call-by-name β-reduction. *TermF* is not an adequate representation of the recursion structure of this algorithm, since the latter only recurses into the left hand side of an *App* constructor, but not the right hand side.

The recursion structure of this algorithm is captured by additionally defining *Term''* as the fixed point of *EvalCtx*. Again, *Term''* is isomorphic to both *Term* and *Term'*, but not equal.

```
data EvalCtx t = Lit'' Int      | Abs'' String Term
               | Var'' String | App'' t Term
type Term'' = Fix EvalCtx
```

Many other functors are possible. Each functor defines a particular view on a datatype. For instance, we can additionally define a type equivalent to *Term* via a functor that focuses on the variable names.

```
data VarTerm t = Lit''' Int | Abs''' String Term
               | Var''' t   | App''' Term Term
type Term''' = VarTerm String
```

Such functors are common when defining *lenses* [36] of a datatype. In general, a datatype with n fields is associated to 2^n functors, 3^n bifunctors, 4^n trifunctors, etc, that is, a *super-exponential* amount of functors.

The datatypes defined via these functors are isomorphic but not equal, which means that programmers have to choose a *dominant* functor ahead of time, and DGP techniques

are only directly available for the dominant functor — in other words, we have a *tyranny of the dominant functor* (analogous to the tyranny of the dominant decomposition [52]). For other decompositions, the programmer would have to manually define and apply the isomorphisms, which is elaborate and error-prone, especially because the number of isomorphisms grows quadratically with the number of functors.

2.1. The Problems

In our example, values of different datatypes are incompatible, first, because different datatypes cannot share data constructors—for instance, Lit'' constructs $Term''$, not $Term'$. This problem can be addressed via polymorphic variants [23] or structural typing.

With polymorphic variants, we next run against *isorecursive types*. A *Term* is not equal to a record that can contain other terms, is only *isomorphic* to it, and data must be explicitly converted across isomorphic datatypes. Outside of DGP, this is a smaller problem because such isomorphisms are part of data constructors. But when using multiple decompositions, users need to combine multiple of these coercions, especially to convert between datatypes with different recursive structure like $Term'$ and $Term''$.

Similarly to some previous work (discussed in Sec. 3.1), we prototyped a Scala library which (a) encoded polymorphic variants and (b) automatically generated coercions between isomorphic datatypes using equal labels, relying on Amadio and Cardelli's algorithm to generate coercions [4, 12]. Thus, users need not write boilerplate converting among *Term*, $Term'$, $Term''$ and $Term'''$; more in general, we could generate conversion between datatype decompositions used in different DGP techniques [13, 25, 40, 41]. Yet, the resulting system was not satisfactory: these coercions had a runtime cost that in some cases was hard to remove. More importantly, users had to constantly invoke coercions by hand at the right point, or confront errors for type mismatches between morally identical types. We decided therefore that, instead of bending over backwards to please a typechecker, the typechecker should take pains to help its users by recognizing more type equivalences, as we describe next.

2.2. Our Approach

We address the problem described above by a novel typed λ-calculus. Our starting point is the higher-order polymorphic λ-calculus F_ω because we need type-level functions to express functors. To F_ω we add record and variant types and, crucially, equirecursive types, through a type-level fixed-point combinator $\mu_k :: (k \to k) \to k$. In the novel resulting calculus, F^μ_ω, the $Term'$ datatype and the *TermF* functor look as follows:

$$
\begin{array}{l}
\textbf{type } \mathit{TermF} = \\
\quad \lambda\tau.\ \langle \mathit{Lit} : \{\, n : \mathit{Int} \,\}, \quad \mathit{Abs} : \{\, x : \mathit{String}, \mathit{body} : \tau \,\}, \\
\qquad\quad \mathit{Var} : \{\, x : \mathit{String} \,\}, \mathit{App} : \{\, \mathit{fun} : \tau, \mathit{arg} : \tau \,\} \quad \rangle \\
\textbf{type } \mathit{Term}' = \mu\ \mathit{TermF} \\
\textbf{type } \mathit{Term} = \mathit{Term}'
\end{array}
$$

The functor is defined as type-level function; its fixed point yields $Term'$. We don't have a distinct construct for datatype definition, so we simply declare that $Term$ is equal to $Term'$.

We can define $Term''$ and $Term'''$ in the same way.

$$\begin{array}{l}\textbf{type } EvalCtx = \\ \quad \lambda\tau.\ \langle Lit\ : \{\, n : Int \,\},\quad Abs\ : \{\, x : String, body : Term \,\}, \\ \qquad\ Var : \{\, x : String \,\}, App : \{\, fun : \tau, arg : Term \,\} \quad \rangle \\ \textbf{type } Term'' = \mu\ EvalCtx\end{array}$$

$$\begin{array}{l}\textbf{type } VarTerm = \\ \quad \lambda\tau.\ \langle Lit\ : \{\, n : Int \,\}, Abs\ : \{\, x : String, body : Term \,\}, \\ \qquad\ Var : \{\, x : \tau \,\}, \quad App : \{\, fun : Term, arg : Term \,\}\rangle \\ \textbf{type } Term''' = VarTerm\ String\end{array}$$

Alternatively, to avoid redundancy we can freely refactor all these type constructors expressing them in terms of $TermBase$:

$$\begin{array}{l}\textbf{type } TermBase = \\ \quad \lambda\rho\ \sigma\ \tau.\ \langle Lit\ : \{\, n : Int \,\}, Abs\ : \{\, x : String, body : \sigma \,\}, \\ \qquad\ Var : \{\, x : \rho \,\}, \quad App : \{\, fun : \tau, arg : \sigma \,\} \quad \rangle \\ \textbf{type } TermF\ \ = \lambda\tau :: *.\ TermBase\ String\ \tau\ \tau \\ \textbf{type } Term'\ \ = \mu\ TermF \\ \textbf{type } Term\ \ = Term' \\ \textbf{type } EvalCtx\ \ = \lambda\tau :: *.\ TermBase\ String\ Term\ \tau \\ \textbf{type } VarTerm = \lambda\tau :: *.\ TermBase\ \tau\ Term\ Term\end{array}$$

2.3. Infinitary Type Equivalence

In F^{μ}_{ω}, $Term'$, $Term''$ and $Term'''$ are equal thanks to a powerful type equivalence relation based on infinitary λ-terms. Intuitively, we identify each recursive datatype $\mu\ F$ with its infinite expansion $F\ (F\ (F ...))$. Hence two datatypes are equal if their infinite expansions have the same variant-record structure and the same field types. This type equivalence extends the one developed by Amadio and Cardelli [4] that we used in our prototype (Sec. 2.1).[1] We are hence confident enough that, thanks to this type equivalence, different DGP techniques can inter-operate.

Instead of defining type equivalence through infinite structures and a type-equality relation formulated coinductively, it is metatheoretically simpler to extend type equality with the μ-unfolding rule $\mu\ F = F\ (\mu\ F)$, interpreted inductively as usual. The resulting

[1] In F^{μ}_{ω}, we use the extended Amadio-Cardelli algorithm to check type equivalence for equirecursive types; in our prototype, we used the same equivalence to recognize when two types would be isomorphic and synthesize a coercion between them.

weak type equality [16, 11] is however strictly weaker [16] and insufficient for our goals. As a minimal example, weak type equality cannot prove the following equations [16, 4, 1]:

$$\mu\ \alpha.\ \alpha \to Int = \mu\ \alpha.\ (\alpha \to Int) \to Int \tag{1}$$

$$\mu\ \alpha.\ \mu\ \beta.\ \alpha \to \beta = \mu\ \alpha.\ \alpha \to \alpha \tag{2}$$

Intuitively, the μ-unfolding rule does not alter the argument of μ, and the two sides of Eq. (1) differ exactly by the different arguments of μ, which no amount of unfolding will equalize.[2] Weak equivalence is sufficient to prove $Term' = Term'''$, but, crucially, $Term' = Term''$ requires infinite unfolding, because any finite amount of unfolding is insufficient to equate the different recursion structures. Hence we conclude that we need *strong type equality*, defined through infinite unfolding, for DGP with multiple simultaneous datatype decompositions.

2.4. DGP in Our Approach

After defining strong type equivalence, we can apply standard DGP techniques. For instance, *fv* is just a fold, and folds can be defined generically. To wit, compare the Haskell definition with the F^{μ}_{ω} version.

```
-- Haskell
fold :: (Functor f) ⇒ (f a → a) → Fix f → a
fold algebra = fix (λdoFold v →
  algebra (fmap doFold (unroll v)))
```

```
-- System F^μ_ω
type Functor f = ∀a b. (a → b) → (f a → f b)
fold : ∀f. (Functor f) → ∀a. (f a → a) → μ f → a
fold = Λ f :: ∗ → ∗. λfmap : Functor f.
         Λ a :: ∗. λalgebra : f a → a.
           fix (λdoFold : μ f → a.
                  λv : μ f.algebra (fmap doFold v))
```

Ignoring superficial differences, in F^{μ}_{ω} we omit invoking the isomorphism *unroll*, since the typechecker knows that $\mu\ f = f\ (\mu\ f)$.

Each of the code above is generic, but depends on an implementation of *fmap* for the relevant functor. Since this implementation is purely boilerplate, in GHC Haskell the programmer can ask the compiler to implement *fmap* through a **deriving** *Functor* clause. Similarly, an automatic implementation of the more general method *Traverse*

[2] Technically, this weakness is usually shown in settings without type constructors. We conjecture weak equality is still "too weak" even in combination with the β-rule, at least in $F^{\mu *}_{\omega}$ since its types still expand to regular trees like for λ_{μ}, and unlike with type-level recursion at K_3 kinds. This conjecture is non-trivial to prove because of the possibility of μ-unfolding in proofs of weak equality, but our attempts yielded no counterexample.

can be requested **deriving** *Traversable*.[3] To provide comparable support, we support traversable functors of arbitrary kinds through a boilerplate-generation mechanism for F^{μ}_{ω}, based on an extension of higher-kinded polytypism [29] (Sec. 6). In our prototype (Sec. 2.1), we found support for *traverse* sufficient to encode a variety of DGP techniques [13, 25, 40, 41].

In the rest of the paper, we demonstrate type soundness for F^{μ}_{ω}, and the decidability of type checking for a the subset $F^{\mu *}_{\omega}$, where μ is restricted to μ_{*}, so that it is only applicable to type-level functions of kind $* \to *$ and can thus only express *first-order* recursive types. This fragment is expressive enough to express all of our examples and the DGP techniques previously mentioned, but not to support nested datatypes (see Sec. 3.4.3). Further extensions to the decidability result appears difficult; a practical system with higher-kinded equirecursive types may require more hints regarding type equivalence from the user.

3. Related Work

We separate related work into three classes: (1) Approaches to synthesize datatype isomorphisms, (2) monomorphization, a technique to avoid the need for isomorphisms, (3) universe construction, the standard generic programming pattern in dependently typed languages, and (4) previous work on equirecursive types.

3.1. Synthesizing Isomorphisms

There are many approaches that try to avoid the boilerplate code that implements certain datatype isomorphisms. Many approaches to datatype genericity are based on the idea of a structural sums-of-products representation of datatypes. Such isomorphisms can be synthesized in Generic Haskell [5]. Recent work in this area has concentrated on a unique sum-of-products representation without nesting [19]. Such isomorphisms are not in the scope of this work; our approach is "nominal": names of labels matter and datatypes with different label names are never equal.

A *generic view* [30, 47] on a datatype T is another type T' together with coercions between T and T'. Generic views can be used to add a new datatype decomposition (and the corresponding isomorphisms) to a datatype, which makes it simpler to define generic functions that require a different view on the data. One supported view is the *fixed point view*, with which the pattern functor can be recovered from a datatype. More sophisticated isomorphisms involving fixed points, such as different functors with the same fixed points, are not supported.

The main difference of this work to all approaches to synthesize isomorphisms is that we strive for a powerful type equality relation which makes it unnecessary to define and apply isomorphisms.

[3] Its design is described at `http://ghc.haskell.org/trac/ghc/ticket/2953`.

3.2. Monomorphization

Monomorphization refers to the process of instantiating a polymorphic value. In the functor decomposition of datatypes, monomorphization means instantiating functor methods like *fmap* so that its type signature refers only to the original datatype, sparing us the need to create a fresh datatype for the functor. As an example, consider the *fmap* method of *TermF*.

$$fmap :: (a \to b) \to TermF\ a \to TermF\ b$$

Note that *Term* is isomorphic to (*TermF Term*). To get rid of the new datatype *TermF* in the signature of *fmap*, we set $a = b = Term$, and replace *TermF a* and *TermF b* by *Term*. The result is a computationally equivalent *fmap* definable in terms of the constructors of *Term* alone. The process is analogous for *fold*.

$$\begin{aligned}
&fmap :: (Term \to Term) \to Term \to Term \\
&fmap\ f\ (Lit\ n) \quad\ = Lit\ n \\
&fmap\ f\ (Var\ x) \quad\ = Var\ x \\
&fmap\ f\ (Abs\ x\ t) \ = Abs\ x\ (f\ t) \\
&fmap\ f\ (App\ t_1\ t_2) = App\ (f\ t_1)\ (f\ t_2) \\
&fold :: (Term \to Term) \to Term \to Term \\
&fold\ f\ t = f\ (fmap\ (fold\ f)\ t)
\end{aligned}$$

Monomorphization is a technique that shows up in several approaches to generic programming, including the lens library [36], *Compos* [13], and *Scrap-your-boilerplate* [38].

Monomorphization avoids the need for isomorphisms, since the monomorphized functions operate on the original algebraic datatype. However, the expressiveness of monomorphized functions is rather limited compared to the polymorphic versions. For instance, the *fold* above supports only recursive term transformations; it does not support the computations of free variables any more. Moreover, through monomorphizing the type signature of *fmap* and *fold*, the *free theorems* of their types no longer dictate their behaviors. In fact, these two very different methods have the same type signature. Nothing warns the user if it calls *fmap* with an algebra by mistake. Similarly, the methods of different functors may not be distinguished by type, risking unintentional misuse.

Furthermore, while monomorphization allows the decomposition of a single datatype into multiple functors (with the limitations described above), it does not allow using the same functor for the definition of multiple datatypes.

3.3. Universe Construction

In many dependently-typed languages, universe constructions [3, 6, 42, 43] allow defining a datatype of codes for a class $\mathcal{C}$ of types. Functions can be defined over every type $\tau \in \mathcal{C}$ by pattern-matching on the code of τ; boilerplate-generators such as *fmap* (Sec. 2.4) or *traverse* (Sec. 6) are definable thus without any special language support. Universe constructions are a promising direction of generic programming and has received much

attention in literature. However, the tyranny of the dominant functor—or the inflexibility of induction principles—persist in the presence of dependent types. Tackling them there would mean confronting the difficulties of coinductive reasoning inside a dependently typed language, difficulties yet to be resolved. Instead, we present dominant functors in the simplest system we could find, that is F^{μ}_{ω}.

3.4. Other Systems with Equirecursive Types

We survey recent systems with equirecursive types; these systems consider a recursive type and their expansions to be interchangeable in all contexts. While some such works discuss subtyping, we will look at them from the simpler perspective of type equivalence, which is sufficient for our purposes. We refer to Brandt and Henglein [12] for earlier work on equirecursive types.

Compared to the surveyed systems, our soundness result holds for the most general class of equirecursive F_{ω} types with a more liberal equivalence relation than those previously investigated. Our decidability result holds for F_{ω} types with first-order recursion, which corresponds to equirecursive F types sprinkled with type-level lambdas and applications.

3.4.1. Equirecursive Simple Types

Amadio and Cardelli [4], Brandt and Henglein [12] and Gapeyev et al. [22] (also in Pierce [45, Ch. 21]) investigated the system of *recursive simple types*, here indicated with λ^{μ}, shown in Fig. 1. Two recursive simple types are equivalent if and only if unrolling them indefinitely produce identical infinite trees. The same type equivalence can also be formulated without infinite unfoldings, using instead the rules of μ-folding interpreted coinductively, (see Fig. 1). This formulation is syntax-directed (technically, *invertible*), so it can be decided efficiently using a general decision procedure for coinductive relations. Both our type equivalence and decision procedure extend this theory, as we discuss in Sec. 5.2.1 and 5.3.1.

Recursive simple types differ from F^{μ}_{ω} types, because:

1. There is no type-level function, or any type-level computation beyond unrolling μ-types.

2. The μ-types are constrained syntactically to be *contractive*; those types that do not unfold to infinite trees are forbidden (e. g., $\mu\alpha.\ \alpha$), for reasons we discuss later.

Despite these differences, a significant part of the metatheory of recursive simple types can be reused for F^{μ}_{ω}. In fact, our proof is based on the presentation in Pierce [45]. However, while we still use the idea of infinite expansion, because of type-level computation its definition must be changed to use infinitary λ-calculus.

3.4.2. Equirecursive F Types

Glew [28] considered adding recursive types to System F. Recursive F types extend

$$
\begin{array}{rcll}
\tau^c & ::= & & \text{simple contractive type} \\
 & & \iota & \text{primitive type} \\
 & | & \tau^s \to \tau^s & \text{function type} \\
 & | & \mu x.\ \tau^c & \mu\text{-type}
\end{array}
$$

$$
\begin{array}{rcll}
\tau^s & ::= & & \text{simple recursive type} \\
 & & \alpha & \text{type variable} \\
 & | & \tau^c &
\end{array}
$$

$$\frac{}{\alpha \equiv \alpha} \qquad \text{(Eq-TVar)}$$

$$\frac{}{\iota \equiv \iota} \qquad \text{(Eq-Prim)}$$

$$\frac{[x \mapsto \mu x.\ \tau^c]\tau^c \equiv \tau^s}{\mu x.\ \tau^c \equiv \tau^s} \qquad \text{(Eq-}\mu_L\text{-Simple)}$$

$$\frac{\tau^s \equiv [x \mapsto \mu\ x.\ \tau^c]\tau^c \qquad \tau^s \text{ does not start with } \mu}{\tau^s \equiv \mu x.\ \tau^c} \qquad \text{(Eq-}\mu_R\text{-Simple)}$$

$$\frac{\tau_1^s \equiv \tau_2^s \qquad \tau_3^s \equiv \tau_4^s}{\tau_1^s \to \tau_3^s \equiv \tau_2^s \to \tau_4^s} \qquad \text{(Eq}\to\text{)}$$

Figure 1: The system of simple recursive types investigated in Amadio and Cardelli [4], Brandt and Henglein [12], Pierce [45], with type equivalence formulated coinductively, through congruence rules and rules for *μ-folding*. This formulation ensures rules are non-overlapping and thus syntax-directed.

recursive simple types as follows:

$$\tau^s = \ldots \mid \forall\alpha.\ \tau^s, \qquad\qquad \tau^c = \ldots \mid \forall\alpha.\ \tau^s.$$

In other words, universal quantification is added as another way to construct contractive types. Like simple types, recursive F types exclude type-level functions, type-level computation and non-contractive μ-types.

Glew interprets recursive F types as *binding trees*, or infinite F types in de Bruijn notation. De Bruijn indices are used to avoid the issue of name binding and α-equivalence. Name binding is present in F^{μ}_{ω} as well, namely in type-level lambdas. Following Czajka [18], we ignore the name binding issue, since standard solutions exist. Glew gave an $O(n^2)$ decision procedure for the equivalence of recursive F types, where n bounds the size of the types. Gauthier and Pottier [24] improve the algorithm to $O(n \log n)$ and generalized it to decide unifiability, so that languages with type inference (e. g., OCaml) may take advantage of recursive F types.

Colazzo and Ghelli [17] added recursive types to System $F_{<:}$. The result is similar to recursive F types, except universal quantifications may include subtype bounds: $\forall\alpha <: \tau_1.\ \tau_2$. Colazzo and Ghelli defined a decidable subtyping relation on recursive $F_{<:}$ types that relates μ-types and their expansions in all contexts, but they gave no infinitary interpretation.

3.4.3. Equirecursive K_3 Types

In modern terms, Solomon [50] considered recursive types that can have parameters of kind $*$, that is, recursive types of K_3 kinds [45, definition 30.4.1]. As discovered later, this allows expressing *nested datatypes* [10] such as perfect binary trees:

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{One}\ a \mid \mathit{Two}\ (\mathit{Tree}\ (a, a))$$

In F^{μ}_{ω}, *Tree* would be the fixed point of a higher-order type:

$$\begin{aligned}&\mu\ (\lambda \mathit{Tree} : * \to *.\\ &\qquad \lambda a : *.\ \langle \mathit{One} : \alpha, \mathit{Two} : \mathit{Tree}\ \{\mathit{fst} : \alpha, \mathit{snd} : \alpha\}\rangle)\end{aligned}$$

Solomon's types are defined by series of potentially recursive type synonyms with parameters and constructed by records, pointers and base types of kind $*$. Despite the lack of explicit lambdas, type-level computation is expressible through types calling each other in the bodies of their definitions.

Solomon showed that equivalence checking for equirecursive K_3 types reduces to equivalence checking for deterministic push-down automata, which Sénizergues proved later to be decidable [48]. Thus equirecursive typing is decidable for nested datatypes. Unfortunately, known algorithms to decide equivalence of deterministic push-down automata [32] are impractical because they have super-exponential time complexity in automaton size (in particular, the algorithms are primitive recursive, but their complexity is not elementary in the automaton size [33, 51]).

F^{μ}_{ω} supports fixed points of arbitrary kinds, but the decidable subset $F^{\mu *}_{\omega}$ only supports recursion for proper types (i.e., only allows using μ where $\kappa = *$), so types still expand to regular trees (see Sec. 5.3). We conjecture that, like for λ_{μ}, the type equivalence problem for $F^{\mu *}_{\omega}$ is still reducible to equivalence of regular languages, while for equirecursion at K_3 kinds goes significantly beyond regular languages; this would explain why supporting equirecursion at K_3 kinds is so much harder. So we exclude recursive K_3 types because of these disproportionate metatheoretic difficulties, and because they are just a small fragment of higher-kinded types.

3.4.4. OCaml-style Equirecursive Types

Im et al. [32] considered $\lambda^{\text{rec}}_{\text{abs}}$, a system with recursive K_3 types, OCaml-style modules and abstract types. They define a term language in addition to the type language and demonstrate type soundness despite the interaction between recursive and abstract types. Although no practical algorithm exists to decide the equivalence of K_3 types, Im et al.'s soundness result also applies to efficiently decidable fragments of $\lambda^{\text{rec}}_{\text{abs}}$. We share their concern for type soundness and follow a similar framework: Our type checking algorithm works only on recursive types of kind $*$, but our soundness result applies to F_{ω} with recursive types of arbitrary kinds.

The distinguishing feature of $\lambda^{\text{rec}}_{\text{abs}}$ is that non-contractive types (i. e., types that do not expand to infinite trees) are not completely forbidden. In fact, abstract types make it impossible to rule out non-contractive types syntactically; instantiating an abstract type may make other types non-contractive. For example, instantiating f by the identity type function produces the non-contractive type $\mu\ (\lambda\alpha.\ \alpha)$ in the type signature of *fold* (Sec. 4). This problem is present in both $\lambda^{\text{rec}}_{\text{abs}}$ and F^{μ}_{ω}. In $\lambda^{\text{rec}}_{\text{abs}}$, infinite proofs relating non-contractive types to every other type are forbidden by construction. In F^{μ}_{ω}, type equivalence is defined in terms of β-equivalence in infinitary λ-calculus, enabling us to reuse existing confluence and normalization results in our soundness proof.

3.4.5. Equirecursive F_{ω} Types

System F_{ω} with equirecursive types (and sometimes subtyping) has been considered in several papers. Bruce et al. [14] presented the syntax of a variant of F_{ω} with subtyping, recursive types, and some other features, but do not consider its metatheory. Hinze [29] considered a variant of F^{μ}_{ω}, but uses the weak type equivalence we discuss in Sec. 2.2, and does not discuss soundness or decidability. Abel [2] also considered a variant of F^{μ}_{ω} and did discuss its metatheory (without decidability of typechecking), but like Hinze he used weak type equivalence, which has a simpler metatheory. Abel's focus is however unrelated from ours (namely, automatic proofs of termination using sized types).

We will prove type soundness for F_{ω} with equirecursive types, but we will only describe an efficient typechecker for a sub-language, where recursive types may only have kind $*$. With recursive types of arbitrary kinds, equivalence between F_{ω} types corresponds to a form of coinductive program equivalence between simply typed λ-terms with a general fixed-point combinator.

K_3 types are a subset of general F^{μ}_{ω} types, and for the latter it is not known whether a sensible, decidable equivalence relation exists [20, section 3.4].

4. System F^{μ}_{ω}

In this section, we define the formal language we propose to support datatype-generic programming (as discussed in Sec. 2). The type signature of *fold*, which we have seen in Sec. 2.2, dictates which language features are necessary:

$$\mathit{fold} : \forall f.\ (\mathit{Functor}\ f) \to \forall a.\ (f\ a \to a) \to \mu\ f \to a$$

The signature of *fold* uses:

- a type-level function f and type-level application $f\ a$,
- universally quantified type variables f, a,
- recursive types, that is, fixed points $\mu\ f$ of arbitrary type functions f. As discussed, we want *equirecursive* types.

Therefore, we have designed System F^{μ}_{ω} combining all 3 features. Fig. 3 to 6 show its syntax, type and kind systems.

In our formal language F^{μ}_{ω}, datatype operations are expressed through records and variants, that are eliminated respectively through projections (Fig. 6, rule T-Proj) and pattern matching (rule T-Case). The language of types of F^{μ}_{ω} is a simply-typed λ-calculus Λ^{μ}, but shifted one level up, just like for F_{ω}. Instead of introducing type constructors (for instance $\to$ for function types or $\forall$ for universal types), we introduce corresponding primitives (Fig. 2). Equirecursive types deserve attention. While we prove type soundness for the language as presented, we can only prove that type checking is decidable for $F^{\mu *}_{\omega}$, where we restrict to recursive types of kind $*$, that is, if we restrict μ (Fig. 4) to the case $\kappa = *$, as discussed in Sec. 3.4.

The typing rule T-Eq (Fig. 6) relies on a notion of type equivalence; we will define it in Sec. 5.

In F^{μ}_{ω}, labels in records and variants are always written in a canonical (alphabetical) order; we will ignore this rule in examples, because label ordering can be canonicalized during desugaring.

$$\{\, x = 3, \mathit{body} = 5 \,\} ::= \{\, \mathit{body} = 5, x = 3 \,\}$$

We formalize the universal quantifier as a collection of type-level constants $\forall_{\kappa}$ indexed by the kind of the type being quantified over. This way, the universal quantifier is treated simply as yet another type-level constant. It is easy to see that our formulation of $\forall$ as a constant is inter-derivable with the standard formulation of $\forall \alpha :: \kappa.\ \tau$ as a syntactic construct [45, fig. 30-1]. When there is no confusion, we will omit the kind index of $\forall_{\kappa}$ and just write $\forall$.

ι	$::=$		Λ^μ constant
		$\to :: * \to * \to *$	function arrow
	$\mid$	$\forall_\kappa :: (\kappa \to *) \to *$	universal quantifier
	$\mid$	$\{\overline{l_i}\} :: \overline{* \to} *$	record type constructor
	$\mid$	$\langle \overline{l_i} \rangle :: \overline{* \to} *$	variant type constructor
	$\mid$	ι_0	*Int*, *Set*, *String* etc.

Labels in record/variant types must formally appear in lexicographic order. This way, types cannot differ only by the order of labels. We shall employ the following syntax sugar:

$$\{\overline{l_i : \tau_i}\} = \{\overline{l_i}\}\ \tau_1 \cdots \tau_n$$
$$\langle \overline{l_i : \tau_i} \rangle = \langle \overline{l_i} \rangle\ \tau_1 \cdots \tau_n$$

Figure 2: Λ^μ constants, the type-level language of F^μ_ω, together with their kinds.

κ	$::=$		kind
		$*$	kind of types
	$\mid$	$\kappa \to \kappa$	kind of type constructors

τ	$::=$		type (constructor)
		$\mu\ \tau$	recursive type
		ι	type-level constant
	$\mid$	α	type-level variable
	$\mid$	$\lambda\alpha :: \kappa.\ \tau$	type-level abstraction
	$\mid$	$\tau\ \tau$	type-level application

Γ	$::=$		typing context
		$\emptyset$	empty context
	$\mid$	$\Gamma, \alpha :: \kappa$	type variable binding
	$\mid$	$\Gamma, x : \tau$	term variable binding

Figure 3: Syntax of Λ^μ, the type-level language of F^μ_ω.

$$\frac{\Gamma \vdash \tau :: \kappa \to \kappa}{\Gamma \vdash \mu\ \tau :: \kappa} \quad \text{(K-Fix)}$$

$$\frac{\Gamma, \alpha :: \kappa_1 \vdash \tau :: \kappa_2}{\Gamma \vdash \lambda\alpha :: \kappa_1.\ \tau :: \kappa_1 \to \kappa_2} \quad \text{(K-Abs)}$$

$$\frac{\Gamma \vdash \tau_1 :: \kappa_2 \to \kappa_3 \qquad \Gamma \vdash \tau_2 :: \kappa_2}{\Gamma \vdash \tau_1\ \tau_2 :: \kappa_3} \quad \text{(K-App)}$$

$$\frac{\alpha :: \kappa \in \Gamma}{\Gamma \vdash \alpha :: \kappa} \quad \text{(K-Var)}$$

$$\frac{}{\Gamma \vdash \iota :: \kappa_\iota} \quad \text{(K-Const)}$$

Figure 4: Kinding rules. Kinds κ_ι for $\iota = \to, \forall, \{\overline{l_i}\}$ and $\langle\overline{l_i}\rangle$ are given next to their syntax definitions (Fig. 2).

t	::=	term
	c	constant
$\mid$	x	variable
$\mid$	$\lambda x : \tau.\ t$	abstraction
$\mid$	$t\ t$	application
$\mid$	$\Lambda\alpha :: \kappa.\ t$	type abstraction
$\mid$	$t\ [\tau]$	type application
$\mid$	$\{\overline{l_i = t_i}\}$	record introduction
$\mid$	$t.l_i$	record elimination (projection)
$\mid$	$\langle l_j = t\rangle$ **as** τ	variant introduction (injection)
$\mid$	**case** t **of** t	variant elimination

c	::=	constant
	$\mathit{fix}_\tau : (\tau \to \tau) \to \tau$	fixed-point combinator
$\mid$	$\cdots$	literals, arithmetic operators, etc.

Figure 5: Syntax of terms of F^μ_ω.

$$\frac{\Gamma \vdash \tau_c : *}{\Gamma \vdash c : \tau_c} \quad \text{(T-Const)}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \text{(T-Var)}$$

$$\frac{\Gamma, x : \sigma \vdash t : \tau \qquad \Gamma \vdash \sigma :: *}{\Gamma \vdash (\lambda x : \sigma.\ t) : \sigma \to \tau} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t : \sigma \to \tau \qquad \Gamma \vdash s : \sigma}{\Gamma \vdash t\ s : \tau} \quad \text{(T-App)}$$

$$\frac{\Gamma, \alpha :: \kappa \vdash t : \tau}{\Gamma \vdash (\Lambda \alpha :: \kappa.\ t) : \forall_\kappa\ (\lambda \alpha :: \kappa.\ \tau)} \quad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash t : \forall_\kappa\ \tau \qquad \Gamma \vdash \sigma :: \kappa}{\Gamma \vdash t\ [\sigma] : \tau\ \sigma} \quad \text{(T-TApp)}$$

$$\frac{\overline{\Gamma \vdash t_i : \tau_i}}{\Gamma \vdash \{\overline{l_i = t_i}\} : \{\overline{l_i : \tau_i}\}} \quad \text{(T-Record)}$$

$$\frac{\Gamma \vdash t : \{\overline{l_i : \tau_i}\}}{\Gamma \vdash t.l_j : \tau_j} \quad \text{(T-Proj)}$$

$$\frac{\Gamma \vdash t : \tau_j \qquad \Gamma \vdash \tau :: * \qquad \tau \equiv \langle \overline{l_i : \tau_i} \rangle}{\Gamma \vdash \langle l_j = t \rangle\ \mathbf{as}\ \tau : \tau} \quad \text{(T-Variant)}$$

$$\frac{\Gamma \vdash t : \langle \overline{l_i : \tau_i} \rangle \qquad \Gamma \vdash s : \{\overline{l_i : \tau_i \to \tau}\}}{\Gamma \vdash \mathbf{case}\ t\ \mathbf{of}\ s : \tau} \quad \text{(T-Case)}$$

$$\frac{\Gamma \vdash t : \sigma \qquad \Gamma \vdash \tau :: * \qquad \sigma \equiv \tau}{\Gamma \vdash t : \tau} \quad \text{(T-Eq)}$$

Figure 6: Typing rules of F^{μ}_{ω}. In T-Const with $c = \mathit{fix}$ we have $\tau_c = \tau \to \tau$ for arbitrary types τ.

5. Soundness and Type Checking of F^{μ}_{ω}

In this section, we discuss the metatheory of F^{μ}_{ω}, focusing on the more interesting parts. We are interested in proving both type soundness (through progress and preservation) for F^{μ}_{ω} and decidable typechecking for $F^{\mu *}_{\omega}$. The typing rules of F^{μ}_{ω} are the same standard as for F_{ω}; but the interesting changes are in the type equality relation, since we combine both β-equivalence $(\lambda x.t_1)t_2 \equiv [x \mapsto t_2]t_1$ and equirecursive types $\mu\ f \equiv f\ (\mu\ f)$. Hence, we need to combine the metatheory of System F_{ω} and of equirecursive types, in particular their theories of type equivalence.

5.1. Type Equivalence, Informally

In this subsection, we discuss informally type equivalence in F^{μ}_{ω}.

5.1.1. Equirecursive Simple Types

Before studying the interaction between equirecursive types and β-equivalence, we recapitulate key insights on equirecursive type equivalence on simple types alone (Sec. 3.4.1). Type equivalence ensures that μ-types are equal to their unfolding; that is, it satisfies the μ-unfolding equation $\mu\alpha.\tau = \tau[\alpha := \mu\alpha.\tau]$. However, as discussed (Sec. 2.3), μ-unfolding induces a *weak* type equivalence, which is insufficient to prove some equations, such as Eq. (1):

$$\mu\alpha.\alpha \to Int = \mu\alpha.(\alpha \to Int) \to Int.$$

Intuitively, proving this equation through μ-unfolding would require an infinite number of unfolding steps. To allow proving Eq. (1), one can define formally the infinite unfolding τ^{∞} of a type τ; unfolding τ infinitely often allows us to eliminate all occurrences of μ from τ^{∞}. Two types are then (strongly) equivalent if their infinite unfoldings are equal. Strong equivalence proves Eq. (1) because both sides unfold to

$$((\ldots \to Int) \to Int) \to Int.$$

However, we can't define the infinite unfolding for types such as $\mu\alpha.\alpha$, which are called *non-contractive* μ-types — intuitively, since each unfolding step returns the same term, the unfolding process that should construct the tree achieves no progress.[4] Without special care, non-contractive types can be proved equal to all other types [32], which is undesirable. Therefore, we must either treat them specially or forbid them altogether. In λ_{μ}, non-contractive types are excluded from the syntax of types: They have form $\ldots(\mu\alpha.\mu\alpha_1 \ldots \mu\alpha_n.\alpha)\ldots$ for $n \in \mathbb{N}$, which is illegal in the grammar in Fig. 1.

5.1.2. Extending Equirecursive Types to System F^{μ}_{ω}

To add equirecursive types to F^{μ}_{ω}, we need to extend infinite expansion to type abstractions and applications, and handle non-contractive types in a different way.

[4] In programming terms, the unfolding process is not productive [12].

First, we extend the infinite unfolding to F^{μ}_{ω}'s types. The type level of System F_{ω} is a simply-typed λ-calculus, to which we add the fixed-point combinator μ; hence, the infinite unfolding process will produce terms of an *infinitary λ-calculus*. For technical reasons, we use *untyped* infinitary λ-calculus: F_{ω}'s soundness proof requires a confluent reduction relation for types, and to the best of our knowledge no suitable one has been studied for infinitary simply-typed λ-calculus. Hence, infinite expansion also performs type erasure. Among the available formulations, we adopt the one by Endrullis and Polonsky [21] because it is coinductive and thus more perspicuous and convenient. We rely on the confluence proof by Czajka [18]; some proof steps in appendices are based on the earlier treatment by Kennaway et al. [35].

To expand μf even when f is not a variable, unlike in λ^{μ}, μ expands to a *function* $\mu^{\infty} = \lambda f.f\ (f\ (f\ \cdots))$, which iterates its argument an infinite number of times. To complete the unfolding process, first f must reduce to a λ-abstraction, and then β-reduction will complete the unfolding.

In F^{μ}_{ω} we must regard non-contractive types as syntactically valid, because they can be created during β-reduction. For instance, $\mu\, f$ is contractive, but β-reducing

$$(\lambda f :: * \to *.\ \mu\, f)\ (\lambda x :: *.\ x)$$

produces $\mu\ (\lambda x :: *.\ x)$. However, we treat non-contractive types specially:

- when defining equivalence, we ensure they are equal to no contractive type;
- during equivalence checking, we avoid expanding them, to prevent equating them with all other types as before.

Non-contractive types also threaten confluence of infinitary reduction. When $f :: * \to *$ is non-contractive, the infinite expansion $t = (\mu f)^{\infty}$ is a nasty infinite loop—in particular, each of its reducts has a redex at its root. In the literature, terms such as t are known as *root-active* terms. Infinitary reduction is not confluent unless we identify all such terms. To restore confluence, one uses *Böhm-reduction w.r.t. root-active terms*, that is, one allows root-active terms to reduce to a special symbol $\bot$, obtaining the *Berarducci-tree* [7] of a term, a variant of the better known *Böhm-tree*. Therefore, we define two types to be equivalent if their Berarducci-trees are. Contractive types are never equivalent to $\bot$; this is sufficient to obtain a satisfactory metatheory.

5.2. Type Soundness

We prove type soundness for F^{μ}_{ω}: Well-typed closed terms never get stuck during evaluation. The proof has the same architecture as the one for F_{ω} by Pierce [45, Chapter 30], because F^{μ}_{ω} is basically F_{ω} with the standard record/variant extensions and a non-standard type equivalence relation. Pierce proves preservation and progress for F_{ω}, following Wright and Felleisen [53], and the proof consists of 4 steps.

1. Lemmas 30.3.1–30.3.4 in Pierce [45]: the standard strengthening, weakening and substitution lemmas for F_{ω}. They carry over to F^{μ}_{ω} with minimal change, because they are unrelated to type equivalence.

$$
\begin{array}{rll}
\tau' & ::= & \text{infinitary lambda term} \\
& \bot & \text{bottom} \\
\| & \iota & \Lambda^{\mu} \text{ constant (Fig. 2)} \\
\| & \alpha & \text{variable} \\
\| & \lambda\alpha.\ \tau' & \text{abstraction} \\
\| & \tau'\ \tau' & \text{application}
\end{array}
$$

Figure 7: Λ^{∞}, the language of infinitary lambda terms with a special constant $\bot$. We reuse Greek letters for Λ^{∞} terms, because they correspond to types in F^{μ}_{ω}. Following Czajka [18], we use double bars $\|$ to signal coinductive definitions.

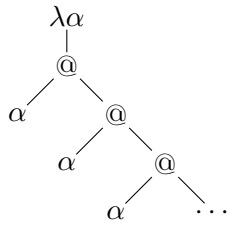

Figure 8: The infinitary lambda term $\mu^{\infty} = \lambda\alpha.\ \alpha\ (\alpha\ (\alpha\ \cdots))$.

2. Lemmas 30.3.5–30.3.11: a confluence proof for the type-level language of F_{ω}, which is a version of simply typed λ-calculus. We obtain an analogous result through interpreting F^{μ}_{ω} types as infinitary λ-terms, and reusing Czajka's confluent reduction.

3. Lemmas 30.3.12, 30.3.13 and theorem 30.3.14: Using confluence of F_{ω} types, Pierce proves an inversion lemma and uses it to establish preservation. We will replicate this step for F^{μ}_{ω}.

4. Lemma 30.3.15 and theorem 30.3.16: Progress is established through a canonical-forms lemma. We will replicate this step for F^{μ}_{ω}.

Step 2 contains the most important idea in our soundness proof, namely the connection between recursive types and infinitary lambda calculus. We detail this connection in Sec. 5.2.1. Steps 3 and 4 are more routine; we summarize the results in Sec. 5.2.2. All proofs are found in Appendix B.

5.2.1. Type Equivalence and Type-level Confluence

In this section, we formalize type equivalence following the ideas sketched in Sec. 5.1, making them precise. The view of F^{μ}_{ω} types as infinitary lambda terms, for example, is formalized as the infinite interpretation function below. Fig. 7 shows the target language Λ^{∞} of infinite interpretation, an untyped infinitary λ-calculus with a special symbol $\bot$.

$$\boxed{\tau' \Rightarrow_\beta \tau'} \qquad \boxed{\tau' \Rightarrow_\bot \tau'}$$

$$\frac{}{(\lambda\alpha.\ \sigma)\ \tau \Rightarrow_\beta [\alpha \mapsto \tau]\sigma}$$

$$\frac{\tau_1 \Rightarrow_\beta \tau_2}{(\lambda\alpha.\ \tau_1) \Rightarrow_\beta (\lambda\alpha.\ \tau_2)}$$

$$\frac{\sigma_1 \Rightarrow_\beta \sigma_2}{\sigma_1\ \tau \Rightarrow_\beta \sigma_2\ \tau}$$

$$\frac{\tau_1 \Rightarrow_\beta \tau_2}{\sigma\ \tau_1 \Rightarrow_\beta \sigma\ \tau_2}$$

$$\frac{\tau \neq \bot \qquad \tau \text{ is root-active (Definition 4)}}{\tau \Rightarrow_\bot \bot}$$

Figure 9: Rules for β- and Böhm-contractions according to Czajka [18]: β-contraction is the relation derivable by $\Rightarrow_\beta$ rules; Böhm-contraction $\Rightarrow_{\beta\bot}$ is the relation derivable by interlacing $\Rightarrow_\beta$ and $\Rightarrow_\bot$ rules.

Definition 1 (infinite interpretation)**.** Let $\tau \in \Lambda^\mu$ be a type of F^μ_ω. The *infinite interpretation* $\tau^\infty \in \Lambda^\infty$ is the infinitary untyped λ-term obtained from τ by erasing kind annotations and replacing each occurrence of $\mu\ \sigma$ by the application $\mu^\infty\ \sigma^\infty$, where μ^∞ is the infinite λ-term in Fig. 8:

$$\begin{aligned} (\lambda\alpha :: \kappa.\ \tau)^\infty &= \lambda\alpha.\ \tau^\infty & \qquad (\iota)^\infty &= \iota \\ (\sigma\ \tau)^\infty &= \sigma^\infty\ \tau^\infty & (\alpha)^\infty &= \alpha \\ (\mu\ \tau)^\infty &= \mu^\infty\ \tau^\infty \end{aligned}$$

Definition 2 (type equivalence)**.** Two F^μ_ω types σ, τ are equivalent, written $\sigma \equiv \tau$, if their infinite interpretations σ^∞ and τ^∞ are Böhm-equivalent (Definition 5).

To define Böhm-equivalence precisely, we need the notion of β-contraction, Böhm-contraction and root-active terms from Czajka [18].[5] The definitions of β- and Böhm contraction are inductive; their redexes must occur at finite depth. Following Czajka, we ignore the issue of α-conversion, as it has standard solutions.

Definition 3 (β-contraction and reduction)**.**

[5]For clarity, we write contraction and reduction relations using $\Rightarrow$ instead of Czajka's $\rightarrow$, which we use for function types.

$$\boxed{\tau' \Rightarrow_\beta^\infty \tau'}$$

$$\frac{\tau \Rightarrow_\beta^* \iota}{\tau \Rightarrow_\beta^\infty \iota} \qquad (\beta\text{-Const})$$

$$\frac{\tau \Rightarrow_\beta^* \alpha}{\tau \Rightarrow_\beta^\infty \alpha} \qquad (\beta\text{-Var})$$

$$\frac{\sigma \Rightarrow_\beta^* (\lambda\alpha.\ \tau) \qquad \tau \Rightarrow_\beta^\infty \tau'}{\sigma \Rightarrow_\beta^\infty (\lambda\alpha.\ \tau')} \qquad (\beta\text{-Abs})$$

$$\frac{\sigma \Rightarrow_\beta^* \tau_1\ \tau_2 \qquad \tau_1 \Rightarrow_\beta^\infty \tau_1' \qquad \tau_2 \Rightarrow_\beta^\infty \tau_2'}{\sigma \Rightarrow_\beta^\infty \tau_1'\ \tau_2'} \qquad (\beta\text{-App})$$

Figure 10: Parallel multistep β-reduction $\Rightarrow_\beta^\infty$ according to Czajka [18], defined coinductively. The relation $\Rightarrow_\beta^*$ is the reflexive transitive closure of β-contraction $\Rightarrow_\beta$ (Fig. 9).

- The (single-step) *β-contraction* relation $\Rightarrow_\beta$ is defined inductively by the $\Rightarrow_\beta$ rules in Fig. 9.
- *Parallel multistep β-reduction* is the relation $\Rightarrow_\beta^\infty$ defined coinductively in Fig. 10.

We call $\Rightarrow_\beta^\infty$ parallel multistep β-reduction because it permits reduction at an infinite number of locations in a term, but at each location permits only a finite number of β-contraction steps.

Root-active terms are $\bot$ and those that can always reduce to β-redexes by parallel multistep β-reduction. This intuition is obtained by simplifying Definition 2 of Czajka [18].

Definition 4 (root-activeness)**.** An infinitary λ-term σ is *root-active* if either $\sigma = \bot$, or else $\sigma \Rightarrow_\beta^\infty \tau$ implies $\tau \Rightarrow_\beta^\infty (\lambda\alpha.\ \tau_0)\ \tau_1$ for some τ_0, τ_1.

Definition 5 (Böhm-contraction, reduction [18] and equivalence)**.**

- The (single-step) *Böhm contraction* relation $\Rightarrow_{\beta\bot}$ is defined inductively by interlacing $\Rightarrow_\beta$ and $\Rightarrow_\bot$ rules in Fig. 9.
- *Parallel multistep Böhm-reduction*, or simply *Böhm reduction*, is the relation $\Rightarrow_{\beta\bot}^\infty$ on infinitary λ-terms defined coinductively in Fig. 11.
- Two infinitary λ-terms σ_1, σ_2 are *Böhm equivalent*, written $\sigma_1 \equiv_{\beta\bot} \sigma_2$, if there exists a term τ such that both $\sigma_1 \Rightarrow_{\beta\bot}^\infty \tau$ and $\sigma_2 \Rightarrow_{\beta\bot}^\infty \tau$.

$$\boxed{\tau' \Rightarrow^{\infty}_{\beta\bot} \tau'}$$

$$\frac{\tau \Rightarrow^{*}_{\beta\bot} \bot}{\tau \Rightarrow^{\infty}_{\beta\bot} \bot} \quad \text{(B-Bot)}$$

$$\frac{\tau \Rightarrow^{*}_{\beta\bot} \iota}{\tau \Rightarrow^{\infty}_{\beta\bot} \iota} \quad \text{(B-Const)}$$

$$\frac{\tau \Rightarrow^{*}_{\beta\bot} \alpha}{\tau \Rightarrow^{\infty}_{\beta\bot} \alpha} \quad \text{(B-Var)}$$

$$\frac{\sigma \Rightarrow^{*}_{\beta\bot} (\lambda\alpha.\ \tau) \qquad \tau \Rightarrow^{\infty}_{\beta\bot} \tau'}{\sigma \Rightarrow^{\infty}_{\beta\bot} (\lambda\alpha.\ \tau')} \quad \text{(B-Abs)}$$

$$\frac{\sigma \Rightarrow^{*}_{\beta\bot} \tau_1\ \tau_2 \qquad \tau_1 \Rightarrow^{\infty}_{\beta\bot} \tau_1' \qquad \tau_2 \Rightarrow^{\infty}_{\beta\bot} \tau_2'}{\sigma \Rightarrow^{\infty}_{\beta\bot} \tau_1'\ \tau_2'} \quad \text{(B-App)}$$

Figure 11: Böhm-reduction $\Rightarrow^{\infty}_{\beta\bot}$ according to Czajka [18], defined coinductively. The relation $\Rightarrow^{*}_{\beta\bot}$ is the reflexive transitive closure of Böhm-contraction $\Rightarrow_{\beta\bot}$ (Fig. 9).

Böhm-reduction is transitive and confluent, so the definition of Böhm-equivalence above is an actual equivalence relation.

Lemma 6. *Böhm-reduction $\Rightarrow^{\infty}_{\beta\perp}$ is transitive.*

Theorem 7 (confluence of Böhm-reduction [18])**.** *If $\sigma \Rightarrow^{\infty}_{\beta\perp} \tau_1$ and $\sigma \Rightarrow^{\infty}_{\beta\perp} \tau_2$, then there exists τ_3 such that $\tau_1 \Rightarrow^{\infty}_{\beta\perp} \tau_3$ and $\tau_2 \Rightarrow^{\infty}_{\beta\perp} \tau_3$.*

Corollary 8.

1. *Böhm-equivalence is reflexive, symmetric and transitive on infinitary λ-terms.*
2. *Type equivalence of F^{μ}_{ω} is reflexive, symmetric and transitive.*

The relation between type equivalence and Böhm reduction is most significant in the *shape-preservation* lemma, which implies that function types are never equivalent to records, and universal types are never equivalent to variants. An analogous statement is Lemma 30.3.12 in Pierce [45]. The shape preservation lemma is important in proving progress and preservation properties of F^{μ}_{ω}, as well as the decidability of typechecking in $F^{\mu *}_{\omega}$.

Lemma 9 (preservation of shape under Böhm equivalence)**.** *If $\iota\ \sigma_1 \cdots \sigma_n \equiv \iota'\ \tau_1 \cdots \tau_n$ as finite F^{μ}_{ω} types, then $\iota = \iota'$ and $\sigma_i \equiv \tau_i$ for all $i \in 1..n$.*

As exemplified in Sec. 2.2, Böhm equivalence is powerful. Here we show a further example, which unlike earlier ones goes beyond the decidable subset of F^{μ}_{ω}. The following definitions of polymorphic lists are intuitively equivalent.

$$\textbf{type } \mathit{List}_1 = \lambda\alpha :: *.\ \mu\ (\lambda\beta :: *.\ \langle \mathit{nil} : \alpha, \mathit{cons} : \beta \rangle)$$
$$\textbf{type } \mathit{List}_2 = \mu\ (\lambda\gamma :: * \to *.\ \lambda\alpha :: *.\ \langle \mathit{nil} : \alpha, \mathit{cons} : \gamma\ \alpha \rangle)$$

In F^{μ}_{ω}, List_1 and List_2 are actually equivalent types, because their infinite interpretations Böhm-reduce to the same infinitary λ-term:

$$\lambda\alpha :: *.\ \langle \mathit{nil} : \alpha, \mathit{cons} : \langle \mathit{nil} : \alpha, \mathit{cons} : \cdots \rangle \rangle.$$

5.2.2. Evaluation, Preservation and Progress

We use a standard call-by-name semantics of F^{μ}_{ω}. Since adding equirecursive types does not affect either the definition of values or the evaluation rules, most evaluation rules are pretty standard and are listed in Fig. 16 on page 39.

The preservation and progress theorem of F^{μ}_{ω} are analogous to Theorems 30.3.14 and 30.3.16 of Pierce [45], both in statement and in proof. Together they imply that whenever progress and preservation hold for constants, no closed, well-typed F^{μ}_{ω} term ever gets stuck.

Theorem 10 (preservation)**.** *Suppose all* E-Delta *rules preserve typing. If $\Gamma \vdash t : \tau$ and $t \Rightarrow t'$, then $\Gamma \vdash t' : \tau$.*

Theorem 11 (progress)**.** *Suppose* E-Delta *rules satisfy progress in the following sense.*

> *If s is a closed, well-typed term of the form $c\ v$, $c\ [\tau]$, $c.l$,* **case** c **of** v*, or* **case** v **of** c*, then s is reducible by an* E-Delta *rule.*

Let t_0 be a closed, well-typed term. Then either t_0 is a value or there exists t_0' such that $t_0 \Rightarrow t_0'$.

5.3. Decidability of Type Checking First-order Recursive Types

As discussed, $F_\omega^{\mu*}$ is the subset of F_ω^μ obtained by restricting the kind of recursive types to $*$. Formally, the kinding rule K-Fix is restricted on μ as follows:

$$\frac{\Gamma \vdash \tau : * \to *}{\Gamma \vdash \mu\ \tau : *} \qquad \text{(K-Fix*)}$$

In this section, we show that typechecking $F_\omega^{\mu*}$ is decidable. The architecture of a type checker for $F_\omega^{\mu*}$ is quite similar to the one for F_ω [45]. It is defined by a set of syntax-directed, *algorithmic* typing rules, which synthesize the type τ from the typing context Γ and the term t such that $\Gamma \vdash t : \tau$ holds. We list the algorithmic typing rules in Appendix C. Here we will only discuss the two subroutines significantly different from an F_ω type checker: deciding type equivalence (Sec. 5.3.1), and discovering type arguments for type-level constants (Sec. 5.3.2). These subroutines correspond to changed parts of the soundness proof, that is, respectively, to Corollary 8 and Lemma 9.

One may attribute the decidability of $F_\omega^{\mu*}$ to the relative simplicity of its types: Their infinite normal forms are *regular trees* [31], that is, each has only a finite number of distinct subtrees [45, Def. 21.7.2]. This is provable by applying section 21.9 of Pierce [45] to $\mathrm{NF}^{\mu*}$ (Theorem 59).

5.3.1. Deciding Type Equivalence

We defined type equivalence as Böhm-equivalence $\equiv_{\beta\bot}$ of types interpreted as terms of infinitary λ-calculus Λ^∞ (Definition 2). Write $\Lambda^{\mu*}$ for the type-level language of $F_\omega^{\mu*}$; then type equivalence is captured in the following diagram.

$$\Lambda^{\mu*}/\equiv \xrightarrow{\qquad (\cdot)^\infty \qquad} \Lambda^\infty/\equiv_\beta^\infty$$

Two components of type equivalence resist algorithmic verification:

1. checking β-equivalence of infinite terms, and
2. detecting root-active terms (Definition 4).

Both problems become decidable when we restrict recursive types to kind $*$. Since recursive types $\mu\ \tau$ may not occur at the operator position of type-level applications, reducing the β-redex $\mu^\infty\ \tau^\infty$ (cf. Fig. 8) never produces new β-redexes. As a result, by β-normalizing

$F_\omega^{\mu*}$ types, we essentially obtain finite representations of normal forms with respect to Böhm-reduction,[6] where all remaining redexes come from subterms $\mu\ \sigma$. Those finite normal forms allow us to verify β-equivalence by traditional algorithms for simple recursive types (Sec. 3.4.1), and detect root-active terms by checking contractiveness, for which (at this point) we can reuse what is essentially the standard definition (Definition 13). Through standard techniques, we characterize the languages of normal forms for $\Lambda^{\mu*}$ and Λ^∞ through their grammars.

Definition 12 (normal form languages, μ-equivalence, infinite expansion)**.**

- $\mathrm{NF}^{\mu*}$ is the language of $F_\omega^{\mu*}$ types in β-normal form (that is, without β-redexes), defined by the nonterminal m in Fig. 12.
- NF^∞ is the language of infinitary λ-terms in Böhm-normal form (that is, without Böhm-redexes), defined by the nonterminal m' in Fig. 12.
- The relation $\equiv_\mu$ on $\mathrm{NF}^{\mu*}$ terms, called *μ-equivalence*, is defined coinductively in Fig. 13. Again, we ignore the issue of α-conversion.
- Each $m \in \mathrm{NF}^{\mu*}$ has an *infinite expansion* $\mathrm{Ex}(m) \in \mathrm{NF}^\infty$ as defined inductively in Fig. 14. The syntactic *contractiveness* criterion is specified in Definition 13.

The μ-equivalence relation is essentially an extension of the type equivalence defined in Fig. 1; rules (Eq-μ_L) and (Eq-μ_R) are reformulations of (Eq-μ_L-Simple) and (Eq-μ_R-Simple); function types need no special handling, because $\to$ is simply treated as a primitive type constructor.

Definition 13. An $\mathrm{NF}^{\mu*}$ term m is *non-contractive* if

$$m = \mu\ (\lambda\alpha_1 :: *.\ \mu\ (\lambda\alpha_2 :: *.\ (\cdots(\mu\ (\lambda\alpha_k :: *.\ \alpha_i))\cdots)))$$

for some $i \in 1..k$. The term m is *contractive* if it is not non-contractive.

The $\equiv_\mu$ rules have extra conditions such as "m does not start with μ" in order to make $\equiv_\mu$ an *invertible* relation, i. e., each judgment $m_1 \equiv_\mu m_2$ has a *unique* derivation tree. Theorem 21.6.2 and Definition 21.6.3 of Pierce [45] present gfp^s, an algorithm that decides coinductively-defined finite-state invertible relations. We use gfp^s to decide $\equiv_\mu$. The algorithm and its termination property are discussed in Appendix D.4.

To decide type equivalence in $F_\omega^{\mu*}$, we decide $\equiv_\mu$ on $\mathrm{NF}^{\mu*}$ terms instead. The strategy is justified in the following theorem.

Theorem 14. *Let σ_1, σ_2 be $F_\omega^{\mu*}$ types with m_1, m_2 as their β-normal forms. Then $\sigma_1 \equiv \sigma_2$ if and only if $m_1 \equiv_\mu m_2$.*

[6] In particular, we use Böhm-reduction w.r.t root-active terms; normal forms for this variant of Böhm-reduction are called *Berarducci-trees*, while normal forms according to usual Böhm-reduction are the better-known Böhm-trees.

$$
\begin{array}{lcll}
m & ::= & & \mathrm{NF}^{\mu *} \text{ term} \\
 & & n & \text{finite neutral term} \\
 & \mid & \lambda\alpha :: \kappa.\ m & \text{annotated abstraction} \\
 & & & \\
n & ::= & & \text{finite neutral term} \\
 & & \iota & \Lambda^{\mu} \text{ constant (Fig. 2)} \\
 & \mid & \alpha & \text{variable} \\
 & \mid & n\ m & \text{application} \\
 & \mid & \mu\ n & \text{fixed-point of neutral term} \\
 & \mid & \mu\ (\lambda\alpha :: *.\ n) & \text{fixed-point of abstraction} \\
 & & & \\
m' & ::= & & \mathrm{NF}^{\infty}\text{-term (Berarducci-tree)} \\
 & & n' & \text{infinite neutral term} \\
 & \parallel & \lambda\alpha.\ m' & \text{unannotated abstraction} \\
 & & & \\
n' & ::= & & \text{infinite neutral term} \\
 & & \bot & \text{bottom} \\
 & \parallel & \iota & \Lambda^{\mu} \text{ constant (Fig. 2)} \\
 & \parallel & \alpha & \text{variable} \\
 & \parallel & n'\ m' & \text{application}
\end{array}
$$

Figure 12: Inductively-defined syntax of $\mathrm{NF}^{\mu *}$-terms m, and coinductively-defined syntax of NF^{∞}-terms m'. As in Czajka [18], double vertical bars signal coinductive definitions.

$$\boxed{m \equiv_\mu m}$$

$$\frac{}{\alpha \equiv_\mu \alpha} \quad \text{(Eq-TVar)}$$

$$\frac{}{\iota \equiv_\mu \iota} \quad \text{(Eq-Prim)}$$

$$\frac{n_1 \equiv_\mu n_2 \qquad m_1 \equiv_\mu m_2}{n_1\ m_1 \equiv_\mu n_2\ m_2} \quad \text{(Eq-AppCong)}$$

$$\frac{m_1 \equiv_\mu m_2}{\lambda\alpha.\ m_1 \equiv_\mu \lambda\alpha.\ m_2} \quad \text{(Eq-}\xi\text{)}$$

$$\frac{n_1\ (\mu\ n_1) \equiv_\mu m_2}{\mu\ n_1 \equiv_\mu m_2} \quad \text{(Eq-}\mu_L\text{-Neutral)}$$

$$\frac{m_1 \equiv_\mu n_2\ (\mu\ n_2) \qquad m_1 \text{ does not start with } \mu}{m_1 \equiv_\mu \mu\ n_2} \quad \text{(Eq-}\mu_R\text{-Neutral)}$$

$$\frac{[\alpha \mapsto \mu\ (\lambda\alpha :: *.\ n_1)]n_1 \equiv_\mu m_2 \qquad \alpha \text{ is contractive in } n_1}{\mu\ (\lambda\alpha :: *.\ n_1) \equiv_\mu m_2} \quad \text{(Eq-}\mu_L\text{)}$$

$$\frac{m_1 \equiv_\mu [\alpha \mapsto \mu\ (\lambda\alpha :: *.\ n_2)]n_2 \qquad \begin{array}{l}\alpha \text{ contractive in } n_2 \\ m_1 \text{ does not start with } \mu\end{array}}{m_1 \equiv_\mu \mu\ (\lambda\alpha :: *.\ n_2)} \quad \text{(Eq-}\mu_R\text{)}$$

$$\frac{\mu\ (\lambda\alpha : *.\ n_1) \text{ and } \mu\ (\lambda\alpha : *.n_2) \text{ are non-contractive}}{\mu\ (\lambda\alpha : *.\ n_1) \equiv_\mu \mu\ (\lambda\alpha : *.\ n_2)} \quad \text{(Eq-}\mu\bot\text{)}$$

Figure 13: Coinductive rules of μ-equivalence.

$$\boxed{\mathrm{Ex}(m) = m'}$$

$$\begin{array}{lll} \mathrm{Ex}(\iota) = \iota & \mathrm{Ex}(n\ m) = \mathrm{Ex}(n)\ \mathrm{Ex}(m) & \mathrm{Ex}(\lambda\alpha :: \kappa.\ m) = \lambda\alpha.\ \mathrm{Ex}(m) \\ \mathrm{Ex}(\alpha) = \alpha & \mathrm{Ex}(\mu\ n) = \mathrm{Ex}(n)\ \mathrm{Ex}(\mu\ n) & \end{array}$$

$$\mathrm{Ex}(\mu\ (\lambda\alpha :: *.\ n)) = \begin{cases} \mathrm{Ex}([\alpha \mapsto \mu\ (\lambda\alpha :: *.\ n)]n) & \text{if } \mu\ (\lambda\alpha :: *.\ n) \text{ is contractive,} \\ \bot & \text{if } \mu\ (\lambda\alpha :: *.\ n) \text{ is non-contractive.} \end{cases}$$

Figure 14: Infinite expansion of $m \in \mathrm{NF}^{\mu *}$ into Berarducci-trees $\mathrm{Ex}(m) \in \mathrm{NF}^{\infty}$.

Theorem 14 is proven in two steps. First we show that infinite expansion Ex captures exhaustive Böhm-reduction $\Rightarrow^{\infty}_{\beta\bot}$, then we show μ-equivalent terms to be exactly those expanding to the same infinite terms in NF^{∞}.

Lemma 15 (commuting diagram)**.** *Let m be the β-normal form of the $F^{\mu *}_{\omega}$-type σ. Then $\sigma^{\infty} \Rightarrow^{\infty}_{\beta\bot} \mathrm{Ex}(m)$.*

$$\begin{array}{ccc} \sigma \in \Lambda^{\mu *} & \xrightarrow{\Rightarrow^{*}_{\beta}} & m \in \mathrm{NF}^{\mu *} \\ {\scriptstyle (\cdot)^{\infty}}\downarrow & & \downarrow{\scriptstyle \mathrm{Ex}(\cdot)} \\ \sigma^{\infty} \in \Lambda^{\infty} & \xrightarrow[\Rightarrow^{\infty}_{\beta\bot}]{} & \mathrm{Ex}(m) \in \mathrm{NF}^{\infty} \end{array}$$

Lemma 16 (μ-expansion)**.** *$m_1 \equiv_{\mu} m_2$ if and only if $\mathrm{Ex}(m_1) = \mathrm{Ex}(m_2)$.*

The remaining proof of Theorem 14 is straightforward: $\sigma_1 \equiv \sigma_2$ iff $\sigma_1^{\infty} \equiv^{\infty}_{\beta\bot} \sigma_2^{\infty}$ iff $\mathrm{Ex}(m_1) = \mathrm{Ex}(m_2)$ iff $m_1 \equiv_{\mu} m_2$.

Pottier [46] already mentioned the idea of reducing types to β-normal forms and reusing algorithms for comparing recursive types, and conjectured that they'd work. We refine and substantiate this conjecture, clarifying some subtle points. In particular, the equivalence checking rules in Fig. 13 needs some extra rules to handle fixed points and unreduced applications of neutral terms. $F^{\mu *}_{\omega}$ type operators can be universally quantified, higher-kinded type variables, so even normal forms can contain applications.

5.3.2. Discovering Type Arguments for Type-level Constants

To decide whether a *simply typed* λ-abstraction $\lambda x : \sigma.\ t$ has type τ, a typechecker must first check that τ is a function type $\sigma_1 \to \sigma_2$, and then verify that $\sigma_1 = \sigma$ and σ_2 is the type of t. In F_{ω} and F^{μ}_{ω}, however, $\lambda x : \sigma.\ t$ may have type τ even if τ is not a function type—it needs only be *equivalent* to a function type. Similar problems arise not just for λ

and $\to$, but for the introduction and elimination forms of all other type constants. Hence, we need a decision procedure for the following question:

> Is a well-kinded $F_\omega^{\mu *}$ type τ equivalent to the application of some type constant ι to types $\sigma_1, \ldots, \sigma_k$? In other words, does $\tau \equiv \iota\ \sigma_1 \cdots \sigma_k$ hold? If it does, then compute k, ι, $\sigma_1, \cdots, \sigma_k$.

In F_ω, a decision procedure for this question only needs to normalize type τ and verify if the result is literally of form $\iota\ \sigma_1 \cdots \sigma_k$. In F_ω^μ, however, we need to handle additional cases for the β-normal forms of types, namely those starting with μ. We deal with the new cases via the following lemma, which is related to Lemma 21.8.6 in Pierce [45].

Lemma 17. *Let $m_1 \in \mathrm{NF}^{\mu *}$ be a contractive $F_\omega^{\mu *}$ type in β-normal form such that $\Gamma \vdash m_1 :: \kappa$. Then there exists $m_2 \in \mathrm{NF}^{\mu *}$ computable from m_1 such that $m_2 \equiv_\mu m_1$, $\Gamma \vdash m_2 :: \kappa$, and m_2 does not start with μ.*

The type m_2 is computed from m_1 by unrolling μ at the top level until a non-recursive type is encountered.

To discover whether $\tau \equiv \iota\ \sigma_1 \cdots \sigma_k$, we normalize τ to $m_1 \in \mathrm{NF}^{\mu *}$. If m_1 is non-contractive, then τ cannot be equivalent to $\iota\ \sigma_1 \cdots \sigma_k$, since the latter is not root-active. If m_1 is contractive, then compute the equivalent type m_2 not starting with μ. Since $F_\omega^{\mu *}$ recursive types have kind $*$, the final type operator n of m_2 is either a constant a variable, or a lambda abstraction. If $n = \iota$, then $\tau \equiv \iota\ \sigma_1 \cdots \sigma_k$ and we can extract $k, \iota, \sigma_1, \ldots, \sigma_k$ by examining m_2. If n is a variable or a lambda abstraction, then τ is not equivalent to any type of the form $\iota\ \sigma_1 \cdots \sigma_n$. Details are in Appendix D.3.

6. From Type Functions to Traversable Functors

A traversable functor $\tau : * \to *$ admits the method

$$\begin{aligned} \mathit{traverse}\langle\tau\rangle : \forall G :: * \to *.\ \mathit{Applicative}\ G \to \\ \forall \alpha_1 \alpha_2.\ (\alpha_1 \to G\ \alpha_2) \to \tau\ \alpha_1 \to G\ (\tau\ \alpha_2) \end{aligned}$$

satisfying certain laws [34, 39]. Traversable functors are a powerful abstraction for datatype operations [27]. The formalism of datatypes in F_ω^μ makes it possible to express generic programming combinators such as `compos` [13], `uniplate` [41], and `gmapT`/`gmapQ`/`gmapM` [38] as instances of *traverse*; details are left as an exercise for the reader.

Despite its power, $\mathit{traverse}\langle\tau\rangle$ can be generated automatically for types designating locations in a datatype built from records, variants, applications, λ and μ. Fig. 15 displays a traversal-generating macro in Hinze's notation of polytypic values [29]. Type arguments make the macro look harder than it really is. To reproduce Fig. 15, programmers need only ask themselves how *traverse* should behave on records, variants and μ-types; the other constructs are handled by a version of the binary parametricity transformation [8, 9]. Due to space constraint, we defer further discussions to Appendix E.

$$\textbf{type}\ \mathit{Applicative}\ G = \{\mathit{pure} : \forall\alpha.\ \alpha \to G\ \alpha, \mathit{call} : \forall\alpha\beta.\ G\ (\alpha \to \beta) \to G\ \alpha \to G\ \beta\}$$
$$\textbf{type}\ \mathit{Traverse}\langle\tau :: \kappa\rangle = \forall G.\ \mathit{Applicative}\ G \to \mathit{Trav}\langle\kappa\rangle\ \tau\ \tau$$

$$\textbf{type}\ \mathit{Trav}\langle * \rangle = \lambda\alpha\beta :: *.\ \alpha \to G\ \beta$$
$$\textbf{type}\ \mathit{Trav}\langle\kappa_1 \to \kappa_2\rangle = \lambda f g :: \kappa_1 \to \kappa_2.\ \forall\alpha\beta : \kappa_1.\mathit{Trav}\langle\kappa_1\rangle\ \alpha\ \beta \to \mathit{Trav}\langle\kappa_2\rangle\ (f\ \alpha)\ (g\ \beta)$$

$$\mathit{traverse}\langle\tau\rangle : \mathit{Traverse}\langle\tau :: \kappa\rangle$$
$$\mathit{traverse}\langle\tau\rangle = \Lambda G :: * \to *.\ \lambda g : \mathit{Applicative}\ G.\ \mathit{trav}\langle\tau\rangle$$

$$\begin{aligned}
\mathit{trav}\langle\alpha\rangle &= p_\alpha \\
\mathit{trav}\langle\lambda\alpha :: \kappa_\alpha.\ \sigma\rangle &= \Lambda\alpha_1 :: \kappa_\alpha.\ \Lambda\alpha_2 : \kappa_\alpha.\ \lambda p_\alpha : \mathit{Trav}\langle\kappa_\alpha\rangle\ \alpha_1\ \alpha_2.\ \mathit{trav}\langle\sigma\rangle \\
\mathit{trav}\langle\sigma\ \tau\rangle &= \mathit{trav}\langle\sigma\rangle\ [\mathrm{rename}_1(\tau)]\ [\mathrm{rename}_2(\tau)]\ \mathit{trav}\langle\tau\rangle \\
\mathit{trav}\langle\mu\ \sigma\rangle &= \mathit{fix}\ (\mathit{trav}\langle\sigma\rangle\ [\mathrm{rename}_1(\mu\ \sigma)]\ [\mathrm{rename}_2(\mu\ \sigma)]) \\
\mathit{trav}\langle\{\mathit{fst}, \mathit{snd}\}\rangle &= \Lambda\alpha_1\alpha_2 :: *.\ \lambda p_\alpha : \alpha_1 \to G\ \alpha_2.\ \Lambda\beta_1\beta_2 :: *.\ \lambda p_\beta : \beta_1 \to G\ \beta_2. \\
&\qquad \lambda x : \{\mathit{fst} : \alpha_1, \mathit{snd} : \beta_1\}. \\
&\qquad\quad g.\mathit{call}\ [\beta_2]\ [\{\mathit{fst} : \alpha_2, \mathit{snd} : \beta_2\}] \\
&\qquad\qquad (g.\mathit{call}\ [\alpha_2]\ [\beta_2 \to \{\mathit{fst} : \alpha_2, \mathit{snd} : \beta_2\}] \\
&\qquad\qquad\quad (g.\mathit{pure}\ [\alpha_2 \to \beta_2 \to \{\mathit{fst} = \alpha_2, \mathit{snd} = \beta_2\}] \\
&\qquad\qquad\qquad (\lambda yz.\ \{\mathit{fst} = y, \mathit{snd} = z\})) \\
&\qquad\qquad\quad (p_\alpha\ x.\mathit{fst})) \\
&\qquad\qquad (p_\beta\ x.\mathit{snd}) \\
\mathit{trav}\langle\langle\mathit{inj}\rangle\rangle &= \Lambda\alpha_1\alpha_2 :: *.\ \lambda p_\alpha : \alpha_1 \to G\ \alpha_2.\ \lambda x : \langle\mathit{inj} : \alpha_1\rangle. \\
&\qquad\qquad \textbf{case}\ x\ \textbf{of}\ \left\{\begin{array}{l} \mathit{inj} = \lambda y_\alpha : \alpha_1.\ g.\mathit{call}\ [\alpha_2]\ [\langle\mathit{inj} : \alpha_2\rangle] \\ \qquad\qquad (g.\mathit{pure}\ [\alpha_2 \to \langle\mathit{inj} : \alpha_2\rangle] \\ \qquad\qquad (\lambda z_\alpha.\ \langle\mathit{inj} = z_\alpha\rangle\ \text{as}\ \langle\mathit{inj} : \alpha_2\rangle))\ (p_\alpha\ y_\alpha) \end{array}\right\}
\end{aligned}$$

$$\mathrm{rename}_i(\tau) = \text{ the result of renaming every free variable } \alpha \neq g \text{ in } \tau \text{ to } \alpha_i$$

Figure 15: Polytypic definition [29] of *traverse*. For clarity, we only show *trav* for a 2-field record and a 1-case variant.

7. Future Work

As we have seen in Sec. 2.2, different type constructors that refer to the same datatype can have some redundancy with each other. To reduce such redundancy, instead of adding all the needed parameterization, type constructor could be specified by "overriding" some parts in another one, similarly to inheritance.

This paper only proves soundness of F^{μ}_{ω} and decidability of a fragment. We expect that a practical implementation would be relatively straightforward. Implementing systems with equirecursive types does not have special impact on the runtime representation of datatypes; data constructors (that is, introduction forms for records and variants) remain unchanged, but do not stop acting as introduction forms for recursive types.

However, some issues deserve some attention. We do not discuss complexity of deciding type equality, which depends on complexity of two steps.

- Normalization of types, like for System F_{ω} and languages with type synonyms. While naive normalization can produce output of exponential size, this issue can be alleviated by representing types as DAGs instead of trees to preserve sharing [49].
- Comparing normalized F^{μ}_{ω}-types: the algorithm we consider takes quadratic instead of linear time. There's work improving this time bound to $O(n \log n)$ [24]; in future work, we plan to investigate how to extend this algorithm to apply to DAGs.

We leave further investigation on these issues to future work.

8. Conclusion

As explained in this paper, when combining datatype-generic programming (DGP) techniques one runs into the tyranny of the dominant functor. Usual workarounds for this tyranny require at least either invoking explicitly isomorphisms explicitly or restricting traversal schemes, and limit the applicability of DGP techniques.

To avoid such drawbacks, we have introduced System F^{μ}_{ω}, a type system combining the expressiveness of System F_{ω} (required for DGP) and strong equirecursive types. We have given a novel proof that this system is sound, by a novel combination of the metatheory of System F_{ω} together with an extension of simple equirecursive types, relying on infinitary λ-calculus. By extending algorithms developed for equirecursive types, we have also shown that if we restrict F^{μ}_{ω} to first-order equirecursive types it enjoys decidable typechecking. We stick to first-order equirecursive types because practical algorithms for type equivalence in more expressive systems are a long-standing research problem.

Finally we have shown how the tyranny of the dominant decomposition does not arise in F^{μ}_{ω}. We have prototyped a design based on analogous ideas in a Scala library, which enabled us to encode different DGP techniques in an interoperable way.

Appendices

The appendices contain details omitted from the main body of the paper. Some theorems are re-stated from the original paper with the same number, but they are numbered according to the place where they first appear, hence theorem numbering is out of order sometimes.

A. Basic Properties of Böhm-reduction

In this section, we prove substitution and transitivity lemmas for Böhm-reduction. The proofs follow the architecture of the transitivity proof for parallel multistep β-reduction in Endrullis and Polonsky [21, Lemma 4]. Some tedious arguments are included for completeness.

The first lemma is immediate by the definition of $\Rightarrow_\beta^\infty$ and $\Rightarrow_{\beta\bot}^\infty$.

Lemma 18.

1. *If* $\sigma_1 \Rightarrow_\beta^* \sigma_2$ *and* $\sigma_2 \Rightarrow_\beta^\infty \sigma_3$, *then* $\sigma_1 \Rightarrow_\beta^\infty \sigma_3$.
2. *If* $\sigma_1 \Rightarrow_{\beta\bot}^* \sigma_2$ *and* $\sigma_2 \Rightarrow_{\beta\bot}^\infty \sigma_3$, *then* $\sigma_1 \Rightarrow_{\beta\bot}^\infty \sigma_3$.

We have so far taken substitution on infinitary terms for granted. In truth, our substitution is defined in Czajka [18] as well as Endrullis and Polonsky [21] by guarded corecursion:

$$\begin{aligned}
[\alpha \mapsto \sigma]\beta &= \begin{cases} \sigma & \text{if } \alpha = \beta \\ \beta & \text{if } \alpha \neq \beta \end{cases} \\
[\alpha \mapsto \sigma]\iota &= \iota \\
[\alpha \mapsto \sigma](\tau_1\ \tau_2) &= [\alpha \mapsto \sigma]\tau_1\ [\alpha \mapsto \sigma]\tau_2 \\
[\alpha \mapsto \sigma](\lambda\beta.\ \tau) &= \lambda\beta.\ [\alpha \mapsto \sigma]\tau \qquad (\beta \notin \mathrm{fv}(\sigma))
\end{aligned}$$

Endrullis and Polonsky [21] mechanize variables as de Bruijn indices so as never to worry about α-conversion. They state the following property.

Property 19 (swapping substitutions)**.** *If* $\beta \notin \mathrm{fv}(\rho)$, *then*

$$[\alpha \mapsto \rho][\beta \mapsto \sigma]\tau = [\beta \mapsto [\alpha \mapsto \rho]\sigma][\alpha \mapsto \rho]\tau.$$

Let us restate Lemma 5 in Czajka [18].

Lemma 20 (substitution preserves root-activeness)**.** *If* τ *is root-active, then so is* $[\alpha \mapsto \sigma]\tau$.

Lemma 21 (substitution preserves $\Rightarrow_{\beta\bot}$)**.** *If* $\sigma \Rightarrow_{\beta\bot} \tau$, *then* $[\alpha \mapsto \rho]\sigma \Rightarrow_{\beta\bot} [\alpha \mapsto \rho]\tau$.

Proof. By induction on $\sigma \Rightarrow_{\beta\bot} \tau$.

Case $\sigma \neq \bot$ is root-active and $\tau = \bot$. By Lemma 20, $[\alpha \mapsto \rho]\sigma$ is root-active and Böhm-contraction to $\bot = [\alpha \mapsto \rho]\tau$.

Case $\sigma = (\lambda\beta.\ \sigma_0)\ \sigma_1$ and $\tau = [\beta \mapsto \sigma_1]\sigma_0$.

$$\begin{aligned}[\alpha \mapsto \rho]((\lambda\beta.\ \sigma_0)\ \sigma_1) &= (\lambda\beta.\ [\alpha \mapsto \rho]\sigma_0)\ [\alpha \mapsto \rho]\sigma_1 \\ &\Rightarrow_{\beta\bot} [\beta \mapsto [\alpha \mapsto \rho]\sigma_1][\alpha \mapsto \rho]\sigma_0 \\ &= [\alpha \mapsto \rho][\beta \mapsto \sigma_1]\sigma_0.\end{aligned}$$

Case $\sigma = \lambda\beta.\ \sigma_0$, $\tau = \lambda\beta.\ \tau_0$, and $\sigma_0 \Rightarrow_{\beta\bot} \tau_0$. By the induction hypothesis, $[\alpha \mapsto \rho]\sigma_0 \Rightarrow_\beta [\alpha \mapsto \rho]\tau_0$. Thus

$$\begin{aligned}[\alpha \mapsto \rho]\sigma &= \lambda\beta.\ [\alpha \mapsto \rho]\sigma_0 \\ &\Rightarrow_{\beta\bot} \lambda\beta.\ [\alpha \mapsto \rho]\tau_0 \\ &= [\alpha \mapsto \rho]\tau.\end{aligned}$$

The cases for the application congruence rules are similar. □

Corollary 22 (substitution preserves $\Rightarrow^*_{\beta\bot}$)**.** *If* $\sigma \Rightarrow^*_{\beta\bot} \tau$*, then* $[\alpha \mapsto \rho]\sigma \Rightarrow^*_{\beta\bot} [\alpha \mapsto \rho]\tau$*.*

Proof. Induction on the number of steps in $\sigma \Rightarrow^*_{\beta\bot} \tau$. □

Lemma 23 (substitution)**.** *If* $\sigma_1 \Rightarrow^\infty_{\beta\bot} \sigma_2$ *and* $\tau_1 \Rightarrow^\infty_{\beta\bot} \tau_2$*, then* $[\alpha \mapsto \sigma_1]\tau_1 \Rightarrow^\infty_{\beta\bot} [\alpha \mapsto \sigma_2]\tau_2$*.*

Proof. We show $[\alpha \mapsto \sigma_1]\tau_1 \Rightarrow^\infty_{\beta\bot} [\alpha \mapsto \sigma_2]\tau_2$ by coinduction, with case analysis on $\tau_1 \Rightarrow^\infty_{\beta\bot} \tau_2$.

Case B-Const: We have $\tau_1 \Rightarrow^*_{\beta\bot} \tau_2 = \iota$. Applying Corollary 22 to the zero-step contraction $\sigma_1 \Rightarrow^*_{\beta\bot} \sigma_1$ yields

$$[\alpha \mapsto \sigma_1]\tau_1 \Rightarrow^*_{\beta\bot} [\alpha \mapsto \sigma_1]\iota = [\alpha \mapsto \sigma_2]\iota,$$

as desired.

Case B-Var: We have $\tau_1 \Rightarrow^*_{\beta\bot} \tau_2$ and τ_2 is a variable. If $\tau_2 \neq \alpha$, then the argument is identical to the case for B-Const. Suppose $\tau_2 = \alpha$. Applying Corollary 22 to the zero-step contraction $\sigma_1 \Rightarrow^*_{\beta\bot} \sigma_1$ yields

$$[\alpha \mapsto \sigma_1]\tau_1 \Rightarrow^*_{\beta\bot} [\alpha \mapsto \sigma_1]\alpha = \sigma_1 \Rightarrow^\infty_{\beta\bot} \sigma_2 = [\alpha \mapsto \sigma_2]\alpha.$$

The desired relation follows from Lemma 18.

Case B-Abs: $\tau_1 \Rightarrow^*_{\beta\bot} (\lambda\beta.\,\tau_{10})$, $\tau_2 = (\lambda\beta.\,\tau_{20})$, and $\tau_{10} \Rightarrow^\infty_{\beta\bot} \tau_{20}$.

$$\begin{aligned}[\alpha \mapsto \sigma_1]\tau_1 &\Rightarrow^*_{\beta\bot} \lambda\beta.\ [\alpha \mapsto \sigma_1]\tau_{10} && \text{(Corollary 22)} \\ &\Rightarrow^\infty_{\beta\bot} \lambda\beta.\ [\alpha \mapsto \sigma_2]\tau_{20} && \text{(coinduction hypothesis)} \\ &= [\alpha \mapsto \sigma_2]\tau_2.\end{aligned}$$

By Lemma 18, $[\alpha \mapsto \sigma_1]\tau_1 \Rightarrow^\infty_{\beta\bot} [\alpha \mapsto \sigma_2]\tau_2$.

Case B-App: Similar to B-Abs. □

Czajka [18, Lemma 36] states that root-activeness propagates backward through Böhm-reduction.

Lemma 24 (propagation of root-activeness)**.** *If $\sigma \Rightarrow^{\infty}_{\beta\bot} \tau$ and τ is root-active, so is σ.*

Lemma 25 (swapping $\Rightarrow^{\infty}_{\beta\bot}$ and $\Rightarrow_{\beta\bot}$)**.** *If $\rho \Rightarrow^{\infty}_{\beta\bot} \sigma$ and $\sigma \Rightarrow_{\beta\bot} \tau$, then $\rho \Rightarrow^{\infty}_{\beta\bot} \tau$.*

Proof. By induction on $\sigma \Rightarrow_{\beta\bot} \tau$.
Case σ is root-active and $\tau = \bot$. By Lemma 24, ρ is root-active and Böhm-contracts to $\bot$.
Case $\sigma = (\lambda\alpha.\ \sigma_0)\ \sigma_1$, and $\tau = [\alpha \mapsto \sigma_1]\sigma_0$. Expanding $\rho \Rightarrow^{\infty}_{\beta\bot} \sigma$ two levels, we obtain

$$\rho \Rightarrow^{*}_{\beta\bot} (\lambda\alpha.\ \rho_0)\ \rho_1, \qquad \rho_0 \Rightarrow^{\infty}_{\beta\bot} \sigma_0, \qquad \rho_1 \Rightarrow^{\infty}_{\beta\bot} \sigma_1.$$

Then

$$\rho \Rightarrow^{*}_{\beta\bot} [\alpha \mapsto \rho_1]\rho_0 \Rightarrow^{\infty}_{\beta} [\alpha \mapsto \sigma_1]\sigma_0$$

by Lemma 23.
Case $\sigma = (\lambda\alpha.\ \sigma_0)$, $\tau = (\lambda\alpha.\ \tau_0)$, and $\sigma_0 \Rightarrow_{\beta\bot} \tau_0$. Expanding $\rho \Rightarrow^{\infty}_{\beta\bot} \sigma$ once, we obtain

$$\rho \Rightarrow^{*}_{\beta\bot} (\lambda\alpha.\ \rho_0), \qquad \rho_0 \Rightarrow^{\infty}_{\beta\bot} \sigma_0.$$

By the induction hypothesis, $\rho_0 \Rightarrow^{\infty}_{\beta\bot} \sigma_0 \Rightarrow_{\beta\bot} \tau_0$ implies $\rho_0 \Rightarrow^{\infty}_{\beta\bot} \tau_0$. Therefore $\rho \Rightarrow^{\infty}_{\beta\bot} \tau$ by B-Abs.
The application congruence rules of $\Rightarrow_{\beta\bot}$ are argued similarly. □

Corollary 26. *If $\rho \Rightarrow^{\infty}_{\beta\bot} \sigma$ and $\sigma \Rightarrow^{*}_{\beta\bot} \tau$, then $\rho \Rightarrow^{\infty}_{\beta\bot} \tau$.*

Proof. Induction on the number of steps in $\sigma \Rightarrow^{*}_{\beta\bot} \tau$. □

Lemma 6. *Böhm-reduction $\Rightarrow^{\infty}_{\beta\bot}$ is transitive.*

Proof. Suppose $\rho \Rightarrow^{\infty}_{\beta\bot} \sigma$ and $\sigma \Rightarrow^{\infty}_{\beta\bot} \tau$. We show $\rho \Rightarrow^{\infty}_{\beta\bot} \tau$ by coinduction, with case analysis on $\sigma \Rightarrow^{\infty}_{\beta\bot} \tau$.
Case B-Bot, B-Const or B-Var: In all 3 cases we have $\sigma \Rightarrow^{*}_{\beta\bot} \tau$, and $\rho \Rightarrow^{\infty}_{\beta\bot} \tau$ follows from Corollary 26.
Case B-Abs: $\sigma \Rightarrow^{*}_{\beta\bot} (\lambda\alpha.\ \sigma_0)$, $\tau = (\lambda\alpha.\ \tau_0)$, and $\sigma_0 \Rightarrow^{\infty}_{\beta\bot} \tau_0$. By Corollary 26 we have $\rho \Rightarrow^{\infty}_{\beta\bot} (\lambda\alpha.\ \sigma_0)$. Expanding this relation gives us

$$\rho \Rightarrow^{*}_{\beta\bot} (\lambda\alpha.\ \rho_0), \qquad \rho_0 \Rightarrow^{\infty}_{\beta\bot} \sigma_0.$$

Now $\rho \Rightarrow^{\infty}_{\beta\bot} (\lambda\alpha.\ \tau_0) = \tau$ follows from B-Abs.
Case B-App. Similar to B-Abs. □

B. Type Soundness of F^{μ}_{ω}

This appendix contains proofs leading to the type soundness of F^{μ}_{ω} omitted from Sec. 5.2.

B.1. Preservation

These two lemmas follow from the definition of Böhm contraction and by induction on the number of contraction steps.

Lemma 27 (preservation of shape under $\Rightarrow_{\beta\perp}$)**.** *If $\iota\ \sigma_1 \cdots \sigma_n \Rightarrow_{\beta\perp} \rho$, then $\rho = \iota\ \rho_1 \cdots \rho_n$ and $\sigma_i \Rightarrow^*_{\beta\perp} \rho_i$ for all i.*

Lemma 28 (preservation of shape under $\Rightarrow^*_{\beta\perp}$)**.** *If $\iota\ \sigma_1 \cdots \sigma_n \Rightarrow^*_{\beta\perp} \rho$, then $\rho = \iota\ \rho_1 \cdots \rho_n$ and $\sigma_i \Rightarrow^*_{\beta\perp} \rho_i$ for all i.*

Now we can prove preservation of shape under Böhm reduction. Its consequence, the preservation of shape under Böhm equivalence, is stated in the main body of the paper.

Lemma 29 (preservation of shape under Böhm-reduction $\Rightarrow^{\infty}_{\beta\perp}$)**.** *If $\iota\ \sigma_1 \cdots \sigma_n \Rightarrow^{\infty}_{\beta\perp} \tau$, then $\tau = \iota\ \tau_1 \cdots \tau_n$ and $\sigma_i \Rightarrow^{\infty}_{\beta\perp} \tau_i$ for all $i \in 1..n$.*

Proof. By induction on n. If $n = 0$, then the lemma holds because ι does not reduce by $\Rightarrow_{\beta}$ or by $\Rightarrow_{\perp}$. If $n > 0$, then consider a derivation of $\iota\ \sigma_1 \cdots \sigma_n \Rightarrow^{\infty}_{\beta\perp} \tau$. There exists a term ρ such that $\iota\ \sigma_1 \cdots \sigma_n \Rightarrow^*_{\beta\perp} \rho$ and the last Böhm-reduction rule was chosen by examining ρ. By Lemma 28, $\rho = \iota\ \rho_1 \cdots \rho_n$ and $\sigma_i \Rightarrow^*_{\beta\perp} \rho_i$ for all $i \leq n$. Therefore the last rule can only be B-App, and we have

$$\tau = \tau'\ \tau_n, \qquad \iota\ \rho_1 \cdots \rho_{n-1} \Rightarrow^{\infty}_{\beta\perp} \tau', \qquad \rho_n \Rightarrow^{\infty}_{\beta\perp} \tau_n.$$

By the induction hypothesis, $\tau' = \iota\ \tau_1 \cdots \tau_{n-1}$ with $\rho_i \Rightarrow^{\infty}_{\beta\perp} \tau_i$ for all $i \leq (n-1)$. Thus

$$\tau = \iota\ \tau_1 \cdots \tau_{n-1}\ \tau_n.$$

By Lemma 18, $\sigma_i \Rightarrow^{\infty}_{\beta\perp} \tau_i$ for all $i \in 1..n$, as desired. □

Lemma 9 (preservation of shape under Böhm equivalence)**.** *If $\iota\ \sigma_1 \cdots \sigma_n \equiv \iota'\ \tau_1 \cdots \tau_n$ as finite F^{μ}_{ω} types, then $\iota = \iota'$ and $\sigma_i \equiv \tau_i$ for all $i \in 1..n$.*

Proof. We have

$$(\iota\ \sigma_1 \cdots \sigma_n)^{\infty} = \iota\ \sigma_1^{\infty} \cdots \sigma_n^{\infty},$$
$$(\iota'\ \tau_1 \cdots \tau_n)^{\infty} = \iota'\ \tau_1^{\infty} \cdots \tau_n^{\infty}.$$

By the definition of type equivalence, there exists an infinitary λ-term ρ such that

$$\iota\ \sigma_1^{\infty} \cdots \sigma_n^{\infty} \Rightarrow^{\infty}_{\beta\perp} \rho, \qquad \iota'\ \tau_1^{\infty} \cdots \tau_n^{\infty} \Rightarrow^{\infty}_{\beta\perp} \rho.$$

By Lemma 29,

$$\rho = \iota\ \rho_1 \cdots \rho_n = \iota'\ \rho_1 \cdots \rho_n,$$

which gives us $\iota = \iota'$. Moreover, $\sigma_i^{\infty} \Rightarrow^{\infty}_{\beta\perp} \rho_i$ and $\tau_i^{\infty} \Rightarrow^{\infty}_{\beta\perp} \rho_i$ for all i. By the definition of type equivalence, $\sigma_i \equiv \tau_i$. □

The inversion lemma is analogous to Lemma 30.3.13 in Pierce [45].

Lemma 30 (inversion)**.**

1. *If* $\Gamma \vdash \lambda x : \sigma.\ t : \tau_1 \to \tau_2$*, then* $\tau_1 \equiv \sigma$ *and* $\Gamma, x : \sigma \vdash t : \tau_2$*. Also,* $\Gamma \vdash \sigma : *$*.*
2. *If* $\Gamma \vdash \Lambda\alpha :: \kappa.\ t : \forall_{\kappa'}\ \tau$*, then* $\kappa = \kappa'$ *and* $\Gamma, \alpha :: \kappa \vdash t : \tau\ \alpha$*.*
3. *If* $\Gamma \vdash \{\overline{l_i = t_i}\} : \{\overline{l_i : \tau_i}\}$*, then* $\Gamma \vdash t_i : \tau_i$ *for all* i*.*
4. *If* $\Gamma \vdash \langle l_j = t_j \rangle$ **as** $\sigma : \langle \overline{l_i : \tau_i} \rangle$*, then* $\sigma \equiv \langle \overline{l_i : \tau_i} \rangle$ *and* $\Gamma \vdash t_j : \tau_j$*.*

Proof. The proof of part 1 is identical to the proof given in Pierce [45, Lemma 30.3.13].

The proof of part 2 is similar to Pierce's. Since we formulated $\forall_\kappa$ as type-level constants, we will reiterate the proof. It is by induction on a more general statement:

$$\text{If } \Gamma \vdash \Lambda\alpha :: \kappa.\ t : \sigma \text{ and } \sigma \equiv \forall_{\kappa'}\ \tau, \text{ then } \kappa = \kappa' \text{ and } \Gamma, \alpha :: \kappa \vdash t : \tau\ \alpha.$$

The induction step is about T-Eq and follows from transitivity of type equivalence (Corollary 8). The base case happens when $\Gamma \vdash \Lambda\alpha :: \kappa.\ t : \sigma$ is derived by T-TAbs. In this case,

$$\sigma = \forall_\alpha(\lambda\alpha :: \kappa.\ \sigma_0), \qquad \Gamma, \alpha :: \kappa \vdash t : \sigma_0$$

for some type σ_0. By Lemma 9, $\forall_\kappa = \forall_{\kappa'}$, which gives us $\kappa = \kappa'$. To obtain $\Gamma, \alpha :: \kappa \vdash t : \tau\ \alpha$, we need only apply T-Eq to the equivalence

$$\sigma_0 \equiv \sigma\ \alpha \equiv \tau\ \alpha.$$

For part 3, we note that the judgement $\Gamma \vdash \{\overline{l_i = t_i}\} : \{\overline{l_i : \tau_i}\}$ is derived by one T-Record followed by a series of T-Eq. Let $\Gamma \vdash \{\overline{l_i = t_i}\} : \{\overline{l_i : \sigma_i}\}$ be the judgement obtained by T-Record; then $\Gamma \vdash t_i : \sigma_i$. By transitivity of type equivalence, $\{\overline{l_i : \sigma_i}\} \equiv \{\overline{l_i : \tau_i}\}$. Recall that $\{\overline{l_i : \sigma_i}\}$ is a shorthand for $\{\overline{l_i}\}\ \sigma_1 \cdots\ \sigma_n$. Lemma 9 gives us $\sigma_i \equiv \tau_i$, from which we can derive $\Gamma \vdash t_i : \tau_i$ via T-Eq.

For part 4, note that the judgement $\Gamma \vdash \langle l_j = t_j \rangle$ **as** $\sigma : \langle \overline{l_i : \tau_i} \rangle$ is derived by one T-Variant followed by a series of T-Eq. Let $\Gamma \vdash \langle l_j = t_j \rangle$ **as** $\sigma : \sigma$ be the judgement obtained by T-Variant; then $\sigma \equiv \langle \overline{l_i : \sigma_i} \rangle$ and $\Gamma \vdash t_j : \sigma_j$. By symmetry and transitivity of type equivalence, $\langle \overline{l_i : \sigma_i} \rangle \equiv \sigma \equiv \langle \overline{l_i : \tau_i} \rangle$. Lemma 9 gives us $\sigma_j \equiv \tau_j$, from which we can derive $\Gamma \vdash t_j : \tau_j$ via T-Eq. □

The term-level evaluation rules are listed in Fig. 16. They are pretty standard; however the δ-reduction rule E-Delta is a slight generalization of the δ-reduction in Wright and Felleisen [53], in that we allow constants to interact with *all* elimination forms, whereas Wright and Felleisen's δ-reduction are only defined for term application, the elimination form of term abstraction. With our more liberal E-Delta, progress and preservation are stated uniformly, and we don't have to artificially forbid polymorphic constants or constant of records/variant types. An example of E-Delta is the evaluation rule schema for fixed-point combinators.

$$\mathit{fix}_\tau\ t \Rightarrow t\ (\mathit{fix}_\tau\ t) \qquad \text{(E-Delta}_{\mathit{fix}}\text{)}$$

$$
\begin{array}{rll}
v & ::= & \text{value} \\
& c & \text{constant} \\
& \mid \lambda x : \sigma.\ t & \text{abstraction} \\
& \mid \Lambda \alpha : \kappa.\ t & \text{type abstraction} \\
& \mid \{\overline{l_i = t_i}\} & \text{record} \\
& \mid \langle l_i\ t \rangle\ \textbf{as}\ \tau & \text{injection}
\end{array}
$$

$$\frac{}{(\lambda x : \sigma.\ s)\ t \Rightarrow [x \mapsto t]s} \quad \text{(E-AppAbs)}$$

$$\frac{s \Rightarrow s'}{s\ t \Rightarrow s'\ t} \quad \text{(E-App1)}$$

$$\frac{}{(\Lambda \alpha :: \kappa.\ t)\ [\tau] \Rightarrow [\alpha \mapsto \tau]t} \quad \text{(E-TAppTAbs)}$$

$$\frac{t \Rightarrow t'}{t\ [\tau] \Rightarrow t'\ [\tau]} \quad \text{(E-TApp)}$$

$$\frac{}{\{\overline{l_i = t_i}\}.l_j \Rightarrow t_j} \quad \text{(E-ProjRcd)}$$

$$\frac{t \Rightarrow t'}{t.l \Rightarrow t'.l} \quad \text{(E-Proj)}$$

$$\frac{\tau \equiv \langle \overline{l_i : \tau_i} \rangle}{\textbf{case}\ (\langle l_j = t \rangle\ \textbf{as}\ \tau)\ \textbf{of}\ \{\overline{l_i = t_i}\} \Rightarrow t_j\ t} \quad \text{(E-Variant)}$$

$$\frac{s \Rightarrow s'}{\textbf{case}\ s\ \textbf{of}\ t \Rightarrow \textbf{case}\ s'\ \textbf{of}\ t} \quad \text{(E-Case1)}$$

$$\frac{t \Rightarrow t'}{\textbf{case}\ s\ \textbf{of}\ t \Rightarrow \textbf{case}\ s\ \textbf{of}\ t'} \quad \text{(E-Case2)}$$

$$\frac{s \Rightarrow \delta(s)}{\begin{array}{c}(s \text{ has the form } c\ t,\ c\ [\tau],\ c.l,\ \textbf{case}\ c\ \textbf{of}\ t, \text{ or } \textbf{case}\ t\ \textbf{of}\ c.)\end{array}} \quad \text{(E-Delta)}$$

Figure 16: Values and call-by-name term-level reduction of F^{μ}_{ω}. Rules and their names match Pierce [45, Chap. 11, 30] where appropriate.

Theorem 10 (preservation)**.** *Suppose all* E-Delta *rules preserve typing. If* $\Gamma \vdash t : \tau$ *and* $t \Rightarrow t'$*, then* $\Gamma \vdash t' : \tau$*.*

Proof. By induction on the typing derivation $\Gamma \vdash t : \tau$. If the judgement is derived by T-Var, T-Abs, T-TAbs, or T-Eq, then the argument is in the proof of Theorem 30.3.14 in Pierce [45]. The judgement cannot be derived by T-Record or T-Variant; otherwise t is already a value and does not reduce. It remains to examine T-App, T-Proj and T-Case.

Case T-App: We know

$$t = t_1\ t_2, \qquad \Gamma \vdash t_1 : \tau_2 \to \tau, \quad \Gamma \vdash t_2 : \tau_2.$$

From Fig. 16, the reduction $t \Rightarrow t'$ may be derived by E-App1, E-AppAbs, or E-Delta. Invoking the induction hypothesis suffices for E-App1. Pierce [45] supplies the argument for E-AppAbs. If E-Delta applies, then $t_1 = c$, $t_2 = v$ and $t' = \delta(c\ v)$ for some constant c and value v. The desired judgement $\Gamma \vdash t' : \tau$ follows from typing preservation of δ-reduction.

Case T-TApp: The reduction $t \Rightarrow t'$ is derived by E-TApp, E-Delta or E-TAppTabs. The case for E-TApp follows from the induction hypothesis, and the case for E-Delta follows from typing preservation of δ-reduction. Suppose, therefore, that E-TAppTabs applies. Then

$$t = (\Lambda\alpha :: \kappa.\ t_0)\ [\sigma], \qquad \Gamma \vdash (\Lambda\alpha :: \kappa.\ t_0) : \forall\ \tau_0,$$
$$t' = [\alpha \mapsto \sigma]t_0, \qquad \Gamma \vdash \sigma :: \kappa.$$

By the inversion lemma, $\Gamma, \alpha :: \kappa \vdash t_0 : \tau_0\ \alpha$. The desired conclusion is $\Gamma \vdash [\alpha \mapsto \sigma]t_0 : [\alpha \mapsto \sigma](\tau_0\ \alpha)$. We obtain it via the standard type substitution lemma [45, lemma 30.3.4(3)], whose proof we shall not reproduce here.

Case T-Proj: We know

$$t = s.l_j, \qquad \Gamma \vdash s : \{\overline{l_i : \tau_i}\}, \qquad \tau = \tau_j.$$

The reduction $t \Rightarrow t'$ may be obtained from E-Proj, E-Delta or E-ProjRcd. The desired judgement follows from the induction hypothesis in the case of E-Proj, and it follows from typing preservation of δ-reduction in the case of E-Delta. Suppose $t \Rightarrow t'$ by E-ProjRcd. Then

$$s = \{\overline{l_i = s_i}\}, \qquad t' = s_j.$$

By part 3 of the inversion lemma (30), $\Gamma \vdash s_j : \tau_j$.

Case T-Case: We know

$$t = \textbf{case}\ u\ \textbf{of}\ s, \qquad \Gamma \vdash u : \langle\overline{l_i : \tau_i}\rangle, \quad \Gamma \vdash s : \{\overline{l_i : \tau_i \to \tau}\}.$$

The reduction $t \Rightarrow t'$ may be obtained from E-Case1, E-Case2, E-Delta or E-Variant. For the first two, the desired judgement follows from the induction hypothesis. For E-Delta, it follows from typing preservation of δ-reduction. Suppose E-Variant is used; then

$$u = \langle l_j = u_j \rangle \text{ as } \sigma, \qquad s = \{\overline{l_i = s_i}\}, \qquad t' = s_j\ u_j.$$

By the inversion lemma, $\Gamma \vdash u_j : \tau_j$ and $\Gamma \vdash s_j : \tau_j \to \tau$. By T-App we conclude $\Gamma \vdash s_j\ u_j : \tau$. □

B.2. Progress

The canonical forms lemma is analogous to Lemma 30.3.15 in Pierce [45].

Lemma 31 (canonical forms)**.** *Suppose $\Gamma \vdash v : \tau_v$ and v is not a constant.*

1. *If $\tau_v = \sigma \to \tau$, then v is an abstraction.*
2. *If $\tau_v = \forall\ \tau$, then v is a type abstraction.*
3. *If τ_v is a record type, then v is a record.*
4. *If τ_v is a variant type, then v is an injection.*

Proof. The canonical forms lemma is all about the following question:

> If $\Gamma \vdash v : \iota\ \tau_1 \cdots \tau_n$ and v is not a constant, then how does the top-level construct of v limits the choice of ι?

To answer this question, we enumerate possible top-level constructs of the value $v \neq c$. Since all typing rules other than T-Eq are syntax-directed, we conclude:

1. If $\Gamma \vdash \lambda x : \sigma.\ t_0 : \tau$, then $\tau \equiv \sigma \to \tau_0$.
2. If $\Gamma \vdash \Lambda\alpha :: \kappa.\ t_0 : \tau$, then $\tau \equiv \forall\ \tau_0$.
3. If $\Gamma \vdash \{\overline{l_i = t_i}\} : \tau$, then $\tau \equiv \{\overline{l_i : \tau_i}\}$.
4. If $\Gamma \vdash \langle l_j = t_j \rangle$ **as** $\sigma : \tau$, then $\tau \equiv \langle \overline{l_i : \tau_i} \rangle$.

Using Lemma 9, we can refine the above as follows.

1. If $\Gamma \vdash \lambda x : \sigma.\ t_0 : \iota\ \tau_1 \cdots \tau_n$, then $\iota = \to$.
2. If $\Gamma \vdash \Lambda\alpha :: \kappa.\ t_0 : \iota\ \tau_1 \cdots \tau_n$, then $\iota = \forall$.
3. If $\Gamma \vdash \{\overline{l_i = t_i}\} : \iota\ \tau_1 \cdots \tau_n$, then $\iota = \{\overline{l_i}\}$.
4. If $\Gamma \vdash \langle l_j = t_j \rangle$ **as** $\sigma : \iota\ \tau_1 \cdots \tau_n$, then $\iota = \langle \overline{l_i} \rangle$.

Assume the antecedent of any part of the canonical forms lemma. Once we have eliminated the impossible using the 5 statements above, whatever remains (however improbable) must be the desired conclusion. □

Theorem 11 (progress)**.** *Suppose* E-DELTA *rules satisfy progress in the following sense.*

> *If s is a closed, well-typed term of the form $c\ v$, $c\ [\tau]$, $c.l$,* **case** c **of** v*, or* **case** v **of** c*, then s is reducible by an* E-DELTA *rule.*

Let t_0 be a closed, well-typed term. Then either t_0 is a value or there exists t_0' such that $t_0 \Rightarrow t_0'$.

Proof. This is a near-perfect clone of the proof of Theorem 30.3.16 in Pierce [45].

We prove progress by induction on the typing derivation $\emptyset \vdash t_0 : \tau$ and case split on the last rule deriving it. The last rule cannot be T-VAR, because t_0 is closed. If it is T-CONST, T-ABS, T-TABS, T-RECORD, or T-VARIANT, then t is a value. If it is T-EQ, then the desired result is immediate from induction hypothesis. The case for T-APP, T-TAPP, T-PROJ and T-CASE are interesting. We will argue for T-CASE; the other cases are argued similarly.

Case T-CASE: We know

$$t_0 = \textbf{case}\ s\ \textbf{of}\ t, \qquad \emptyset \vdash s : \langle \overline{l_i : \tau_i} \rangle, \quad \emptyset \vdash t : \{\overline{l_i : \tau_i \to \tau}\}.$$

By the induction hypothesis, s and t are each a value or reducible. If one of them is reducible, then t_0 is reducible by E-CASE1 or E-CASE2. Suppose both of them are values. If s or t is a constant, then t_0 reduces by some E-DELTA rule. Otherwise Lemma 31 tells us

$$s = \langle l_j = s_j \rangle\ \textbf{as}\ \sigma, \qquad t = \{\overline{l_i = t_i}\}.$$

By Lemma 30, $\sigma \equiv \langle \overline{l_i : \tau_i} \rangle$. The rule E-VARIANT applies and gives us $t_0 \Rightarrow t_j\ s_j$. □

C. Algorithmic Typing Rules

In this appendix, we list the algorithmic typing rules of $F_\omega^{\mu *}$ omitted from Sec. 5.3 (Fig. 17). Our notation is borrowed from Pfenning [44]. The superscript $^+$ mark inputs, and $^-$ marks outputs. Our algorithmic typing judgement always has the form

$$\Gamma^+ \vdash t^+ \uparrow \tau^-.$$

It means that the type checker takes Γ and t as input, and it synthesizes the type τ as output.

An implementation of these rules starts by matching t against the conclusion of a TA-rule, and then checks the conditions from left to right and from top to bottom. By comparing each TA-rule with the corresponding T-rule (Fig. 6), it is possible to verify the following properties.

$$\frac{\Gamma^+ \vdash \tau_c^+ :: *}{\Gamma^+ \vdash c^+ \uparrow \tau_c^-} \quad \text{(TA-Const)}$$

$$\frac{x^+ : \tau^- \in \Gamma^+}{\Gamma^+ \vdash x^+ \uparrow \tau^-} \quad \text{(TA-Var)}$$

$$\frac{\Gamma^+, x^+ : \sigma^+ \vdash t^+ \uparrow \tau^- \qquad \Gamma^+ \vdash \sigma^+ :: *}{\Gamma^+ \vdash (\lambda x^+ : \sigma^+.\ t^+) \uparrow \sigma^- \to \tau^-} \quad \text{(TA-Abs)}$$

$$\frac{\begin{array}{l}\Gamma^+ \vdash t_1^+ \uparrow \tau_1^- \\ \Gamma^+ \vdash t_2^+ \uparrow \tau_2^- \qquad \tau_1^+ \equiv \sigma_2^- \to \tau^- \qquad \sigma_2^+ \equiv \tau_2^+\end{array}}{\Gamma \vdash t_1\ t_2 \uparrow \tau^-} \quad \text{(TA-App)}$$

$$\frac{\Gamma^+, \alpha^+ :: \kappa^+ \vdash t^+ \uparrow \tau^-}{\Gamma^+ \vdash (\Lambda \alpha^+ :: \kappa^+.\ t^+) \uparrow \forall_{\kappa^-}\ (\lambda \alpha^- :: \kappa^-.\ \tau^-)} \quad \text{(TA-TAbs)}$$

$$\frac{\Gamma^+ \vdash t^+ \uparrow \tau_0^- \qquad \tau_0^+ \equiv \forall_{\kappa^-}\ \tau^- \qquad \Gamma^+ \vdash \sigma^+ :: \kappa^-}{\Gamma^+ \vdash t^+\ [\sigma^+] \uparrow \tau^-\ \sigma^-} \quad \text{(TA-TApp)}$$

$$\frac{\overline{\Gamma^+ \vdash t_i^+ \uparrow \tau_i^-}}{\Gamma^+ \vdash \{\overline{l_i = t_i^+}\} \uparrow \{\overline{l_i : \tau_i^-}\}} \quad \text{(TA-Record)}$$

$$\frac{\Gamma^+ \vdash t^+ \uparrow \tau^- \qquad \tau^+ \equiv \{\overline{l_i^- : \tau_i^-}\}}{\Gamma^+ \vdash t^+.l_j \uparrow \tau_j^-} \quad \text{(TA-Project)}$$

$$\frac{\Gamma^+ \vdash t^+ \uparrow \sigma^- \qquad \Gamma^+ \vdash \tau^+ :: * \qquad \tau^+ \equiv \langle \overline{l_i^- : \tau_i^-} \rangle \qquad \sigma^+ \equiv \tau_j^+}{\Gamma^+ \vdash \langle l_j = t^+ \rangle \textbf{ as } \tau^+ \uparrow \tau^-} \quad \text{(TA-Variant)}$$

$$\frac{\begin{array}{lll} & & \overline{l_i^+ = l_i'^+} \\ \Gamma^+ \vdash t^+ \uparrow \rho^- \qquad \rho^+ \equiv \langle \overline{l_i^- : \rho_i^-} \rangle & & \overline{\rho_i^+ \equiv \sigma_i^+} \\ \Gamma^+ \vdash s^+ \uparrow \sigma^- \qquad \sigma^+ \equiv \{\overline{l_i'^- : \sigma_i^- \to \tau_i^-}\} & & \overline{\tau_1^+ \equiv \tau_i^+}\end{array}}{\Gamma^+ \vdash \textbf{case } t^+ \textbf{ of } s^+ \uparrow \tau_1^-} \quad \text{(TA-Case)}$$

Figure 17: Algorithmic typing rules for $F_\omega^{\mu *}$.

- *Soundness*: If $\Gamma^+ \vdash t^+ \uparrow \tau^-$, then $\Gamma \vdash t : \tau$.
- *Completeness*: If $\Gamma \vdash t : \tau$, then $\Gamma^+ \vdash t^+ \uparrow \sigma^-$ for some $\sigma \equiv \tau$.

Furthermore, the TA-rules are syntax-directed, and the TA-judgements in their conditions are always about a smaller term. Therefore we have

- *Termination*: The number of TA-rules in any algorithmic typing judgement $\Gamma^+ \vdash t^+ : \tau^-$ is at most the size of t.

The conditions of TA-rules include kinding judgements and two forms of type equivalence relations. They are to be checked by subroutines.

- Kinding judgemnts $\Gamma^+ \vdash \tau^+ :: \kappa^+$ and $\Gamma \vdash \tau^+ : \kappa^-$: Since the type language of $F_\omega^{\mu *}$ is a version of simply typed lambda calculus, we take it for granted that kinding judgements can be checked and kinds can be synthesized.
- Type equivalence of the form $\sigma^+ \equiv \tau^+$: Given two $F_\omega^{\mu *}$ types, decide whether they are equivalent. The problem is discussed in Sec. 5.3.1 and the subroutine is listed at the end of Appendix D.
- Type equivalence of the form $\sigma^+ \equiv \iota^- \ \sigma_1^- \cdots \sigma_k^-$. Examples include $\tau_1^+ \equiv \sigma_2^- \to \tau^-$ in TA-App, and $\tau^+ \equiv \{\overline{l_i^- : \tau_i^-}\}$ in TA-Project. These relations ask whether τ is equivalent to the result of applying some type constant ι, and if it is, what ι and its arguments are. The problem and its solution are described in Sec. 5.3.2.

While the TA-rules are only concerned about type synthesis themselves, the decidability of type equivalence means that they are usable also for verifying a fully known judgement $\Gamma \vdash t : \tau$, in that we compare τ with the type of t synthesized under Γ.

D. Decidability of Type Equivalence on $F_\omega^{\mu *}$

Here we supply details omitted from Sec. 5.3.1 and 5.3.2. To recapitulate, the decision procedure for equivalence between well-kinded $F_\omega^{\mu *}$-types proceeds in two steps.

1. Convert the well-kinded $F_\omega^{\mu *}$-types to β-normal forms in $\mathrm{NF}^{\mu *}$ (Fig. 12).
2. Decide a coinductively defined relation $\equiv_\mu$ (Fig. 13) on $\mathrm{NF}^{\mu *}$-types with the algorithm gfp^s (Fig. 18) from Pierce [45].

The procedure is justified in 3 steps. First we establish the commuting diagram lemma $\sigma^\infty \Rightarrow_{\beta\bot}^\infty \mathrm{Ex}(\mathrm{nf}(\sigma))$ (Lemma 15), which implies that Böhm-equivalence between σ^∞ and τ^∞ can be decided by comparing $\mathrm{Ex}(\mathrm{nf}(\sigma))$ and $\mathrm{Ex}(\mathrm{nf}(\tau))$. Then we establish the μ-expansion lemma (Lemma 16), namely that Ex translates μ-equivalent terms precisely into equal infinite terms. The μ-expansion lemma allow us to decide $\mathrm{Ex}(\mathrm{nf}(\sigma)) = \mathrm{Ex}(\mathrm{nf}(\tau))$ by deciding $\mathrm{nf}(\sigma) \equiv_\mu \mathrm{nf}(\tau)$. Finally we show that the decision procedure for $\equiv_\mu$ terminates by counting "subterms" of $\mathrm{NF}^{\mu *}$ terms.

From the μ-expansion lemma, we know that μ-equivalence $\equiv_\mu$ is an actual equivalence relation, i. e., it is reflexive, symmetric and transitive. That enables us to implement the other subroutine in the $F_\omega^{\mu*}$ typechecker, namely the discovery of type arguments, simply by unrolling μ at the outermost level.

The rest of the section will use the following notions from the main paper:

- the language $\mathrm{NF}^{\mu*}$ of $F_\omega^{\mu*}$ types in normal form, and the language NF^∞ of infinite terms in normal form (Fig. 12);
- the infinite term μ^∞ (Fig. 8);
- the infinite interpretation function $(\cdot)^\infty$ (Definition 1);
- the infinite expansion function $\mathrm{Ex}(\cdot)$ (Fig. 14);
- Böhm reduction and Böhm equivalence (Definition 5);
- μ-equivalence $\equiv_\mu$ (Definition 12);
- contractiveness (Definition 13).

D.1. Commuting Diagram Lemma

We will work toward the commuting diagram lemma. Infinite expansion Ex presents the greatest difficulty: It essentially performs an infinite amount of β-reduction at different locations. The challenge is to demonstrate that Ex preserves substitution, normal forms, and Böhm-reduction. Once the preservation properties are shown, the commuting diagram lemma is immediate.

D.1.1. Infinite Expansion Preserves Substitution

$\mathrm{NF}^{\mu*}$ and NF^∞ terms are not closed under substitution in general, because substitution can create β-redexes. However, as seen from the next lemma, neutral terms are safe to substitute with. We will only substitute neutral terms into normal forms for the rest of the section.

Lemma 32.

- *Let $m \in \mathrm{NF}^{\mu*}$ and let n be a finite neutral term. Then $[\alpha \mapsto n]m$ belongs to $\mathrm{NF}^{\mu*}$. Moreover, if m is neutral, so is $[\alpha \mapsto n]m$.*
- *Let $m' \in \mathrm{NF}^\infty$ and let n' be an infinite neutral term. Then $[\alpha \mapsto n']m' \in \mathrm{NF}^\infty$. Moreover, if m' is neutral, so is $[\alpha \mapsto n']m'$.*

Proof. Every occurrence of the variable α in m is ultimately derived from the production rule $n ::= \alpha$, thus occurs where any finite neutral term is allowed. Since finite neutral terms are defined by a context-free grammar, replacing α by n in m will produce another term derivable from the grammar of $\mathrm{NF}^{\mu*}$. Suppose m is neutral. If $m \neq \alpha$, then

$[\alpha \mapsto n]m$ has the top-level construct of m and has to be neutral. If $m = \alpha$, then $[\alpha \mapsto n]m = n$, which is neutral by assumption.

The argument about infinite terms n', m' is analogous. □

Lemma 33. *Let m be an $\mathrm{NF}^{\mu *}$ term and let n be a finite neutral term. Then*

$$\mathrm{Ex}([\alpha \mapsto n]m) = [\alpha \mapsto \mathrm{Ex}(n)]\mathrm{Ex}(m).$$

To prove Lemma 33, we do induction with a stronger hypothesis.

Lemma 34. *Let m be an $\mathrm{NF}^{\mu *}$ term and let $n_1, \ldots, n_k$ be finite neutral terms. Then*

$$\mathrm{Ex}([\beta_1 \mapsto n_1] \cdots [\beta_k \mapsto n_k]\ m) = [\beta_1 \mapsto \mathrm{Ex}(n_1)] \cdots [\beta_k \mapsto \mathrm{Ex}(n_k)]\ \mathrm{Ex}(m).$$

Proof. By induction on m. We will abbreviate

$$[\beta_1 \mapsto n_1] \cdots [\beta_k \mapsto n_k]\ m$$

into

$$\overline{[\beta_i \mapsto n_i]}m.$$

Case $m = \iota$ or $m = \beta \notin \{\beta_1, \ldots, \beta_k\}$. The lemma degenerates to $\mathrm{Ex}(m) = \mathrm{Ex}(m)$.

Case $m = \beta_i$. The lemma degenerates to $\mathrm{Ex}(n_i) = \mathrm{Ex}(n_i)$.

Case $m = n_0\ m_0$. Then

$$\mathrm{Ex}(m) = \mathrm{Ex}(n_0)\ \mathrm{Ex}(m_0).$$

By the induction hypothesis,

$$\begin{aligned}\mathrm{Ex}(\overline{[\beta_i \mapsto n_i]}n_0) &= \overline{[\beta_i \mapsto \mathrm{Ex}(n_i)]}\mathrm{Ex}(n_0),\\ \mathrm{Ex}(\overline{[\beta_i \mapsto n_i]}m_0) &= \overline{[\beta_i \mapsto \mathrm{Ex}(n_i)]}\mathrm{Ex}(m_0).\end{aligned}$$

Thus

$$\begin{aligned}&\mathrm{Ex}(\overline{[\beta_i \mapsto n_i]}m)\\ &= \overline{[\beta_i \mapsto \mathrm{Ex}(n_i)]}\mathrm{Ex}(n_0) \quad \overline{[\beta_i \mapsto \mathrm{Ex}(n_i)]}\mathrm{Ex}(m_0)\\ &= \overline{[\beta_i \mapsto \mathrm{Ex}(n_i)]}\mathrm{Ex}(m).\end{aligned}$$

The argument is similar if $m = \lambda\alpha :: \kappa.\ m_0$.

Case $m = \mu\ n_0$. Then

$$\begin{aligned}\mathrm{Ex}(\mu\ n_0) &= \mathrm{Ex}(n_0)\ \mathrm{Ex}(m)\\ &= \mathrm{Ex}(n_0)\ (\mathrm{Ex}(n_0)\ (\mathrm{Ex}(n_0)\ \cdots)).\end{aligned}$$

By the induction hypothesis,

$$\mathrm{Ex}(\overline{[\beta_i \mapsto n_i]}n_0) = \overline{[\beta_i \mapsto \mathrm{Ex}(n_i)]}\mathrm{Ex}(n_0).$$

Therefore

$$\begin{aligned}
&\overline{[\beta_i \mapsto \mathrm{Ex}(n_i)]}\mathrm{Ex}(\mu\ n_0)\\
&= \overline{[\beta_i \mapsto \mathrm{Ex}(n_i)]}\mathrm{Ex}(n_0)\ (\overline{[\beta_i \mapsto \mathrm{Ex}(n_i)]}\mathrm{Ex}(n_0)\ (\overline{[\beta_i \mapsto \mathrm{Ex}(n_i)]}\mathrm{Ex}(n_0)\ \cdots))\\
&= \mathrm{Ex}(\overline{[\beta_i \mapsto n_i]}n_0)\ (\mathrm{Ex}(\overline{[\beta_i \mapsto n_i]}n_0)\ (\mathrm{Ex}(\overline{[\beta_i \mapsto n_i]}n_0)\ \cdots))\\
&= \mathrm{Ex}(\overline{[\beta_i \mapsto n_i]}m).
\end{aligned}$$

Case $m = \mu\ (\lambda\alpha :: *.\ n)$. If m is non-contractive, then m has no free variable and the lemma degenerates to $\mathrm{Ex}(m) = \mathrm{Ex}(m)$. Assume m to be contractive; then it is also neutral. By the induction hypothesis,

$$\begin{aligned}
\overline{[\beta_i \mapsto \mathrm{Ex}(n_i)]}[\alpha \mapsto \mathrm{Ex}(m)]\mathrm{Ex}(n) &= \mathrm{Ex}(\overline{[\beta_i \mapsto n_i]}[\alpha \mapsto m]n),\\
[\alpha \mapsto \mathrm{Ex}(m)]\mathrm{Ex}(n) &= \mathrm{Ex}([\alpha \mapsto m]n).
\end{aligned}$$

Thus

$$\begin{aligned}
\overline{[\beta_i \mapsto \mathrm{Ex}(n_i)]}\mathrm{Ex}(m) &= \overline{[\beta_i \mapsto \mathrm{Ex}(n_i)]}\mathrm{Ex}([\alpha \mapsto m]n)\\
&= \overline{[\beta_i \mapsto \mathrm{Ex}(n_i)]}[\alpha \mapsto \mathrm{Ex}(m)]\mathrm{Ex}(n)\\
&= \mathrm{Ex}(\overline{[\beta_i \mapsto n_i]}[\alpha \mapsto m]n)\\
&= \mathrm{Ex}([\alpha \mapsto \overline{[\beta_i \mapsto n_i]}m]\overline{[\beta_i \mapsto n_i]}n)\\
&= \mathrm{Ex}(\mu\ (\lambda\alpha :: *.\ \overline{[\beta_i \mapsto n_i]}n)).\\
&= \mathrm{Ex}(\overline{[\beta_i \mapsto n_i]}m).
\end{aligned}$$

□

D.1.2. Infinite Expansion Preserves Normal Forms

This subsection establishes that infinite expansion Ex maps the normal form of $F_\omega^{\mu *}$ types into infinite terms without β-redexes.

We extend β-contraction to finite types, leaving type constants and μ uninterpreted.

Definition 35. The β-contraction relation $\Rightarrow_\beta$ is extended to finite, $F_\omega^{\mu *}$ types by augmenting the $\Rightarrow_\beta$ rules in Fig. 9 with the following congruence rule for recursive types.

$$\frac{\tau_1 \Rightarrow_\beta \tau_2}{\mu\ \tau_1 \Rightarrow_\beta \mu\ \tau_2}$$

Lemma 36. *Let σ be a well-kinded $F_\omega^{\mu *}$-type. Then σ has a β-normal form in the language* $\mathrm{NF}^{\mu *}$.

Proof. Since β-contraction between $F_\omega^{\mu *}$-types does not involve constants, we can think of constants as variables of a bigger context. We apply preservation and strong normalization of pure simply typed λ-calculus to conclude that σ has a well-kinded β-normal form τ. To see that τ belongs to the language $\mathrm{NF}^{\mu *}$, we carry out induction on $\Gamma \vdash \tau : \kappa$.

Case K-FIX*: We know $\tau = \mu\ \tau_0$, $\Gamma \vdash \tau_0 : * \to *$. If τ_0 is not an abstraction, then by induction hypothesis, $\tau_0 = n_0$ is a neutral term and $\tau = \mu\ n_0$. If $\tau_0 = \lambda\alpha :: \kappa_\alpha.\ \tau_1$, then $\kappa = *$ and $\Gamma, \alpha :: * \vdash \tau_1 : *$. Since no abstraction has kind $*$, $\tau_1 = n_1$ is a neutral term by the induction hypothesis and $\tau = \mu\ (\lambda\alpha :: *.\ n_1)$, as desired.

Case K-ABS: $\tau = \lambda\alpha :: \kappa.\ \tau_0$. By the induction hypothesis, $\tau_0 = m \in \mathrm{NF}^{\mu *}$ and $\tau = \lambda\alpha :: \kappa.\ m$.

Case K-APP: $\tau = \tau_1\ \tau_2$. By the induction hypothesis, $\tau_2 = m \in \mathrm{NF}^{\mu *}$. Since τ is not a β-redex, $\tau_1 = n$ is a neutral term by the induction hypothesis and $\tau = n\ m$.

Case K-VAR and K-CONST: Immediate, since $\Lambda^{\mu *}$ variables and constants belong to $\mathrm{NF}^{\mu *}$. □

Lemma 37. *If $\tau \in \Lambda^\infty$ has no β-redex, then $\tau \in \mathrm{NF}^\infty$. Moreover, if τ is not an abstraction, then τ is an infinite neutral term.*

Proof. By coinduction on τ.

If $\tau = \bot$, ι or α, then τ is an infinitary neutral term.

If $\tau = \lambda\alpha.\ \tau_0$, then by coinduction hypothesis, $\tau_0 \in \mathrm{NF}^\infty$. Therefore $\tau \in \mathrm{NF}^\infty$.

If $\tau = \tau_1\ \tau_2$, then τ_1 is not an abstraction, as τ has no β-redex. By the coinduction hypothesis, τ_1 is an infinite neutral term and $\tau_2 \in \mathrm{NF}^\infty$. Therefore τ is an infinite neutral term. □

Lemma 38. *Let $m \in \mathrm{NF}^{\mu *}$. Then*

1. $\mathrm{Ex}(m) \in \mathrm{NF}^\infty$, *and*
2. $\mathrm{Ex}(m)$ *is an infinite neutral term whenever m is a finite neutral term.*

In particular, $\mathrm{Ex}(m)$ *contains no β-redex.*

Proof. By coinduction on the grammar of NF^∞ with case analysis on m.

Case $m = \alpha$ or $m = \iota$. Immediate.

Case $m = \lambda\alpha :: \kappa.\ m_0$. Immediate from the coinduction hypothesis.

Case $m = n\ m_0$. We have $\mathrm{Ex}(m) = \mathrm{Ex}(n)\ \mathrm{Ex}(m_0)$. By the coinduction hypothesis, $\mathrm{Ex}(n)$ is an infinite neutral term and $\mathrm{Ex}(m_0) \in \mathrm{NF}^\infty$. Therefore $\mathrm{Ex}(m) \in \mathrm{NF}^\infty$ by the application production.

Case $m = \mu\ n$. By definition,

$$\mathrm{Ex}(m) = \mathrm{Ex}(n)\ \mathrm{Ex}(m).$$

By the coinduction hypothesis, $\mathrm{Ex}(n)$ is an infinite neutral term and $\mathrm{Ex}(m) \in \mathrm{NF}^\infty$, which gives us $\mathrm{Ex}(m) \in \mathrm{NF}^\infty$ by the application production.

Case $m = \mu\ (\lambda\alpha_1 :: *.\ n_2)$. If m is non-contractive, then $\mathrm{Ex}(m) = \bot$ and the lemma is immediate. If m is contractive, then

$$m = \mu\ (\lambda\alpha_1 :: *. \cdots \mu\ (\lambda\alpha_k :: *.\ n) \cdots)$$

such that $n \notin \{\alpha_1, \ldots, \alpha_k\}$ and it does not have the form $\mu\ (\lambda\beta.\ \cdots)$. Write

$$\begin{aligned} m_1 &= m, \\ m_{i+1} &= \mu\ (\lambda\alpha_{i+1} :: *. \cdots \mu\ (\lambda\alpha_k :: *.\ [\alpha_i \mapsto m_i, \ldots, \alpha_1 \mapsto m_1]n) \cdots), \\ n' &= [\alpha_j \mapsto \mathrm{Ex}(m_j) \quad {}^{i \in k..1}]\ n. \end{aligned}$$

Then

$$\begin{aligned} \mathrm{Ex}(m) &= \mathrm{Ex}([\alpha_i \mapsto \mathrm{Ex}(m_i) \quad {}^{i \in k..1}]\ n) \\ &= \mathrm{Ex}(n'). \end{aligned}$$

The constraints on n means that the neutral term n' is either a variable, a constant, an application, or has the form $\mu\ n'_0$. Using previous arguments for these cases, we conclude that $\mathrm{Ex}(m) = \mathrm{Ex}(n')$ is an infinite neutral term. □

D.1.3. Eliminating μ

The trickiest cases in the proofs of Lemmas 33 and 38 are about types starting with μ. This subsection will allow us to avoid much of the difficulty as long as we reason only up to Böhm-equivalence.

The next lemma is shown by a straightforward induction on Λ^μ terms.

Lemma 39.

- $([\alpha \mapsto n]m)^\infty = [\alpha \mapsto n^\infty]m^\infty$.
- *If* $\sigma \Rightarrow_\beta \tau$, *then* $\sigma^\infty \Rightarrow_\beta \tau^\infty$.

Lemma 40 (μ-elimination)**.** *For all contractive* $m \in \mathrm{NF}^{\mu *}$ *there exists* $m_0 \in \mathrm{NF}^{\mu *}$ *computable from* m *such that* $m^\infty \equiv_{\beta\perp} m_0^\infty$, $\mathrm{Ex}(m) = \mathrm{Ex}(m_0)$, *and* m_0 *does not start with* μ.

Proof. We construct m_0 by induction on the number of μ at the front of m. Since Böhm equivalence and equality on infinite terms are both transitive, it suffices to reduce the number of μ at the front of m in each induction step.

If m does not start with μ, then choose $m_0 = m$ and we are done.

If $m = \mu\ n$, then choose $m_0 = n\ (\mu\ n)$. Then $\mathrm{Ex}(m) = \mathrm{Ex}(m_0)$ by definition, and

$$\begin{aligned} m^\infty &= \mu^\infty\ n^\infty \\ &\Rightarrow_\beta n^\infty\ (n^\infty\ (n^\infty\ \cdots)) \\ &\Leftarrow_\beta n^\infty\ (\mu^\infty\ n^\infty) \\ &= (n\ (\mu\ n))^\infty. \end{aligned}$$

If $m = \mu\ (\lambda\alpha :: *.\ n)$, then choose $m_0 = [\alpha \mapsto m]n$. Since m is contractive, m_0 has one fewer μ at the front. Again $\mathrm{Ex}(m) = \mathrm{Ex}(m_0)$ by definition. We have

$$\begin{aligned}
m^\infty &= \mu^\infty\ (\lambda\alpha :: *.\ n^\infty) \\
&\Rightarrow_\beta (\lambda\alpha :: *.\ n^\infty)\ ((\lambda\alpha :: *.\ n^\infty)\ ((\lambda\alpha :: *.\ n^\infty)\ \cdots)) \\
&\Rightarrow_\beta [\alpha \mapsto (\lambda\alpha :: *.\ n^\infty)\ ((\lambda\alpha :: *.\ n^\infty)\ ((\lambda\alpha :: *.\ n^\infty)\ \cdots))]\ n^\infty \\
&\Leftarrow_\beta^\infty [\alpha \mapsto m^\infty]n^\infty \\
&= ([\alpha \mapsto m]n)^\infty,
\end{aligned}$$

where the last step follows from Lemma 39. □

D.1.4. Infinite Expansion Preserves Böhm Reduction

This subsection implements infinite expansion Ex by Böhm reduction.

Lemma 41. *$m^\infty \equiv_{\beta\bot} \mathrm{Ex}(m)$ for all $m \in \mathrm{NF}^{\mu *}$.*

Proof. By coinduction on $\equiv_{\beta\bot}$ with case analysis on m.

Case $m = \iota$ or $m = \alpha$. Immediate.

Case $m = n_0\ m_0$. By the coinduction hypothesis, $n_0^\infty \equiv_{\beta\bot} \mathrm{Ex}(n_0)$ and $m_0^\infty \equiv_{\beta\bot} \mathrm{Ex}(m_0)$. We have

$$m^\infty = n_0^\infty\ m_0^\infty \equiv_{\beta\bot} \mathrm{Ex}(n_0)\ \mathrm{Ex}(m_0) = \mathrm{Ex}(m)$$

by two uses of B-App (Fig. 11).

Case $m = \lambda\alpha :: \kappa.\ n$. By the coinduction hypothesis, $n^\infty \equiv_{\beta\bot} \mathrm{Ex}(n)$. We obtain

$$m^\infty = \lambda\alpha :: \kappa.\ n^\infty \equiv_{\beta\bot} \lambda\alpha :: \kappa.\ \mathrm{Ex}(n) = \mathrm{Ex}(m)$$

by two uses of B-Abs.

Case $m = \mu\ n$ or $m = \mu\ (\lambda\alpha :: *.\ n)$. If m is non-contractive, then m^∞ is root-active, $\mathrm{Ex}(m) = \bot$, and they are Böhm-equivalent by definition. If m is contractive, then by Lemma 40, there exists m_0 such that $m^\infty \equiv_{\beta\bot} m_0^\infty$, $\mathrm{Ex}(m) = \mathrm{Ex}(m_0)$, and m_0 does not start with μ. Use one of the previous cases to establish $m_0^\infty \equiv_{\beta\bot} \mathrm{Ex}(m_0)$; the desired equivalence $m^\infty \equiv_{\beta\bot} \mathrm{Ex}(m)$ follows from transitivity. □

Lemma 42. *$m^\infty \Rightarrow_{\beta\bot}^\infty \mathrm{Ex}(m)$ for all $m \in \mathrm{NF}^{\mu *}$.*

Proof. By Lemma 41, $m^\infty \equiv_{\beta\bot} \mathrm{Ex}(m)$. Therefore there exists m' such that $m^\infty \Rightarrow_{\beta\bot}^\infty m'$ and $\mathrm{Ex}(m) \Rightarrow_{\beta\bot}^\infty m'$. Since $\mathrm{Ex}(m)$ has no β-redex, $m' = \mathrm{Ex}(m)$ and the lemma follows. □

Lemma 15 (commuting diagram)**.** *Let m be the β-normal form of the $F_\omega^{\mu*}$-type σ. Then $\sigma^\infty \Rightarrow_{\beta\perp}^\infty \mathrm{Ex}(m)$.*

$$\begin{array}{ccc} \sigma \in \Lambda^{\mu*} & \xrightarrow{\Rightarrow_\beta^*} & m \in \mathrm{NF}^{\mu*} \\ {\scriptstyle (\cdot)^\infty}\downarrow & & \downarrow{\scriptstyle \mathrm{Ex}(\cdot)} \\ \sigma^\infty \in \Lambda^\infty & \xrightarrow[\Rightarrow_{\beta\perp}^\infty]{} & \mathrm{Ex}(m) \in \mathrm{NF}^\infty \end{array}$$

Proof. By Lemmas 39 and 42,

$$\sigma^\infty \Rightarrow_\beta^* m^\infty \Rightarrow_{\beta\perp}^\infty \mathrm{Ex}(m).$$

By Lemma 18, $\sigma^\infty \Rightarrow_{\beta\perp}^\infty \mathrm{Ex}(m)$. □

Corollary 43. *Let m be the β-normal form of the $F_\omega^{\mu*}$-type σ. Then $\sigma^\infty \equiv_{\beta\perp} \mathrm{Ex}(m)$.*

D.2. μ-Expansion Lemma

We will show that equality on NF^∞ terms (i. e., Berarducci trees) are captured by the coinductive rules of μ-equivalence.

The next lemma is analogous to Lemma 21.8.6 in Pierce [45]. It leads to the μ-expansion lemma, whence we establish the reflexivity, symmetry and transitivity of $\equiv_\mu$ by relating it to equality on infinite terms. Alternatively, if we were to prove the transitivity of $\equiv_\mu$ beforehand, then Lemma 44 would follow from the proof of Lemma 40.

Lemma 44. *Let m_1, m_2 be contractive $\mathrm{NF}^{\mu*}$ terms such that $m_1 \equiv_\mu m_2$. Then there exists m_3, m_4 such that*

$$m_3 \equiv_\mu m_4, \qquad \mathrm{Ex}(m_1) = \mathrm{Ex}(m_3), \qquad \mathrm{Ex}(m_2) = \mathrm{Ex}(m_4),$$

and neither m_3 nor m_4 starts with μ.

Proof. By induction on the total number of μ at the front of m_1 and m_2.

If neither m_1 nor m_2 starts with μ, then take $m_3 = m_1$ and $m_4 = m_2$.

Suppose $m_1 = \mu\,(\lambda\alpha : *.\ n_1)$. Since we assumed it to be contractive,

$$[\alpha \mapsto m_1]n_1 \equiv_\mu m_2.$$

The $\mathrm{NF}^{\mu*}$ term $[\alpha \mapsto m_1]n_1$ has one fewer μ at the front than m_1. By the induction hypothesis, there exist $m_3 \equiv_\mu m_4$ such that neither m_3 nor m_4 starts with μ and

$$\mathrm{Ex}(m_3) = \mathrm{Ex}([\alpha \mapsto m_1]n_1) = \mathrm{Ex}(m_1), \qquad \mathrm{Ex}(m_4) = \mathrm{Ex}(m_2).$$

Therefore m_3 and m_4 are the pair we need.

Suppose $m_1 = \mu\ n_1$. Then

$$\mathrm{Ex}(m_1) = \mathrm{Ex}(n_1)\ \mathrm{Ex}(m_1) = \mathrm{Ex}(n_1\ m_1).$$

The term $n_1\ m_1$ does not start with μ. The induction hypothesis gives us the desired m_3, m_4 when invoked on $n_1\ m_1$ and m_2.

The argument is similar in cases where m_2 starts with μ and m_1 does not. □

Lemma 45. *If $m_1 \equiv_\mu m_2$, then either both are non-contractive or both are contractive.*

Proof. Suppose m_1 is contractive and m_2 is non-contractive. Eq-$\mu\bot$ cannot be used to establish $m_1 \equiv_\mu m_2$. No other rule's conclusion has a non-contractive term on the right hand side, therefore $m_1 \not\equiv_\mu m_2$. A similar argument is used if m_1 is non-contractive and m_2 is contractive. □

We are ready to prove that infinite expansion Ex translates $\equiv_\mu$ verbatim into equality. It is related to Theorem 21.8.7 in Pierce [45], and its proof is very similar.

Lemma 16 (μ-expansion)**.** *$m_1 \equiv_\mu m_2$ if and only if* $\mathrm{Ex}(m_1) = \mathrm{Ex}(m_2)$.

Proof. **Part I: "only-if."** $m_1 \equiv_\mu m_2$ implies $\mathrm{Ex}(m_1) = \mathrm{Ex}(m_2)$. We prove it by coinduction on equality of infinitary lambda terms, with case distinction on $m_1 \equiv_\mu m_2$.

Case one of the Eq-μ rules. By Lemma 45, either m_1, m_2 are both non-contractive or are both contractive. If both are non-contractive, then $\mathrm{Ex}(m_1) = \mathrm{Ex}(m_2) = \bot$. If both are contractive, then by Lemma 44, there exists m_3, m_4 such that

$$m_3 \equiv_\mu m_4, \qquad \mathrm{Ex}(m_1) = \mathrm{Ex}(m_3), \qquad \mathrm{Ex}(m_2) = \mathrm{Ex}(m_4),$$

and neither m_3 nor m_4 starts with μ. Use one of the cases below to establish $\mathrm{Ex}(m_3) = \mathrm{Ex}(m_4)$, and we have $\mathrm{Ex}(m_1) = \mathrm{Ex}(m_2)$ by symmetry and transitivity of equality.

Case Eq-TVar or Eq-Prim: $m_1 = m_2 = \iota$ or α. In either case, $\mathrm{Ex}(m_1) = \mathrm{Ex}(m_2)$ by definition.

Case Eq-AppCong:

$$\begin{aligned} &m_1 = n_1\ m_{10}, && \mathrm{Ex}(m_1) = \mathrm{Ex}(n_1)\ \mathrm{Ex}(m_{10}), \\ &m_2 = n_2\ m_{20}. && \mathrm{Ex}(m_2) = \mathrm{Ex}(m_2)\ \mathrm{Ex}(m_{20}). \end{aligned}$$

By the coinduction hypothesis, $\mathrm{Ex}(n_1) = \mathrm{Ex}(n_2)$ and $\mathrm{Ex}(m_{10}) = \mathrm{Ex}(m_{20})$. The desired equality follows.

Case Eq-ξ:

$$\begin{aligned} &m_1 = \lambda\alpha :: \kappa.\ m_{10}, && \mathrm{Ex}(m_1) = \lambda\alpha.\ \mathrm{Ex}(m_{10}), \\ &m_2 = \lambda\alpha :: \kappa.\ m_{20}, && \mathrm{Ex}(m_2) = \lambda\alpha.\ \mathrm{Ex}(m_{20}). \end{aligned}$$

By the coinduction hypothesis, $\mathrm{Ex}(m_{10}) = \mathrm{Ex}(m_{20})$. The desired equality follows.

Part II: "if." $\mathrm{Ex}(m_1) = \mathrm{Ex}(m_2)$ implies $m_1 \equiv_\mu m_2$. We prove it by coinduction on $\equiv_\mu$, with case distinction on the top-level constructs of m_1 and m_2.

We know $\mathrm{Ex}(m_1)$ and $\mathrm{Ex}(m_2)$ have the same top-level constructors. By inspecting Ex, (m_1, m_2) must have one of the following forms:

$$\begin{array}{ll} (\alpha, \alpha), & (\iota, \iota), \\ (n_1\ m_{10}, n_2\ m), & (\lambda\alpha :: \kappa.\ m_{10}, \lambda\alpha :: \kappa.\ m_{20}), \\ (\mu\ n_1, m_2), & (m_1, \mu\ m_2), \\ (\mu\ (\lambda\alpha :: *.\ n_1), m_2), & (m_1, \mu\ (\lambda\alpha :: *.\ n_2)). \end{array}$$

Case $m_1 = m_2 = \alpha$ or ι. Then $m_1 \equiv_\mu m_2$ by definition.

Case $m_1 = n_1\ m_{10}$, $m_2 = n_2\ m_{20}$. Then

$$\mathrm{Ex}(m_1) = \mathrm{Ex}(n_1)\ \mathrm{Ex}(m_{10}), \qquad \mathrm{Ex}(m_2) = \mathrm{Ex}(n_2)\ \mathrm{Ex}(m_{20}).$$

By the coinduction hypothesis, $n_1 \equiv_\mu n_2$ and $m_{10} \equiv_\mu m_{20}$. Eq-AppCong gives us $m_1 \equiv_\mu m_2$.

Case $m_1 = \lambda\alpha :: \kappa.\ m_{10}$, $m_2 = \lambda\alpha :: \kappa.\ m_{20}$. By the coinduction hypothesis, $m_{10} \equiv_\mu m_{20}$. We derive $m_1 \equiv_\mu m_2$ by Eq-ξ.

Case $m_1 = \mu\ n_1$. Then

$$\mathrm{Ex}(m_1) = \mathrm{Ex}(n_1)\ \mathrm{Ex}(m_1) = \mathrm{Ex}(n_1\ m_1)$$

by the definition of Ex. From $\mathrm{Ex}(n_1\ m_1) = \mathrm{Ex}(m_2)$, the coinduction hypothesis gives us $n_1\ m_1 \equiv_\mu m_2$, with which we derive $m_1 \equiv_\mu m_2$ by Eq-μ_L-Neutral.

Case $m_1 = \mu\ (\lambda\alpha.\ n_1)$ is contractive. Then

$$\mathrm{Ex}(m_1) = \mathrm{Ex}([\alpha \mapsto m_1]n_1).$$

From $\mathrm{Ex}([\alpha \mapsto m_1]n_1) = \mathrm{Ex}(m_2)$, the coinduction hypothesis gives us $[\alpha \mapsto m_1]n_1 \equiv_\mu m_2$, with which we derive $m_1 \equiv_\mu m_2$ by Eq-μ_L.

Case $m_1 = \mu\ (\lambda\alpha.\ n_1)$ is non-contractive. Then $\mathrm{Ex}(m_1) = \mathrm{Ex}(m_2) = \bot$. By the definition of Ex, m_2 is non-contractive and we derive $m_1 \equiv_\mu m_2$ by Eq-$\mu\bot$.

Case m_1 does not start with μ and m_2 does. The argument is similar to the preceding 2 cases. We require m_1 not to start with μ so that Eq-μ_R-Neutral and Eq-μ_R apply. □

D.3. Type Argument Discovery

Given the connection between type equivalence and μ-equivalence, type argument discovery becomes easy (Sec. 5.3.2): We need only normalize the input type and unroll top-level μ using the Eq-μ rules until a non-recursive or non-contractive type is encountered.

Lemma 17. *Let $m_1 \in \mathrm{NF}^{\mu*}$ be a contractive $F_\omega^{\mu*}$ type in β-normal form such that $\Gamma \vdash m_1 :: \kappa$. Then there exists $m_2 \in \mathrm{NF}^{\mu*}$ computable from m_1 such that $m_2 \equiv_\mu m_1$, $\Gamma \vdash m_2 :: \kappa$, and m_2 does not start with μ.*

Proof. By Lemma 40, there exists m_2 computable from m_1 such that $\mathrm{Ex}(m_1) = \mathrm{Ex}(m_2)$. By Lemma 16, $m_1 \equiv_\mu m_2$. □

Theorem 46. *Given any $F_\omega^{\mu*}$ type σ, one can decide whether $\sigma \equiv \iota\ \sigma_1 \cdots \sigma_k$. If the answer is yes, then one can compute $\iota, k, \sigma_1, \ldots, \sigma_k$.*

Proof. Compute the β-normal form m_1 of σ. If m_1 is non-contractive, then σ^∞ is root-active and cannot be Böhm-equivalent to any type of the form $\iota\ \sigma_1 \cdots \sigma_k$. If m_1 is contractive, then use Lemma 17 to compute $m_2 \equiv_\mu m_1$ such that m_2 does not start with μ. By Theorem 14, $m_2 \equiv \sigma$. Let n be the final operator of m_2. Since recursive types in $F_\omega^{\mu*}$ have kind $*$, n is either a constant ι, a variable, or a lambda abstraction. If $n = \iota$, then $m_2 = \iota\ \sigma_1 \cdots \sigma_k$, and we can read $\iota, k, \sigma_1, \ldots, \sigma_k$ from m_2. If m_2 is a variable or an abstraction, then σ is not equivalent to any type of the form $\iota\ \sigma_1 \cdots \sigma_k$, for otherwise there would be a conflict in the top-level construct of the common Böhm-redux of m_2 and $\iota\ \sigma_1 \cdots \sigma_k$. □

D.4. Type Equivalence Verification Algorithm

The promised *gfp*^{s} algorithm to verify μ-equivalence is listed in Fig. 18. It calls the subroutine "support" listed in Fig. 19.

Here it suffices to think of "support" as a concrete subroutine. To understand the formal definition of support and why *gfp*^{s} decides μ-equivalence, however, one would have to go into the low-level machinary of coinduction (Definition 74 in Appendix F). Pierce [45, chapter 21, definition 21.6.3] demonstrates the correctness of the *gfp*^{s} algorithm for general coinductive relations, which applies directly to our μ-equivalence. We will not go into details of the correctness argument.

The rest of the subsection investigates the termination behavior of *gfp*^{s}. Theorem 21.5.12 in Pierce [45] asserts the termination of *gfp*^{s}-like algorithms for all finite-state coinductive relations. We will establish that μ-equivalence is finite-state, namely that the transitive "support" of every pair of types is finite (Lemma 57). Here is the intuition of Pierce's theorem 21.5.12 in our scenario: The parameter X of *gfp*^{s} (Fig. 18) stays a subset of a finite set throughout execution. If the algorithm does not return *false*, then all members of X are eventually added to the set A, and subsequently deleted from X, leading to termination with *true*. To prove that μ-equivalence is finite-state, one may simply apply section 21.9 of Pierce [45] to the language $\mathrm{NF}^{\mu*}$ of β-normal forms. We will nevertheless reproduce the argument for verification.

Definition 47 (reachability)**.** A pair (m_n, m'_n) of $\mathrm{NF}^{\mu*}$ terms are *reachable* from the pair (m_0, m'_0) if there exists $\mathrm{NF}^{\mu*}$ terms $m_{n-1}, m'_{n-1}, \ldots, m_1, m'_1$ such that

$$(m_n, m'_n) \in \mathrm{support}(m_{n-1}, m'_{n-1}),$$
$$\vdots$$
$$(m_1, m'_1) \in \mathrm{support}(m_0, m'_0).$$

Definition 48 (finite state)**.** The μ-equivalence relation is *finite-state* if the number of pairs reachable from every pair of $\mathrm{NF}^{\mu*}$ terms is finite.

As stated above, we will show μ-equivalence to be finite-state. Appendix F explains how μ-equivalence is related to reachability defined in terms of "support" (Fig. 19).

$$m_1 \equiv_\mu m_2 \qquad \text{if and only if} \qquad \mathit{gfp}^s(\emptyset, \{(m_1, m_2)\}) = \mathit{true}.$$

$$\begin{aligned}
\mathit{gfp}^s(A, X) = {} & \text{if } X = \emptyset, \text{ then } \mathit{true} \\
& \text{else choose arbitrary } x \in X \\
& \quad \text{if } x \in A, \text{ then } \mathit{gfp}^s(A, X - \{x\}) \\
& \quad \text{else if support}(x) \text{ is undefined, then } \mathit{false} \\
& \quad \text{else } \mathit{gfp}^s(A \cup \{x\}, X \cup \text{support}(x)).
\end{aligned}$$

Figure 18: The gfp^s algorithm Pierce [45, Definition 21.6.3]. The function support is defined in Fig. 19 and discussed in Examples 72 and 76.

$$\begin{aligned}
\text{support}(x, x) &= \emptyset && \text{(if } x = \alpha \text{ or } x = \iota\text{)} \\
\text{support}(n_1\ m_1, n_2\ m_2) &= \{(n_1, n_2), (m_1, m_2)\} \\
\\
\text{support}(\lambda\alpha :: \kappa.\ m_1, \lambda\alpha :: \kappa.\ m_2) &= \{(m_1, m_2)\} \\
\\
\text{support}(\mu\ n, m) &= \{(n\ (\mu\ n), m)\} \\
\\
\text{support}(\mu\ (\lambda\alpha :: *.\ n), m) &= \{([\alpha \mapsto \mu\ (\lambda\alpha :: *.\ n)]n, m)\} \\
& && \text{(if } \mu\ (\lambda\alpha :: *.\ n) \text{ is contractive)} \\
\text{support}(m, \mu\ n) &= \{(m, n\ (\mu\ n))\} \\
& && \text{(if } m \text{ does not start with } \mu\text{)} \\
\text{support}(m, \mu\ (\lambda\alpha :: *.\ n)) &= \{(m, [\alpha \mapsto \mu\ (\lambda\alpha :: *.\ n)]n)\} \\
& && \text{(if } m \text{ does not start with } \mu \text{ and} \\
& && \mu\ (\lambda\alpha :: *.\ n) \text{ is contractive)} \\
\text{support}(\mu\ (\lambda\alpha :: *.\ n_1), \mu\ (\lambda\alpha :: *.\ n_2)) &= \emptyset \\
& && \text{(if } \mu\ (\lambda\alpha :: *.\ n_1) \text{ and } \mu\ (\lambda\alpha :: *.\ n_2) \\
& && \text{are non-contractive)} \\
\text{support}(m_1, m_2) &= \text{ undefined} \\
& && \text{(otherwise)}
\end{aligned}$$

Figure 19: The support function of $G_\mu^{\equiv}$ (Example 72), the generating function of μ-equivalence $\equiv_\mu$ on $\text{NF}^{\mu *}$.

$$m \leq m \qquad (\leq\text{-Refl})$$

$$\frac{m \leq m_0}{m \leq \lambda\alpha :: \kappa.\ m_0} \qquad (\leq\text{-Abs})$$

$$\frac{m \leq n_0}{m \leq n_0\ m_0} \qquad (\leq\text{-App1})$$

$$\frac{m \leq m_0}{m \leq n_0\ m_0} \qquad (\leq\text{-App2})$$

Figure 20: Congruence subterm rules.

$$\frac{m \leq_\downarrow n\ (\mu\ n)}{m \leq_\downarrow \mu\ n} \qquad (\leq_\downarrow\text{-}\mu\text{-Neutral})$$

$$\frac{m \leq_\downarrow [\alpha \mapsto \mu\ (\lambda\alpha :: *.\ n)]n}{m \leq_\downarrow \mu\ (\lambda\alpha :: *.\ n)} \qquad (\leq_\downarrow\text{-}\mu)$$

Figure 21: Additional top-down subterm rules.

$$n\ (\mu\ n) \leq_\uparrow \mu\ n \qquad (\leq_\uparrow\text{-}\mu\text{-Axiom})$$

$$\frac{m \leq_\uparrow n}{m \leq_\uparrow \mu\ n} \qquad (\leq_\uparrow\text{-}\mu\text{-Neutral})$$

$$\frac{m \leq_\uparrow n}{[\alpha \mapsto \mu\ (\lambda\alpha :: *.\ n)]m \leq_\uparrow \mu\ (\lambda\alpha :: *.\ n)} \qquad (\leq_\uparrow\text{-}\mu)$$

Figure 22: Additional bottom-up subterm rules.

Definition 49 (subterm relations)**.**

- The top-down subterm relation $\leq_\downarrow$ is defined inductively by the $\leq$-rules (Fig. 20) together with the $\leq_\downarrow$-rules (Fig. 21).
- The bottom-up subterm relation $\leq_\uparrow$ is defined inductively by the $\leq$-rules (Fig. 20) together with the $\leq_\uparrow$-rules (Fig. 22).

Note that the $\leq_\downarrow$- and $\leq_\uparrow$-rules have no contractiveness side condition. This is safe because inductive relations permit only finite derivation trees, which cannot relate non-contractive types to everything else.

Lemma 50. *If* $(s_1, s_2) \in \mathrm{support}(m_1, m_2)$*, then*

$$(s_1 \leq_\downarrow m_1 \vee s_1 \leq_\downarrow m_2) \wedge (s_2 \leq_\downarrow m_1 \vee s_2 \leq_\downarrow m_2).$$

Proof. By scrutinizing Fig. 19. Note that if $m_1 = \mu\ n$, then $s_1 = n\ (\mu\ n) \leq_\downarrow m_1$ is derived via $\leq$-Refl followed by $\leq_\downarrow$-μ-Neutral; and if $m_1 = \mu\ (\lambda\alpha :: *.\ n)$, then $s_1 \leq_\downarrow m_1$ is derived via $\leq$-Refl followed by $\leq_\downarrow$-μ. □

Lemma 51. *The top-down subterm relation* $\leq_\downarrow$ *is transitive.*

Proof. Suppose $m_1 \leq_\downarrow m_2 \leq_\downarrow m_3$. We will show $m_1 \leq_\downarrow m_3$ by induction on $m_2 \leq_\downarrow m_3$.

Case $\leq$-Refl. Trivial.

Case $\leq$-Abs: $m_1 \leq_\downarrow m_2 \leq_\downarrow (\lambda\alpha :: \kappa.\ m_{30})$ such that $m_2 \leq_\downarrow m_{30}$. By the induction hypothesis, $m_1 \leq_\downarrow m_{30}$ and we can derive $m_1 \leq_\downarrow (\lambda\alpha :: \kappa.\ m_{30})$ by $\leq$-Abs.

Case $\leq$-App1 and $\leq$-App2. Similar to the case for $\leq$-Abs.

Case $\leq_\downarrow$-μ-Neutral: $m_1 \leq_\downarrow m_2 \leq_\downarrow \mu\ n$ such that $m_2 \leq_\downarrow n\ (\mu\ n)$. By the induction hypothesis, $m_1 \leq_\downarrow n\ (\mu\ n)$ and we can derive $m_1 \leq_\downarrow \mu\ n$ by $\leq_\downarrow$-μ-Neutral.

Case $\leq_\downarrow$-μ: $m_1 \leq_\downarrow m_2 \leq_\downarrow \mu\ (\alpha :: *.\ n) = m_3$ such that $m_2 \leq_\downarrow [\alpha \mapsto m_3]n$. By the induction hypothesis, $m_1 \leq_\downarrow [\alpha \mapsto m_3]n$ and we can derive $m_1 \leq_\downarrow m_3$ by $\leq_\downarrow$-μ. □

Lemma 52. *If* (s_1, s_2) *is reachable from* (m_1, m_2)*, then*

$$(s_1 \leq_\downarrow m_1 \vee s_1 \leq_\downarrow m_2) \wedge (s_2 \leq_\downarrow m_1 \vee s_2 \leq_\downarrow m_2).$$

Proof. By induction on the lenght of the "support chain", using transitivity of $\leq_\downarrow$. □

Lemma 53. *If* $m \leq_\uparrow m'$*, then each free variable* α *of* m *occurs either free or bound in* m'*.*

Proof. By induction on $m \leq_\uparrow m'$.

Case $\leq$-Refl. Obvious.

Case $\leq$-Abs: $m' = \lambda\beta :: \kappa.\ m_0$ such that $m \leq_\uparrow m_0$. If $\alpha = \beta$, then α occurs bound in m'. If $\alpha \neq \beta$, then by the induction hypothesis, α occurs free or bound in m_0.

Case $\leq$-App1: $m' = n_0\ m_0$ such that $m \leq_\uparrow n_0$. By the induction hypothesis, α occurs free or bound in n_0.

Case $\leq$-App2. Similar.

Case $\leq_\uparrow$-μ-Axiom: $m' = \mu\ n$ and $m = n\ m'$. Then α occurs free either in n or in m'.

Case $\leq_\uparrow$-μ-Neutral: $m' = \mu\ n$ and $m \leq_\uparrow n$. By the induction hypothesis, α occurs free or bound in n.

Case $\leq_\uparrow$-μ: $m' = \mu\ (\beta\alpha :: *.\ n)$ and $m = [\beta \mapsto m']m_0$ for some $m_0 \leq_\uparrow n$. Then α occurs free in either m_0 or m'. In the former case, the induction hypothesis tells us that α occurs free in n. □

The next lemma is the most tricky of the subsection. Its proof follows closely the proof of lemma 21.9.9 in Pierce [45].

Lemma 54. *If $s \leq_\uparrow [\alpha \mapsto q]m$ then either $s \leq_\uparrow q$, or else $s = [\alpha \mapsto q]m'$ for some $m' \leq_\uparrow m$.*

Proof. By induction on m.

Case $m = \iota$ or $m = \beta \neq \alpha$. Then $[\alpha \mapsto q]m = m$. The relation $s \leq_\uparrow m$ is derivable only by $\leq$-Refl, whence $s = m$. The lemma follows by choosing $m' = s = m$.

Case $m = \alpha$. Then $[\alpha \mapsto q]m = q$ and $s \leq_\uparrow q$ by assumption.

Case $m = \lambda\beta :: \kappa.\ m_0$. The relation $s \leq_\uparrow [\alpha \mapsto q]m$ is derivable by either $\leq$-Refl or $\leq$-Abs. If it is $\leq$-Refl, choose $m' = m$ and we are done. If it is $\leq$-Abs, then $s \leq_\uparrow [\alpha \mapsto q]m_0$. By the induction hypothesis, either $s \leq_\uparrow q$ or $s = [\alpha \mapsto q]m'$ for some $m' \leq_\uparrow m_0$. In the latter case, $m' \leq_\uparrow m$ by $\leq$-Abs.

Case $m = n_0\ m_0$. The relation $s \leq_\uparrow [\alpha \mapsto q]m$ is derivable by $\leq$-Refl, $\leq$-App1 or $\leq$-App2. The case for $\leq$-Refl is again easy. In the case for $\leq$-App1, $s \leq_\uparrow [\alpha \mapsto q]n_0$. By the induction hypothesis, either $s \leq_\uparrow q$ or $s = [\alpha \mapsto q]m'$ for some $m' \leq_\uparrow n_0$, and we obtain $m' \leq_\uparrow m$ by $\leq$-App1. The case for $\leq$-App2 is similar.

Case $m = \mu\ n$. The relation $s \leq_\uparrow [\alpha \mapsto q](\mu\ n)$ is derived by $\leq$-Refl, or $\leq_\uparrow$-μ-Axiom, or $\leq_\uparrow$-μ-Neutral. In the first two cases, choose $m' = \mu\ n$ and $m' = n\ (\mu\ n)$ respectively, and we obtain $s = [\alpha \mapsto q]m'$ with $m' \leq_\uparrow m$. In the last case, we have $s \leq_\uparrow [\alpha \mapsto q]n$. By the induction hypothesis, either $s \leq_\uparrow q$ or $s = [\alpha \mapsto q]m'$ for some $m' \leq_\uparrow n$. In the latter case, $m' \leq_\uparrow m$ by $\leq_\uparrow$-μ-Neutral.

Case $m = \mu\ (\lambda\beta :: *.\ n)$. The relation $s \leq_\uparrow [\alpha \mapsto q]m$ is derived by either $\leq$-Refl or $\leq_\uparrow$-μ. The case for $\leq$-Refl is again easy. In the case for $\leq_\uparrow$-μ, there exists $s' \leq_\uparrow [\alpha \mapsto q]n$ such that

$$s = [\beta \mapsto [\alpha \mapsto q]m]s'.$$

By the induction hypothesis, either $s' \leq_\uparrow q$ or $s' = [\alpha \mapsto q]s''$ for some $s'' \leq_\uparrow n$.

Suppose $s' \leq_\uparrow q$. By convention, β occurs neither free nor bound in q. By Lemma 53, β is not free in s'. Then $s = s' \leq_\uparrow q$.

Suppose $s' = [\alpha \mapsto q]s''$ and $s'' \leq_\uparrow n$. Then

$$\begin{aligned} s &= [\beta \mapsto [\alpha \mapsto q]m][\alpha \mapsto q]s'' \\ &= [\alpha \mapsto q][\beta \mapsto m]s''. \end{aligned}$$

Choose $m' = [\beta \mapsto m]s''$; we have $m' \leq_\uparrow m$ by $\leq_\uparrow$-μ. □

Lemma 55. *If $s \leq_{\downarrow} m$, then $s \leq_{\uparrow} m$.*

Proof. By induction on $s \leq_{\downarrow} m$.

Case $\leq$-Refl, $\leq$-Abs, $\leq$-App1 or $\leq$-App2 . Immediate from the induction hypothesis.

Case $\leq_{\downarrow}$-μ-Neutral: $m = \mu\ n$ and $s \leq_{\downarrow} n\ (\mu\ n)$. By the induction hypothesis, $s \leq_{\uparrow} n\ (\mu\ n)$, which is derivable by $\leq$-Refl, $\leq$-App1 or $\leq$-App2. If the rule was $\leq$-Refl, then $s = n\ (\mu\ n)$ and $s \leq_{\uparrow} m$ follows from $\leq_{\uparrow}$-μ-Axiom. If the rule was $\leq$-App1, then $s \leq_{\uparrow} n$ and the desired relation holds by $\leq_{\uparrow}$-μ-Neutral. If the rule was $\leq$-App2, then we obtain a derivation for $s \leq_{\uparrow} (\mu\ n)$ as a sub-derivation of $s \leq_{\uparrow} n\ (\mu\ n)$.

Case $\leq_{\downarrow}$-μ: $m = \mu\ (\lambda\alpha :: *.\ n)$ and $s \leq_{\downarrow} [\alpha \mapsto m]n$. By the induction hypothesis, $s \leq_{\uparrow} [\alpha \mapsto m]n$. By Lemma 54,[7] either $s \leq_{\uparrow} m$ or $s = [\alpha \mapsto m]s'$ such that $s' \leq_{\uparrow} n$. In the latter case, we have $s \leq_{\uparrow} m$ by $\leq_{\uparrow}$-μ. □

Lemma 56. *For each $m \in \mathrm{NF}^{\mu *}$, only a finite number of $\mathrm{NF}^{\mu *}$ terms s satisfy $s \leq_{\uparrow} m$.*

Proof. The bottom-up subterm relation $\leq_{\uparrow}$ is defined via well-founded recursion on the right-hand-side. Moreover, the rule deriving $s \leq_{\uparrow} m$ is completely determined by the top-level construct of m except when m is an application, in which case we can choose between $\leq$-App1 and $\leq$-App2. If m contains k application constructs, then there are at most 2^k derivation trees with a final conclusion of the form $s \leq_{\uparrow} m$. □

Lemma 57. *For each pair (m_1, m_2) of $\mathrm{NF}^{\mu *}$ terms, the number of term-pairs reachable from them is finite. In other words, μ-equivalence is finite-state.*

Proof. For each pair (s_1, s_2) reachable from (m_1, m_2), the terms s_1 and s_2 must be chosen from the finite number of bottom-up subterms of m_1 and m_2. □

Having established that μ-equivalence is finite state, theorem 21.5.12 of Pierce [45] implies that it is possible to decide whether $m_1 \equiv_{\mu} m_2$ for all $m_1, m_2 \in \mathrm{NF}^{\mu *}$. We proved type argument discovery to be decidable along the way (Theorem 46). The two subroutines of the $F^{\mu *}_{\omega}$ typechecker are now complete.

D.5. $F^{\mu *}_{\omega}$ Types Have Regular Berarducci Trees

A consequence of μ-equivalence being finite state (Lemma 57) is that the Berarducci trees of $F^{\mu *}_{\omega}$ types have finitely many subtrees. It provides intuition why $F^{\mu *}_{\omega}$ has decidable typechecking (Sec. 5.3); we will prove it also.

Lemma 58. *If s is a subtree of $\mathrm{Ex}(m)$, then there exists $m' \leq_{\downarrow} m$ such that $s = \mathrm{Ex}(m')$.*

Proof. By strong induction on the smallest depth d of s inside $\mathrm{Ex}(m)$. If $d = 0$, then $s = \mathrm{Ex}(m)$ and we choose $m' = m$. If $d > 0$, case-split on m.

Case $m = \iota$ or $m = \alpha$. This case is impossible. By the definition of Ex, $s = \mathrm{Ex}(m)$, contradicting the assumption $d > 0$.

[7] In the substitution $[\alpha \mapsto m]n$, we assume the convention that α to occur neither bound nor free in m. One can imagine α-converting all bound variables in m to be distinct from α before substitution.

Case $m = n\ m_0$. Since $s \neq \mathrm{Ex}(m)$, either $\mathrm{Ex}(n)$ or $\mathrm{Ex}(m_0)$ contains s as a subtree at a depth smaller than d. For both cases, it suffices to invoke the induction hypothesis.

Case $m = \lambda\alpha :: \kappa.\ m_0$. Similar to the previous case.

Case $m = \mu\ n$. Then s is a subtree of $\mathrm{Ex}(m) = \mathrm{Ex}(n)\ \mathrm{Ex}(m)$. The shallowest occurence of s cannot be in the operand $\mathrm{Ex}(m)$, for that would lead to the contradiction $d < d$. Therefore s is a subtree of $\mathrm{Ex}(n)$. By the induction hypothesis, there exists $m' \leq_\downarrow n$ such that $s = \mathrm{Ex}(m')$. We derive the desired relation $m' \leq_\downarrow m$ by $\leq$-App1 and $\leq_\downarrow$-μ-Neutral.

Case $m = \mu\ (\lambda\alpha :: *.\ n)$. If $\mu\ (\lambda\alpha :: \kappa.\ n)$ is non-contractive, then $\mathrm{Ex}(m) = \bot$ and $d = 0$, contradicting assumption. Therefore $\mu\ (\lambda\alpha :: *.\ n)$ is contractive. By successive unrolling of top-level μ, we obtain a term $n_0 \leq_\downarrow m$ such that n_0 does not have the form $\mu\ (\lambda\beta \cdots)$ and $\mathrm{Ex}(n_0) = \mathrm{Ex}(m)$. Use one of the other cases to discover $n_0' \leq_\downarrow n_0$ such that $s = \mathrm{Ex}(n_0')$; by transitivity of $\leq_\downarrow$ we also have $n_0' \leq_\downarrow m$. □

Theorem 59. *For every F_ω^μ type σ, the Berarducci tree of σ^∞ is regular.*

Proof. Let m be the β-normal form of σ. Since $\sigma^\infty \Rightarrow_{\beta\bot}^\infty \mathrm{Ex}(m)$ and $\mathrm{Ex}(m)$ is irreducible by $\Rightarrow_{\beta\bot}^\infty$, the Berarducci tree of σ^∞ is $\mathrm{Ex}(m)$. Let s be a subtree of $\mathrm{Ex}(m)$. By Lemma 58, there exists $m' \leq_\downarrow m$ such that $s = \mathrm{Ex}(m')$. Since the choice of m' is finite, so is the choice of s. □

E. Polytypism with Relevance Tracking

In this section, we continue the discussion about generating traversable functors with macros from Sec. 6. We extend the polytypism theory of Hinze [29] to support traversable functors, and to handle unsupported type constants and free type variables robustly. We will show that Hinze's main theorem continues to hold—that polytypic terms have polykinded types. We close the section with some remaining issues for future investigation.

E.1. Motivation

We will make two extensions to Hinze's theory of polytypism: the *initial context* and *relevance tracking*. Here we motivate both extensions.

The initial context is added to support traversable functors. The type of *traverse* is not *polykinded* in Hinze's sense; it is polytypic only on line (4) below, where G occurs free, and the first argument $g : \mathit{ApplicativeG}$ would be bound at the term level already. Therefore we will allow each polytypic value to transform the empty context into a designated *initial context*. An example is shown immediately after Definition 60.

$$\mathit{traverse}\langle\tau\rangle : \forall G :: * \to *.\ \mathit{Applicative}\ G \to \tag{3}$$

$$\forall\alpha_1\alpha_2.\ (\alpha_1 \to G\ \alpha_2) \to \tau\ \alpha_1 \to G\ (\tau\ \alpha_2) \tag{4}$$

Relevance tracking is added to handle free variables and unsupported constants. In Fig. 15, the macro *traverse* is only defined on types built without function arrows $\to$ or

universal quantifiers $\forall$. Moreover, each free type variable α of τ generates an ubound term variable p_α in $\mathit{traverse}\langle\tau\rangle$, and so the macro is only meaningful on closed types. Let

$$\tau = \lambda\alpha :: *.\ \{\mathit{fst} : \alpha, \mathit{snd} : \beta\}.$$

Then τ has a free variable β. Yet there is a meaningful *traverse* method for the functor corresponding to τ:

$$\begin{aligned}
&\mathit{traverse} : \forall G :: * \to *.\ \mathit{Applicative}\ G \to \forall\alpha_1\alpha_2.\ (\alpha_1 \to G\ \alpha_2) \to \tau\ \alpha_1 \to G\ (\tau\ \alpha_2)\\
&\mathit{traverse} = \Lambda G :: * \to *.\ \lambda g : \mathit{Applicative}\ G.\\
&\qquad \Lambda\alpha_1\alpha_2 :: *.\ \lambda p_\alpha : \alpha_1 \to G\ \alpha_2.\ \lambda x : \{\mathit{fst} : \alpha_1, \mathit{snd} : \beta\}.\\
&\qquad\quad g.\mathit{call}\ [\beta]\ [\{\mathit{fst} = \alpha_2, \mathit{snd} = \beta\}]\\
&\qquad\qquad (g.\mathit{call}\ [\alpha_2]\ [\beta \to \{\mathit{fst} = \alpha_2, \mathit{snd} = \beta\}]\\
&\qquad\qquad\quad (g.\mathit{pure}\ [\alpha_2 \to \beta \to \{\mathit{fst} = \alpha_2, \mathit{snd} = \beta\}]\ (\lambda yz.\ \{\mathit{fst} = y, \mathit{snd} = z\}))\\
&\qquad\qquad\quad (p_\alpha\ x.\mathit{fst}))\\
&\qquad\qquad (g.\mathit{pure}\ x.\mathit{snd})
\end{aligned}$$

The rest of the section describes how to generate terms like *traverse* despite the free variable β.

E.2. Polykinded Types and Polytypic Terms

Definition 60. An n-nary *polykinded type* $\mathit{Poly}\langle\kappa\rangle$ is a family of types indexed by kinds κ satisfying the following properties.

1. There exists a context Γ_0, called the *initial context*, such that for every kind κ,

$$\Gamma_0 \vdash \mathit{Poly}\langle\kappa\rangle :: \underbrace{\kappa \to \cdots \to \kappa}_{n \text{ times}} \to *$$

2. *Poly* satisfies the following equation for all kinds κ_1, κ_2:

$$\begin{aligned}
\mathit{Poly}\langle\kappa_1 \to \kappa_2\rangle = \lambda f_1 \cdots f_n : \kappa_1 \to \kappa_2.\ \forall\alpha_1 \cdots \alpha_n : \kappa_1.\\
\mathit{Poly}\langle\kappa_1\rangle\ \alpha_1 \cdots \alpha_n \to \mathit{Poly}\langle\kappa_2\rangle\ (f_1\ \alpha_1) \cdots (f_n\ \alpha_n).
\end{aligned}$$

The type macro *Trav* in Fig. 15 is a binary polykinded type with initial context

$$\Gamma_0 = G :: * \to *, g : \mathit{Applicative}\ g.$$

Definition 61. Let V be a set of type variables. For each type τ and integer i, the type $\tau|_{V=i}$ is obtained from τ by systematically renaming every free occurrence of α to α_i for all $\alpha \in V$.

Definition 62. Let *Poly* be an n-nary polykinded type with initial context Γ_0 and let V be a set of type variables. For each context Γ, the extended context $\mathit{Poly}\langle\Gamma \mid V\rangle$ is defined inductively as follows.

$$\begin{aligned}
\mathit{Poly}\langle\epsilon \mid V\rangle &= \Gamma_0 \\
\mathit{Poly}\langle\Gamma, \alpha : \kappa \mid V\rangle &= \mathit{Poly}\langle\Gamma \mid V\rangle, \alpha : \kappa && (\text{if } \alpha \notin V) \\
\mathit{Poly}\langle\Gamma, \alpha : \kappa \mid V\rangle &= \mathit{Poly}\langle\Gamma \mid V\rangle, \alpha : \kappa, \alpha_1 : \kappa, \ldots, \alpha_n : \kappa, p_\alpha : \mathit{Poly}\langle\kappa\rangle\ \alpha_1 \cdots \alpha_n && (\text{if } \alpha \in V)
\end{aligned}$$

We model polytypic values as a 3-place relation between kinding judgements, sets of type variables, and terms. The set of type variables in the relation represents the variable we *care about*. In the example in Appendix E.1, we care about α but not β. Although β is free in τ, we should not, and in fact cannot, instantiate *traverse* on the abstract type β.

Definition 63. A *polytypic term poly* with n-nary polykinded type *Poly* is a 3-place relations between kinding judgements, type variable sets, and terms. If $(\Gamma \vdash \tau :: \kappa, V, t) \in \mathit{poly}$, then we write

$$\mathit{poly}\langle\Gamma \vdash \tau :: \kappa \mid V\rangle = t.$$

Where Γ and κ are unimportant, we will also write

$$\mathit{poly}\langle\tau \mid V\rangle = t.$$

Every polytypic term is defined inductively by the following *default rules* together with an unspecified set of *custeom rules*.

$$\frac{\alpha \in V}{\mathit{poly}\langle\alpha \mid V\rangle = p_\alpha} \tag{P-Var}$$

$$\frac{\mathit{poly}\langle\sigma \mid V\rangle = t}{\mathit{poly}\langle\mu\ \sigma \mid V\rangle = \mathit{fix}\ (t\ [\mu\ \sigma\ |_{V=1}] \cdots [\mu\ \sigma\ |_{V=n}])} \tag{P-Fix}$$

$$\frac{\mathit{poly}\langle\sigma\rangle = t_\sigma \qquad \mathit{poly}\langle\tau\rangle = t_\tau}{\mathit{poly}\langle\sigma\ \tau\rangle = t_\sigma\ [\tau|_{V=1}] \cdots [\tau|_{V=n}]\ t_\tau} \tag{P-App}$$

$$\frac{\mathit{poly}\langle\Gamma, \alpha :: \kappa_\alpha \vdash \sigma :: \kappa \mid V \cup \alpha\rangle = t}{\mathit{poly}\langle\Gamma \vdash \lambda\alpha :: \kappa_\alpha.\ \sigma :: \kappa_\alpha \to \kappa \mid V\rangle = \Lambda\alpha_1 \cdots \alpha_n :: \kappa_\alpha.\ \lambda p_\alpha : \mathit{Poly}\langle\kappa_\alpha\rangle\ \alpha_1 \cdots \alpha_n.\ t} \tag{P-Abs}$$

Despite the notation $\mathit{poly}\langle\tau \mid V\rangle = t$, our polytypic terms are relations and not necessarily functions. They are allowed to have terms $t_1 \neq t_2$ such that $\mathit{poly}\langle\tau \mid V\rangle = t_1$ and $\mathit{poly}\langle\tau \mid V\rangle = t_2$ at the same time.

The the macro *trav* in Sec. 6 is a polytypic term with $V = \mathrm{fv}(\tau)$. In fact, we obtain Hinze's notion of polytypic terms if we set $V = \mathrm{fv}(\tau)$ and set the initial context Γ_0 to the empty context.

Definition 64. Let *poly* be a polytypic term of n-nary polykinded type *Poly*. A judgement $poly\langle \Gamma \vdash \tau :: \kappa \mid V \rangle = t$ is *well-typed* if

$$Poly\langle \Gamma \mid V \rangle \vdash t : Poly\langle \kappa \rangle\ (\tau|_{V=1}) \cdots (\tau|_{V=n}).$$

We will now show a result analogous to Theorem 1 in Hinze [29], that the polytypic terms are well-typed. The main difference is in how variables are renamed in the type arguments $(\tau|_{V=1}) \cdots (\tau|_{V=n})$. In Hinze [29], all free variables are renamed. In our extension, only relevant variables (i. e., members of V) are renamed. Nevertheless, the proof remains similar.

Theorem 65. *Let poly be a polytypic term of n-nary polykinded type Poly. If all custom rules of poly derive well-typed judgements, then all members of poly are well-typed.*

Proof. Suppose

$$poly\langle \Gamma \vdash \tau :: \kappa \mid V \rangle = t.$$

We will prove

$$Poly\langle \Gamma \mid V \rangle \vdash t : Poly\langle \kappa \rangle\ (\tau|_{V=1}) \cdots (\tau|_{V=n})$$

by induction on $poly\langle \tau \mid V \rangle = t$. Since judgements produced by custom rules are well-typed by assumption, we need only reason about the default rules.

Case P-Var: We have $\tau = \alpha$, and $\alpha : \kappa \in \Gamma$, and $\alpha \in V$. Then

$$t = p_\alpha,$$
$$Poly\langle \kappa \rangle\ (\alpha|_{V=1}) \cdots (\alpha|_{V=n}) = Poly\langle \kappa \rangle\ \alpha_1 \cdots \alpha_n.$$

Since $\alpha \in V$,

$$p_\alpha : Poly\langle \kappa \rangle\ \alpha_1 \cdots \alpha_n \in Poly\langle \Gamma \mid V \rangle.$$

The desired typing judgement follows from T-Var.

Case P-Fix: We have $\tau = \mu\ \sigma$. Write

$$\sigma_i = \sigma|_{V=i}, \qquad \tau_i = \tau|_{V=i} = \mu\ \sigma_i.$$

Then

$$t = fix\ (t_\sigma\ [\tau_1] \cdots [\tau_n])$$

for some t_σ such that

$$poly\langle \Gamma \vdash \sigma :: \kappa \to \kappa \mid V \rangle = t_\sigma.$$

By the induction hypothesis,

$$Poly\langle \Gamma \mid V \rangle \vdash t_\sigma : Poly\langle \kappa \to \kappa \rangle\ \sigma_1 \cdots\ \sigma_n.$$

By the definition of polykinded types,

$$Poly\langle \kappa \to \kappa \rangle\ \sigma_1 \cdots\ \sigma_n = \forall \alpha \cdots \alpha_n : \kappa.\ Poly\langle \kappa \rangle\ \alpha_1 \cdots \alpha_n \to Poly\langle \kappa \rangle\ (\sigma_1\ \alpha_1) \cdots (\sigma_n\ \alpha_n).$$

By T-TApp and T-Eq,

$$Poly\langle\Gamma \mid V\rangle \vdash t_\sigma\ [\tau_1]\cdots[\tau_n] : Poly\langle\kappa\rangle\ \tau_1\cdots\tau_n \to Poly\langle\kappa\rangle\ (\sigma_1\ \tau_1)\cdots(\sigma_n\ \tau_n).$$

Since type equivalence is a congruence relation, from $\tau_i \equiv \sigma_i\ \tau_i$ we can derive

$$Poly\langle\kappa\rangle\ \tau_1\cdots\tau_n \equiv Poly\langle\kappa\rangle\ (\sigma_1\ \tau_1)\cdots(\sigma_n\ \tau_n).$$

By T-Eq,

$$Poly\langle\Gamma \mid V\rangle \vdash t_\sigma\ [\tau_1]\cdots[\tau_n] : Poly\langle\kappa\rangle\ \tau_1\cdots\tau_n \to Poly\langle\kappa\rangle\ \tau_1\cdots\tau_n.$$

We may now apply T-App after T-Const for the fixed-point combinator *fix* to obtain the desired typing judgement.

Case P-App: We have $\tau = \rho\ \sigma$ and

$$t = t_\rho\ [\sigma|_{V=1}]\cdots[\sigma|_{V=n}]\ t_\sigma$$

for some t_ρ, t_σ such that

$$poly\langle\Gamma \vdash \rho : \kappa_\sigma \to \kappa \mid V\rangle = t_\rho, \qquad poly\langle\Gamma \vdash \sigma : \kappa_\sigma \mid V\rangle = t_\sigma.$$

Write

$$\rho_i = \rho|_{V=i}, \qquad \sigma_i = \sigma|_{V=i}, \qquad \tau_i = \tau|_{V=i} = \rho_i\ \sigma_i.$$

By the induction hypothesis,

$$Poly\langle\Gamma \mid V\rangle \vdash t_\rho : Poly\langle\kappa_\sigma \to \kappa\rangle\ \rho_1\cdots\rho_n,$$
$$Poly\langle\Gamma \mid V\rangle \vdash t_\sigma : Poly\langle\kappa_\sigma\rangle\ \sigma_1\cdots\sigma_n.$$

From there, it is possible to derive the desired conclusion by expanding $Poly\langle\kappa_\sigma \to \kappa\rangle$ and using T-TApp, T-App and T-Eq as appropriate.

Case P-Abs: We have $\tau = \lambda\alpha : \kappa_\alpha.\ \sigma$ and

$$t = \Lambda\alpha_1\cdots\alpha_n :: \kappa_\alpha.\ \lambda p_\alpha : Poly\langle\kappa_\alpha\rangle\ \alpha_1\cdots\alpha_n.\ t_\sigma$$

for some t_σ such that

$$poly\langle\Gamma, \alpha : \kappa_\alpha \vdash \sigma : \kappa_\sigma \mid V \cup \{\alpha\}\rangle = t_\sigma.$$

Write

$$\begin{aligned}\tau_i &= \tau|_{V=i} & \sigma_i &= \sigma|_{V\cup\{\alpha\}=i}\\ &= \lambda\alpha : \kappa_\alpha.\ \sigma|_{V=i}, & &\equiv \tau_i\ \alpha_i.\end{aligned}$$

By the induction hypothesis,

$$Poly\langle\Gamma \mid V \cup \{\alpha\}\rangle \vdash t_\sigma : Poly\langle\kappa_\sigma\rangle\ \sigma_1\cdots\sigma_n.$$

Applying T-TAbs and T-Abs several times, we obtain

$$\begin{aligned}&\mathit{Poly}\langle\Gamma \mid V\rangle \vdash \\ &\quad \Lambda\alpha_1 \cdots \alpha_n : \kappa.\ \lambda p_\alpha : \mathit{Poly}\langle\kappa_\alpha\rangle\ \alpha_1 \cdots \alpha_n.\ t_\sigma : \\ &\qquad \forall\alpha_1 \cdots \alpha_n : \kappa_\alpha.\ \mathit{Poly}\langle\kappa_\alpha\rangle\ \alpha_1 \cdots \alpha_n \to \mathit{Poly}\langle\kappa_\sigma\rangle\ \sigma_1 \cdots \sigma_n\end{aligned}$$

The desired typing judgement follows from T-Eq and the congruence property of type equality, using the fact that $\sigma_i \equiv \tau_i\ \alpha_i$. □

E.3. Generating Traversable Functors

We generate traversable functors using the type macro *Traverse* and the term macro *traverse*. They are defined using polykinded type *Trav* and polytypic term *trav*. The definition of *Trav* is in Fig. 15 on page 32. The polytypic term *trav* is defined by custom rules for record, variant and μ-types in Fig. 15 together with the following rule for irrelevant positions.

$$\frac{V \cap \mathrm{fv}(\tau) = \emptyset}{\mathit{trav}\langle\Gamma \vdash \tau :: * \mid V\rangle = g.\mathit{pure}\ [\tau]} \qquad \text{(P-Irrelevant)}$$

The rule P-Irrelevant produce well-typed judgements, because

$$\mathit{Trav}\langle *\rangle\ (\tau|_{V=1})\ (\tau|_{V=2}) = \mathit{Trav}\langle *\rangle\ \tau\ \tau \equiv \tau \to G\ \tau.$$

The macro *traverse* invokes the polytypic term *trav* that does not care about any variable:

$$\mathit{traverse}\langle\tau\rangle = \Lambda G :: * \to *.\ \lambda g : \mathit{Applicative}\ G.\ \mathit{trav}\langle\tau \mid \emptyset\rangle$$

Instantiating τ to the example in Appendix E.1, the polytypic term *trav* starts to care about α at its binding site, but never cares about β. We have $\mathit{trav}\langle\beta \mid \{\alpha\}\rangle = g.\mathit{pure}\ [\beta]$, which eventually results in the expected code for *traverse* in Appendix E.1.

E.4. Discussion

In this subsection, we discuss the relation between polytypism and *parametricity transformation*, and issues related to *faithfulness*, an indicator for whether the behavior of polytypic values match user expection.

Parametricity Transformation

We have redeveloped polytypism specifically for $F^{\mu *}_\omega$, but polytypic values are in fact definable in all pure type systems, as extensions to the *parametricity transformation* [8, 9]. Since System $F^{\mu *}_\omega$ is itself a pure type system extended by constants, one can reformulate polytypism *without* relevance tracking in terms of Bernardy et al.'s *reflecting systems* [9, definition 3.3]. The input language is taken to be Λ^μ, the type-level language of F^μ_ω (Fig. 3). To make it a pure type system, we add a top-level sort $\Box$ such that

$\kappa : \Box$ for all kinds κ. If we reflect $\Box$ into $*$ in F^{μ}_{ω}, then the parametricity transformation provides exactly the default rules of polytypic terms (Definition 63). The parametricity transformation is undefined for Λ^{μ}-constants that are not sorts ($*$, $\to$, $\forall$, $\{\overline{l_i}\}$, $\langle\overline{l_i}\rangle$); one may supply custom rules for them to complete a polytypic term. In addition, the empty context needs to transform to the initial context for macros like *traverse*. The abstraction theorem of the parametricity transformation [9, theorem 3.12] can be extended to allow custom rules and initial contexts, and it would coincide with the correctness theorem of polytypism, namely that polytypic values have polykinded types.

While it is possible to make the paragraph above precise, we shall remain informal and leave the details for another occasion. The reader is encouraged to work out the formalism for themselves.

Faithfulness

Let us call a program transformation F *faithful* if it transforms equivalent programs into equivalent programs and different programs into different programs:

$$s \approx_0 t \qquad \text{if and only if} \qquad F(s) \approx_1 F(t).$$

Faithfulness is a desirable property. If we view each polytypic term as a domain-specific language for program generation, then faithfulness corresponds to watertight abstraction: The user may carry out equational reasoning entirely within the source language, without ever thinking about the generated code.

Faithfulness is a bi-implication. The forward direction is the preservation of certain equivalence relations. The parametricity transformation preserves β-equivalence [8, 9], which is cause to believe that forward faithfulness holds for Hinze's polytypism as well. Unfortunately, the custom rule P-Irrelevant destroys forward faithfulness. There are two non-equivalent expansions of $\mathit{traverse}\langle\sigma\rangle$ for the type σ below depending on whether P-Irrelevant or P-App is used first. To restore forward faithfulness, we may have to incorporate the laws of traversable functors.

$$\sigma = (\lambda\alpha :: *.\ \{\mathit{fst} : \alpha\})\ \mathit{Int}$$

The backward direction of faithfulness means that non-equivalent programs are transformed into non-equivalent programs. Compilers typically care more about equalities than inequalities, since they want to optimize by rewriting. We do not know any backward faithfulness result about polytypism or the parametricity transformation.

F. Digression: Coinduction on Sets and Terms

So far we have defined coinductive relations in terms of inference rules, and carried out coinductive arguments in the informal style advocated in Kozen and Silva [37]. In this section, we descend into a lower level of abstraction, and view coinductive relations as the greatest fixed points of monotone set functions. This low-level view allows us to explain why Fig. 19 is called the *support* of $\equiv_{\mu}$, and to give an alternative proof of Lemma 42, a key step of the commuting diagram lemma.

F.1. Coinduction on Sets

We will recall the presentation of coinduction in chapter 21 of Pierce [45] and add a few examples. Here are the definitions of monotone set functions, F-closure and F-consistency from Pierce.

Definition 66 (monotonicity)**.** Fix a set $\mathcal{U}$ as the universe of discourse. Let $2^{\mathcal{U}}$ be the power set of $\mathcal{U}$. A *monotone set function* F is a function from $2^{\mathcal{U}}$ to $2^{\mathcal{U}}$ such that $F(X) \subseteq F(Y)$ for all $X, Y \subseteq \mathcal{U}$ satisfying $X \subseteq Y$.

Definition 67 (closure, consistency and fixed points)**.** Suppose $X \subseteq \mathcal{U}$ and let F be a monotone set function.

1. X is *F-closed* if $F(X) \subseteq X$.
2. X is *F-consistent* if $X \subseteq F(X)$.
3. X is a *fixed point* of F if $F(X) = X$. In other words, X is both F-closed and F-consistent.
4. X is the *least fixed point* of F, written $X = \mu F$, if X is a fixed point of F such that every other fixed point of F is a superset of X.
5. X is the *greatest fixed point* of F, written $X = \nu F$, if X is a fixed point of F such that every other fixed point of F is a subset of X.

Recall Knaster-Tarski theorem (Thm. 21.1.4 in Pierce [45]) and the principle of coinduction (Cor. 21.1.8 in Pierce [45]).

Theorem 68 (Knaster-Tarski)**.** *Every monotone set function F has a greatest fixed point, namely the union of all F-consistent sets.*

Corollary 69 (principle of coinduction)**.** *If X is F-consistent, then $X \subseteq \nu F$.*

Example 70. Coinductively defined terms are the greatest fixed points of their *generating functions*. Let the universe of discourse $\mathcal{U}$ be the set of all syntax trees. Infinitary λ-calculus Λ^∞ is generated by the monotone function G^∞_Λ. We use L as the parameter of G^∞_Λ because a collection of syntax trees form a language. Write C for the set of constants ι, and V for the set of variables α. Note how components of $G^\infty_\Lambda(L)$ correspond to the grammar rules defining Λ^∞ (Fig. 7).

$$\begin{aligned} G^\infty_\Lambda(L) = \{\bot\} \cup C \cup V & \qquad \text{(bottom, constants and variables)} \\ \cup \{\lambda\alpha.\ \tau \mid \alpha \in V \text{ and } \tau \in L\} & \qquad \text{(abstraction)} \\ \cup \{\sigma\ \tau \mid \sigma, \tau \in L\} & \qquad \text{(application)} \end{aligned}$$

Example 71. Let C be the set of constants and V the set of variables. Let the universe of discourse $\mathcal{U}$ be the universal relation $\Lambda^\infty \times \Lambda^\infty$ on infinitary lambda terms. Let us write down the generating function $G^\infty_{\beta\bot}$ of Böhm-reduction (Definition 5). Since subsets

of $\mathcal{U}$ are relations on infinitary λ terms, we use the letter R for the argument of $G^{\infty}_{\beta\bot}$. Note how components of $G^{\infty}_{\beta\bot}(R)$ correspond to the coinduction rules of $\Rightarrow^{\infty}_{\beta\bot}$ (Fig. 11). In this sense, generating functions are descriptions of coinductively-defined objects at a lower level of abstraction.

$$
\begin{aligned}
G^{\infty}_{\beta\bot}(R) = {} & \{(\tau, x) \mid x \in \{\bot\} \cup V \cup C \text{ and } \tau \Rightarrow^{*}_{\beta\bot} x\} \\
& \hspace{12em} \text{(B-Bot, B-Const and B-B-Var)} \\
& \cup \{(\sigma, \lambda\alpha.\ \tau') \mid \sigma \Rightarrow^{*}_{\beta\bot} (\lambda\alpha.\ \tau) \text{ and } (\tau, \tau') \in R\} \qquad \text{(B-Abs)} \\
& \cup \{(\sigma, \tau_1'\ \tau_2') \mid \sigma \Rightarrow^{*}_{\beta\bot} \tau_1\ \tau_2 \text{ and } (\tau_1, \tau_1'), (\tau_2, \tau_2)' \in R\} \qquad \text{(B-App)}
\end{aligned}
$$

Example 72. Let $\mathcal{U} = \mathrm{NF}^{\mu *} \times \mathrm{NF}^{\mu *}$, the universal relation on finite, $F^{\mu *}_{\omega}$ types in β-normal form. Let us write down the generating function $G^{\equiv}_{\mu}$ of μ-equivalence $\equiv_\mu$ (Definition 12). As usual, we tag components of $G^{\equiv}_{\mu}(R)$ by corresponding inference rules (Fig. 13).

$$
\begin{aligned}
G^{\equiv}_{\mu}(R) = {} & \{(x, x) \mid x = \alpha \text{ or } x = \iota\} \qquad \text{(EQ-TVar and EQ-Prim)} \\
& \cup \{(n_1\ m_1, n_2\ m_2) \mid (n_1, n_2) \in R, (m_1, m_2) \in R\} \qquad \text{(Eq-AppCong)} \\
& \cup \{(\lambda\alpha : \kappa.\ m_1, \lambda\alpha : \kappa.\ m_2) \mid (m_1, m_2) \in R\} \qquad \text{(Eq-}\xi\text{)} \\
& \cup \{(\mu\ n_1, m_2) \mid (n_1\ (\mu\ n_1), m_2) \in R\} \qquad \text{(Eq-}\mu_L\text{-Neutral)} \\
& \cup \left\{ (m_1, \mu\ n_2) \;\middle|\; \begin{array}{l} m_1 \text{ does not start with } \mu \\ (m_1, n_2\ (\mu\ n_2)) \in R \end{array} \right\} \qquad \text{(Eq-}\mu_R\text{-Neutral)} \\
& \cup \left\{ (\mu\ (\lambda\alpha : *.\ n_1), m_2) \;\middle|\; \begin{array}{l} \mu\ (\lambda\alpha : *.\ n_1) \text{ is contractive} \\ ([\alpha \mapsto \mu\ (\lambda\alpha : *.\ n_1)]n_1, m_2) \in R \end{array} \right\} \qquad \text{(Eq-}\mu_L\text{)} \\
& \cup \left\{ (m_1, \mu\ (\lambda\beta : *.\ n_2)) \;\middle|\; \begin{array}{l} m_1 \text{ does not start with } \mu \\ \mu\ (\lambda\beta : *.\ n_2) \text{ is contractive} \\ (m_1, [\beta \mapsto \mu\ (\lambda\beta : *.\ n_2)]n_2) \in R \end{array} \right\} \qquad \text{(Eq-}\mu_R\text{)} \\
& \cup \{(m_1, m_2) \mid m_1 \text{ and } m_2 \text{ are both non-contractive}\} \qquad \text{(Eq-}\mu\bot\text{)}
\end{aligned}
$$

A term relation is *syntax-directed* if each judgement between two concrete terms has a unique rule deriving it, discovered usually by examining the top-level construct of the terms. *Invertible functions* are the analogous concept for monotone set functions. The *support* of an invertible function is analogous to the subroutine computing the last rule in a judgement from concrete terms in a syntax-directed relation.

Definition 73 (invertibility)**.** A monotone set function F is *invertible* if for all $x \in \mathcal{U}$, either $x \notin F(\mathcal{U})$ or else there exists $X \subseteq \mathcal{U}$ such that $x \in F(X)$, and we have $X \subseteq Y$ whenever $x \in F(Y)$.

Definition 74 (support)**.** Let F be an invertible monotone set function. The *support* of F is a partial function from $\mathcal{U}$ to its powerset $2^{\mathcal{U}}$ mapping each $x \in F(\mathcal{U})$ to the smallest X such that $x \in F(X)$. If no such X exists, then the support of F is undefined on x.

Example 75. The generating function G^{∞}_{Λ} of Λ^{∞} is invertible. Here is its support.

$$\mathrm{support}_{G^{\infty}_{\Lambda}}(x) = \begin{cases} \emptyset & \text{if } x \in \{\bot\} \cup C \cup V, \\ \{\tau\} & \text{if } x = \lambda\alpha.\ \tau, \\ \{\sigma, \tau\} & \text{if } x = \sigma\ \tau. \end{cases}$$

Example 76. Böhm-reduction $\Rightarrow^{\infty}_{\beta\bot}$ is not invertible. However, μ-equivalence $\equiv_{\mu}$ is designed so that its generating function $G^{\equiv}_{\mu}$ is invertible. Fig. 19 on page 55 shows its support.

F.2. Recursively Defined Infinitary Terms and Böhm Reduction

We investigate the fixed points of contexts in infinitary lambda calculus. By *contexts* we mean term transformations f of the form $f(\sigma) = [\alpha \mapsto \sigma]\tau$. We will work toward a proof that taking fixed points preserve Böhm reduction (Theorem 80). The result is used in the alternative proof of the commuting diagram lemma, and is interesting on its own.

Definition 77. Write $\sigma =_n \tau$ if the Λ^{∞} terms σ, τ are equal up to depth n.

Clearly $=_n$ is reflexive, symmetric and transitive. We shall take it for granted that $\sigma = \tau$ if and only if $\sigma =_n \tau$ for all $n \in \mathbb{N}$; this is provable given a fixed encoding of infinitary λ-terms (e. g., as a co-datatype with de Bruijn indices).

Lemma 78 (fixed point of infinite context)**.** *Let $\tau \neq \alpha$ be an infinite term. There exists a unique infinite term σ such that $\sigma = [\alpha \mapsto \sigma]\tau$. Moreover, $\alpha \notin \mathrm{fv}(\sigma)$.*

We call σ the *fixed point* of τ with respect to α and write

$$\sigma = Fix_{\alpha}\tau.$$

Proof. Let f be the function on Λ^{∞} such that $f(\rho) = [\alpha \mapsto \rho]\tau$. Since $\tau \neq \alpha$,

$$f(\rho_1) =_{n+1} f(\rho_2) \qquad \text{if and only if} \qquad \rho_1 =_n \rho_2 \tag{5}$$

for all $\rho_1, \rho_2 \in \Lambda^{\infty}$ and $n \in \mathbb{N}$. As a result,

$$f^n(\bot) =_n f^{n+k}(\bot)$$

for all $n, k \in \mathbb{N}$. Moreover, $\rho_1 =_n \rho_2$ implies $f(\rho_1) =_n f(\rho_2)$.

Define σ as the infinitary λ-term whose depth-i constructs agree with $f^i(\bot)$. Since α does not occur free in $f^i(\bot)$ for all i, it does not occur free in σ. Moreover, $\sigma =_n f^m(\bot)$ for all $m \geq n$. Therefore for all $n \in \mathbb{N}$,

$$f(\sigma) =_n f(f^n(\bot)) = f^{n+1}(\bot) =_n \sigma.$$

Since $f(\sigma) =_n \sigma$ for all n, we conclude that $f(\sigma) = \sigma$. This establishes the existence of $Fix_{\alpha}\tau$.

For uniqueness, let σ' be another fixed point of f. Then we have $\sigma =_n f^n(\bot) =_n \sigma'$ for all $n \in \mathbb{N}$, which gives us $\sigma = \sigma'$. □

Example 79. The notation $Fix_\alpha\tau$ allows us to write the infinite tree μ^∞ in Fig. 8 (page 21) formally:

$$\mu^\infty = \lambda\alpha.\ Fix_\beta(\alpha\ \beta).$$

Theorem 80. *Let σ, τ be infinitary lambda terms such that $\tau \neq \alpha$ and $\sigma \Rightarrow^\infty_{\beta\bot} \tau$. Then $Fix_\alpha\sigma \Rightarrow^\infty_{\beta\bot} Fix_\alpha\tau$.*

Proof. By the principle of coinduction (Corollary 69), it suffices to construct a set containing $(Fix_\alpha\sigma, Fix_\alpha\tau)$ that is consistent with respect to the generating function $G^\infty_{\beta\bot}$ of Böhm reduction (Example 71). We will construct this set as the greatest fixed point of a monotone set function G, defined below.

Let R_0 be the relation such that

$$R_0 = \{(\sigma_0, \tau_0) \mid \sigma_0 \Rightarrow^\infty_{\beta\bot} \tau_0 \text{ and } \tau_0 \neq \alpha \text{ is a subterm of } \tau\}.$$

G is the set function

$$G(R) = \left\{ ([\alpha \mapsto \rho_1]\sigma_0, [\alpha \mapsto \rho_2]\tau_0) \;\middle|\; \begin{array}{l} (\sigma_0, \tau_0) \in R_0 \\ (\rho_1, \rho_2) \in R \\ \alpha \notin \mathrm{fv}(\rho_1) \cup \mathrm{fv}(\rho_2) \end{array} \right\}.$$

G is clearly monotone. We will show that $(Fix_\alpha\sigma, Fix_\alpha\tau)$ is contained in νG and that νG is $G^\infty_{\beta\bot}$-consistent.

Containment. By the coinduction principle, it suffices to show that

$$S = \{(Fix_\alpha\sigma, Fix_\alpha\tau)\}$$

is G-consistent. Choose

$$\rho_1 = Fix_\alpha\sigma, \qquad \sigma_0 = \sigma,$$
$$\rho_2 = Fix_\alpha\tau, \qquad \tau_0 = \tau.$$

Since $\tau \neq \alpha$ and $\sigma \Rightarrow^\infty_{\beta\bot} \tau$, we have $(\sigma_0, \tau_0) \in R_0$. By the definition of S, $(\rho_1, \rho_2) \in S$. Thus

$$([\alpha \mapsto \rho_1]\sigma_0, [\alpha \mapsto \rho_2]\tau_0) \in G(S).$$

However,

$$[\alpha \mapsto \rho_1]\sigma_0 = Fix_\alpha\sigma, \qquad [\alpha \mapsto \rho_2]\tau_0 = Fix_\alpha\tau,$$

whence $S \subseteq G(S)$.

$G^{\infty}_{\beta\bot}$-**Consistency.** We want to show $\nu G \subseteq G^{\infty}_{\beta\bot}(\nu G)$. Since $\nu G \subseteq G(\nu G)$, every member of νG has the form

$$([\alpha \mapsto \rho_1]\sigma_0, [\alpha \mapsto \rho_2]\tau_0)$$

such that $(\rho_1, \rho_2) \in \nu G$ and $(\sigma_0, \tau_0) \in R_0$. Let us case-split on τ_0.

Case $\tau_0 = \bot$, ι or β. We have $\tau_0 \neq \alpha$ and $\sigma_0 \Rightarrow^*_{\beta\bot} \tau_0$. By Corollary 22,

$$\begin{aligned}[\alpha \mapsto \rho_1]\sigma_0 &\Rightarrow^*_{\beta\bot} [\alpha \mapsto \rho_1]\tau_0 \\ &= \tau_0 \\ &= [\alpha \mapsto \rho_2]\tau_0.\end{aligned}$$

Applying B-Bot, B-Const or B-Var as appropriate, we obtain

$$([\alpha \mapsto \rho_1]\sigma_0, [\alpha \mapsto \rho_2]\tau_0) \in G^{\infty}_{\beta\bot}(X)$$

for whatever $X \subseteq \Lambda^{\infty} \times \Lambda^{\infty}$.

Case $\tau_0 = \lambda\beta.\ \tau_{00}$. By the definition of R_0,

$$\sigma_0 \Rightarrow^*_{\beta\bot} \lambda\beta.\ \sigma_{00}, \qquad\qquad \sigma_{00} \Rightarrow^{\infty}_{\beta\bot} \tau_{00}.$$

We split the argument further into the subcase $\tau_{00} \neq \alpha$ and the subcase $\tau_{00} = \alpha$.

Subcase $\tau_{00} \neq \alpha$. Then $(\sigma_{00}, \tau_{00}) \in R_0$ and

$$([\alpha \mapsto \rho_1]\sigma_{00}, [\alpha \mapsto \rho_2]\tau_{00}) \in G(\nu G) = \nu G. \tag{6}$$

By Corollary 22,

$$[\alpha \mapsto \rho_1]\sigma_0 \Rightarrow^*_{\beta\bot} [\alpha \mapsto \rho_1](\lambda\beta.\ \sigma_{00}).$$

Using Eq. (6) as the 2nd condition of B-Abs in Example 71, we obtain

$$([\alpha \mapsto \rho_1]\sigma_0, [\alpha \mapsto \rho_2](\lambda\beta.\ \tau_{00})) \in G^{\infty}_{\beta\bot}(\nu G).$$

Subcase $\tau_{00} = \alpha$. Then $\sigma_{00} \Rightarrow^{\infty}_{\beta\bot} \alpha$, and hence $\sigma_{00} \Rightarrow^*_{\beta\bot} \alpha$. We have to examine $(\rho_1, \rho_2) \in \nu G$:

$$\begin{aligned}\rho_1 &= [\alpha \mapsto \rho_{11}]\rho_{10}, & (\rho_{10}, \rho_{20}) &\in R_0, \\ \rho_2 &= [\alpha \mapsto \rho_{21}]\rho_{20}, & (\rho_{11}, \rho_{21}) &\in \nu G.\end{aligned}$$

Note that $\alpha \notin \mathrm{fv}(\rho_{11})$ by the definition of G. By Property 19,

$$\begin{aligned}[\alpha \mapsto \rho_1]\sigma_{00} &= [\alpha \mapsto [\alpha \mapsto \rho_{11}]\rho_{10}]\sigma_{00} \\ &= [\alpha \mapsto \rho_{11}][\alpha \mapsto \rho_{10}]\sigma_{00}.\end{aligned}$$

Recall that $\sigma_{00} \Rightarrow^*_{\beta\bot} \alpha$. By Corollary 22,

$$\begin{aligned}[\alpha \mapsto \rho_{10}]\sigma_{00} &\Rightarrow^*_{\beta\bot} [\alpha \mapsto \rho_{10}]\alpha \\ &= \rho_{10} \Rightarrow^{\infty}_{\beta\bot} \rho_{20},\end{aligned}$$

therefore $[\alpha \mapsto \rho_{10}]\sigma_{00} \Rightarrow^{\infty}_{\beta\bot} \rho_{20}$. Since $(\rho_{10}, \rho_{20}) \in R_0$, we know $\rho_{20} \neq \alpha$ is a subterm of τ. Thus

$$([\alpha \mapsto \rho_{10}]\sigma_{00}, \rho_{20}) \in R_0.$$

This is enough to establish

$$([\alpha \mapsto \rho_{11}][\alpha \mapsto \rho_{10}]\sigma_{00}, [\alpha \mapsto \rho_{21}]\rho_{20}) \in G(\nu G) = \nu G.$$

Recall that

$$\begin{aligned}[\alpha \mapsto \rho_1]\sigma_{00} &= [\alpha \mapsto \rho_{11}][\alpha \mapsto \rho_{10}]\sigma_{00},\\ [\alpha \mapsto \rho_2]\tau_{00} &= \rho_2 = [\alpha \mapsto \rho_{21}]\rho_{20}.\end{aligned}$$

Thus we have, in fact,

$$([\alpha \mapsto \rho_1]\sigma_{00}, [\alpha \mapsto \rho_2]\tau_{00}) \in \nu G.$$

Since $[\alpha \mapsto \rho_1]\sigma_0 \Rightarrow^{*}_{\beta\bot} (\lambda\beta.\ [\alpha \mapsto \rho_1]\sigma_{00})$, B-Abs gives us

$$([\alpha \mapsto \rho_1]\sigma_0, \lambda\beta.\ [\alpha \mapsto \rho_2]\tau_{00}) \in G^{\infty}_{\beta\bot}(\nu G),$$

as desired.

Case $\tau_0 = \tau_{01}\ \tau_{02}$. The argument is similar. Suppose $\sigma_0 \Rightarrow^{*}_{\beta\bot} \sigma_{01}\ \sigma_{02}$ such that $\sigma_{0i} \Rightarrow^{\infty}_{\beta\bot} \tau_{0i}$ for $i \in \{1, 2\}$. Then the intermediate goal is

$$([\alpha \mapsto \rho_1]\sigma_{0i}, [\alpha \mapsto \rho_2]\tau_{0i}) \in \nu G,$$

which is shown by splitting the subcases $\tau_{0i} \neq \alpha$ and $\tau_{0i} = \alpha$. □

F.3. Alternative Proof of a Step in the Commuting Diagram Lemma

We provide an alternative proof of Lemma 42, a key step in the commuting diagram lemma. Using Theorem 80, we need not rely on structural properties of $\mathrm{NF}^{\mu *}$ any more.

Lemma 81. *Let $n \neq \alpha$ be an infinite neutral term in NF^{∞}. Then $Fix_{\alpha} n$ is an infinite neutral term.*

Proof. Let $f(\sigma) = [\alpha \mapsto \sigma]n$. By Lemma 78, $Fix_{\alpha} n = f^i(Fix_{\alpha})$ for all $i \in \mathbb{N}$. Since n is neutral and $n \neq \alpha$, we know $f^i(\sigma)$ contains no β-redex at depth i. Thus Fix_{α} contains no β-redex at any depth. By Lemma 37, Fix_{α} is an infinite neutral term. □

Lemma 42. $m^{\infty} \Rightarrow^{\infty}_{\beta\bot} \mathrm{Ex}(m)$ *for all* $m \in \mathrm{NF}^{\mu *}$.

Alternative proof. By induction on m. We omit routine arguments and showing the interesting cases, where m starts with μ.

Case $m = \mu\ n$. We have $m^\infty = \mu^\infty\ n^\infty$. By Example 79,

$$\mu^\infty\ n^\infty \Rightarrow_\beta Fix_\alpha(n^\infty\ \alpha)$$

for some fresh name α. By the induction hypothesis, $n^\infty \Rightarrow^\infty_{\beta\bot} \mathrm{Ex}(n)$. Theorem 80 gives us

$$\begin{aligned} Fix_\alpha(n^\infty\ \alpha) &\Rightarrow^\infty_{\beta\bot} Fix_\alpha(\mathrm{Ex}(n)\ \alpha) \\ &= \mathrm{Ex}(n)\ \mathrm{Ex}(\mu\ n) = \mathrm{Ex}(\mu\ n). \end{aligned}$$

Thus

$$m^\infty \Rightarrow_\beta Fix_\alpha(n^\infty\ \alpha) \Rightarrow^\infty_{\beta\bot} \mathrm{Ex}(m).$$

The desired relation $m^\infty \Rightarrow^\infty_{\beta\bot} \mathrm{Ex}(m)$ follows from Lemma 18.

Case $m = \mu\ (\lambda\alpha :: *.\ n)$ and m is contractive. Then $n \neq \alpha$, $n^\infty \neq \alpha$ and $\mathrm{Ex}(n) \neq \alpha$. By Lemma 33,

$$\mathrm{Ex}(m) = [\alpha \mapsto \mathrm{Ex}(m)]\mathrm{Ex}(n) = Fix_\alpha \mathrm{Ex}(n).$$

By the induction hypothesis,

$$n^\infty \Rightarrow^\infty_{\beta\bot} \mathrm{Ex}(n).$$

By Theorem 80,

$$Fix_\alpha n^\infty \Rightarrow^\infty_{\beta\bot} Fix_\alpha \mathrm{Ex}(n).$$

By transitivity of Böhm-reduction (Lemma 6), for $m^\infty \Rightarrow^\infty_{\beta\bot} \mathrm{Ex}(m)$ to hold, it suffices to show

$$\mu^\infty\ (\lambda\alpha.\ n^\infty) \Rightarrow^\infty_{\beta\bot} Fix_\alpha n^\infty.$$

Note from Example 79 that

$$\mu^\infty\ (\lambda\alpha.\ n^\infty) \Rightarrow_\beta Fix_\alpha\ ((\lambda\alpha.\ n^\infty)\ \alpha)\,.$$

Using the fact that $(\lambda\alpha.\ n^\infty)\ \alpha \Rightarrow_\beta n^\infty$, we appeal to Theorem 80 and conclude

$$Fix_\alpha\ ((\lambda\alpha.\ n^\infty)\ \alpha) \Rightarrow^\infty_{\beta\bot} Fix_\alpha n^\infty.$$

Case $m = \mu\ (\alpha :: *.\ n)$ and m is non-contractive. We know $\mathrm{Ex}(m) = \bot$ and $m^\infty \neq \bot$. To prove $m^\infty \Rightarrow^\infty_{\beta\bot} \mathrm{Ex}(m)$, we need to show that m^∞ is root-active. Since m is non-contractive,

$$m^\infty = \mu^\infty\ (\lambda\alpha_1.\ (\cdots(\mu^\infty\ (\lambda\alpha_k.\ \alpha_i))\cdots)).$$

Since $\mu^\infty\ (\lambda\alpha_j.\ m_0) \Rightarrow^*_\beta m_0$ whenever $\alpha_j \notin \mathrm{fv}(m_0)$, we have

$$m^\infty \Rightarrow^*_\beta \mu^\infty\ (\lambda\alpha_i.\ \alpha_i) \Rightarrow_\beta Fix_\alpha((\lambda\alpha_i.\ \alpha_i)\ \alpha)$$

and hence

$$m^\infty \Rightarrow^\infty_{\beta\bot} Fix_\alpha((\lambda\alpha_i.\ \alpha_i)\ \alpha).$$

Since $Fix_\alpha((\lambda\alpha_i.\ \alpha_i)\ \alpha)$ only β-reduces to itself, it is root-active. By Lemma 24, m^∞ is root-active as well. □

References

[1] M. Abadi and M. P. Fiore. Syntactic considerations on recursive types. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 242–252. IEEE Computer Society, 1996.

[2] A. Abel. *A polymorphic lambda-calculus with sized higher-order types.* PhD thesis, Ludwig-Maximilians-Universität München, 2006.

[3] T. Altenkirch, C. McBride, and P. Morris. Generic programming with dependent types. In *Datatype-Generic Programming*, pages 209–257. Springer, 2007. Revised lectures of International Spring School SSDGP 2006.

[4] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Trans. Program. Lang. Syst.*, 15(4):575–631, 1993.

[5] F. Atanassow and J. Jeuring. Inferring Type Isomorphisms Generically. In D. Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, chapter 4, pages 32–53. Springer Berlin / Heidelberg, 2004.

[6] M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4):265–289, 2003.

[7] A. Berarducci. Infinite-calculus and non-sensible models. In A. Ursini and P. Agliano, editors, *Logic and Algebra (Pontignano, 1994)*, volume 180 of *Lecture Notes in Pure and Applied Mathematics*, pages 339–378. Marcel Dekker Inc., 1996.

[8] J.-P. Bernardy and M. Lasson. Realizability and parametricity in pure type systems. In *Foundations of Software Science and Computational Structures*, pages 108–122. Springer LNCS 6604, 2011.

[9] J.-P. Bernardy, P. Jansson, and R. Paterson. Proofs for free. *Journal of Functional Programming*, 22(2):107–152, 2012.

[10] R. Bird and L. Meertens. Nested datatypes. In J. Jeuring, editor, *Mathematics of Program Construction*, number 1422 in Lecture Notes in Computer Science, pages 52–67. Springer Berlin Heidelberg, 1998.

[11] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. In P. d. Groote and J. R. Hindley, editors, *Typed Lambda Calculi and Applications*, number 1210 in Lecture Notes in Computer Science, pages 63–81. Springer Berlin Heidelberg, 1997.

[12] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundam. Inform.*, 33(4):309–338, 1998.

[13] B. Bringert and A. Ranta. A pattern for almost compositional functions. *Journal of Functional Programming*, 18(5-6):567–598, 2008.

[14] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software*, pages 415–438. Springer, 1997.

[15] Y. Cai, P. G. Giarrusso, and K. Ostermann. 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*. ACM, 2016.

[16] F. Cardone and M. Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92(1):48–80, May 1991.

[17] D. Colazzo and G. Ghelli. Subtyping recursive types in kernel Fun. In *Proceedings of Symposium on Logic in Computer Science*, pages 137–146. IEEE, 1999.

[18] Ł. Czajka. A coinductive confluence proof for infinitary lambda-calculus. In *Rewriting and Typed Lambda Calculi*, pages 164–178. Springer, 2014.

[19] E. de Vries and A. Löh. True sums of products. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*, pages 83–94. ACM, 2014.

[20] D. Dreyer. A type system for recursive modules. In *Proceedings of International Conference on Functional Programming*, pages 289–302. ACM, 2007.

[21] J. Endrullis and A. Polonsky. Infinitary Rewriting Coinductively. In N. A. Danielsson and B. Nordström, editors, *18th International Workshop on Types for Proofs and Programs (TYPES 2011)*, volume 19 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16–27. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013.

[22] V. Gapeyev, M. Y. Levin, and B. C. Pierce. Recursive subtyping revealed. *Journal of Functional Programming*, 12(06):511–548, 2002.

[23] J. Garrigue. Programming with polymorphic variants. In *ML Workshop*, volume 13. Baltimore, 1998.

[24] N. Gauthier and F. Pottier. Numbering matters: First-order canonical forms for second-order recursive types. In *Proceedings of International Conference on Functional Programming*, pages 150–161. ACM, 2004.

[25] J. Gibbons. Design patterns as higher-order datatype-generic programs. In *Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming*, pages 1–12. ACM, 2006.

[26] J. Gibbons. Datatype-generic programming. In *Spring School on Datatype-Generic Programming*. Springer LNCS 4719, 2007.

[27] J. Gibbons and B. C. d. S. Oliveira. The essence of the Iterator pattern. *Journal of Functional Programming*, 19:377–402, 2009.

[28] N. Glew. A theory of second-order trees. In *Programming Languages and Systems*, pages 147–161. Springer, 2002.

[29] R. Hinze. Polytypic values possess polykinded types. In *Mathematics of Program Construction*, pages 2–27. Springer, 2000.

[30] S. Holdermans, J. Jeuring, A. Löh, and A. Rodriguez Yakushev. Generic views on data types. In *Mathematics of Program Construction*, pages 209–234. Springer, 2006.

[31] G. Huet. Regular Böhm trees. *Mathematical Structures in Computer Science*, 8(06): 671–680, 1998.

[32] H. Im, K. Nakata, and S. Park. Contractive signatures with recursive types, type parameters, and abstract types. In *Automata, Languages, and Programming*, pages 299–311. Springer, 2013.

[33] P. Jancar. Decidability of DPDA language equivalence via first-order grammars. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, pages 415–424. IEEE Computer Society, 2012.

[34] M. Jaskelioff and O. Rypacek. An investigation of the laws of traversals. In *Proceedings of the Fourth Workshop on Mathematically Structured Functional Programming*, volume 76, pages 40–49. Open Publishing Association, 2012.

[35] J. Kennaway, J. W. Klop, M. R. Sleep, and F.-J. de Vries. Infinitary lambda calculus. *Theoretical Computer Science*, 175(1):93–125, 1997.

[36] E. A. Kmett. The lens package. `http://hackage.haskell.org/package/lens`.

[37] D. Kozen and A. Silva. Practical coinduction. Technical Report `http://hdl.handle.net/1813/30510`, Computing and Information Science, Cornell University, November 2012.

[38] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 26–37. ACM, 2003.

[39] C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

[40] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer LNCS 523, 1991.

[41] N. Mitchell and C. Runciman. Uniform boilerplate and list processing. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 49–60. ACM, 2007.

[42] U. Norell. Dependently typed programming in agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009. Revised lectures of the 4th International School AFP 2002.

[43] N. Oury and W. Swierstra. The power of pi. In *ACM Sigplan Notices*, volume 43, pages 39–50. ACM, 2008.

[44] F. Pfenning. Lecture notes on bidirectional type checking. In Carnegie Mellon University course 15-312, "Foundations of programming languages," fall 2004. `http://www.cs.cmu.edu/~fp/courses/15312-f04/handouts/15-bidirectional.pdf`, retrieved on 10 July 2015.

[45] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[46] F. Pottier. [TYPES] System F omega with (equi-)recursive types. `http://lists.seas.upenn.edu/pipermail/types-list/2011/001525.html`, 2011. Retrieved on 2 July 2015.

[47] A. Rodriguez Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *Proceedings of International Conference on Functional Programming*, pages 233–244. ACM, 2009.

[48] G. Sénizergues. Some applications of the decidability of DPDA's equivalence. In *Machines, Computations, and Universality*, pages 114–132. Springer, 2001.

[49] Z. Shao, C. League, and S. Monnier. Implementing typed intermediate languages. In *Proceedings of International Conference on Functional Programming*, pages 313–323. ACM, 1998.

[50] M. Solomon. Type definitions with parameters. In *Proceedings of Symposium on Principles of Programming Languages*. ACM, 1978.

[51] C. Stirling. Deciding DPDA equivalence is primitive recursive. In *Automata, languages and programming*, pages 821–832. Springer, 2002.

[52] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119. ACM, 1999.

[53] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.