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

Yufei Cai Paolo G. Giarrusso Klaus Ostermann

University of Tübingen, Germany

Abstract

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_ω^μ, 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_ω^μ enables using DGP techniques incrementally, instead of planning for them up-front or doing invasive refactoring.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.2 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

Keywords Datatype-generic programming, equirecursive types, functors

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 [24, 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_ω^μ, an extension of System F_ω with equirecursive datatypes (Sec. 4).
- We define and study the *coinductive equational theory* of F_ω^μ types, based on the theory of infinitary λ-calculus. Using this theory, we prove type soundness for F_ω^μ (Sec. 5.2).
- We show that $F_\omega^{\mu *}$, that is F_ω^μ 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_ω^μ and gives a high-level overview. Sec. 3 discusses related work on DGP and equirecursive types. Sec. 4 formalizes the static semantics of F_ω^μ. Sec. 5 discusses the soundness of F_ω^μ and the decidability of typechecking in $F_\omega^{\mu *}$. Sec. 6 is about boilerplate generation. Sec. 7 lists future work. Sec. 8 concludes.

POPL'16, January 20–22, 2016, St. Petersburg, FL, USA
ACM. 978-1-4503-3549-2/16/01...
http://dx.doi.org/10.1145/2837614.2837660

This paper only contains proof sketches. Full proofs, together with other material which we will point to, are in the appendices of the accompanying technical report.

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.

$$\begin{aligned}
&\mathbf{data}\ \mathit{Term} = \mathit{Lit}\ \mathit{Int} \quad\ \ \mid \mathit{Abs}\ \mathit{String}\ \mathit{Term}\\
&\qquad\qquad\quad \mid\ \mathit{Var}\ \mathit{String} \mid \mathit{App}\ \mathit{Term}\ \mathit{Term}
\end{aligned}$$

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

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:

$$\begin{aligned}
&\mathbf{data}\ \mathit{TermF}\ t = \mathit{Lit}'\ \mathit{Int} \quad\ \ \mid \mathit{Abs}'\ \mathit{String}\ t\\
&\qquad\qquad\qquad\ \mid\ \mathit{Var}'\ \mathit{String} \mid \mathit{App}'\ t\ t\\
&\mathbf{type}\ \mathit{Term}' = \mathit{Fix}\ \mathit{TermF}\\
&\mathbf{newtype}\ \mathit{Fix}\ f = \mathit{Roll}\ \{\mathit{unroll} :: f\ (\mathit{Fix}\ f)\}
\end{aligned}$$

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 [37] 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 $\mathit{EvalCtx}$. Again, Term'' is isomorphic to both Term and Term', but not equal.

$$\begin{aligned}
&\mathbf{data}\ \mathit{EvalCtx}\ t = \mathit{Lit}''\ \mathit{Int} \quad\ \ \mid \mathit{Abs}''\ \mathit{String}\ \mathit{Term}\\
&\qquad\qquad\qquad\quad\ \mid\ \mathit{Var}''\ \mathit{String} \mid \mathit{App}''\ t\ \mathit{Term}\\
&\mathbf{type}\ \mathit{Term}'' = \mathit{Fix}\ \mathit{EvalCtx}
\end{aligned}$$

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.

$$\begin{aligned}
&\mathbf{data}\ \mathit{VarTerm}\ t = \mathit{Lit}'''\ \mathit{Int} \mid \mathit{Abs}'''\ \mathit{String}\ \mathit{Term}\\
&\qquad\qquad\qquad\qquad \mid\ \mathit{Var}'''\ t \quad \mid \mathit{App}'''\ \mathit{Term}\ \mathit{Term}\\
&\mathbf{type}\ \mathit{Term}''' = \mathit{VarTerm}\ \mathit{String}
\end{aligned}$$

Such functors are common when defining *lenses* [35] 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 [50]). 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 [22] 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, 24, 38, 39]. 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_ω^μ, the Term' datatype and the TermF functor look as follows:

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

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.

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

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

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

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

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 *weak type equality* [11, 15] is however strictly weaker [15] and insufficient for our goals. As a minimal example, weak type equality cannot prove the following equations [1, 4, 15]:

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

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

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 $\mathit{Term}' = \mathit{Term}'''$, but, crucially, $\mathit{Term}' = \mathit{Term}''$ 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.

$$
\begin{aligned}
&\text{-- Haskell} \\
&\mathit{fold} :: (\mathit{Functor}\ f) \Rightarrow (f\ a \to a) \to \mathit{Fix}\ f \to a \\
&\mathit{fold}\ \mathit{algebra} = \mathit{fix}\ (\lambda \mathit{doFold}\ v \to \\
&\quad \mathit{algebra}\ (\mathit{fmap}\ \mathit{doFold}\ (\mathit{unroll}\ v)))
\end{aligned}
$$

$$
\begin{aligned}
&\text{-- System } F^{\mu}_{\omega} \\
&\mathbf{type}\ \mathit{Functor}\ f = \forall a\ b.\ (a \to b) \to (f\ a \to f\ b) \\
&\mathit{fold}\ :\ \forall f.\ (\mathit{Functor}\ f) \to \forall a.\ (f\ a \to a) \to \mu\ f \to a \\
&\mathit{fold} = \Lambda\ f :: * \to *.\ \lambda \mathit{fmap} : \mathit{Functor}\ f. \\
&\qquad\quad \Lambda\ a :: *.\ \lambda \mathit{algebra} : f\ a \to a. \\
&\qquad\qquad \mathit{fix}\ (\lambda \mathit{doFold} : \mu\ f \to a. \\
&\qquad\qquad\qquad \lambda v : \mu\ f. \mathit{algebra}\ (\mathit{fmap}\ \mathit{doFold}\ v))
\end{aligned}
$$

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* 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 [28] (Sec. 6). In our prototype (Sec. 2.1), we found support for *traverse* sufficient to encode a variety of DGP techniques [13, 24, 38, 39].

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 [18]. 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* [29, 45] 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.

[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.

[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.

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

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.

3.2 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.

$$\mathit{fmap} :: (a \to b) \to \mathit{TermF}\ a \to \mathit{TermF}\ b$$

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

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

$$\begin{aligned}
&\mathit{fold} :: (\mathit{Term} \to \mathit{Term}) \to \mathit{Term} \to \mathit{Term}\\
&\mathit{fold}\ f\ t = f\ (\mathit{fmap}\ (\mathit{fold}\ f)\ t)
\end{aligned}$$

Monomorphization is a technique that shows up in several approaches to generic programming, including the lens library [35], *Compos* [13], and *Scrap-your-boilerplate* [36].

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, 40, 41] 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 $\mathit{traverse}$ (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.

$$\begin{array}{rlr}
\tau^c ::= & & \text{simple contractive type}\\
& \iota & \text{primitive type}\\
| & \tau^s \to \tau^s & \text{function type}\\
| & \mu x.\ \tau^c & \mu\text{-type}\\[1em]
\tau^s ::= & & \text{simple recursive type}\\
& \alpha & \text{type variable}\\
| & \tau^c &
\end{array}$$

$$\frac{}{\alpha \equiv \alpha}\ (\text{EQ-TVAR})$$

$$\frac{}{\iota \equiv \iota}\ (\text{EQ-PRIM})$$

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

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

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

Figure 1. The system of simple recursive types investigated in Amadio and Cardelli [4], Brandt and Henglein [12], Pierce [43], with type equivalence formulated coinductively, through congruence rules and rules for μ-*folding*. This formulation ensures rules are non-overlapping and thus syntax-directed.

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. [21] (also in Pierce [43, 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_ω^μ. In fact, our proof is based on the presentation in Pierce [43]. 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 [27] considered adding recursive types to System F. Recursive F types extend recursive simple types as follows:

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

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_ω^μ as well, namely in type-level lambdas. Following Czajka [17], 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 [23] 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 [16] 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 [48] considered recursive types that can have parameters of kind $*$, that is, recursive types of K_3 kinds [43, definition 30.4.1]. As discovered later, this allows expressing *nested datatypes* [10] such as perfect binary trees:

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

In F_ω^μ, *Tree* would be the fixed point of a higher-order type:

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

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 [46]. Thus equirecursive typing is decidable for nested datatypes. Unfortunately, known algorithms to decide equivalence of deterministic push-down automata [31] 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 [32, 49]).

F_ω^μ supports fixed points of arbitrary kinds, but the decidable subset $F_\omega^{\mu *}$ 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 $F_\omega^{\mu *}$ 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. [31] considered $\lambda_{\text{abs}}^{\text{rec}}$, 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 $\lambda_{\text{abs}}^{\text{rec}}$. 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 $\lambda_{\text{abs}}^{\text{rec}}$ 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 $\lambda_{\text{abs}}^{\text{rec}}$ and F_ω^μ. In $\lambda_{\text{abs}}^{\text{rec}}$, infinite proofs relating non-contractive types to every other type are forbidden by construction. In F_ω^μ, 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 [28] considered a variant of F_ω^μ, 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_ω^μ 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 sublanguage, 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_ω^μ types, and for the latter it is not known whether a sensible, decidable equivalence relation exists [19, section 3.4].

4. System F_ω^μ

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.

$$
\begin{array}{lll}
\iota ::= & & \Lambda^\mu \text{ constant} \\
& \to \;::\; * \to * \to * & \text{function arrow} \\
| & \forall_\kappa \;::\; (\kappa \to *) \to * & \text{universal quantifier} \\
| & \{\overline{l_i}\} :: \overline{* \to} * & \text{record type constructor} \\
| & \langle\overline{l_i}\rangle :: \overline{* \to} * & \text{variant type constructor} \\
| & \iota_0 & \textit{Int}, \textit{Set}, \textit{String} \text{ etc.}
\end{array}
$$

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.

$$
\begin{array}{lll}
\kappa ::= & & \text{kind} \\
& * & \text{kind of types} \\
| & \kappa \to \kappa & \text{kind of type constructors}
\end{array}
$$

$$
\begin{array}{lll}
\tau ::= & & \text{type (constructor)} \\
& \mu\,\tau & \text{recursive type} \\
& \iota & \text{type-level constant} \\
| & \alpha & \text{type-level variable} \\
| & \lambda\alpha :: \kappa.\, \tau & \text{type-level abstraction} \\
| & \tau\; \tau & \text{type-level application}
\end{array}
$$

$$
\begin{array}{lll}
\Gamma ::= & & \text{typing context} \\
& \emptyset & \text{empty context} \\
| & \Gamma, \alpha :: \kappa & \text{type variable binding} \\
| & \Gamma, x : \tau & \text{term variable binding}
\end{array}
$$

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

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,

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

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

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

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

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

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).

$$
\begin{array}{lll}
t ::= & & \text{term} \\
& c & \text{constant} \\
| & x & \text{variable} \\
| & \lambda x : \tau.\, t & \text{abstraction} \\
| & t\; t & \text{application} \\
| & \Lambda\alpha :: \kappa.\, t & \text{type abstraction} \\
| & t\; [\tau] & \text{type application} \\
| & \{\overline{l_i = t_i}\} & \text{record introduction} \\
| & t.l_i & \text{record elimination (projection)} \\
| & \langle l_j = t\rangle\; \mathbf{as}\; \tau & \text{variant introduction (injection)} \\
| & \mathbf{case}\; t\; \mathbf{of}\; t & \text{variant elimination}
\end{array}
$$

$$
\begin{array}{lll}
c ::= & & \text{constant} \\
& \mathit{fix}_\tau : (\tau \to \tau) \to \tau & \text{fixed-point combinator} \\
| & \cdots & \text{literals, arithmetic operators, etc.}
\end{array}
$$

Figure 5. Syntax of terms of F^μ_ω.

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 [43, fig. 30-1]. When there is no confusion, we will omit the kind index of $\forall_\kappa$ and just write $\forall$.

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 equirecur-

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

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

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

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

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

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

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

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

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

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

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

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

sive types, in particular their theories of type equivalence.

5.1 Type Equivalence, Informally

In this subsection, we discuss informally type equivalence in F_ω^μ.

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):

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

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

$$((\ldots \to \mathit{Int}) \to \mathit{Int}) \to \mathit{Int}.$$

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 [31], 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_ω^μ

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

First, we extend the infinite unfolding to F_ω^μ'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 [20] because it is coinductive and thus more perspicuous and convenient. We rely on the confluence proof by Czajka [17]; some proof steps in the accompanying technical report are based on the earlier treatment by Kennaway et al. [34].

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_ω^μ 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 $\perp$, 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 $\perp$; this is sufficient to obtain a satisfactory metatheory.

5.2 Type Soundness

We prove type soundness for F_ω^μ: Well-typed closed terms never get stuck during evaluation. The proof has the same architecture as the one for F_ω by Pierce [43, Chapter 30], because F_ω^μ is basically F_ω with the standard record/variant extensions and a non-standard

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

τ' ::=		infinitary lambda term
	$\bot$	bottom
	$\Vert\ \iota$	Λ^μ constant (Fig. 2)
	$\Vert\ \alpha$	variable
	$\Vert\ \lambda\alpha.\,\tau'$	abstraction
	$\Vert\ \tau'\,\tau'$	application

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 [17], we use double bars $\Vert$ to signal coinductive definitions.

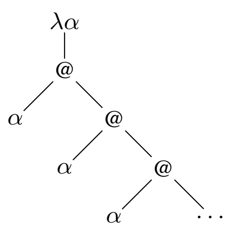

Figure 8. The infinitary lambda term $\mu^\infty = \lambda\alpha.\,\alpha\,(\alpha\,(\alpha\ \cdots))$.

type equivalence relation. Pierce proves preservation and progress for F_ω, following Wright and Felleisen [51], and the proof consists of 4 steps.

1. Lemmas 30.3.1–30.3.4 in Pierce [43]: 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.
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 the technical report.

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 $\bot$.

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:

$$(\lambda\alpha :: \kappa.\,\tau)^\infty = \lambda\alpha.\,\tau^\infty \qquad (\iota)^\infty = \iota$$
$$(\sigma\,\tau)^\infty = \sigma^\infty\,\tau^\infty \qquad (\alpha)^\infty = \alpha$$
$$(\mu\,\tau)^\infty = \mu^\infty\,\tau^\infty$$

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 [17].[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.

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

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

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

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

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

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

Figure 9. Rules for β- and Böhm-contractions according to Czajka [17]: 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.

$$\boxed{\tau' \Rightarrow^\infty_\beta \tau'}$$

$$\frac{\tau \Rightarrow^*_\beta \iota}{\tau \Rightarrow^\infty_\beta \iota} \quad (\beta\text{-}\textsc{Const})$$

$$\frac{\tau \Rightarrow^*_\beta \alpha}{\tau \Rightarrow^\infty_\beta \alpha} \quad (\beta\text{-}\textsc{Var})$$

$$\frac{\sigma \Rightarrow^*_\beta (\lambda\alpha.\,\tau) \qquad \tau \Rightarrow^\infty_\beta \tau'}{\sigma \Rightarrow^\infty_\beta (\lambda\alpha.\,\tau')} \quad (\beta\text{-}\textsc{Abs})$$

$$\frac{\sigma \Rightarrow^*_\beta \tau_1\,\tau_2 \qquad \tau_1 \Rightarrow^\infty_\beta \tau'_1 \qquad \tau_2 \Rightarrow^\infty_\beta \tau'_2}{\sigma \Rightarrow^\infty_\beta \tau'_1\,\tau'_2} \quad (\beta\text{-}\textsc{App})$$

Figure 10. Parallel multistep β-reduction $\Rightarrow^\infty_\beta$ according to Czajka [17], defined coinductively. The relation $\Rightarrow^*_\beta$ is the reflexive transitive closure of β-contraction $\Rightarrow_\beta$ (Fig. 9).

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

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

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

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

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

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

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

Figure 11. Böhm-reduction $\Rightarrow^{\infty}_{\beta\perp}$ according to Czajka [17], defined coinductively. The relation $\Rightarrow^{*}_{\beta\perp}$ is the reflexive transitive closure of Böhm-contraction $\Rightarrow_{\beta\perp}$ (Fig. 9).

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

- 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 $\Rightarrow^{\infty}_{\beta}$ defined coinductively in Fig. 10.

We call $\Rightarrow^{\infty}_{\beta}$ 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 $\perp$ and those that can always reduce to β-redexes by parallel multistep β-reduction. This intuition is obtained by simplifying Definition 2 of Czajka [17].

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

Definition 5 (Böhm-contraction, reduction [17] and equivalence)**.**

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

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 [17])**.** *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 [43]. 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 the technical report.

The preservation and progress theorem of F^{μ}_{ω} are analogous to Theorems 30.3.14 and 30.3.16 of Pierce [43], 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, $F^{\mu*}_{\omega}$ is the subset of F^{μ}_{ω} obtained by restricting the kind of recursive types to $*$. Formally, the kinding rule K-FIX is restricted on μ as follows:

$$\frac{\Gamma \vdash \tau : * \to *}{\Gamma \vdash \mu\,\tau : *} \quad \text{(K-FIX*)}$$

In this section, we show that typechecking $F^{\mu*}_{\omega}$ is decidable. The architecture of a type checker for $F^{\mu*}_{\omega}$ is quite similar to the one for F_ω [43]. 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 the technical report. 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 $F^{\mu*}_{\omega}$ to the relative simplicity of its types: Their infinite normal forms are *regular trees* [30], that is, each has only a finite number of distinct subtrees [43, Def. 21.7.2]. This is provable by applying section 21.9 of Pierce [43] to $\mathrm{NF}^{\mu*}$ (see accompanying technical report).

5.3.1 Deciding Type Equivalence

We defined type equivalence as Böhm-equivalence $\equiv_{\beta\perp}$ 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.

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

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 [43] 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 the technical report.

[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.

$m\ ::=$		$\mathrm{NF}^{\mu*}$ term
	n	finite neutral term
	$\mid\ \lambda\alpha :: \kappa.\ m$	annotated abstraction
$n\ ::=$		finite neutral term
	ι	Λ^μ constant (Fig. 2)
	$\mid\ \alpha$	variable
	$\mid\ n\ m$	application
	$\mid\ \mu\ n$	fixed-point of neutral term
	$\mid\ \mu\ (\lambda\alpha :: *.\ n)$	fixed-point of abstraction
$m'\ ::=$		NF^∞-term (Berarducci-tree)
	n'	infinite neutral term
	$\Vert\ \lambda\alpha.\ m'$	unannotated abstraction
$n'\ ::=$		infinite neutral term
	$\perp$	bottom
	$\Vert\ \iota$	Λ^μ constant (Fig. 2)
	$\Vert\ \alpha$	variable
	$\Vert\ n'\ m'$	application

Figure 12. Inductively-defined syntax of $\mathrm{NF}^{\mu*}$-terms m, and coinductively-defined syntax of NF^∞-terms m'. As in Czajka [17], double vertical bars signal coinductive definitions.

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 $m_1 \equiv_\mu m_2$.*

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

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

$$\begin{tikzcd}
\sigma \in \Lambda^{\mu*} \arrow[r, "\Rightarrow_\beta^*"] \arrow[d, "(\cdot)^\infty"'] & m \in \mathrm{NF}^{\mu*} \arrow[d, "\mathrm{Ex}(\cdot)"] \\
\sigma^\infty \in \Lambda^\infty \arrow[r, "\Rightarrow_{\beta\perp}^\infty"'] & \mathrm{Ex}(m) \in \mathrm{NF}^\infty
\end{tikzcd}$$

Lemma 16. *Let m_1, m_2 be $\mathrm{NF}^{\mu*}$ terms. Then $m_1 \equiv_\mu m_2$ 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 $\sigma_1^\infty \equiv_{\beta\perp}^\infty \sigma_2^\infty$ iff $\mathrm{Ex}(m_1) = \mathrm{Ex}(m_2)$ iff $m_1 \equiv_\mu m_2$.

Pottier [44] 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

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

$$\mathrm{Ex}(\iota) = \iota \qquad \mathrm{Ex}(n\, m) = \mathrm{Ex}(n)\,\mathrm{Ex}(m) \qquad \mathrm{Ex}(\lambda\alpha :: \kappa.\, m) = \lambda\alpha.\,\mathrm{Ex}(m)$$

$$\mathrm{Ex}(\alpha) = \alpha \qquad \mathrm{Ex}(\mu\, n) = \mathrm{Ex}(n)\,\mathrm{Ex}(\mu\, n)$$

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

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

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

$$\frac{}{\alpha \equiv_\mu \alpha} \quad \text{(EQ-TVAR)}$$

$$\frac{}{\iota \equiv_\mu \iota} \quad \text{(EQ-PRIM)}$$

$$\frac{n_1 \equiv_\mu n_2 \qquad m_1 \equiv_\mu m_2}{n_1\, m_1 \equiv_\mu n_2\, m_2} \quad \text{(EQ-APPCONG)}$$

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

$$\frac{n_1\,(\mu\, n_1) \equiv_\mu m_2}{\mu\, n_1 \equiv_\mu m_2} \quad \text{(EQ-}\mu_L\text{-NEUTRAL)}$$

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

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

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

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

Figure 13. Coinductive rules of μ-equivalence.

handle fixed points and unreduced applications of neutral terms. $F_\omega^{\mu*}$ 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_ω^μ, 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 $k, \iota, \sigma_1, \cdots, \sigma_k$.

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 [43].

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 $m_2 \equiv m_1$, $\Gamma \vdash m_2 :: \kappa$, 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 or a variable. 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 $n = \alpha$, then τ is not equivalent to any type of the form $\iota\,\sigma_1 \cdots \sigma_n$.

6. From Type Functions to Traversable Functors

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

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

satisfying certain laws [33, 37]. Traversable functors are a powerful abstraction for datatype operations [26]. The formalism of datatypes in F_ω^μ makes it possible to express generic programming combinators such as `compos` [13], `uniplate` [39], and `gmapT/gmapQ/gmapM` [36] as instances of *traverse*; details are left as an exercise for the reader.

Despite its power, *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 [28]. 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 the accompanying technical report.

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

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

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

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

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

Figure 15. Polytypic definition [28] 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_ω^μ 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 [47].
- Comparing normalized F_ω^μ-types: the algorithm we consider takes quadratic instead of linear time. There's work improving this time bound to $O(n \log n)$ [23]; 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_ω^μ, 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_ω^μ 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_ω^μ. 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.

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] F. Cardone and M. Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92(1):48–80, May 1991.

[16] 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.

[17] Ł. Czajka. A coinductive confluence proof for infinitary lambda-calculus. In *Rewriting and Typed Lambda Calculi*, pages 164–178. Springer, 2014.

[18] 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.

[19] D. Dreyer. A type system for recursive modules. In *Proceedings of International Conference on Functional Programming*, pages 289–302. ACM, 2007.

[20] 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.

[21] V. Gapeyev, M. Y. Levin, and B. C. Pierce. Recursive subtyping revealed. *Journal of Functional Programming*, 12(06):511–548, 2002.

[22] J. Garrigue. Programming with polymorphic variants. In *ML Workshop*, volume 13. Baltimore, 1998.

[23] 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.

[24] 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.

[25] J. Gibbons. Datatype-generic programming. In *Spring School on Datatype-Generic Programming*. Springer LNCS 4719, 2007.

[26] J. Gibbons and B. C. d. S. Oliveira. The essence of the Iterator pattern. *Journal of Functional Programming*, 19:377–402, 2009.

[27] N. Glew. A theory of second-order trees. In *Programming Languages and Systems*, pages 147–161. Springer, 2002.

[28] R. Hinze. Polytypic values possess polykinded types. In *Mathematics of Program Construction*, pages 2–27. Springer, 2000.

[29] 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.

[30] G. Huet. Regular Böhm trees. *Mathematical Structures in Computer Science*, 8(06):671–680, 1998.

[31] 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.

[32] 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.

[33] 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.

[34] J. Kennaway, J. W. Klop, M. R. Sleep, and F.-J. de Vries. Infinitary lambda calculus. *Theoretical Computer Science*, 175(1):93–125, 1997.

[35] E. A. Kmett. The lens package. `http://hackage.haskell.org/package/lens`.

[36] 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.

[37] C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

[38] 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.

[39] N. Mitchell and C. Runciman. Uniform boilerplate and list processing. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 49–60. ACM, 2007.

[40] 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.

[41] N. Oury and W. Swierstra. The power of pi. In *ACM Sigplan Notices*, volume 43, pages 39–50. ACM, 2008.

[42] 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.

[43] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[44] 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.

[45] 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.

[46] G. Sénizergues. Some applications of the decidability of DPDA's equivalence. In *Machines, Computations, and Universality*, pages 114–132. Springer, 2001.

[47] Z. Shao, C. League, and S. Monnier. Implementing typed intermediate languages. In *Proceedings of International Conference on Functional Programming*, pages 313–323. ACM, 1998.

[48] M. Solomon. Type definitions with parameters. In *Proceedings of Symposium on Principles of Programming Languages*. ACM, 1978.

[49] C. Stirling. Deciding DPDA equivalence is primitive recursive. In *Automata, languages and programming*, pages 821–832. Springer, 2002.

[50] 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.

[51] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.