Mendler-style Recursion Schemes for Mixed-Variant Datatypes

Ki Yung Ahn[1], Tim Sheard[1], and Marcelo Fiore[2]

1 **Department of Computer Science, Portland State University Oregon, USA**
`{kya,sheard}@cs.pdx.edu`

2 **Computer Laboratory, University of Cambridge Cambridge, UK**
`Marcelo.Fiore@cl.cam.ac.uk`

Abstract

Some concepts, such as Higher-Order Abstract Syntax (HOAS), are most naturally expressed by *mixed-variant datatypes* (a.k.a. negative (recursive) datatypes). Unfortunately, mixed-variant datatypes are often outlawed in formal reasoning systems based on the Curry–Howard correspondence (e.g., Coq, Agda), because the conventional recursion schemes (or induction principles) supported in such systems cannot guarantee termination for mixed-variant datatypes.

There is an alternative style of formulating recursion schemes, known as the Mendler style, that can guarantee termination for arbitrary datatypes. Ahn and Sheard [8] formulated a Mendler-style recursion scheme (***msfit***), and provided examples involving regular (i.e., non-indexed) mixed-variant datatypes (e.g., untyped λ-calculus in HOAS). Their examples demonstrate an advantage of the Mendler style – a termination guarantee for arbitrary datatypes, including mixed-variant ones. They proved termination of the examples via an embedding into System F_ω.

Another advantage of the Mendler style is that recursion schemes naturally extend to non-regular (i.e., indexed) datatypes. In this paper, we provide another example: a type-preserving evaluator for a simply-typed HOAS defined as a type-indexed mixed-variant datatype. This example demonstrates both advantages of the Mendler style.

This example illustrates a novel discovery that the simply-typed HOAS evaluator is expressible within System F_ω. To our knowledge, this is the first example of a simply-typed HOAS evaluator (without translation through first-order syntax) that is equipped with correct-by-construction proofs (in the Curry–Howard sense) of both type-preservation and normalization. We also develop further theoretical discussions on the F_ω-embedding of ***msfit*** and introduce further studies on two new recursion schemes (***mprsi*** and ***mphit***), which are also useful for mixed-variant datatypes. We hope our work motivates future design of logical reasoning systems that support a wider range of datatypes, including mixed-variant ones.

1998 ACM Subject Classification D.3.3 [Programming Languages]: Language Constructs and Features — Data types and structures, Polymorphism, and Recursion; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs – Functional constructs, Program and recursion schemes, and Type structure; F.4.1 [Mathematical Logic and Formal Systems]: Mathematical Logic — Lambda calculus and related systems

Keywords and phrases Mendler-style recursion, higher-order abstract syntax, HOAS, mixed-variant datatypes, negative datatypes, termination, normalization

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

Conference title on which this volume is based on.
Editors: Billy Editor, Bill Editors; pp. 1–27
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Inspired by Mendler [25], Uustalu, Matthes, and others [28, 29, 5, 6, 4] have studied and generalized Mendler's formulation of primitive recursion. They coined the term *Mendler style* for this new way of formulating recursion schemes and called the previous prevalent approach *conventional style* (e.g., the Squiggol school and structural/lexicographic termination checking as used in proof assistants). Advantages of the Mendler style, in contrast to the conventional style, include:

- Admitting arbitrary recursive datatype definitions (including mixed-variant ones),
- succinct and intuitive usability of recursion schemes (code looks like general recursion),
- uniformity of recursion scheme definition across all datatypes (including indexed ones), and
- type-based termination (not relying on any external theories other than type checking).

The primary focus of this work is on the first advantage, but other advantages are discussed and demonstrated by examples throughout this paper.

Early work [28, 29, 5, 6, 4] on the Mendler style noticed the first advantage but focused on examples using positive datatypes. Recently, Ahn and Sheard [8] discovered a Mendler-style recursion scheme ***msfit*** over mixed-variant datatypes (inspired by earlier work [24, 18, 30] in the conventional setting). Using ***msfit***, they demonstrated a HOAS formatting example (§2.2) over a non-indexed HOAS. This example was adapted from earlier work [18, 30] in the conventional style. In this paper, we demonstrate that ***msfit*** is useful over indexed datatypes as well (§3).

Ahn and Sheard [8] gave a semi-formal termination proof by embedding ***msfit*** into subset of Haskell that is believed to be a subset of System F_ω. Here, we investigate its properties in a more rigorous theoretical setting (§5).

In this paper, we give an introduction to the Mendler style by reviewing Mender-style iteration (***mit***) and iteration with syntactic inverses (***msfit***) over regular (i.e., non-indexed) datatypes. Next, we demonstrate the usefulness of the Mendler-style recursion scheme ***msfit*** over indexed and mixed-variant datatypes (§3). We report our novel discovery that a type-preserving evaluator for a simply-typed HOAS can be defined using ***msfit***. which indicates that a simply-typed HOAS evaluator can be embedded in System F_ω with its correct-by-construction proof of type-preservation and strong normalization. We can show its strong normalization by embedding ***msfit*** into System F_ω (§5.2). We also show that the equational properties of ***msfit*** are faithfully transferred to its F_ω-embedding (§5.3, §5.4). Moreover, we discuss the relationship between ordinary fixpoints and the inverse-augmented fixpoints used in ***msfit*** (§7), and introduce two new recursion schemes over mixed-variant datatypes (§7).

Our contributions can be listed as follows:

1. Demonstrating the usefulness of the Mendler style over indexed and mixed-variant datatypes,
2. writing a simply-typed HOAS evaluator using ***msfit***, whose type-preservation and termination properties are guaranteed simply by type checking (§3),
3. clarifying the relation between fixpoints of ***mit*** and fixpoints of ***msfit*** (§4),
4. embedding ***msfit*** into System F_ω (§5.2),
5. proving equational properties regarding the F_ω-embedding of ***msfit*** (§5.3, §5.4),
6. formulating the Mendler-style primitive recursion with a size-index (§7.1), and
7. formulating another Mendler-style iteration with syntactic inverses *à la* PHOAS (§7.2).

2 Mendler-style recursion schemes

In this section, we introduce basic concepts of two Mendler-style recursion schemes: iteration (***mit***) and iteration with syntactic inverses (***msfit***). Further details on Mendler-style recursion schemes, including these two and more, can be found in [8, 6, 29, 4].

In Listing 1, we illustrate the two recursion schemes, ***mit*** and ***msfit***, using Haskell. We use a subset of Haskell, where we restrict the use of certain language features and some of the definitions we introduce. We will explain the details and motivation of these restrictions as we discuss Listing 1.

Each Mendler-style recursion scheme is described by a pair: a type fixpoint (e.g., μ_*, μ'_*) and its constructors (e.g., $\boldsymbol{In}_*$, $\boldsymbol{In}'_*$), and the recursion scheme itself (e.g., $\boldsymbol{mit}_*$, $\boldsymbol{msfit}_*$). A Mendler-style recursion scheme is characterized by the abstract operations it supports. The types of these abstract operations are evident in the type signature of the recursion scheme. In Listing 1, we emphasize this by factoring out the type of the first argument (φ) as a type synonym prefixed by *Phi*. Note the various synonyms for each recursion scheme – Phi_* has one abstract operation and Phi'_* has two.

Mendler-style recursion schemes take two arguments. The first is a function[1] that will be applied to concrete implementations of the abstract operators, then uses these operations to describe the computation. The second argument is a recursive value to compute over. One programs by supplying specific instances of the first argument φ.

2.1 Mendler-style iteration

Mendler-style iteration (***mit***) operates on recursive types constructed by the fixpoint μ. The fixpoint μ is indexed by a kind. We describe μ at kind $*$ and $* \to *$ in Listing 1. We enforce two restrictions on the Haskell code in the Mendler style examples:

- Recursion is allowed only in the definition of the fixpoint at type-level, and in the definition of the recursion scheme at term-level. The type constructor μ_* expects a non-recursive base structure $f :: * \to *$ to construct a recursive type ($\mu_* f :: *$). The type constructor $\mu_{*\to*}$ expects a non-recursive base structure $f :: (* \to *) \to (* \to *)$ to construct a recursive type constructor ($\mu_{*\to*} f :: * \to *$), which expects one type index ($i :: *$). We do not use recursive datatype definitions (as natively supported by Haskell) elsewhere. We do not use recursive function definitions either, except to define Mendler-style recursion schemes.
- Elimination of recursive values is only allowed via the recursion scheme. One is allowed to freely introduce recursive values using ***In***-constructors, but not allowed to freely eliminate (i.e., pattern match against ***In***) those recursive values. Note that $\boldsymbol{mit}_*$ and $\boldsymbol{mit}_{*\to*}$ are defined using pattern matching against $\boldsymbol{In}_*$ and $\boldsymbol{In}_{*\to*}$. Pattern matching against them elsewhere is prohibited.

The type synonyms Phi_* and $\mathit{Phi}_{*\to*}$ describe the types of the first arguments of $\boldsymbol{mit}_*$ and $\boldsymbol{mit}_{*\to*}$. These type synonyms indicate that Mendler-style iteration supports one abstract operation: abstract recursive call ($r \to a$). The type variable r stands for an abstract recursive value, which could be supplied to the abstract recursive call as an argument. Since r is universally quantified within Phi_* and $\mathit{Phi}_{*\to*}$, functions of type $\mathit{Phi}_* f\ a$ and $\mathit{Phi}_{*\to*} f\ a$ must be parametric over r (i.e., must not rely on examining any details of r-values). In Phi_*, $(r \to a)$ is the type for an abstract recursive call, which computes an answer of type a from

[1] By convention, we denote the function as φ. Which is why the type synonyms are prefixed by *Phi*.

Listing 1 Mendler-style iteration (***mit***) and Mendler-style iteration with syntactic inverses (***msfit***) at kind $*$ and $* \to *$ transcribed in Haskell

```
data μ*     (f::(* → *))               = In*     (f (μ*    f)   )
data μ*→*   (f::(* → *) → (* → *)) i   = In*→*   (f (μ*→* f)  i)

type a .→ b = ∀ i . a i → b i
                              -- call
type Phi*    f a = ∀ r . (r → a) → (f r → a)
type Phi*→*  f a = ∀ r . (r .→ a) → (f r .→ a)

mit*    :: Phi*    f a → μ*    f    → a
mit*→*  :: Phi*→*  f a → μ*→*  f .→ a i

mit*    φ (In*    x) = φ (mit*    φ) x
mit*→*  φ (In*→*  x) = φ (mit*→*  φ) x
-- ------------------------------------------------------------------------
data μ'*    f a   = In'*    (f (μ'*    f a)  ) | Inverse*     a
data μ'*→*  f a i = In'*→*  (f (μ'*→* f a) i) | Inverse*→*  (a i)
                              -- inverse        -- call
type Phi'*    f a = ∀ r . (a → r a) → (r a → a) → f (r a) → a
type Phi'*→*  f a = ∀ r . (a .→ r a) → (r a .→ a) → f (r a) .→ a

msfit*    :: Phi'*    f a → (∀ a . μ'*    f a  ) → a
msfit*→*  :: Phi'*→*  f a → (∀ a . μ'*→*  f a i) → a i

msfit*    φ r = msfit φ r where
  msfit φ (In'*   x)       = φ Inverse*  (msfit φ)  x
  msfit φ (Inverse*  z)    = z

msfit*→*   φ r = msfit φ r   where
  msfit   :: Phi'*→* f a → μ'*→* f a .→ a
  msfit φ (In'*→*  x)       = φ Inverse*→*  (msfit φ)  x
  msfit φ (Inverse*→*  z)   = z
```

Note. The formulation of $\mu'_{*\to*}$ and $\boldsymbol{msfit}_{*\to*}$ in the previous work by Ahn and Sheard [8] should be adjusted as shown above. Although the previous formulation is type correct, we realized that one cannot write useful examples over indexed datatypes such as the type-preserving evaluator example in this paper. It was an oversight due to the lack of testing their formulation by examples over indexed mixed-variant datatypes.

Listing 2 List length example using $\textbf{\textit{mit}}_*$

```
data L p r = N | C p r
type List p = μ* (L p)
nil         = In* N
cons x xs   = In* (C x xs)
-- length :: List p → Int
length = mit* φ where
  φ len N          = 0
  φ len (C x xs)   = 1 + len xs
```

the abstract recursive type r. This abstract recursive call is used to implement a function of type $f\ r \to a$, which computes an answer (a) from f-structures filled with abstract recursive values (r). Similarly, $(\forall\ i.r\ i \to a\ i)$ in $\mathit{Phi}_{*\to*}$ is the type for an abstract recursive call, which is an index preserving function that computes an indexed answer $(a\ i)$ from an indexed recursive value $(r\ i)$. In the Haskell definitions of $\textbf{\textit{mit}}_*$ and $\textbf{\textit{mit}}_{*\to*}$, these abstract operations are made concrete by a native recursive call. Note that the first arguments to φ in the definitions of $\textbf{\textit{mit}}_*$ and $\textbf{\textit{mit}}_{*\to*}$ are $(\textbf{\textit{mit}}_*\varphi)$ and $(\textbf{\textit{mit}}_{*\to*}\varphi)$.

Uses of Mendler-style recursion schemes are best explained by examples. Listing 2 is a well-known example of a list length function defined in terms of $\textbf{\textit{mit}}_*$. The recursive type for lists $(\mathit{List}\ \ p)$ is defined as a fixpoint of $(L\ p)$, where L is the base structure for lists. The data constructors of *List*, *nil* and *cons*, are defined in terms of $\textbf{\textit{In}}_*$ and the data constructors of L. We define *length* by applying $\textbf{\textit{mit}}_*$ to the φ function. The function φ is defined by two equations, one for the N-case and the other for the C-case. When the list is empty (N-case), the φ function simply returns 0. When the list has an element (C-case), we first compute the length of the tail (i.e., the list excluding the head, that is, the first element) by applying the abstract recursive call $(\mathit{len} :: r \to \mathit{Int})$[2] to the (abstract) tail $(\mathit{xs} :: r)$,[3] and, then, we add 1 to the length of the tail $(\mathit{len}\ \ \mathit{xs})$.

2.2 Mendler-style iteration with syntactic inverses

Mendler-style iteration with syntactic inverses ($\textbf{\textit{msfit}}$) operates on recursive types constructed by the fixpoint μ'. The fixpoint μ' is a non-standard fixpoint additionally parametrized by the answer type (a) and has two constructors $\textbf{\textit{In}}'$ and *Inverse*. $\textbf{\textit{In}}'$-constructors are analogous to $\textbf{\textit{In}}$-constructors of μ. *Inverse*-constructors hold answers to be computed by $\textbf{\textit{msfit}}$. For example,the result of computing $\textbf{\textit{msfit}}\,\varphi\ (\mathit{Inverse}_*\,5)$ is 5 regardless of φ. The stylistic restrictions on the Haskell code involving $\textbf{\textit{msfit}}$ are:

- Recursion is only allowed by the fixpoint at type-level (μ') and by the recursion scheme $(\textbf{\textit{msfit}})$ at term-level. We do not rely on recursive datatype definitions and function definitions defined by the general recursion natively supported in Haskell.
- Elimination of recursive values is allowed via the recursion scheme. One is allowed to freely construct recursive values using $\textbf{\textit{In}}'$-constructors, but not allowed to freely eliminate (i.e., pattern match against $\textbf{\textit{In}}'$) them. Pattern matching against *Inverse* is also forbidden.

These restrictions are similar to the stylistic restrictions involving $\textbf{\textit{mit}}$.

The abstract operations supported by $\textbf{\textit{msfit}}$ are evident in the first argument type – Phi'_* and $\mathit{Phi}'_{*\to*}$ are the type synonyms for the first argument types of $\textbf{\textit{msfit}}_*$ and $\textbf{\textit{msfit}}_{*\to*}$.

[2] Here, the answer type is Int.
[3] Note that $C\ x\ \mathit{xs} :: L\ p\ r$ since $\mathit{xs} :: r$.

Listing 3 Formatting an untyped HOAS expression into a *String* (adopted from [8])

```
data ExpF r = Lam (r → r) | App r r
type Exp' a = μ'_* ExpF a    -- Inverse_*-free expressions enforced by parametricty
type Exp = ∀ a . Exp' a      -- pre-expressions that may contain Inverse_*
-- lam :: (Exp' a → Exp' a) → Exp' a
lam e    = In'_* (Lam e)
-- app :: Exp' a → Exp' a → Exp' a
app f e  = In'_* (App f e)

showExp :: Exp → String
showExp e = msfit_* φ e vars where
  -- φ :: Phi'_* ExpF ([String] → String)
  φ inv show' (App x y) = λvs      → "(" ++ show' x vs ++ " "
                                       ++ show' y vs ++ ")"
  φ inv show' (Lam z)   = λ(v : vs) → "(\\" ++ v ++ "." ++
                                   show' (z (inv (const v))) vs ++ ")"
```

Note that the abstract recursive type r is also additionally parametrized by the answer type a in the type signatures of $\boldsymbol{msfit}_*$ and $\boldsymbol{msfit}_{*\to*}$, since μ' is additionally parametrized by a. In addition to the abstract recursive call, $\boldsymbol{msfit}$ also supports the abstract inverse operation. Note that the types for abstract inverse ($(a \to r\ a)$ and $(a\ i \to r\ a\ i)$) are indeed the types for inverse functions of abstract recursive call ($(r\ a \to a)$ and $(r\ a\ i \to a\ i)$). Instead of using actual inverse functions to compute inverse images from answer values during computation, one can hold intermediate answer values, whose inverse images are irrelevant, inside *Inverse*-constructors during the computation using $\boldsymbol{msfit}$.

The type signature of $\boldsymbol{msfit}$ expects the second argument to be parametric over the answer type. Note the second argument types $(\forall\ a.\ \mu'_* f\ a)$ and $(\forall\ a.\ \mu'_{*\to*} f\ a\ i)$ in the type signatures of $\boldsymbol{msfit}_*$ and $\boldsymbol{msfit}_{*\to*}$. Using *Inverse* to construct recursive values elsewhere is, in a way, prohibited due to the second argument type of $\boldsymbol{msfit}$. Using *Inverse* to construct concrete recursive values makes the answer type specific. For example, $(\mathit{Inverse}_*\, 5) :: \mu'_* f\ \mathit{Int}$, whose answer type made specific to *Int*, cannot be passed to $\boldsymbol{msfit}_*$ its second argument. The constructor *Inverse* is only intended to define $\boldsymbol{msfit}$ and its first argument (φ). One can indirectly access *Inverse* via the abstract inverse operation supported by $\boldsymbol{msfit}$. Note, in the Haskell definitions of $\boldsymbol{msfit}_*$ and $\boldsymbol{msfit}_{*\to*}$, the second arguments to φ are $\mathit{Inverse}_*$ and $\mathit{Inverse}_{*\to*}$. That is, the abstract inverse operation is implemented by the *Inverse*-constructor.

The HOAS formatting is a "hello world" example repeatedly formulated in studies on recursion schemes over HOAS; e.g., [18, 30, 12] to mention a few in the conventional style. This example is interesting because it is a simplification of a recurring pattern (or functional pearl [10]) of conversion from higher-order syntax to first-order syntax, which is often found in implementations of embedded domain specific languages. Listing 3 illustrates a Mendler-style formulation (*showExp*) of this example using $\boldsymbol{msfit}$.

The key characteristic of *showExp* is apparent in the user-defined combining function φ. From the type of φ, we know that the result of iteration over a HOAS term e is a function; more specifically, $\boldsymbol{msfit}_*\ \varphi\ e :: [\mathit{String}] \to \mathit{String}$. An infinite list of fresh variable names (vars)[4] is supplied as an argument to $\boldsymbol{msfit}_*\ \varphi\ e$ to obtain a string that represents e.

[4] To be strictly complacent to the conventions of the Mendler style, we would have to formulate a co-recursive datatype that generates infinite list of variable names. Here, we simply use Haskell's lazy

Definition of φ consists of two equations. The first equation for *App* is a typical structural recursion over positive occurrences of recursive subcomponents. The second equation for *Lam* exploits the abstract inverse ($inv :: ([String] \to String) \to r\ ([String] \to String)$) provided by $\textbf{\textit{msfit}}$ to handle the negative recursive occurrence. When formatting a *Lam*-expression, one should supply a fresh variable to represent the bounded variable (which is the negative recursive occurrence) introduced by *Lam*. Here, we consume one fresh name from the supplied list of fresh names by pattern matching $(v{:}\,vs)$, and take an inverse of a constant function that will return the name $(inv(const\ \ v))$, which has an appropriate type to pass into the function z contained in constructor *Lam*. Since the result of this application $z(inv(const\ \ v))$ corresponds to a positive recursive occurrence, we simply apply the abstract recursive call $show'$.

3 Type-preserving evaluation of the simply-typed HOAS

Listing 4 Simply-typed HOAS evaluation using $\textbf{\textit{msfit}}_{*\to*}$

```
data ExpF r t where  Lam :: (r t₁ → r t₂) → ExpF r (t₁ → t₂)
                     App :: r (t₁ → t₂) → r t₁ → ExpF r t₂
type Exp' a t = μ'_{*→*} ExpF a t
type Exp t = ∀ a . Exp' a t
-- lam :: (Exp' a t₁ → Exp' a t₂) → Exp' a (t₁ → t₂)
lam e     = In'_{*→*} (Lam e)
-- app :: Exp' a (t₁ → t₂) → Exp' a t₁ → Exp' a t₂
app f e   = In'_{*→*} (App f e)

data K t = η {η⁻¹ :: t}
-- eval :: Exp t → K t
eval = msfit_{*→*} φ where
  φ :: Phi'_{*→*} ExpF K
  φ inv ev (Lam f)   = η (λv → η⁻¹(ev (f (inv (η v)))))
  φ inv ev (App f x) = η (η⁻¹(ev f) (η⁻¹(ev x)))
```

We can write an evaluator for a simply-typed HOAS in a simple manner using $\textbf{\textit{msfit}}_{*\to*}$, as illustrated in Listing 4. We first define the simply-typed HOAS as a recursive indexed datatype $Exp\ :: * \to *$. We take the fixpoint using $\mu'_{*\to*}$ (the fixpoint with a syntactic inverse). This fixpoint is taken over a non recursive base structure $ExpF :: (* \to *) \to (* \to *)$. Note that expressions ($Exp\ t$) by their types (t). Recursive types defined using $\mu'_{*\to*}$, such as Exp' is also parametrized by the type of the answer (a). The use of the $\textbf{\textit{msfit}}_{*\to*}$ requires that *Exp* should be parametric in this answer type by defining $Exp\ t$ as $\forall\ a.\ Exp'\ a$.

The definition of *eval* specifies how to evaluate an HOAS expression to a host-language value (i.e., Haskell) wrapped by the identity type (K). In the description below, we ignore the wrapping (η) and unwrapping (η^{-1}) of K. See the Listing 4 (where they are not omitted) if you care about these details. We discuss the evaluation for each of the constructors of *Exp*:

- Evaluating an HOAS abstraction ($Lam\ f$) lifts an object-language function (f) over *Exp* into a host-language function over values: $(\lambda v \to ev\ \ (f(inv\ \ v)))$. In the body of this host-language lambda abstraction, the inverse of the (host-language) argument value v is passed to the object-language function f. The resulting HOAS expression $(f(inv\ \ v))$ is evaluated by the recursive caller (ev) to obtain a host-language value.

lists because our focus here is not co-recursion but introducing an example using $\textbf{\textit{msfit}}$.

- Evaluating an HOAS application $(App\ f\ x)$ lifts the function f and argument x to host-language values $(ev\ f)$ and $(ev\ x)$, and uses host-language application to compute the resulting value. Note that the host-language application $((ev\ f)\ (ev\ x))$ is type correct since $ev\ f :: a \to b$ and $ev\ x :: a$, thus the resulting value has type b.

We know that *eval* indeed terminates since $\mu'_{* \to *}$ and $\boldsymbol{msfit}_{* \to *}$ can be embedded into System F_ω in a manner similar to the embedding of μ'_* and $\boldsymbol{msfit}_*$ into System F_ω.

Listing 4 highlights two advantages of the Mendler style over the conventional style in one example. This example shows that the Mendler-style iteration with syntactic inverses is useful for both *negative* and *indexed* datatypes. *Exp* in Listing 4 has both negative recursive occurrences and type indices.

The *showExp* example in Listing 3, which we discussed in the previous section, has appeared in the work of Fegaras and Sheard [18], written in the conventional style. So, the *showExp* example, only shows that the Mendler style is as expressive as the conventional style (although it is perhaps syntactically more pleasant than the conventional style). Although it is possible to formulate such a recursion scheme over indexed datatypes in the conventional style (e.g., the simply-typed HOAS evaluation example of Bahr and Hvited [12]), it is not quite elegant as in the Mendler style because the conventional style is based on ad-hoc polymorphism, using type classes in Haskell. In contrast, $\boldsymbol{msfit}$ is uniformly defined over indexed datatypes of arbitrary kinds. Both $\boldsymbol{msfit}_{* \to *}$, used in the *eval*, and $\boldsymbol{msfit}_*$, used in the *showExp*, have exactly the same syntactic definition, differing only in their type signatures, as illustrated in Listing 1.

4 μ'-fixpoint is a subtype of μ-fixpoint

We discussed the usefulness of $\boldsymbol{msfit}$ by the illustrating examples on HOAS. If one is to design a language based on Mendler-style recursion schemes, one would want to support as many useful recursion schemes available, including $\boldsymbol{mit}$ and $\boldsymbol{msfit}$. One issue in such design is that we have two different fixpoints μ and μ'. The standard fixpoint μ does not come with syntactic inverses while μ' comes with its syntactic inverse. It would be a bad design choice to provide two unrelated fixpoints and let users deal with them manually. We would like to apply as many recursion schemes to one recursive value without manual conversion.

We discovered a coercion from μ'-values to μ-values, as illustrated in Listing 5. In Listing 5, we define a mapping from *Exp* (i.e., $\forall\ a.\mu'_* ExpF\ a$) to *Expr* (i.e., $\mu_* ExpF$) using $\boldsymbol{msfit}_*$, where *ExpF* is a base structure for the untyped HOAS. Since we have two fixpoints, μ'_* and μ_*, we can define two recursive datatypes from the base structure *ExpF*. One is *Exp* defined as $(\forall\ a.\mu'_* ExpF\ a)$ and the other is *Expr* defined as $\mu_* ExpF$. The function $exp2expr :: Exp \to Expr$ implements the mapping from μ'_*-based HOAS expressions to μ_*-based HOAS expressions. Note, *exp2expr* is defined using $\boldsymbol{msfit}_*$. Since there exists an embedding of μ_* and $\boldsymbol{msfit}_*$ into System F_ω [8], *exp2expr* is admissible in System F_ω. However, it is unlikely that we can embed a coercion function for an arbitrary base structure f, $mu2rec :: (\forall\ a.\mu'_* f\ a) \to \mu_* f$, in System F_ω[5].

The converse coercion from μ-values to μ'-values is not likely to exist in general, but the conversion might be possible when the answer type of the μ'-values (e.g., a in $\mu'_* ExpF\ a$) has been specialized to the final answer value. For instance, we attempted to convert from

[5] The discussions in §5 on the embedding of $\boldsymbol{msfit}$ suggests why the *mu2rec* is unlikely to be embedded in System F_ω, but its specific instances, such as *exp2expr*, can be embedded in System F_ω.

Listing 5 Coercion from μ'-values to μ-values using $\boldsymbol{msfit}_*$

```
data ExpF r = Lam (r → r) | App r r
type Expr = μ∗ ExpF
type Exp′ a = μ′∗ ExpF a
type Exp = (∀ a. Exp′ a) −− (∀ a. μ′∗ ExpF a)

exp2expr :: Exp → Expr    −− (∀ a. μ′∗ ExpF a) → μ∗ ExpF
exp2expr = msfit∗ φ where
  φ inv p2r (Lam f)     = In∗(Lam(λx → p2r (f (inv x))))
  φ inv p2r (App e1 e2) = In∗(App (p2r e1) (p2r e2))
```

Listing 6 An incomplete attempt to convert from μ-values to μ'-values

```
msfit′ :: Phi′∗ f a → μ′∗ f a → a
msfit′ φ (In′∗ x)        = φ Inverse∗ (msfit′ φ) x
msfit′ φ (Inverse∗ z) = z

exp′2expr :: Exp′ Expr → Expr   −− i.e, μ′∗ ExpF (μ∗ ExpF) → μ∗ ExpF
exp′2expr = msfit′ φ where
  φ inv p2r (Lam f)      = In∗(Lam((λx → p2r (f (inv x)))))
  φ inv p2r (App e1 e2) = In∗(App (p2r e1) (p2r e2))

expr2exp′ :: Expr → Exp′ Expr   −− i.e., μ∗ ExpF → μ′∗ ExpF (μ∗ ExpF)
expr2exp′ (In∗(Lam f))      = In′∗ (Lam (λx→ expr2exp′ (f (exp′2expr x))))
expr2exp′ (In∗(App e1 e2)) = In′∗ (App (expr2exp′ e1) (expr2exp′ e2))
```

$Exp'\ Expr$ to $Expr$, rather than from Exp (i.e., $\forall\ a.Exp'\ a$) to $Expr$.[6] We illustrate this idea in Listing 6, which is still an incomplete attempt since there is no termination guarantee for $expr2exp'$. Note that $expr2exp'$ is not defined using a Mendler-style recursion scheme but using general recursion.

The coercion from $(\forall\ a.\mu'_* ExpF\ a)$ to $(\mu_* ExpF)$ exists. We conjecture that it should be possible to derive a coercion function from μ'-values to μ-values when given a specific instance of the base structure. Therefore, when designing a language based on Mendler-style recursion schemes, we may support coercion from μ'-values to μ-values.

We believe that $\boldsymbol{msfit}_*$ can express more functions than $\boldsymbol{mit}_*$ (e.g., *showExp* in Listing 3). Then, it may be the case that the set of values of $(\forall\ a.\mu'_* f\ a)$ is in fact more restrictive than the set of values of $(\mu_* f)$. The additional expressiveness of $\boldsymbol{msfit}_*$ may be a compensation for the restrictions on the value of $(\forall\ a.\mu'_* f\ a)$. In summary, $(\forall\ a.\mu'_* f\ a)$ is a subset of $(\mu_* f)$. We believe that this generalizes to arbitrary kinds other than $*$.

5 Embedding *msfit* into System F_ω

We first review the embedding of Mendler-style iteration ($\boldsymbol{mit}_*$), before discussing the embedding of Mendler-style iteration with syntactic inverses ($\boldsymbol{msfit}_*$). The embedding of Mendler-style iteration consists of a polymorphic encoding of the fixpoint operator (μ_*) and term encodings (as functions) of its constructor ($\boldsymbol{In}_*$) and eliminator ($\boldsymbol{mit}_*$). We also

[6] Also note that a in $(\mu'_*\ ExpF\ a)$ in the type signature of $\boldsymbol{msfit}'$ is not quantified, c.f. $((\forall\ a.\mu'_* f\ a)$ in the type signature of $\boldsymbol{msfit}_*$.

show that one can derive the equational properties of $\boldsymbol{mit}_*$, which correspond to its Haskell definition discussed earlier.

Next, we discuss the embedding of $\boldsymbol{msfit}_*$ into System F_ω. The embedding of Mendler-style iteration with syntactic inverses should consist of a polymorphic encoding of the inverse-augmented fixpoint operator (μ'_*) and term encodings of its two constructors ($\mathit{Inverse}_*$ and $\boldsymbol{In}'_*$) and the eliminator ($\boldsymbol{msfit}_*$). The embedding is not as simple as the embedding of μ_* and $\boldsymbol{mit}_*$ because we have not found an F_ω-term that embeds $\boldsymbol{In}'_*$. However, we can embed each recursive type (e.g., Exp'), when given a concrete base structure (e.g., ExpF), and deduce general rules of how to embed inverse-augmented recursive types. We also show that we can derive the expected equational properties for a specific example (assuming that the section-retraction pair of the identity type is equivalent to an identity function); the example we use is the untyped HOAS (Exp') discussed in earlier sections.

Our discussion in this section is focused at kind $*$, but the embeddings for Mendler-style recursion schemes at higher-kinds (e.g., $\boldsymbol{mit}_{*\to*}$ and $\boldsymbol{msfit}_{*\to*}$) would be similar to the embeddings of them at kind $*$. In fact, the term definitions for data constructors and eliminators (i.e., recursion schemes) are always exactly the same regardless of their kinds. Only their types become richer as we move to higher kinds, having more indices applied to type constructors.

5.1 The embedding of $\boldsymbol{mit}_*$ and its equational property

Mendler-style iteration ($\boldsymbol{mit}_*$) can be embedded into System F_ω as follows [6, 8]:

$$
\begin{aligned}
&\mu_* \;=\; \lambda F^{*\to*}.\forall X^*.(\forall R^*.(R \to X) \to F R \to X) \to X\\
&\boldsymbol{mit}_* \;:\; \forall A^*.(\forall R^{*\to*}.(R \to A) \to F R \to A) \to \mu_* F \to A\\
&\boldsymbol{mit}_*\ \varphi\ r \;=\; r\ \varphi\\
&\boldsymbol{In}_* \;:\; \forall F^{*\to*}.F(\mu_* F) \to \mu_* F\\
&\boldsymbol{In}_*\ x\ \varphi \;=\; \varphi\ (\boldsymbol{mit}_*\ \varphi)\ x
\end{aligned}
$$

From the above embedding, one can derive the equational property of $\boldsymbol{mit}_*$ apparent in the Haskell definition (Listing 1) as follows: $\boldsymbol{mit}_*\ \varphi\ (\boldsymbol{In}_*\ x) = \boldsymbol{In}_*\ x\ \varphi = \varphi\ (\boldsymbol{mit}_*\ \varphi)\ x$.

5.2 Embedding $\boldsymbol{msfit}_*$

The aim is to embed Mendler-style iteration with static inverses ($\boldsymbol{msfit}_*$) into System F_ω along the following lines.[7] The embeddings for μ'_* and $\boldsymbol{msfit}_*$ can given as follows:

$$
\begin{aligned}
&\mu'_* \;=\; \lambda F^{*\to*}.\lambda A^*.KA + ((KA \to A) \to F(KA) \to A) \to A\\
&\boldsymbol{msfit}_* \;:\; \forall A^*.(\forall R^{*\to*}.(A \to RA) \to (RA \to A) \to F(RA) \to A) \to (\forall A^*.\mu'_* FA) \to A\\
&\boldsymbol{msfit}_*\ \varphi\ r \;=\; r\ \eta^{-1}\ \underbrace{(\lambda f.f(\varphi\ \eta))}_{g}
\end{aligned}
$$

where $K = \lambda A^*.A$ is an identity type constructor, therefore, both $\eta : A \to KA$ and $\eta^{-1} : KA \to A$ are identity functions. We could have just erased K in the embedding of μ'_* above, but having K makes it syntactically more evident of the correspondence between this F_ω-embedding and the Haskell transcription in Listing 4.[8] It is also easier to notice that KA matches with RA through polymorphic instantiation while type checking the definition of

[7] A Haskell transcription of this embedding appears in the previous work of Ahn and Sheard [8].

[8] The purpose of identity datatype K in Listing 4 is to avoid higher-order unification during type inference so that GHC can type check.

$\boldsymbol{msfit}_*$. In the embedding of $\boldsymbol{msfit}_*$, note that $r : \mu'_* \, F \, A$ and that μ'_* is defined using a sum type $(+)$, whose polymorphic embedding is $A + B = \forall X^*.(A \to X) \to (B \to X) \to X$ and its two constructors $in_L : \forall A^*.\forall B^*.A \to A + B$ (left injection) and $in_R : \forall A^*.\forall B^*.B \to A + B$ (right injection) are defined as $in_L = \lambda a.\lambda f_1.\lambda f_2.f_1 \; a$ and $in_R = \lambda b.\lambda f_1.\lambda f_2.f_2 \; b$. The value r selects $\eta^{-1} : KA \to A$ to handle $Inverse_*$-values and selects g to handle $\boldsymbol{In}'_*$-values.

Next, we need to embed the two data constructors of μ_*, $Inverse_*$ and $\boldsymbol{In}'_*$.

We were able to define a universal embedding of $Inverse_*$ that works for arbitrary F:

$$Inverse_* \;:\; \forall F^{*\to*}.\forall A^*.A \to \mu'_* F A$$
$$Inverse_* \; a \;=\; in_L(\eta \; a)$$

From the embedding of $Inverse_*$, we can derive the equational property of $\boldsymbol{msfit}_*$ over $Inverse_*$-values, which is apparent in the Haskell definition of $\boldsymbol{msfit}_*$ in Listing 1, as below:

$$\boldsymbol{msfit}_* \; \varphi \; (Inverse_* a) = (Inverse_* \; a) \; \eta^{-1} \; g = in_L \; (\eta \; a) \; \eta^{-1} \; g = \eta^{-1}(\eta \; a) = a$$

However, we have not been able to define a universal embedding of $\boldsymbol{In}'_*$ in System F_ω. What we know is that the embedding of $\boldsymbol{In}'_*$ must be in the form of a right injection (in_R):[9]

$$\boldsymbol{In}'_* \;:\; \forall F^{*\to*}.\forall A^*.F(\mu'_* F A) \to \mu'_* F A$$
$$\boldsymbol{In}'_* \; x \;=\; in_R(\cdots \text{ missing complete definition } \cdots)$$

We believe that we can find an embedding of $\boldsymbol{In}'_*$ for each F when the definition of F is given concretely (see Appendix A). That is, we can embed constructor functions of a recursive type $\mu{*}'F$ for each specific F.[10] For instance, we can embed the constructor functions of Exp' in Listing 3 and its two data constructors lam and app into System F_ω, as below:[11]

$$lam \;:\; \forall A^*.(Exp' \; A \to Exp' \; A) \to Exp' \; A$$
$$lam \; f \;=\; \boldsymbol{In}'_*(Lam \; f) = in_R \; \underbrace{(\lambda\varphi'.\varphi' \; \eta^{-1} \; \overbrace{(Lam(\lambda y.lift \; \varphi' \; (f(in_L \; y))))}^{v})}_{w}$$
$$app \;:\; \forall A^*.Exp' \; A \to Exp' \; A \to Exp' \; A$$
$$app \; r_1 \; r_2 = \boldsymbol{In}'_*(App \; r_1 \; r_2) = in_R \; \underbrace{(\lambda\varphi'.\varphi' \; \eta^{-1} \; (App \; (lift \; \varphi' \; r_1) \; (lift \; \varphi' \; r_2)))}_{h}$$

where $lift$ is defined as follows:

$$lift \;:\; (\forall A^*.(KA \to A) \to F(KA) \to A) \to \mu'_* F A \to KA$$
$$lift \; \varphi' \; r \;=\; r \; id \; (\lambda z.\eta(z \; \varphi'))$$

Recall that μ'_* is a sum type. The $lift$ function converts $(\mu'_* F A)$-values to (KA)-values when given a function $\varphi' : \forall A^*.(KA \to A) \to F(KA) \to A$. Observe that the type of φ' matches with the partial application of φ, the first argument of $msfit$, applied to η. Since $\varphi : \forall R^*.(A \to RA) \to (RA \to A) \to F(RA) \to A$ and $\eta : A \to KA$, we first instantiate R with K in the type of φ, that is, $(A \to KA) \to (KA \to A) \to F(KA) \to A$. Then, $(\varphi\eta) : (KA \to A) \to F(KA) \to A$, which matches the type of φ', the first argument of $lift$.

We use $lift$ for the recursive values that are covariant, in order to convert from $F(\mu'_* F A)$-structures, or $F(RA)$-structures, to $F(KA)$-structures – recall the type of the φ'. We lift recursive values r_1 and r_2, which are both covariant, in the embedding of app. We also lift

---

[9] It was also the case in the previous work of Ahn and Sheard [8], but was not clearly stated in the text.
[10] Similarly, all regular recursive types can be embedded into System F, but not μ_* itself.
[11] The use of $\boldsymbol{In}'_*$ here is only a conceptual illustration because we have embedded $\boldsymbol{In}'_*$ itself into System F_ω. We also labeled some of the subterms (v, w, and h) for later use in the discussion.

the value resulting from f, whose return type is $F(\mu'_* FA)$, in the embedding of *lam*, since the right-hand side of the function type is covariant.

For recursive values needed in contravariant positions, we simply left inject answer values. For example, y in the embedding of *lam* has type KA since we expect the argument to *Lam* be of type $KA \to KA$ because we expect $v : F(KA)$, which is the second argument to be applied to φ'. To convert from (KA) to $\mu'_* FA$, we only need to left inject, that is, $(in_L\ y)$, which can be applied to $f : \mu'_* FA \to \mu'_* FA$.

We believe that it is possible to give an embedding for any recursive type in this way, that is, by lifting $(\mathit{lift}\ \varphi)$ the recursive values in covariant positions and by left injecting (in_L) the answer values when recursive values are needed in contravariant positions. A type-directed algorithm for deriving the embeddings of the constructor functions of $\mu'_* F$ for each given $F : * \to *$ is described in Appendix A). It would be an interesting theoretical quest to search for a calculus that can directly embed the constructor $\boldsymbol{In}'_* : \forall F^{* \to *}. \forall A^*. F(\mu'_* FA) \to \mu'_* FA$.

In the remainder of this section, we discuss the equational properties of $\boldsymbol{msfit}_*$ over $\boldsymbol{In}'_*$-values of the type *Exp*. That is, when $\boldsymbol{msfit}_*$ is applied to the values constructed either by *app* or by *lam*.

5.3 Equational properties of *msfit*$_*$ over values constructed by *lam*

When applied to $(\mathit{lam}\ f)$, we expect $\boldsymbol{msfit}_*$ to satisfy the following equation:

$$\boldsymbol{msfit}_*\ \varphi\ (\mathit{lam}\ f) \stackrel{?}{=} \varphi\ \eta\ \eta^{-1}\ (\mathit{Lam}(\lambda y.\eta(\boldsymbol{msfit}\ \varphi\ (f(in_L\ y))))) \tag{1}$$

We use η to convert answer values of type A, resulting from $(\boldsymbol{msfit}\ \varphi\ (f(in_L\ y)))$, to values of type KA, since we need $(\mathit{Lam}(\lambda y.\eta(\boldsymbol{msfit}\ \varphi\ (f(in_L\ y)))))$ to be of type $F(KA)$. The type of φ expects a value of type $F(RA)$ as its third argument, where R is a polymorphic type variable, which instantiates to K in the right-hand side of Equation (1). We use in_L to convert $y : KA$, to a value of $\mu'_* FA$ in order to apply it to $f : \mu'_* FA \to \mu'_* FA$.

The left-hand side of Equation (1) can be expanded using the definitions of $\boldsymbol{msfit}_*$, in_R, g, and w, as below:

$$\begin{aligned}
\boldsymbol{msfit}_*\ \varphi\ (\mathit{lam}\ f) &= (\mathit{lam}\ f)\ \eta^{-1}\ g \\
&= in_R\ w\ \eta^{-1}\ g\ =\ g\ w\ =\ w(\varphi\eta) \\
&= \varphi\ \eta\ \eta^{-1}\ (\mathit{Lam}(\lambda y.\mathit{lift}\ (\varphi\eta)\ (f(in_L\ y)))) \\
&= \varphi\ \eta\ \eta^{-1}\ (\mathit{Lam}(\lambda y.\psi))
\end{aligned}$$

where $\psi = (f(in_L\ y))\ \mathit{id}\ (\lambda z.\eta(z(\varphi\eta)))$.

The resulting equation is similar in structure to the right-hand side of Equation (1). Thus, justifying Equation (1) amounts to showing:

$$\psi \stackrel{?}{=} \eta(\boldsymbol{msfit}\ \varphi\ (f(in_L\ y))) \tag{2}$$

The right-hand side of Equation (2) expands as follows:

$$\eta(\boldsymbol{msfit}\ \varphi\ (f(in_L\ y)))) = \eta(in_L\ \psi\ \eta^{-1}\ g) = \eta(\eta^{-1}\ \psi) = \psi$$

In the last step of $\eta(\eta^{-1}\psi) = \psi$, we relied on the fact that η and η^{-1} are identity functions.

5.4 Equational properties of *msfit*$_*$ over values constructed by *app*

When applied to $(\mathit{app}\ r_1\ r_2)$, we expect $\boldsymbol{msfit}_*$ to recurse on each of r_1 and r_2, as follows:

$$\boldsymbol{msfit}_*\ \varphi\ (\mathit{app}\ r_1\ r_2) \stackrel{?}{=} \varphi\ \eta\ \eta^{-1}\ (\mathit{App}\ (\eta(\boldsymbol{msfit}_*\ \varphi\ r_1))\ (\eta(\boldsymbol{msfit}_*\ \varphi\ r_2))) \tag{3}$$

We need η to convert answer values of type A to values of type KA, since we need $(App\ (\eta(\boldsymbol{msfit}_*\ \varphi\ r_1))\ (\eta(\boldsymbol{msfit}_*\ \varphi\ r_2)))$ to have type $F(KA)$. The type of φ expects a value of type $F(RA)$ as its third argument, where R is a polymorphic type variable, which instantiates to K in the right-hand side of Equation (3). By using the definitions of $\boldsymbol{msfit}_*$, in_R, g, and h, the left-hand side of Equation (3) expands as follows:

$$\begin{aligned} \boldsymbol{msfit}_*\ \varphi\ (app\ x\ y) \ &=\ (app\ r_1\ r_2)\ \eta^{-1}\ g \\ &=\ in_R\ h\ \eta^{-1}\ g \ =\ g\ h\ =\ h(\varphi\ \eta) \\ &=\ \varphi\ \eta\ \eta^{-1}\ (App\ (lift\ (\varphi\eta)\ r_1)\ (lift\ (\varphi\eta)\ r_2)) \end{aligned}$$

The resulting expression is similar in structure to the right-hand side of Equation (3). Thus, justifying Equation (3) amounts to showing:

$$\eta(\boldsymbol{msfit}_*\ \varphi\ r) \stackrel{?}{=} lift\ (\varphi\eta)\ r \tag{4}$$

When $r = in_R\ z$, Equation (4) is justified as follows:

$$\begin{aligned} \eta(\boldsymbol{msfit}_*\ \varphi\ (in_R\ z)) \ &=\ \eta(in_R\ z\ \eta^{-1}\ g) \ =\ \eta(g\ z) \ =\ \eta(z(\varphi\eta)) \\ &=\ (in_R\ z)\ id\ (\lambda z.\eta(z.(\varphi\eta))) \ =\ lift\ (\varphi\eta)\ (in_R\ z) \end{aligned}$$

When $r = in_L\ z$, the right-hand side of Equation (4) expands as below:

$$lift\ \varphi\ (in_L\ z) \ =\ (in_L\ z)\ id\ (\lambda z.\eta(z.(\varphi\eta))) \ =\ id\ z = z$$

and the left-hand side of Equation (4) expands as below

$$\eta(\boldsymbol{msfit}_*\ \varphi\ r) \ =\ \eta(in_L\ z\ \eta^{-1}\ g) = \eta(\eta^{-1} z) = z$$

In the last step of $\eta(\eta^{-1} z) = z$, we relied on the fact that η and η^{-1} are identity functions.

6 Related work

Here, we discuss several related work. In §6.1, we introduce Mendler-style primitive recursion ($\boldsymbol{mpr}$) to lead up the discussion of $\boldsymbol{mprsi}$ (§7.1). In §6.2, we summarize type-based termination and sized-type approach (in relation to $\boldsymbol{mpr}$). Lastly, in §6.3, we discuss a generic programming library in Haskell that supports binders using parametric HOAS, which leads up the discussion of $\boldsymbol{mphit}$ (§7.2). We also mention recent breakthrough regarding self-evaluation of System F_ω (§6.4).

6.1 Mendler-style primitive recursion

Termination of the Mendler-style iteration ($\boldsymbol{mit}$) can be proved by embedding $\boldsymbol{mit}$ into System F_ω as discussed in §2.1. The embedding of $\boldsymbol{mit}$ in §2.1 is *reduction preserving*: the number of reduction steps using the embedding and using the equational specification should differ no more than constant time factor. A reduction preserving embedding of primitive recursion into System F_ω cannot exist because it is known that "induction is not derivable in second order dependent type theory" [20] and that "primitive recursion can be seen as the computational interpretation of induction through the Curry-Howard interpretation of propositions-as-types" [21]. Although it is possible to simulate primitive recursion in terms of iteration, it may become computationally inefficient. For example, *pred* in Listing 7 could be defined using $\boldsymbol{mit}$ but its time complexity would be at least linear to the size of the input rather than constant. A constant time *pred* is definable due to the abstract cast operation

Listing 7 Examples using Mendler-style primitive recursion ***mpr*** at kind $*$: a constant time *pred* and a *factorial* function.

```
mpr* :: (∀ r. (r → μ* f) → (r → a) → f r → a) → μ* f → a
mpr* φ (In* x) = φ id (mpr* φ) x

data N r = Z | S r
type Nat = μ* N
zero     = In* Z
succ n   = In* (S n)

pred = mpr* φ where
  φ cast pr Z       = zero
  φ cast pr (S n)   = cast n

factorial = mpr* φ where
    φ cast fac Z       = succ zero
    φ cast fac (S n)   = times (succ (cast n)) (fac n)
```

provided by ***mpr***. This operation casts abstract recursive values of type r into concrete recursive values of type $\mu_* f$; its type $(r \to \mu_* f)$ is apparent from the type signature of $\boldsymbol{mpr}_*$. This cast operation computes in constant time because it is implemented as the identity function (id) in the definition of $\boldsymbol{mpr}_*$. A representative example of ***mpr*** that actually uses recursion is the factorial function. The multiplication function *times* used in the definition of *factorial* can be defined in terms of $\boldsymbol{mit}_*$ and an addition function; in turn, the addition function can be defined in terms of $\boldsymbol{mit}_*$ as well. Mendler-style primitive recursion generalizes to higher kinds in the same manner as ***mit*** and ***msfit*** (see Listing 1 in §2).

Abel and Matthes [4] discovered a reduction preserving embedding of the Mendler-style primitive recursion in System Fix_ω, which is a strongly normalizing calculus extending System F_ω with polarized kinds and equi-recursive fixpoints. Polarized kinds extend the kind arrow with polarities in the form of $p\,\kappa_1 \to \kappa_2$ where polarity p is either $+$, $-$, or 0; meaning that the argument must be used in positive, negative, or any position, respectively. For example, in a polarized system, the base structure $N :: * \to *$ for natural numbers in Listing 7 could be assigned $+* \to *$ because its argument r is only used covariantly, and, base structure $ExpF :: * \to *$ in Listing 3 for the untyped HOAS (see §2.2) must be assigned kind $0* \to *$ because its argument r is used in both covariant and contravariant positions. The equi-recursive fixpoint $\mathsf{fix}_\kappa : +\kappa \to \kappa$ in System Fix_ω can be applied only to positive base structures.[12] Abel and Matthes encoded the iso-recursive fixpoint operator μ in terms of the equi-recursive fixpoint operator fix, by converting base structures of arbitrary polarities into base structures of positive polarities, in order to embed ***mpr*** into System Fix_ω.

6.2 Type-based termination and sized types

Type-based termination (coined by Barthe and others [13]) stands for approaches that integrate termination into type checking, as opposed to syntactic approaches that reason about termination over untyped term structures. The Mendler-style approach is, of course,

[12] Otherwise, equi-recursive types are able to express diverging computations when they are not restricted to positive polarity.

type-based. In fact, the idea of type-based termination was inspired by Mendler [25, 26]. In the Mendler style, we know that well-typed functions defined using Mendler-style recursion schemes always terminate. This guarantee follows from the design of the recursion scheme, where the use of higher-rank polymorphic types in the abstract operations enforce the invariants necessary for termination.

Abel [2, 3] summarizes the advantages of type-based termination as: *communication* (programmers think using types), *certification* (types are machine-checkable certificates), *a simple theoretical justification* (no additional complication for termination other than type checking), *orthogonality* (only small parts of the language are affected, e.g., principled recursion schemes instead of general recursion), *robustness* (type system extensions are less likely to disrupt termination checking), *compositionality* (one needs only types, not the code, for checking the termination), and *higher-order functions and higher-kinded datatypes* (works well even for higher-order functions and non-regular datatypes). In his dissertation [2] (Section 4.4) on sized types, Abel views the Mendler-style approach as enforcing size restrictions using higher-rank polymorphism as follows:

- The abstract recursive type r in the Mendler style corresponds to $\mu^\alpha F$ in his sized-type system (System $\mathsf{F}_\omega\hat{}$), where the sized type for the value being passed in corresponds to $\mu^{\alpha+1} F$.
- The concrete recursive type μF in the Mendler style corresponds to $\mu^\infty F$ since there is no size restriction.
- By subtyping, a type with a smaller size-index can be cast to the same type with a larger size-index.

The same intuition holds for the termination behaviors of Mendler-style recursion schemes over positive datatypes. For positive datatypes, Mendler-style recursion schemes terminate because r-values are direct subcomponents of the value being eliminated. They are always smaller than the value being passed in. Types enforce that recursive calls are only well-typed, when applied to smaller subcomponents.

Abel's System $\mathsf{F}_\omega\hat{}$ can express primitive recursion quite naturally using subtyping. The casting operation $(r \to \mu F)$ in Mendler-style primitive recursion corresponds to an implicit conversion by subtyping from $\mu^\alpha F$ to $\mu^\infty F$ because $\alpha \leq \infty$. System $\mathsf{F}_\omega\hat{}$ [2] is closely related to System Fix_ω [4]. Both of these systems are base on equi-recursive fixpoint types over positive base structures. Both of these systems are able to embed (or simulate) Mendler-style primitive recursion (which is based on iso-recursive types) via the encoding [19] of arbitrary base structures into positive base structures.

Abel's sized-type approach evidences good intuition concerning the reasons that certain recursion schemes terminate over positive datatypes. But, we have not gained a useful intuition of whether or not those recursion schemes would terminate for negative datatypes, unless there is an encoding that can translate negative datatypes into positive datatypes. For primitive recursion, this is possible (as we mentioned above). However, for our recursion scheme ***msfit***, which is especially useful over negative datatypes, we do not know of an appropriate encoding that can map the inverse-augmented fixpoints into positive fixpoints. So, it is not clear whether the sized-type approach based on positive equi-recursive fixpoints can provide a good explanation for the termination of ***msfit***.

In §7.1, we will discuss another Mendler-style recursion scheme (***mprsi***), which is also useful over negative datatypes and believed to have a termination property (not yet proved) based on the size of the index in the datatype.

6.3 Parametric compositional data types

Bahr and Hvited developed a generic programming library in Haskell, *compositional data types* (CDT) [11], which builds on Wouter Swierstra's ideas of *data types à la carte* [27]. Recently, they extended CDT to handle binders by adopting Adam Chlipala's idea of PHOAS [16], naming thier new extension as *parametric compositional data types* (PCDT). In Section 3 of their paper on PCDT [12], they give an enlightening comparative summary on a series of studies on recursion schemes over mixed-variant datatypes in the conventional setting — Meijer and Hutton [24], Fegaras and Sheard [18], Washburn and Weirich [30], and their own.

PCDT is based on the conventional style, relying on ad-hoc polymorphism. That is, they need to derive a class instance of an appropriate algebra in order to define a desired recursion scheme for each datatype definition. For example, a functor instance for iteration and a difunctor (or profunctor) instance for iteration with inverses over regular datatypes. Since conventional-style recursion schemes do not generalize naturally to non-regular datatypes such as GADTs, they also need to derive different class instances, that is, higher-order functor and difunctor instances for non-regular datatypes. To alleviate this drawback of the conventional style, they automate instance derivation by meta-programming using Template Haskell for the PCDT library user.

On the contrary, the Mendler style, being based on higher-order parametric polymorphism, enjoys uniform definitions of recursion schemes across arbitrary kinds of datatypes, naturally generalizing from regular to non-regular datatypes. In §7.2, we demonstrate this elegance of the Mendler style by formulating a Mendler-style counterpart of the conventional-style recursion scheme in PHOAS. Here, we summarize the key idea how Bahr and Hvited [12] factored out the fixpoint operator from recursive formulations of PHOAS,[13] in order to lead up the discussion in §7.2.

In PCDT, they transfer the essence of PHOAS using two-level types that are equipped with an extra parameter in base functors as well as in the fixpoint operator. For example, their fixpoint operator and the base functor for the untyped HOAS would be defined as:[14]

$$\mathbf{data}\ \hat{\mu}_*\ (f :: * \to * \to *)\ a = \boldsymbol{In\hat{n}}_*\ (f\ a\ (\hat{\mu}_*\ f\ a))\ \mid\ \mathit{Var}_*\ a$$
$$\mathbf{data}\ \mathit{ExpF}\ r_-\ r = \mathit{Lam}\ (r_- \to r)\ \mid\ \mathit{App}\ r\ r$$

Their fixpoint operator $\hat{\mu}_*$ takes a type constructor of kind $* \to * \to *$ as an argument, unlike the previously discussed fixpoint operators (e.g., μ_* or μ'_*) that take arguments of kind $* \to *$. Note the use of two parameters r_- and r used in contravariant and covariant positions respectively in the definition of *ExpF*; the additional parameter r_- is used in a contravariant recursive position in the argument of the *Lam* constructor.

Then, the recursive type for the untyped HOAS is defined as the fixpoint of base *ExpF*:

$$\mathbf{type}\ \mathit{Exp}'\ a = \hat{\mu}_*\ \mathit{ExpF}\ a \qquad \text{-- pre-expressions that may contain } \mathit{Var}_*$$
$$\mathbf{type}\ \mathit{Exp} = \forall\ a.\ \mathit{Exp}'\ a \qquad \text{-- } \mathit{Var}_*\text{-free expressions enforced by parametricty}$$

When *ExpF* is applied to $\hat{\mu}_*$, the parameter r_- matches with the answer type a and the parameter r matches with the recursive type ($\hat{\mu}_* \mathit{ExpF}\ a$). Their Var_* constructor for $\hat{\mu}_*$ serves the same purpose (i.e., injecting an inverse of an answer value) as our $\mathit{Inverse}_*$ for μ'_*.

---

[13] An online posting of Edward Kmett [22] also discusses PHOAS in a formulation very similar to PCDT.

[14] Bahr and Hvited named their fixpoint operator *Trm* in PCDT. Here, we call it $\hat{\mu}$ in order to use a notation similar to the other operators (μ and $\check{\mu}$) in this paper. In addition, they compose base functors with multiple constructors such as *ExpF* from several single constructor functors; hence, their library is named *compositional*. Here, we focus the discussions on the *parametric* flavor of their contribution.

Finally, the constructor functions for the untyped HOAS are defined as follows:

$$
\begin{array}{l}
lam\ \ f = \boldsymbol{\hat{In}}_*\ (Lam\ (f\ .\ Var_*))\ \text{--}\ ::\ (\hat{\mu}_*\ f\ a \to \hat{\mu}_*\ ExpF\ a) \to \hat{\mu}_*\ ExpF\ a \\
app\ \ e_1\ e_2 = \boldsymbol{\hat{In}}_*\ (App\ e_1\ e_2)\ \ \text{--}\ ::\ \hat{\mu}_*\ ExpF\ a \to \hat{\mu}_*\ ExpF\ a \to \hat{\mu}_*\ ExpF\ a
\end{array}
$$

Note the similarities between the types of the constructor functions above and the types of the constructor functions in Listing 3. A notable difference is where the inverse injection is used: their Var_* is used in the constructor function implementation (lam), while our $Inverse_*$ is used in the recursion scheme implementation ($\boldsymbol{msfit}_*$).

6.4 Self-interpreter of System F_ω

Recently, there has been a breakthrough in normalization barrier of defining a self-interpreter within a strongly normalizing language. Previously, it was believed that self-interpreters were definable only in Turing-complete languages. Brown and Palsberg [14] successfully defined a self-interpretation of the System F_ω within System F_ω. Interestingly, they also used HOAS repesentation of terms and a subset of Haskell (which is believed to be a subset of F_ω) to semi-formally prove their theories, similarly to the previos work of [8]. In perspective of this recent breaktrough, the existence of an F_ω-embedding for *msfit*, which can express simply-typed HOAS evaluation, is indeed probable.

7 Further work

We present two threads of further work regarding Mendler-style recursion schemes over mixed-variant datatypes — Mendler-style primitive recursion with a sized index (§7.1) and Mendler-style parametric iteration (§7.2).

7.1 Mendler-style primitive recursion with a sized index

In §2 and §3, we discussed Mendler-style iteration with a syntactic inverse, ***msfit***, which is particularly useful for defining functions over negative (or mixed-variant) datatypes. We demonstrated the usefulness of ***msfit*** by defining functions over HOAS:

- the string formatting function *showExp* for the untyped HOAS using $\boldsymbol{msfit}_*$ (Listing 3),
- the type-preserving evaluator *eval* for the simply-typed HOAS using $\boldsymbol{msfit}_{*\to*}$ (Listing 4).

In this subsection, we speculate about another Mendler-style recursion scheme, ***mprsi***, motivated by an example similar to the *eval* function. The name ***mprsi*** stands for Mendler-style primitive recursion with a sized index.

Listing 8 A simply-typed HOAS evaluation via a user-defined value domain.

$$
\begin{array}{l}
\mathbf{data}\ \ ExpF\ \ r\ \ t\ \ \mathbf{where}\ \ Lam :: (r\ t_1 \to r\ t_2) \to ExpF\ r\ (t_1 \to t_2) \\
\qquad\qquad\qquad\qquad\qquad App :: r\ (t_1 \to t_2) \to r\ t_1 \to ExpF\ r\ t_2 \\
\mathbf{type}\ \ Exp'\ a\ t = \mu'_{*\to*}\ ExpF\ a\ t \\
\mathbf{type}\ \ Exp\ t = \forall\ a\ .\ Exp'\ a\ t \\
\\
\mathbf{data}\ \ V\ r\ t\ \ \mathbf{where}\ \ VFun :: (r\ t_1 \to r\ t_2) \to V\ r\ (t_1 \to t_2) \\
\mathbf{type}\ \ Val\ \ t = \mu_{*\to*}\ V\ t\ \ \ \text{--- user defined value domain} \\
val\ \ f = \boldsymbol{In}_{*\to*}\ (VFun\ f) \\
\\
veval :: Exp\ t \to Val\ t \\
veval\ \ e = \boldsymbol{msfit}_{*\to*}\ \varphi\ e\ \ \mathbf{where} \\
\quad \varphi :: Phi'_{*\to*}\ ExpF\ (\mu_{*\to*}\ V)
\end{array}
$$

```
  φ inv ev (Lam f) = val(λv → ev (f (inv v)))
  φ inv ev (App e₁ e₂) = unVal(ev e₁) (ev e₂)
-- unVal does not follow the restrictions of the Mendler style.
-- Its definition relies on pattern matching against In_{*→*}.
unVal :: Val (t₁ → t₂) → (Val t₁ → Val t₂)
unVal (In_{*→*}(VFun f)) = f
```

We review the *eval* example and then compare it to our motivating example *veval* for ***mprsi***. Both *eval* and *veval* are illustrated in Listing 8. Recall that this code is written in Haskell, following the Mendler-style conventions. The function $eval :: Exp\ t \to K\,t$ is a type preserving evaluator that evaluates an HOAS expression of type t to a (Haskell) value of type t. The *eval* function always terminates because $\boldsymbol{msfit}_{*\to*}$ always terminates. Recall that $\boldsymbol{msfit}_{*\to*}$ and $\mu'_{*\to*}$ can be embedded into System F_ω.

The motivating example $veval :: Exp\ t \to Val\ t$ is also a type-preserving evaluator. Unlike *eval*, it evaluates to a user-defined value domain *Val* of type t (rather than a Haskell value). The definition of *veval* is similar to *eval*; both of them are defined using $\boldsymbol{msfit}_{*\to*}$. The first equation of φ for evaluating the *Lam*-expression is essentially the same as the corresponding equation in the definition of *eval*. The second equation of φ for evaluating the *App*-expression is also similar in structure to the corresponding equation in the definition of *eval*. However, the use of *unVal* is problematic. In particular, the definition of *unVal* relies on pattern matching against $\boldsymbol{In}_{*\to*}$. Recall that one cannot freely pattern match against a recursive value in the Mendler style. Recursive values must be analyzed (or eliminated) by using Mendler-style recursion schemes. It is not a problem to use η^{-1} in the definition of *eval* because K is non-recursive.

It is unlikely that *unVal* can be defined using any of the existing Mendler-style recursion schemes. So, we designed a new Mendler-style recursion scheme that can express *unVal*. The new recursion scheme ***mprsi*** extends ***mpr*** with an additional uncast operation. Recall that ***mpr*** has two abstract operations, call and cast. So, ***mprsi*** has three abstract operations, call, cast, and uncast. In the following paragraphs, we explain the design of ***mprsi*** step-by-step.

Let us try to define *unVal* using $\boldsymbol{mpr}_{*\to*}$ and examine where it falls short: $\boldsymbol{mpr}_{*\to*}$ provides two abstract operations, *cast* and call, as it can be seen from the type signature:

```
mpr_{*→*} :: (∀ r i. (∀ i. r i → μ_{*→*} f i)   -- cast
                   → (∀ i. r i → a i)           -- call
                   → (f r i → a i)              ) → μ_{*→*} f i → a i
```

We attempt to define *unVal* using $\boldsymbol{mpr}_{*\to*}$ as follows:

```
unVal :: μ_{*→*} V (t₁ → t₂) → (μ_{*→*} V t₁ → μ_{*→*} V t₂)
unVal = mpr_{*→*} φ where
  φ cast call (VFun f) = ...
```

Inside the φ function, we have a function $f :: (r\ t_1 \to r\ t_2)$ over abstract recursive values. We need to cast f into a function over concrete recursive values $(\mu_{*\to*}\,V\,t_1 \to \mu_{*\to*}\,V\,t_2)$. We should not need to use *call*, since we do not expect to use any recursion to define *unVal*. So, the only available operation is $cast :: (\forall\ i. r\ i \to \mu_{*\to*} f\ i)$. Composing *cast* with f, we can get $(cast\,.\ f) :: (r\ t_1 \to \mu_{*\to*}\,V\,t_2)$, whose codomain $(\mu_{*\to*}\,V\,t_2)$ is exactly what we want. But, the domain is still abstract $(r\ t_1)$ rather than being concrete $(\mu_{*\to*}\,V\,t_1)$. We are stuck.

What additional abstract operation would help us complete the definition of *unVal*? We need an abstract operation to cast from $(r\ t_1)$ to $(\mu_{*\to*}\,V\,t_1)$ in a contravariant position. If we had an inverse of cast, $uncast :: (\forall\ i. \mu_{*\to*} f\ i \to r\ i)$, we can complete the definition of *unVal* by composing *uncast*, *f*, and *cast*. That is, *uncast* . *f* . *cast* :: $(\mu_{*\to*}\, V\, t_1 \to \mu_{*\to*}\, V\, t_2)$. Thus, we can formulate $\boldsymbol{mprsi}_{*\to*}$ with a naive type signature as follows:

$$
\begin{array}{lll}
\boldsymbol{mprsi}_{*\to*} :: (\forall\; r\; i.\; & (\forall\; i.\; r\; i \to \mu_{*\to*}\; f\; i) & \text{-- } cast \\
 & \to (\forall\; i.\; \mu_{*\to*}\; f\; i \to r\; i) & \text{-- } uncast \\
 & \to (\forall\; i.\; r\; i \to a\; i) & \text{-- } call \\
 & \to (f\; r\; i \to a\; i) &) \to \mu_{*\to*}\; f\; i \to a\; i \\
\end{array}
$$
$$\boldsymbol{mprsi}_{*\to*}\; \varphi\; (\boldsymbol{In}_{*\to*}\; x) = \varphi\; id\; id\; (\boldsymbol{mprsi}_{*\to*}\; \varphi)\; x$$

Although the type signature above is type-correct, it is too powerful. The Mendler-style uses types to forbid non-terminating computations as ill-typed. Having both *cast* and *uncast* supports the same ability as freely pattern matching over recursive values, which can lead to non-termination. To recover the guarantee of termination, we need to restrict the use of either *cast* or *uncast*, or both.

Let us see how this non-termination might occur. If we allowed $\boldsymbol{mprsi}_{*\to*}$ with the naive type signature above, we could write an evaluator (similar to *veval* but for an untyped HOAS), which does not always terminate. This evaluator would diverge for terms with self application. Here, we walk through the process of defining an untyped HOAS. The base structures of the untyped HOAS and its value domain can be defined as follows:

$$
\begin{array}{l}
\mathbf{data}\; \mathit{ExpF}_u\; r\; t = \mathit{Lam}_u\; (r\; t \to r\; t) \mid \mathit{App}_u\; (r\; t)\; (r\; t) \\
\mathbf{data}\; V_u\; r\; t = \mathit{VFun}_u\; (r\; t \to r\; t)
\end{array}
$$

Fixpoints of the structures above represent the untyped HOAS and its value domain. Here, the index t is bogus; that is, it does not track the types of terms but remains constant everywhere. Using the naive version of $\boldsymbol{mprsi}_{*\to*}$ above, we can write an evaluator similar to *veval* for the untyped HOAS $(\mu_{*\to*}\mathit{ExpF}_u())$ via the value domain $(\mu_{*\to*}\, V_u())$, which would obviously not terminate for some inputs.

Why did we believe that *veval* always terminates? Because it evaluates a well-typed HOAS, whose type is encoded as an index t in the recursive datatype $(\mathit{Exp}\; t)$. That is, the use of indices as types is the key to the termination property. Therefore, our idea is to restrict the use of the abstract operations by enforcing constraints over their indices; in that way, we would still be able to write *veval* for the typed HOAS, but would get a type error when we try to write an evaluator for the untyped HOAS.

We suggest that some of the abstract operations of $\boldsymbol{mprsi}_{*\to*}$ should only be applied to the abstract values whose indices are smaller in size compared to the size of the argument index. For the *veval* example, the structural ordering over types can be given as $t_1 < (t_1 \to t_2)$ and $t_2 < (t_2 \to t_1)$. We have two candidates for the type signature of $\boldsymbol{mprsi}_{*\to*}$:

- Candidate 1: restrict uses of both *cast* and *uncast*

$$
\begin{array}{lll}
\boldsymbol{mprsi}_{*\to*} :: (\forall\; r\; j.\; & (\forall\; i.\; (i<j) \Rightarrow r\; i \to \mu_{*\to*}\; f\; i) & \text{-- } cast \\
 & \to (\forall\; i.\; (i<j) \Rightarrow \mu_{*\to*}\; f\; i \to r\; i) & \text{-- } uncast \\
 & \to (\forall\; i.\; r\; i \to a\; i) & \text{-- } call \\
 & \to (f\; r\; j \to a\; j) &) \to \mu_{*\to*}\; f\; i \to a\; i \\
\end{array}
$$

- Candidate 2: restrict the use of *uncast* only

$$
\begin{array}{lll}
\boldsymbol{mprsi}_{*\to*} :: (\forall\; r\; j.\; & (\forall\; i.\; r\; i \to \mu_{*\to*}\; f\; i) & \text{-- } cast \\
 & \to (\forall\; i.\; (i<j) \Rightarrow \mu_{*\to*}\; f\; i \to r\; i) & \text{-- } uncast \\
 & \to (\forall\; i.\; r\; i \to a\; i) & \text{-- } call \\
 & \to (f\; r\; j \to a\; j) &) \to \mu_{*\to*}\; f\; i \to a\; i \\
\end{array}
$$

We strongly believe that the first candidate always terminates, but it might be overly restrictive. Maybe the second candidate is enough to guarantee termination? Both candidates allow defining *veval*, since one can define *unVal* using $\boldsymbol{mprsi}_{*\to*}$ with either one of the candidates. Both candidates forbid the definition of an evaluator over the untyped HOAS, because neither supports extracting functions from the untyped value domain.

We need further studies to prove termination properties of ***mprsi***. The sized-type approach, discussed in the related work section, seems to be relevant to showing termination of ***mprsi***. However, existing theories on sized-types are not directly applicable to ***mprsi*** because they are focused on positive datatypes, but not negative datatypes.

Listing 9 Mendler-style parametric iteration (***mphit***) at kind $*$ and $* \to *$.

```
data μ̂_*     f a    = În_*    (f a (μ̂_*    f a))     | Var_*    a
data μ̂_{*→*} f a i  = În_{*→*} (f a (μ̂_{*→*} f a) i) | Var_{*→*} (a i)
type Phi_*     f a = ∀ r.(r a → a) → f a (r a) → a
type Phi_{*→*} f a = ∀ r.(r a .→ a) → f a (r a) .→ a
-- În_*^{-1} :: μ̂_* f a → Maybe (f a (μ̂_* f a))
În_*^{-1} (În_* x) = Just x
În_*^{-1} _        = Nothing
-- În_{*→*}^{-1} :: μ̂_{*→*} f a i → Maybe (f a (μ̂_{*→*} f a) i)
În_{*→*}^{-1} (În_{*→*} x) = Just x
În_{*→*}^{-1} _            = Nothing

mphit_* :: Phi_* f a → (∀ a. μ̂_* f a) → a
mphit_* φ x = mphit φ x where
  mphit φ (În_* x)  = φ (mphit φ) x
  mphit φ (Var_* a) = a

mphit_{*→*} :: Phi_{*→*} f a → (∀ a. μ̂_{*→*} f a i) → a i
mphit_{*→*} φ x = mphit φ x where
  mphit :: Phi_{*→*} f a → μ̂_{*→*} f a .→ a
  mphit φ (În_{*→*} x)  = φ (mphit φ) x
  mphit φ (Var_{*→*} a) = a
```

Listing 10 The *eval* example revisited using $\boldsymbol{mphit}_{*\to*}$.

```
data ExpF r_ r t where Lam :: (r_ t1 → r t2) → ExpF r_ r (t1→t2)
                       App :: r (t1→t2) → r t1 → ExpF r_ r t2
type Exp' a t = μ̂_{*→*} ExpF a t
type Exp t = ∀ a. Exp' a t
-- lam :: (μ̂_{*→*} f a t1 → μ̂_{*→*} ExpF a t2) → μ̂_{*→*} ExpF a (t1 → t2)
lam f = În_{*→*} (Lam (f . Var_{*→*}))
-- app :: μ̂_{*→*} ExpF a (t1 → t2) → μ̂_{*→*} ExpF a t1 → μ̂_{*→*} ExpF a t2
app e1 e2 = În_{*→*} (App e1 e2)

data K a = η {η^{-1} :: a}
-- eval :: Exp t → K t
eval = mphit_{*→*} φ where
  φ :: Phi_{*→*} ExpF K
  φ ev (Lam f)   = η (λv → η^{-1}(ev (f (η v))))
  φ ev (App f x) = η (η^{-1}(ev f) (η^{-1}(ev x)))
```

7.2 Mendler-style parametric iteration

Inspired by the conventional style iteration over PHOAS [12] (discussed in §6.3), we formulate its Mendler-style counterpart *Mendler-style parametric iteration* (***mphit***). Listing 9 illustrates Haskell transcription of ***mphit*** at kind $*$ and $* \to *$. Note that datatype definitions of $\hat{\mu}$ and type signatures of ***mphit*** at both kinds have virtually identical structure except for the index i and that the implementations of $\boldsymbol{mphit}_*$ and $\boldsymbol{mphit}_{*\to*}$ have exactly the same structure.

A notable difference from ***msfit***, besides the extra parameter (r_-) in base functors (discussed in §6.3), is that ***mphit*** provides only one abstract operation (abstract recursive call) as you can observe from the type synonym definitions of Phi_* and $Phi_{*\to*}$. For instance, $(r\ a \to a)$ is the type of the abstract recursive call provided by $\boldsymbol{mphit}_*$. Recall that ***msfit*** provides two abstract operations (abstract inverse and recursive call) while ***mit*** provides one (recursive call only). As a result, the first equations of ***mphit*** in the definitions of $\boldsymbol{mphit}_*$ and $\boldsymbol{mphit}_{*\to*}$ are exactly the same in structure as the definitions of $\boldsymbol{mit}_*$ and $\boldsymbol{mit}_{*\to*}$ in Listing 1. Hence, the revisited example of *eval* (Listing 10) is more succinct than its corresponding example using ***msfit*** (Listings 4), omitting *inv* in the definition of the φ functions. Note that the uses of *inv* in ***mphit*** are delegated to the constructor functions of $\hat{\mu}$-types, which involve contravariant recursive occurrences; for instance, *lam* in Listing 10 is defined in terms of $Var_{*\to*}$, which is the constructor of $\hat{\mu}_{*\to*}$ for injecting inverses.

Listing 11 Constant folding using $\boldsymbol{mphit}_*$.

```
data ExprF r_ r = LET r (r_ → r) | ADD r r | LIT Int
type Expr' a = μ̂* ExprF a
type Expr = ∀ a. Expr' a
eLet e f = In̂* (LET e (f . Var*))
eAdd e1 e2 = In̂* (ADD e1 e2)
eLit n     = In̂* (LIT n)
-- constfold :: Expr → Expr
constfold e = mphit* φ e where
  φ cf (LET e f)     = eLet (cf e) (cf . f)
  φ cf (LIT n)       = eLit n
  φ cf (ADD e1 e2) = case (In̂*⁻¹ e1', In̂*⁻¹ e2') of
                      (Just(LIT n), Just(LIT m)) → eLit (n + m)
                      _                          → eAdd e1' e2'
                    where e1' = cf e1
                          e2' = cf e2
```

Bahr and Hvited [12] exemplified the strength of their PHOAS-based iteration, compared to those [18, 30, 8] based on ordinary (or strong) HOAS, by defining a constant folding over a small language. In Listing 11, we illustrate the constant folding example in the Mendler style. Note the simplicity of our Mendler-style version — no need for class instances for functor, difunctor, or higher-kinded versions of such algebras. One can freely use $\boldsymbol{In̂}_*^{-1}$ with $\boldsymbol{mphit}_*$ assuming that the recursive type $\hat{\mu}_* f$ is constructed following the convention that the first and second arguments of the base structure $f :: * \to * \to *$ are contravariant and covariant. For instance, r_- is contravariant and r is covariant in the definition of *ExpF*. Because $\hat{\mu}_*$ is defined to be recursive only over the second covariant argument, the type system prevents $\boldsymbol{In̂}_*^{-1}$ from being applied to the first contravariant argument, assuming that the first argument type is different from the second arguments. Within the context of $\varphi :: (\forall\ r.\ (r\ a {\to} a) \to f\ a\ (r\ a) \to a)$, the first and second arguments of f cannot be the same type because a is clearly not unifiable with $(r\ a)$.

Although we transcribed ***mphit*** in Haskell, we have not yet proved its termination property (neither did Bahr and Hvited [12] for their conventional version). To prove its termination, we should find an embedding of ***mphit*** in a strongly normalizing calculus and study the equational properties of that embedding, just as we did for ***msfit*** in §5. We think Fix_ω and $\mathsf{F}_\omega^{\hat{}}$ are good candidate calculi for embedding ***mphit*** because they support polarized kinds. With polarized kinds, we can ensure that parameters r and r_{-} are always used in covariant and contravariant positions, respectively, in base functor definitions. Studying the relation between μ and $\hat{\mu}$, as we did for μ and μ' in §4, is another subject of future work.

8 Summary and future work

We reviewed Mendler-style iteration (***mit***) and primitive recursion (***mpr***) with their typical examples, the list length function (Listing 2) and the factorial function (Listing 7), respectively. ***mpr*** extends ***mit*** with the additional cast operation that converts abstract recursive values to concrete recursive values. Moreover, we reviewed Mendler-style iteration with syntactic inverses (***msfit***) with the HOAS formatting example (Listing 3); this is the "hello world" example of recursion schemes over mixed-variant datatypes. The abstract inverse operation provided by ***msfit***, which is not present in ***mit***, makes it useful over mixed-variant datatypes.

We formulated the type-preserving evaluator for the simply-typed HOAS (Listing 4). This evaluator demonstrates the usefulness of ***msfit*** over indexed mixed-variant datatypes. Moreover, this example is a novel theoretic discovery that type-preserving HOAS evaluation can directly (i.e., without via translation to/from intermediate first-order syntax) embedded into System F_ω because we proved termination of the HOAS evaluator by embedding ***msfit*** into System F_ω (§5.2). Moreover, we studied the equational properties of the embedding (§5.2-5.4) and the subtype relation between ordinary fixpoint types for ***mit*** and their corresponding inverse-augmented fixpoint types for ***msfit*** (§4).

We introduced the idea of Mender-style iteration with a sized index (***mprsi***) motivated by the example of type-preserving evaluation via semantic domain (Listing 8), in contrast to the evaluation example via native values of the host language using ***msfit*** (Listing 4). ***mprsi*** extends ***mpr*** with the additional abstract uncast operation, which is the inverse of the abstract cast operation provided by ***mpr*** as well. However, the uncast operation needs to be restricted in order to guarantee termination. Termination proof for ***mprsi*** needs further investigation. Termination proof for ***mprsi*** is expected to be more challenging than ***msfit*** and ***mphit*** because it involves size measure constraint unlike other Mendler-style recursion schemes we have stuided so far. Our startegy for the termination proof of ***mprsi*** is to first come up with a version of ***mprsi*** that distinguishes between positive and negative recursive occurrences as in ***mphit*** and then apply theries developed in such context (e.g., [1]).

We introduced Mendler-style iteration over PHOAS (***mphit***) and demonstrated its usefulness by writing the type-preserving evaluator over typed PHOAS (Listing 10); this is similar to the HOAS evaluator using ***msfit*** (Listing 4) but even more succinct because abstract inverses are not needed. Moreover, we can write examples using ***mphit*** that are not expressible using ***msfit*** such as the constant folding example (Listing 11). We hope to show termination of ***mphit*** by finding its embedding in System Fix_ω, which is an extension of System F_ω that can embed ***mpr***.

Mendler-style recursion schemes naturally extends term-indexed datatypes (e.g., length-indexed lists) so that one can express more fine-grained properties of programs in their types. Ahn, Sheard, Fiore, and Pitts [9] developed a term-indexed calculi System F_i by extending

System F_ω with term indices in order to embed Mendler-style recursion schemes such as ***mit*** and ***msfit*** over term-indexed datatypes. System Fix_i [7] is a similar extension to System Fix_ω that can embed ***mpr*** and (hopefully) ***mphit*** over term-indexed datatypes.

Based on the theories of term-indexed calculi, we have been developing a language called Nax, named after *Nax P. Mendler*, that supports Mendler-style recursion schemes over both type- and term-indexed datatypes as native language constructs. The Nax language [7] is designed to adopt advantages of both functional programming languages (e.g., mixed-variant datatypes, type inference) and dependently-typed proof assistants (e.g., fine-grained properties, logical consistency). The semantics of Nax can be understood by embedding its key constructs such as datatypes and recursion schemes into the term-indexed calculi.

One of the challenges in the language design is to choose as many useful set of Mendler-style recursion schemes, including ones for mixed-variant datatypes, that have compatible embeddings in a term-indexed calculus. Not all recursion schemes would necessarily have close relationship between their fixpoint types, such as the subtyping relation between fixpoints of ***mit*** and ***msfit*** discussed in §4. ***mit*** and ***mpr*** are compatible as well. However, we think it may be difficult to find compatible embeddings for both ***mpr*** and ***msfit***. We hope to discover an embedding of ***mphit*** that is compatible with the embedding of ***mpr***, hopefully using the same calculus (Fix_ω), which is used for showing termination of ***mpr***.

There are several other features in consideration to develop Nax to become a more powerful and practical language. Some have already been implemented and awaiting theoretical clarifications, while others are just preliminary thoughts: restrictive form of kind polymorphsim, pattern match coverage checking, generalization of arrow (i.e., function) types in abstract operations to generalize Mendler-style recursion schemes even further (e.g., monadic recursion [11]), and handling computations that cannot (or need not) be internally proved terminating by the type system (e.g., bar types [17], mobile types [15]).

Acknowledgements

Thanks to Gabor Greif and Keewon Seo for their proofreading, and Ralph Matthes for his suggestion on looking into interpolation.

References

1 Martin Abadi and Marcelo Fiore. Syntactic considerations on recursive types. In *LICS: IEEE Symposium on Logic in Computer Science*, 1996.
2 Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.
3 Andreas Abel. Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types, February 15 2012.
4 Andreas Abel and Ralph Matthes. Fixed points of type constructors and primitive recursion. In *CSL*, volume 3210 of *LNCS*, pages 190–204. Springer, 2004.
5 Andreas Abel, Ralph Matthes, and Tarmo Uustalu. Generalized iteration and coiteration for higher-order nested datatypes. In *FoSSaCS*, volume 2620 of *LNCS*, pages 54–69. Springer, 2003.
6 Andreas Abel, Ralph Matthes, and Tarmo Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theoretical Computer Science*, 333(1-2):3 – 66, 2005.
7 Ki Yung Ahn. *The Nax language*. PhD thesis, Department of Computer Science, Portland State University, PO Box 751, Portland, OR, 97207 USA, November 2014.
8 Ki Yung Ahn and Tim Sheard. A hierarchy of Mendler-style recursion combinators: Taming inductive datatypes with negative occurrences. In *ICFP '11*, pages 234–246. ACM, 2011.

9 Ki Yung Ahn, Tim Sheard, Marcelo Fiore, and Andrew M. Pitts. System Fi: a higher-order polymorphic lambda calculus with erasable term indices. In *Proceedings of the 11th international conference on Typed lambda calculi and applications*, TLCA '13, 2013.

10 Emil Axelsson and Koen Claessen. Using circular programs for higher-order syntax: functional pearl. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 257–262. ACM, 2013.

11 Patrick Bahr and Tom Hvitved. Compositional data types. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming*, WGP '11, pages 83–94, New York, NY, USA, September 2011. ACM.

12 Patrick Bahr and Tom Hvitved. Parametric compositional data types. In *MSFP*, pages 3–24, 2012.

13 Gilles Barthe, Maria João Frade, E. Giménez, Luis Pinto, and Tarmo Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004.

14 Matt Brown and Jens Palsberg. Breaking through the normalization barrier: A self-interpreter for F-omega. In *POPL '16*. ACM, 2016.

15 Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. *SIGPLAN Not.*, 49(1):33–45, January 2014.

16 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP '08*, pages 143–156. ACM, 2008.

17 Robert L Constable and Scott Fraser Smith. Partial objects in constructive type theory. Technical report, Cornell University, 1987.

18 Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 284–294, New York, NY, USA, 1996. ACM.

19 Herman Geuvers. Inductive and coinductive types with iteration and recursion. In B. Nordström, K. Pettersson, and G. Plotkin, editors, *Informal Proceedings Workshop on Types for Proofs and Programs, Båstad, Sweden, 8–12 June 1992*, pages 193–217. Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ., 1992.

20 Herman Geuvers. Induction is not derivable in second order dependent type theory. In *Proceedings of the 5th international conference on Typed lambda calculi and applications*, TLCA'01, pages 166–181, Berlin, Heidelberg, 2001. Springer-Verlag.

21 Lars Hallnäs. On systems of definitions, induction and recursion. *BIT Numerical Mathematics*, 32(1):45–63, 1992.

22 Edward Kemptt. PHOAS for free, December 2013. FP Complete™ online article available at `https://www.fpcomplete.com/user/edwardk/phoas`, Accessed: 2014-08-03.

23 Ralph Matthes. Interpolation for natural deduction with generalized eliminations. In *International Seminar on Proof Theory in Computer Science (PTCS'01)*, LNCS, 2001.

24 Erik Meijer and Graham Hutton. Bananas in space: extending fold and unfold to exponential types. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, FPCA '95, pages 324–333, New York, NY, USA, 1995. ACM.

25 N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *LICS*, pages 30–36, 1987.

26 N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Ann. Pure Appl. Logic*, 51(1-2):159–172, 1991.

27 Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18:423–436, 7 2008.

28 Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nordic Journal of Computing*, 6(3):343–361, 1999.

29 Tarmo Uustalu and Varmo Vene. Coding recursion à la Mendler (extended abstract). In Johan Jeuring, editor, *Proc. of 2nd Workshop on Generic Programming*, Tech. Report UU-CS-2000-19, Dept. of Computer Science, Utrecht Univ., pages 69–85. 2000.

30 Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: encoding higher-order abstract syntax with parametric polymorphism. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, ICFP '03, pages 249–262, New York, NY, USA, 2003. ACM.

Listing 12 An embedding algorithm in Haskell

```
data Sig = P | M -- P and M stands for + and −

data Ty = TV Var | All Var Ty | Ty :→ Ty
        | Unit | Ty :×: Ty -- These can be encoded too but
        | Void | Ty :+: Ty -- for convenience of presentation.

data Tm = Fn (Tm → Tm) | Tm :$ Tm -- term representation using HOAS
        | Cunit | Tm :× Tm | Fst Tm | Snd Tm
        | L Tm | R Tm | CaseLR Tm (Tm → Tm) (Tm → Tm)
        | Lift Tm Tm | Phi -- Added constants for lift and φ.

type Var = Int -- Give some suitable type for variable.

flipSig :: Sig → Sig
flipSig M = P
flipSig P = M

-- For a base structure F defined as  data F r = C t₁ ... tₙ | ... ,
-- the embedding of the constructors of μ'* F has the form of
-- c = in_R (λφ.φ η⁻¹ (C (rEm t₁) ··· (rEm tₙ)))) where rEm is defined as below:
rEm :: Sig → Var → Ty → Tm → Tm
rEm _ _ Unit         = id -- or const Cunit
rEm _ _ Void         = id
rEm _ r (TV x) | r≠x = id -- Ignore variables other than the recursive one.
rEm P _ (TV _)       = Lift Phi -- Apply lift φ in positive occurrence
rEm M _ (TV _)       = L        -- and apply in_L in negative occurrence.
rEm p r (a :→ b)     = λf → Fn (λx → rEm p r b (f :$ rEm p' r a x))
                       where p' = flipSig p
rEm p r (a :×: b)    = λx → rEm p r a x :× rEm p r b x
rEm p r (a :+: b)    = λx → CaseLR x (rEm p r a) (rEm p r b)
rEm p r (All x b) | r≠x = rEm p r b
rEm _ _ (All _ _)    = error "should␣have␣been␣alpha␣renamed"
```

A Appendix: a type-directed embedding algorithm for the constructors of regular datatypes used with $msfit_*$

In §5.2, we embedded the type constructors of the untyped HOAS, which is a mixed-variant datatype with both positive and negative occurrences, as annotated by $+$ and $-$ in $App\ r^+\ r^+$ and $Abs\ (r^- \to r^+)$. The HOAS example discussed in §5.2 has these recursive occurrences either at topmost level, as in $App\ r^+\ r^+$ occurring twice positively, or on both sides of the arrow type at topmost level, as in $Abs\ (r^- \to r^+)$ occurring negatively on the left-hand side and positively on the right-hand side. Positive and negative occurrences are embedded differently – recall that we used $lift\,\varphi$ for positive occurrences and in_L for negative occurrences (see p11 in §5.2).

In general, recursive occurrences may occur more deeply inside the type structure. For example, consider $\mu'_* F$ where **data** $F\ r = \ C\ ((r^+ \to r^-) \to r^+)$. The leftmost occurrence of r in the definition of F is positive because it is on the left hand side of the arrow at negative position (negative of negative considered positive). Other type structures such as sums, products, and universal quantifications do not have affect on the sign of recursive occurrences in its subcomponents. That is, the subcomponents maintain the same sign for recursive occurrences as their outermost position.

In Listing 12, we describe an algorithm implementing the idea discussed in the previous two paragraphs using Haskell. This algorithm is type-directed, that is, it analyzes the given base structure F to derive the embeddings for the constructors of $\mu'_* F$. Here, we only consider regular datatypes. By convention, the recursive argument r always comes at the last. For instance, the base structure for lists **data** $L\ a\ r = ...$ where we take its fixpoint as $\mu_*(L\ a)$ for the list datatype. Therefore without loss of generality, we assume that the base structures are defined as **data** $F\ r = ...$. Since our target calculus is polymorphic, we need variables (Var) and universal quantification (All) to represent types (Ty). We have sums (:+:) and products (:×:) and their identities $Unit$ and $Void$ because base structures are defined as sums of products of types. We can inline the embeddings of recursive types of the form $\mu'_* G$ occurring in the definition of F, provided that G is defined prior to F, because we already know the embedding of μ'_* (see p10 in §5.2) and G can also be embedded into System F_ω as sums of products. Therefore, it suffice for the embedding function rEm in Listing 12 to analyze type structures (Ty) in order to generate the embedded terms (Tm).

We think it would be possible to prove the equational properties of this type-directed embedding using interpolation, as used in the paper by Matthes [23]. In fact, his paper has been a hint to derive our algorithm in Listing 12. Although we have only demonstrated the algorithm for regular datatypes, we do not expect difficulties in generalizing this algorithm to include non-regular datatypes, except for truly nested datatypes (e.g., **data** $Bush\ a = BNil \mid BCons\ Bush\ (Bush\ a)$). Embeddings for truly nested datatypes are going to be trickier than the embeddings for the other datatypes because truly nested datatype are indexed by their own types.