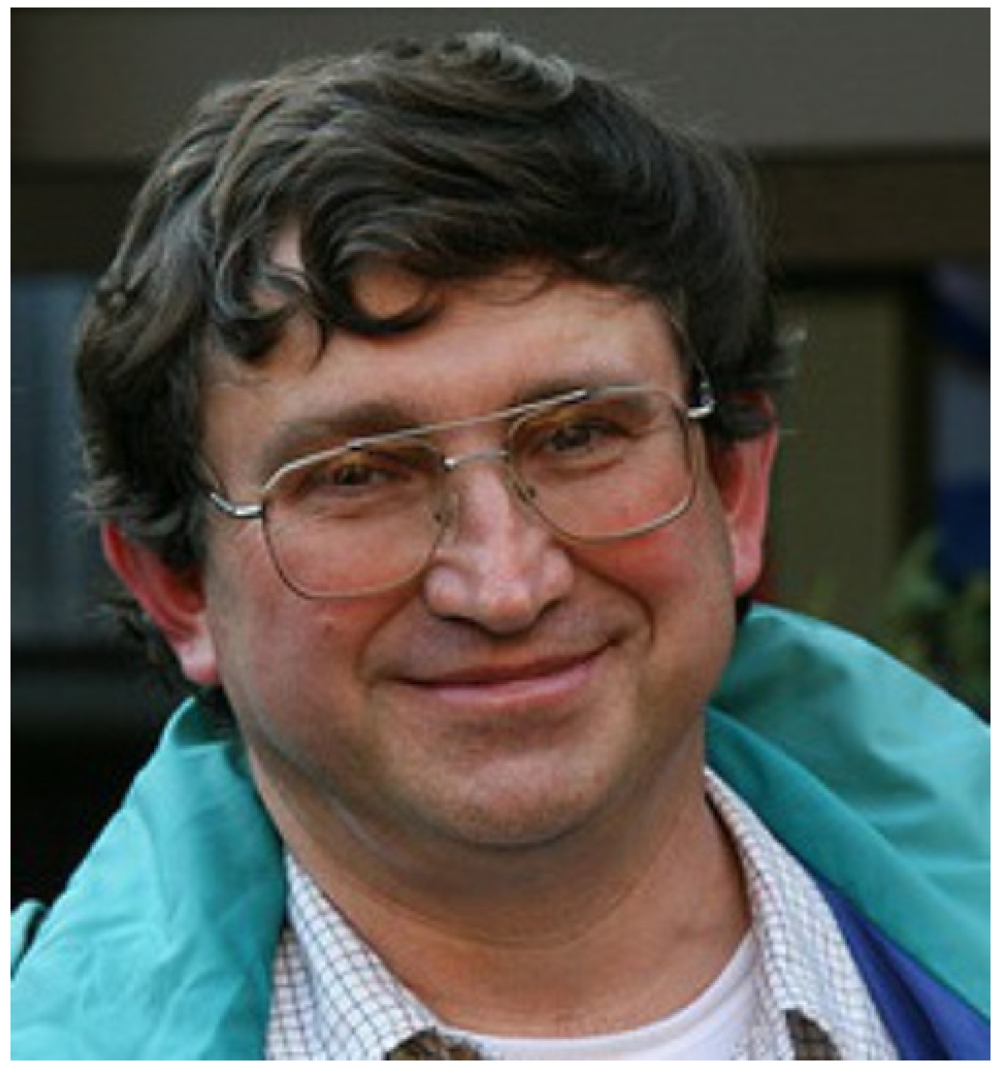
Mendler-style Recursion Schemes for Mixed-Variant Datatypes

Ki Yung Ahn

kya@cs.pdx.edu

Tim Sheard



sheard@cs.pdx.edu

Marcelo P. Fiore

Marcelo.Fiore@cl.cam.ac.uk

Outline

Mendler-style Recursion Schemes

- mit (iteration)
- msfit (iteration with syntactic Inverse)

Untyped HOAS (Regular Mixed-Variant Datatype)

Formatting Untyped HOAS (using msfit at kind $*$)

Simply-Typed HOAS (Indexed Mixed-Variant Datatype)

Formatting Untyped HOAS (using msfit at kind $* \to *$)

The Nax language

Need for yet another Mendler-style Recursion Scheme

(iso-) Recursive Types in Functional Languages

kinding: $(\mu\text{-form})\dfrac{\Gamma \vdash F : * \to *}{\Gamma \vdash \mu F : *}$

typing: $(\mu\text{-intro})\dfrac{\Gamma \vdash t : F(\mu F)}{\Gamma \vdash \mathsf{In}\ t : \mu F}$ $\quad$ $(\mu\text{-elim})\dfrac{\Gamma \vdash t : \mu F}{\Gamma \vdash \mathsf{unIn}\ t : F(\mu F)}$

reduction: $(\mathsf{unIn}\text{-}\mathsf{In})\dfrac{}{\mathsf{unIn}\ (\mathsf{In}\ t) \rightsquigarrow t}$

already inconsistent as a logic (via Curry--Howard correspondence)
without even having to introduce any term-level recursion
since we can already define general recursion using this

Having both **unrestricted formation** and **unrestricted elimination** of μ leads to inconsistency

data T a r = C (r → a)

w : μ (T a) → a   -- encoding of (λx. x x) in the untyped λ-calc
w = λv. case (unIn v) of (C x) → f (C x)

-- w (C w)   amounts to a well-known diverging term in λ-calculus

fix : (a → a) → a   -- the Y combinator (λf.(λx.f(xx))(λx.f(xx))))
fix = λf. (λx. f (w x)) (In (C (λx. f (w x))))

To recover consistency, one could
either restrict formation   (conventional approach)
or   restrict elimination   (Mendler-style approach)

Conventional Iteration over Recursive Types

kinding: $$(\mu\text{-form}^+)\ \frac{\Gamma \vdash F : * \to * \qquad \mathsf{positive}(F)}{\Gamma \vdash \mu F : *}$$

allow formation of only positive recursive types (i.e., when F has a map)

typing: (μ-intro) and (μ-elim) same as functional language

$$(\mathbf{It})\ \frac{\Gamma \vdash t : \mu F \qquad \Gamma \vdash \varphi : F\,A \to A}{\Gamma \vdash \mathbf{It}\ \varphi\ t : A}$$

reduction: (unIn-In) same as functional language

freely eliminate (i.e. use unIn) recursive values

$$(\mathbf{It}\text{-}\mathsf{In})\ \frac{}{\mathbf{It}\ \varphi\ (\mathsf{In}\ t) \rightsquigarrow \varphi\ (\mathsf{map}_F\ (\mathbf{It}\ \varphi)\ t)}$$

Mendler-style Iteration over Recursive Types

kinding: (μ-form) same as functional language

freely form ANY recursive type!!

typing: (μ-intro) same as functional language

note, no (μ-elim) rule

$$(\mathbf{mit})\,\frac{\Gamma \vdash t : \mu F \qquad \Gamma \vdash \varphi : \forall X.(X \to A) \to F X \to A}{\Gamma \vdash \mathbf{mit}\ \varphi\ t : A}$$

elimination is possible only through mit

reduction: $(\mathbf{mit}\text{-}\mathsf{In})\,\dfrac{}{\mathbf{mit}\ \varphi\ (\mathsf{In}\ t) \rightsquigarrow \varphi\ (\mathbf{mit}\ \varphi)\ t}$

Mendler-style Iteration

```
-- Mu at different kinds (there are many more, one at each kind)
data Mu0 (f :: * → *)                     = In0 (f (Mu0 f ))
data Mu1 (f :: (*→*)→(*→*)) i= In1 (f (Mu1 f ) i)

-- Mendler-style iterators at different kinds
mit0 :: Phi0 f a → Mu f → a
mit0 phi (In0 x) = phi (mit0 phi) x

mit1 :: Phi1 f a i → Mu f i → a i
mit1 phi (In1 x) = phi (mit1 phi) x

                              recursive call
type Phi0 = ∀r.         (r   →  a ) →         (f r   → a  )
type Phi1 = ∀r. (∀i. r i → a i) → (∀i. f r i → a i)
```

eliminating In0 or In1 by pattern matching is only allowed in Mendler-style iterators

- ✔ Uniformly defined at different kinds (can handle non-regular datatypes quite the same way as handling regular datatypes)
- ✔ Terminates for ANY datatype (can embed Mu and mit into Fw. see Abel, Matthes and Uustallu TCS'04)
- ✔ However, not very useful for mixed-variant datatypes

Mendler-style Iteration with a syntactic inverse

```
-- Mu' at different kinds (we only use two of them in this talk)
data Mu'0 (f :: * → *)           a   = In'0 (f (Mu'0 f a))    | Inverse0 a
data Mu'1 (f :: (*→*)→(*→*)) a i  = In'1 (f (Mu'1 f a) i) | Inverse1 (a i)

-- msfit at different kinds
msfit0 :: Phi'0 f a → Mu'0 f → a
msfit0 phi (In'0 x) = phi Inverse0 (msfit0 phi) x

msfit1 :: Phi'1 f a i → Mu'1 f i → a i
msfit1 phi (In'1 x) = phi Inverse1 (msfit1 phi) x

                         inverse              recursive call
type Phi'0 f a = ∀r.     (a    → r a  ) →     (r a    → a  ) →    (f r a      → a  )
type Phi'1 f a = ∀r. (∀i. a i → r a i) → (∀i. r a i → a i) → (∀i. f (r a) i → a i)
```

- ✔ Terminates for ANY datatype (can embed Mu and msfit into Fw. see Ahn and Sheard ICFP'11)
- ✔ msfit is quite useful for mixed-variant datatypes, due to "inverse" operation in addition to "recursive call"

eliminating In'0 or In'1 by pattern matching is only allowed in Mendler-style iterators

Untyped HOAS

(a regular mixed-variant datatype)

```
-- using general recursion at type level
data Exp = Lam (Exp → Exp) | App Exp Exp

-- using fixpoint (Mu'0) over non-recursive base structure (ExpF)
data ExpF r = Lam (r → r) | App r r
type Exp' a = Mu'0 ExpF a           -- (Exp' a) may contain Inverse
type Exp = ∀ a . Exp' a   -- Exp does not contain Inverse

lam :: (∀ a. Exp' a → Exp' a) -> Exp        -- it's not  (Exp → Exp) → Exp
lam  f  = In'0 (Lam f )                 -- f can handle Inverse containg values
                                        -- but can never examine its content

app :: Exp → Exp → Exp
app e₁ e₂ = In'0 (App e₁ e₂)
```

Formatting Untyped HOAS using **msfit0** at kind *

```
showExp :: Exp → String
showExp e = msfit0  phi e vars   where
  phi :: Phi'0 ExpF ([String] → String)
  phi inv showE (Lam z)     =
        λ(v:vs) → "(λ"++v++"→"++ showE (z (inv (const v))) vs ++")"
  phi inv showE (App x y)  =
        λvs     → "("++ showE x vs ++" "++ showE y vs ++")"

                           inverse      recursive call
type Phi0 f a =  ∀ r. (a → r a) -> (r a -> a) → (f r a → a)

vars = [ "x"++show n | n<-[0..] ] :: [String]
```

Simply-Typed HOAS

(a non-regular mixed-variant datatype)

```
-- using general recursion at type level
data Exp t where                -- Exp :: * → *
  Lam :: (Exp t1 → Exp t2) → Exp (t1 → t2)
  App :: Exp (t1 → t2) → Exp t1 → Exp t2

-- using fixpoint (Mu'1) over non-recursive base structure (ExpF)
data ExpF r t where             -- ExpF :: (* → *) → (* → *)
  Lam :: (r t1 → r t2) → ExpF r (t1 → t2)
  App :: r (t1 → t2) → r t1 → ExpF r t2
type Exp' a t = Mu'1 ExpF a t        -- (Exp' a t) might contain Inverse
type Exp t =  ∀ a . Exp' a t         -- (Exp t) does not contain Inverse

lam :: (forall a . Exp' a t1 -> Exp' a t2) -> Exp (t1 → t2)
lam f = In'1 (Lam f )                -- f can handle Inverse containg values
                                     -- but can never examine its content

app :: Exp (t1 → t2) → Exp t1 → Exp t2
app e1 e2  = In'1 (App e1 e2)
```

Evaluating Simply-Typed HOAS using msfit1 at kind $* \to *$

newtype Id a = MkId { unId :: a }

evalHOAS :: Exp t → Id t
evalHOAS e = msfit1 phi e   where
  phi :: Phi'1 ExpF Id   {- v :: t_1 -}   {- f :: r Id t_1 -> r Id t_2 -}
  phi inv ev (Lam f) = MkId(λv → unId(ev (f (inv (MkId v)))))
  phi inv ev (App e_1 e_2) = MkId(unId(ev e_1) (unId(ev e_2)))

inverse   recursive call

type Phi'1 f a = ∀r. (∀i. a i → r a i) → (∀i. r a i → a i) → (∀i. f (r a) i → a i)

This is example is a super awesome cooooool discovery that System Fw can express Simply-Typed HOAS evaluation!!!

The Nax language

- ✔ Built upon the idea of Mendler-style recursion schemes
- ✔ Supports an extension of Hindley--Milner type inference to make it easier to use Mendler-style recursion schemes and indexed datatypes
- ✔ Handling different notions of recursive type operators (Mu, Mu') still needs more rigorous clarification in the theory, but intuitively,

  Mu0 f = ∀a. Mu'0 f a
  Mu1 f = ∀a. Mu'1 f a i
  ...

  So, we hide Mu' from users as if there is one kind of Mu and Mu' is only used during the computation of msfit
- ✔ Supports **term-indexed types** as well as type-indexed types. Our term indexed calculus, System Fi (to appear in TLCA 2013), extends System Fw with erasable term indices.

Need for yet another Mendler-style recursion scheme

There are more recursion schemes other than **mit** and **msfit**

E.g., Mendler-style primitive recursion (**mpr**) can cast from abstract recursive type (r) to concrete recursive type (Mu f).

$$\text{Phi0 f a} = \forall r.\ (r \to \text{Mu f}) \to (r \to a) \to (\text{f r} \to a)$$

$$\text{Phi1 f a} = \forall r.\ (\forall i.\ r\ i \to \text{Mu f i}) \to (\forall i.\ r\ i \to a\ i) \to (\forall i.\ \text{f r i} \to a\ i)$$

With **mpr**, one can write constant time predecessor for natural numbers and tail function for lists by using the cast operation.

Next example motivates an extension to mpr that can uncast from concrete recursive type (Mu f i) to abstract recursive type (r i)

$$\text{Phi1 f a} = \forall r.\ (\forall i.\ \text{Mu f i} \text{ -> } r\ i) \to (\forall i.\ r\ i \to \text{Mu f i}) \to (\forall i.\ r\ i \to a\ i) \to (\forall i.\ \text{f r i} \to a\ i)$$

Recursion scheme with above Phi1 type does not terminate for mixed-variant datatypes -- needs some additional restriction on uncast.

Evaluating Simply-Typed HOAS into a user defined value domain

data V r t where V :: (r t_1 → r t_2) → V r (t_1 → t_2)
type Val t = Mu V t
val f = In1 (V f)

evalHOAS :: Exp t → Val t
evalHOAS e = msfit1 phi e   where
  phi :: Phi'1 ExpF (Mu1 V)   -- f :: r Val t_1 -> r Val t_2 ,   v :: Val t_1
  phi inv ev (Lam f) = val(λv → ev (f (inv v)))
  phi inv ev (App e_1 e_2) = unVal (ev e_1) (ev e_2)

Only if we had unVal :: Val (t_1 → t_2) → (Val t_1 → Val t_2) it would be possible to write this, but unVal does not seem to be definable using any known Mendler-style recursion scheme.

New recursion scheme to write unVal

```
data V r t where V :: (r t₁ → r t₂) → V r (t₁ → t₂)
type Val t = Mu V t
val f = In1 (V f )
unVal v = unId(mprsi phi v) where
                    phi :: Phi1 V Id
                    phi uncast cast (V f) = Id(λv. cast (f (uncast v)))

mprsi :: Phi1 f a → Mu f i → a i
mprsi phi (In1 x) = phi  id  id  (mrpsi phi)  x
-- size restriction over indices in both uncast and cast
type Phi1 f a =  ∀  r  j.    (∀ i. (i < j) ⇒ Mu f i → r i) →       -- uncast
                             (∀ i. (i < j) ⇒ r i → Mu f i) →       -- cast
                             (∀ i. r i → a i) → (r j → a j)
-- or, maybe without the size restriction over indices in cast
type Phi1 f a =  ∀  r  j.    (∀ i. (i < j) ⇒ Mu f i → r i) →       -- uncast
                             (∀ i.            r i → Mu f i) →      -- cast
                             (∀ i. r i → a i) → (f r j → a j)
```

Preliminary idea that still need further studies for the termination proof

Questions or Suggestions?

Thanks for listening.

===============< **Advertisement** >=============

Ki Yung Ahn <kya@cs.pdx.edu> is graduating soon this summer and openly looking for research positions worldwide.