A Hierarchy of Mendler style iteration/recursion combinators:

taming recursive types with negative occurrences

Ki Yung Ahn kya@cs.pdx.edu
Tim Sheard sheard@cs.pdx.edu

2011-11-04
Functional Programming Lab Seminar,
University of Nottingham

Nax language design: collaborating with
Andrew.Pitts@cl.cam.ac.uk
Marcelo.Fiore@cl.cam.ac.uk

Context of the work: **Trellys** Project

- Dependently typed language aiming for both a programming language and a reasoning system
- Able to logically reason about programs and compute over data structure containing proofs

Portland State University

Tim Sheard

Ki Yung Ahn   Nathan Collins

Penn University of Pennsylvania

Stephanie Weirich

Chris Casinghino   Vilhelm Sjöberg

The University of Iowa

Aaron Stump   Garrin Kimmell

Harley D. Eades III   Peng Fu

Characteristics of programming languages and reasoning systems

Typed Functional Programming Languages (e.g., ML, Haskell)

- Can define arbitrary recursive types definition
- No guarantee for normalization

Typed Logical Reasoning Systems (e.g., Coq, HOL)

- Can NOT define arbitrary recursive types (only positive ones)
- Guaranteed to normalize

Nax: a middle ground taking good properties of the both worlds

Typed Functional Programming Languages (e.g., ML, Haskell)

- Can define arbitrary recursive types definition
- No guarantee for normalization

Typed Logical Reasoning Systems (e.g., Coq, HOL)

- Can NOT define arbitrary recursive types (only positive ones)
- Guaranteed to normalize

Spoiler: Aren't there well known calculi with such properties?

Like System F and F_ω ...

Yes, Nax is going to be closely related to them.

Nax

- Named after Nax P. Mendler
- An extension of System F (or, F_ω)
- Allow formation of any recursive types (i.e. both positive and negative recursive types)
- Strongly normalizing
  - A rich family of principled iteration/recursion combinators over recursive types
  - These combinators were discovered and designed following the style of Mendler's

Why are we designing Nax?

in the context of Trellys project

- The goal of the Trellys project is to design and implement a unified system that is both
  - a full functional programming language
    Prog (programmatic sublanguage): $G \vdash_{\mathrm{Prog}} e : t$
  - and, a sound logical reasoning system
    Log (logical sublanguage): $G \vdash_{\mathrm{Log}} e : t$
- What are the minimal requirements of Log?

Nax $\left\{ \right.$

  - Normalizing (logical proof should be finite)
  - Support arbitrary Recursive Datatypes
    (to be able to refer to any program in Prog)

Outline

- Background & Motivation
- Preliminary Concepts
    - Recursive types
    - Mendler style iteration/recursion
- Current design of Nax
- Future work

Recursive Types (a.k.a. Fixpoint Types)

- Solutions for for recursive type equations
  - $X = 1 + X$ natural number
  - $X = 1 + (A \times X)$ list containing A element
  - $X = A + (X \times X)$ binary tree with A leaves
- That is, fixpoint μF such that $\mu F = F(\mu F)$
  - $N(X) = 1 + X$ natrual number type: μN
  - $L(X) = 1 + (A \times X)$ list type: μL
  - $T(X) = A + (X \times X)$ binary tree type: μT

Two styles of recursive types

- Equi-recursive (implicit conversion)

```
G |- e : μF
---------------- μ-elim
G |- e : F(μF)
```

```
G |- e : F(μF)
---------------- μ-intro
G |- e : μF
```

- Iso-recursive (explicit conversion)

```
G |-      e : μF
--------------------- μ-elim
G |- unIn e : F(μF)
```

```
G |- e    : F(μF)
------------------- μ-intro
G |- In e : μF
```

```
unIn (In e) → e
```

Two styles of recursive types

- Equi-recursive (implicit conversion)

```
type X = Either () X
data Either a b = Left a | Right b
```

This is only an analogy ... cyclic type synonym is a type error in Haskell

- Iso-recursive (explicit conversion)

```
data N r = Z | S r
type Nat = Mu N
zero     = In Z
succ n  = In (S n)
newtype Mu f = In (f (Mu f))   -- definition of μ
unIn (In x) = x          -- recall the reduction rule
```

Encoding of iso-recursive types a.k.a. two-level types

- Usual one-level recursive type definition Nat can be thought as an abstract interface (Nat, zero, succ) of the two-level implementation that hides more primitive constructs, that is, the recursion operator (Mu, In, out) and the base structure (N, Z, S)

```
data Nat = Zero | Succ Nat
```

```
data N r = Z | S r
type Nat = Mu N
zero     = In Z
succ n  = In (S n)
newtype Mu f = In (f (Mu f))   -- definition of μ
unIn (In x) = x         -- recall the reduction rule
```

Exercise on two-level types

- Natrual numbers

```
data Nat  =   Zero
          |   Succ Nat
```

```
data N = Z | S r
type Nat = Mu N
zero   = In N
succ n = In (S n)
```

- Lists

```
data List a =  Nil
            |  Cons a (List a)
```

```
data L a r = N | C a r
type List a = Mu (L a)
nil         = In N
cons x xs = In (C x xs)
```

- Trees

```
data Tree a
     =  Leaf a
     |  Node (Tree a) (Tree a)
```

```
data T a r = L a | N r r
type Tree a = Mu (T a)
leaf x      = In (L x)
node tl tr = In (N tl tr)
```

Recursive types and Normalization

- Unrestricted use of general recursion at term level can cause diverging computation
  - e.g. “`let f x = f x in f 0`” loops
- Unrestricted formation and elimination (i.e., pattern matching) over recursive types can also cause diverging computation, even without any use recursion at term level
  - With recursive types, it is possible to encode diverging self application $(\lambda x.xx)\ (\lambda x.xx)$ of the untyped lambda calculus in a type correct way
    - Also observed by Nax P. Mendler

Diverging computation just using Recursive types

- Mendler's example in Haskell: encoding of a classical self application $(\lambda x.xx)\ (\lambda x.xx)$

$$
\begin{array}{l|l}
\textbf{data}\ T = C\ (T \to ()) & \quad w\ (C\ w) \\
p :: T \to (T \to ()) & \leadsto (p\ (C\ w))\ (C\ w)) \\
p\ (C\ f) = f & \leadsto w\ (C\ w) \\
w :: T \to () & \leadsto (p\ (C\ w))\ (C\ w)) \\
w\ x = (p\ x)\ x & \leadsto \cdots
\end{array}
$$

- Ability to pattern match (eliminate) freely over recursive types is enough to cause divergence
  - didn't have to use term level recursion at all

Two design choices of normalization with recursive types

- Restrict Formation rule

$$\frac{G \mid\!- F : * \to *}{G \mid\!- \mu F : *}\ \mu\text{-form}$$

- Restrict Elimination rule

$$\frac{G \mid\!- e : \mu F}{G \mid\!- \mathrm{unIn}\ e : F(\mu F)}\ \mu\text{-elim}$$

$$\frac{G \mid\!- e : F(\mu F)}{G \mid\!- \mathrm{In}\ e : \mu F}\ \mu\text{-intro}$$

$$\mathrm{unIn}\ (\mathrm{In}\ e) \to e$$

Positive vs. Negative occurrences in recursive types

- Interpreting $(A \rightarrow B)$ logically as implication, which is equivalent to $(\neg A \bigvee B)$
- So, left of → is negative position and right of → is positive position
- Positive datatype: all recursive occurrences are in positive position
  data Tree = Leaf Int | InfBranch (Nat→Tree)
- Negative datatype: exist recursive occurrences in one or more negative positions
  data Exp = Lam (Exp→Exp) | App Exp Exp

Strictly Positive vs. Positive

- data $A = C\ ((A \to \mathrm{Bool}) \to \mathrm{Bool})$
  - Positive since A is in doubly negated position, but not strictly positive since A appears inside the left hand side of the top level $\to$
  - Considered to be non-set theoretic since it asserts the proposition that powerset of powerset of A being isomorphic to A, which is a set theoretic nonsense
- All strictly positive types have set theoretic interpretation
- Some positive, but not strictly positive, types CAN be considered set theoretically
  - data $SN = SN\ (\forall b.\ b \to (SN \to b) \to b)$
    Scott Numerals - an encoding of natural numbers

Diverging computation using Negative recursive types

- Mendler's example in Haskell: encoding of a classical self application $(\lambda x.xx)\ (\lambda x.xx)$

$$
\begin{array}{l|l}
\mathbf{data}\ T = C\ (T \to ()) & \quad w\ (C\ w) \\
p :: T \to (T \to ()) & \leadsto (p\ (C\ w))\ (C\ w)) \\
p\ (C\ f) = f & \leadsto w\ (C\ w) \\
w :: T \to () & \leadsto (p\ (C\ w))\ (C\ w)) \\
w\ x = (p\ x)\ x & \leadsto \cdots
\end{array}
$$

- Ability to pattern match (eliminate) freely over recursive types is enough to cause divergence
    - didn't have to use term level recursion at all

Why care about negative datatypes?
(Example 1: Reducibility)

- Definition of Reducibility for System T
  - Red{Nat}(M) iff M reduce to canonical form of Nat
  - Red{A→B}(M) iff for all N, Red{A}(N) implies Red{B}(M N)

- In proof assistants like Coq, this most natural definition will be rejected

```
Inductive Red: ty → exp → Prop
 := RedN : forall n, Const n → Red nat n
  | RedA : forall e A B, (forall A e', Red A e' → Red B (e e'))
                          → Red (A → B)
```

Why care about negative datatypes?
(Example 2: HOAS)

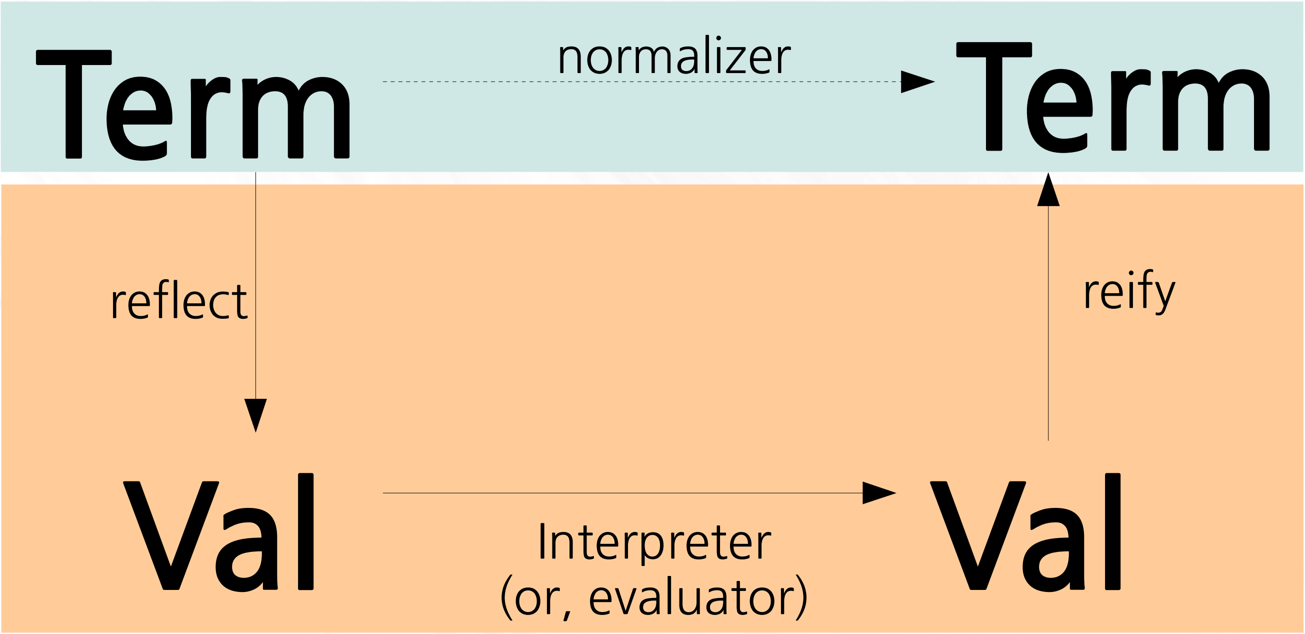
- HOAS for untyped lambda calculus (in Haskell)

  $\mathrm{data\ Exp} = \mathrm{Lam\ (Exp} \to \mathrm{Exp)} \mid \mathrm{App\ Exp\ Exp}$

  - Since Exp models the untyped lambda calculus, its eval function $\mathrm{eval} :: \mathrm{Exp} \to \mathrm{Exp}$ is partial
  - But, there can be many useful total functions over Exp, such as $\mathrm{showExp} :: \mathrm{Exp} \to \mathrm{String}$ that formats an HOAS term into a printable string
- More complex transformations using HOAS for typed languages have been studied in the context of type preserving compilers

Why care about negative datatypes?
(Example 3: Normalization by Evaluation)

- Define normalization of terms (positive datatype) using evaluation of values (negative datatype)



Outline

- Background & Motivation
- Preliminary Concepts
    - Recursive types (equi/iso, positive/negative)
    - Mendler style iteration/recursion
- Current design of Nax
- Future work

Two styles of iteration/recursion Squiggol style vs. Mendler style

- Developed in the context of Functional Programming Languages with Type Inference (Hindley-Milner type system)

  No wonder why this style has been more popular in functional programming

- Iteration/Recursion well-defined for positive datatypes, but not for negative datatypes
- Defined for regular datatypes, but not easy to generalize to non-regular datatypes (e.g. nested datatypes, GADTs)

- Developed in the context of Nuprl, an interactive theorem prover for constructive math, with a very powerful type system (dependent types, higher rank polymorphism)
- Iteration/Recursion well-defined for arbitrary datatypes including negative datatypes

  Mendler didn't notice this himself, later discovered by others

- Naturally generalize to non-regular datatypes (more generally to type constructors of arbitrary higher kinds)

Exercise on two-level types

- Natrual numbers

```
data Nat  =   Zero
          |   Succ Nat
```

```
data N = Z | S r
type Nat = Mu N
zero    = In N
succ n = In (S n)
```

- Lists

```
data List a =  Nil
            |  Cons a (List a)
```

```
data L a r = N | C a r
type List a = Mu (L a)
nil          = In N
cons x xs = In (C x xs)
```

- Trees

```
data Tree a
     =  Leaf a
     |  Node (Tree a) (Tree a)
```

```
data T a r = L a | N r r
type Tree a = Mu (T a)
leaf x       = In (L x)
node tl tr = In (N tl tr)
```

Iteration (a.k.a. catamorhpism) in Squiggol style

```
iter :: Functor f ⇒
        (f a → a) → Mu f → a
iter φ (In x) =
  φ (fmap (iter φ) x)
```

```
instance Functor (L x) where
  -- fmap :: (a→ b) → L x a → L x b
  fmap h N       = N
  fmap h (C x a) = C x (h a)
```

```
lenList = Mu (L x) → Int
lenList = iter phi
  where
    phi :: L x Int → Int
    phi N            = 0
    phi (C x xslen) = 1 + xslen
```

- Generalization of folds, expressed using 2-level types
- Recursion is captured by `iter` at term level, and `Mu` at type level (non-recursive elsewhere)
- `fmap` guides where to invoke recursive calls
- User supplies `phi :: f a → a`, which defines how to process the base structure containing answers of the already processed subcomponents

Iteration (a.k.a. catamorhpism) in Mendler style

```
miter :: (∀r.(r → a) → f r → a) →
          Mu f → a
miter φ (In x) =
  φ (miter φ) x
```

```
lenList = Mu (L x) → Int
lenList = miter phi
  where
   phi :: ∀r.(r→Int) → L x r → Int
   phi len N        = 0
   phi len (C x xs) = 1 + len xs
```

- Key idea: φ expects another argument, which enables user to control recursive calls instead of relying on predefined fmap
- No requirement on the base (f) being a positive functor
- Higher rank polymorphism (∀r. ...) enforce recursive subcomponents in the base structure (f r) be abstract inside φ, which is Mendler's ingenious idea to guarantee normalization
- phi looks almost the same as what you'd write in a functional language with general recursion

Does miter really normalize?

Isn't it dangerous to allow users to control recursive calls?

```
miter :: (     (Mu f → a) → f (Mu f) → a) → Mu f → a   -- Naïve type
miter :: (∀r. ( r      → a) → f  r       → a) → Mu f → a   -- Mendler's type
```

```
cons :: x → Mu (L x) → Mu (L x)

lenListBad = mcata phi
  where
   phi len N          = 0
   phi len (C x xs) = 1 + len (cons x xs)
```

The function lenListBad diverges when applied to a non-empty list, since it recursively calls on the same list (cons x xs) rather than its tail (xs).

- lenListBad will type check with the naïve type of miter
- len (cons x xs) is a type error with Mendler's type
  - cons expects its 2nd arg of type Mu (L x) but xs :: r, where r is parametric (or, abstract)
  - len :: (r→a) expects an arg of abstract type r but the result type of cons is Mu (L x)

can't cons with xs

won't work even if you could cons

Does miter really normalize?

- Yes, we've seen the intuition
- The formal proof can be done by a reduction preserving embedding into a well known normalizing language Fω (i.e., one reduction step involving miter is simulated by one or more constantly bound reduction steps in Fω)

```
newtype Mu f = In (f (Mu f))

mcata :: (∀r.(r → a) → f r → a) → Mu f
→ a
mcata φ (In x) = φ (mcata φ) x

{- lambda abstraction, application, and
   non recursive base structures have
   trivial embeddings into Fw -}
```

Fω

Primitive Recursion vs. Iteration

- Primitive Recursion gives you access to both the values of the subcomponents and the results of processing the subcomponents
- Iteration only gives you access to the results of processing the subcomponents
- Although miter looks like giving you access to the the subcomponent values, it really isn't. (try to define factorial if you are in doubt)

```
lenList = Mu (L x) → Int
lenList = miter phi
  where
   phi :: ∀r.(r→Int) → L x Int →
Int
   phi len N          = 0
   phi len (C x xs) = 1 + len xs
```

xs :: r is an abstract handle to the tail whose only sensible use is to be applied to the abstract interface to the recursive call

len :: r → Int

Mendler style Primitive Recursion

$$\text{miter} :: (\forall r. (r \to a) \to f\ r \to a) \to \text{Mu}\ f \to a$$
$$\text{miter}\ \varphi\ (\text{In}\ x) = \varphi\ (\text{miter}\ \varphi)\ x$$

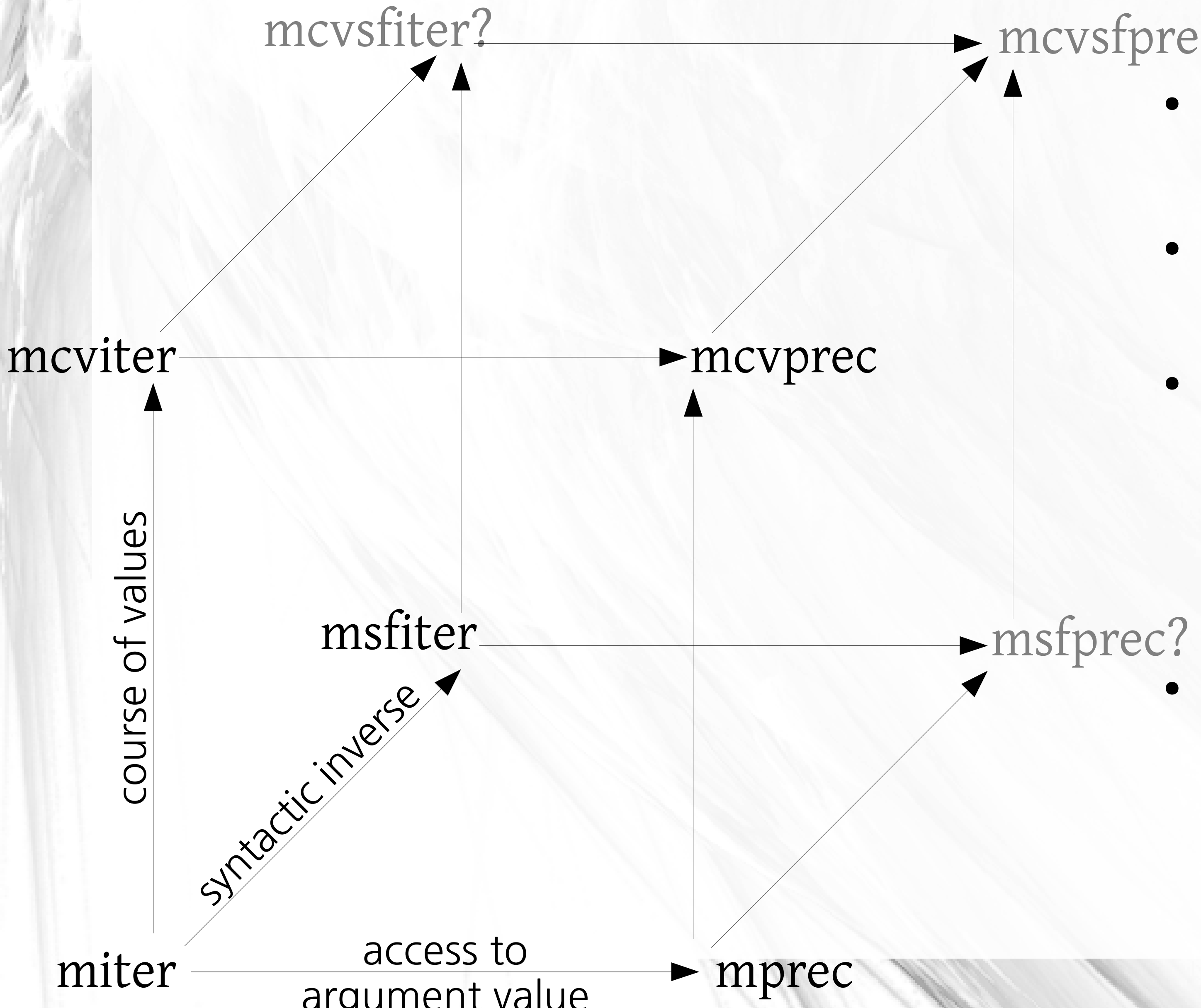
$$\text{mprec} :: (\forall r. (r \to \text{Mu}\ f) \to (r \to a) \to f\ r \to a) \to \text{Mu}\ f \to a$$
$$\text{mprec}\ \varphi\ (\text{In}\ x) = \varphi\ \text{id}\ (\text{mprec}\ \varphi)\ x$$

$$(\times) :: \text{Mu}\ N \to \text{Mu}\ N \to \text{Nat}$$

$$\text{fact} = \text{Mu}\ N \to \text{Nat}$$
$$\begin{aligned}\text{fact} = \text{mprec phi} \quad \text{where}\ & \text{phi} :: \forall r. (r \to \text{Mu}\ N) \to (r \to \text{Nat}) \to N\ r \to \text{Nat} \\ & \text{phi cast fac}\ Z = 0 \\ & \text{phi cast fac}\ (S\ n) = \text{succ}(\text{cast}\ n) \times \text{fac}\ n\end{aligned}$$

- φ for mprec expects yet another argument, which is a type casting function from an abstract type (r) to the concrete recursive type ($\text{Mu}\ f$)
- Mendler's original work (LICS '87) is about mprec

A Hierarchy of Mendler style Iteration/Recursion Combinators



- miter, msfiter can be embeded into F_ω
- mprec embeds into Fix_ω (Abel & Matthes CSL '04)
- mcv- combinators only normalize for positive datatypes (other non-cv combinators normalize for arbitrary datatypes)
- Naturally extends to higher kinds where Mu and related combinators are index by kinds
  Mu_{*} , In_{*} , miter_{*} , ...
  $\mathrm{Mu}_{*\to*}$, $\mathrm{In}_{*\to*}$, $\mathrm{miter}_{*\to*}$, ..

Mendler style course of values Iteration

```
miter   :: (∀r.           (r → a) → f r → a) → Mu f → a
miter   φ (In x) = φ      (miter φ)   x
```

```
mcviter :: (∀r. (r → f r) → (r → a) → f r → a) → Mu f → a
mcviter φ (In x) = φ unIn (mcviter φ) x
```

```
(+) :: Mu N → Mu N → Nat

fibo = Mu N → Nat
fibo = mcviter phi where phi :: ∀r.(r→N r) → (r→Nat) → N r → Nat
                         phi  out  fib Z     = succ zero
                         phi  out  fib (S n) = case out n of  Z    → succ zero
                                                              S n' → fib n + fib n'
```

- φ for mcviter expects yet another argument, which is an abstract eliminator out :: r → f r passing around unIn :: Mu f → f (Mu f), giving the ability to abstractly eliminate (i.e., pattern match away) In constructor of Mu

only normalizes for positive datatype.
A counter example for negative datatype in our ICFP'11 paper.

Mendler style Sheard-Fegaras Iteration

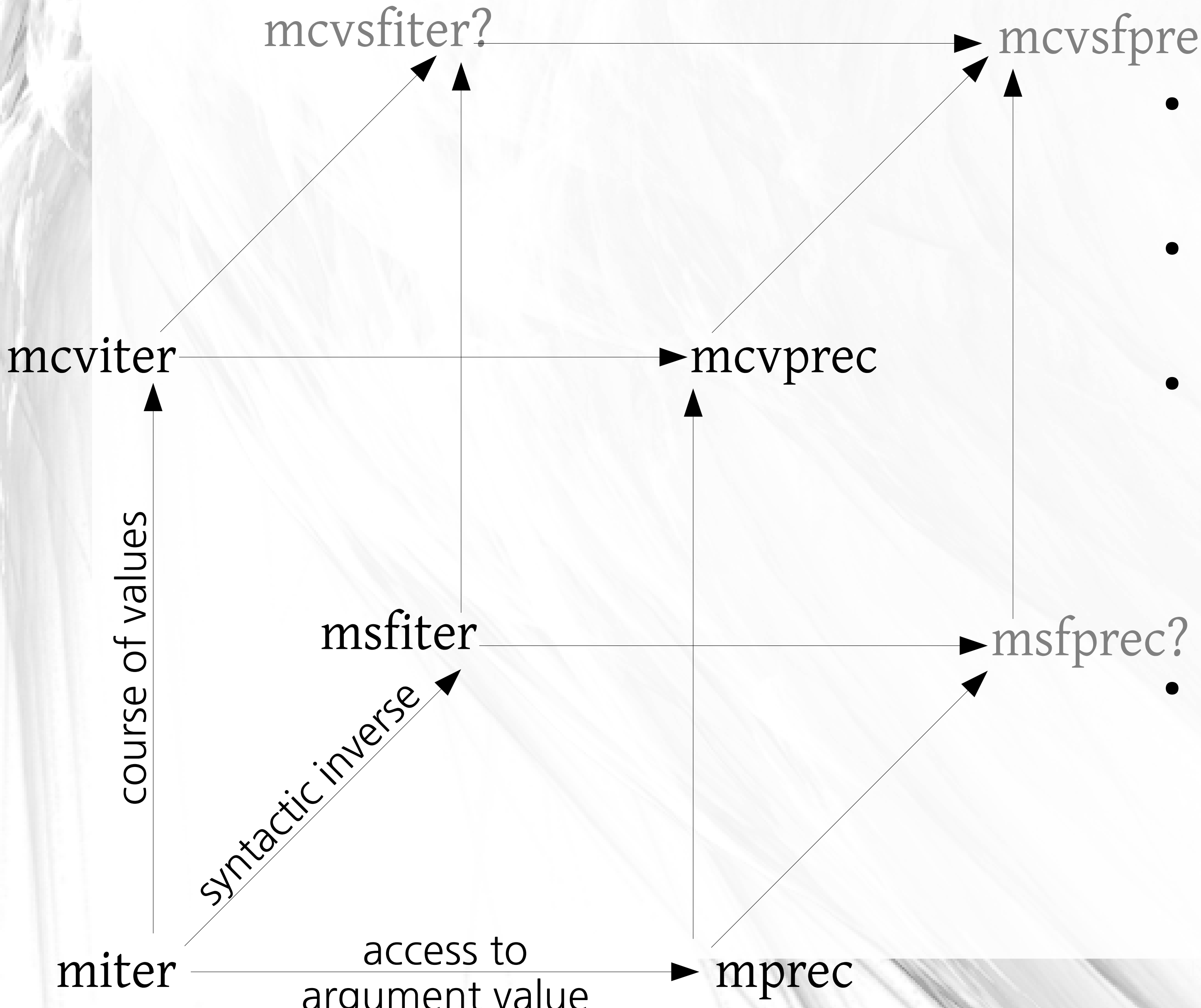
```
data Mu' f a = In' (f (Mu' f a)) | Inverse a  -- Mu with syntactic inverse

msfiter :: (∀r. (a → r a) → (r a → a) → f (r a) → a) → (∀a. Mu' f a) → a
msfiter φ (In' x)       = φ Inverse (msfiter φ) x
msfiter φ (Inverse x) = x
```

```
data E r = A r r | L (r → r)      -- base structure for HOAS
type Exp = ∀a. Mu' E a
countVar :: Exp → Int  -- count the no. of variable use. (λx.xx) is 2, (λx.λy.x) is 1
countVar = msfiter phi  where  phi :: ∀r.(Int→r Int)→(r Int→Int)→E (r Int)→Int
                               phi  inv  count (L g)        = count (g (inv 1))
                               phi  inv  count (A e1 e2) = count e1 + count e2
```

- φ for msfiter expects yet another argument, which is a syntactic Inverse allowing you to instantly create an abstract recursive value (inv 1 :: r Int) from an expected result value (1 :: Int) so that you can supply it to a function (g:: r Int → r Int) expecting an abstract recursive value

See our ICFP'11 paper for further details.

A Hierarchy of Mendler style Iteration/Recursion Combinators



- miter, msfiter can be embeded into F_ω
- mprec embeds into Fix_ω (Abel & Matthes CSL '04)
- mcv- combinators only normalize for positive datatypes (other non-cv combinators normalize for arbitrary datatypes)
- Naturally extends to higher kinds where Mu and related combinators are index by kinds
  Mu_*, In_*, miter_*, ...
  $\mathrm{Mu}_{*\to*}$, $\mathrm{In}_{*\to*}$, $\mathrm{miter}_{*\to*}$, ..

Outline

- Background & Motivation
- Preliminary Concepts
  - Recursive types (equi/iso, positive/negative)
  - Mendler style iteration/recursion
    - Well-defined for negative datatypes
    - Naturally generalize to non-regular datatypes
    - Variations of iteration/recursion schemes (course of values, syntactic inverse) have been discovered and studied
- Current design of Nax
- Future work

Some trivia

Why design a new language when you can embed that new language into F_ω or Fix_ω? Why not just use F_ω or Fix_ω?

Same reason you don't want to use Turing machine or plain lambda calculus instead of programming in high level languages

- Embedding into F_ω or Fix_ω is only a tool for showing normalization
- Encoding datatypes in F_ω or Fix_ω is tedious
- Some language design decisions can make type inference/checking more convenient
- Some recursion combinators can be simplified when we define them as language constructs

Mendler style Sheard-Fegaras Iteration

```
data Mu' f a = In' (f (Mu' f a)) | Inverse a  -- Mu with syntactic inverse

msfiter :: (∀r. (a → r a) → (r a → a) → f (r a) → a) → (∀a. Mu' f a) → a
msfiter φ (In' x)        = φ Inverse (msfiter φ) x
msfiter φ (Inverse x) = x
```

Instead of above implementation in Haskell, we can define msfiter as a Nax language primitive of the following type using one same Mu, and reduction rules defined as follows:

```
msfiter :: (∀r. (a → r) → (r → a) → f r → a) → Mu f → a

msfiter φ (In' x)        → φ Inverse (msfiter φ) x
msfiter φ (Inverse x) → x
```

Inverse is a transient term, which only appear during computation but cannot appear in the source code

Syntax:

Curry style System F with some extensions

$$Dec ::= F\,\overline{X}\,X.\{\,\overline{C\,\overline{T}}\,\}$$

$$Decs ::= \cdot \mid Dec; Decs$$

$$T ::= F\,\overline{T}\,T \mid T \to T \mid X \mid \forall X.T \mid \mu(F\,\overline{T})$$

$$\begin{aligned} M ::=\ & x \mid C \mid \mathsf{case}\ M\ \{\,\overline{C\ \overline{x}.M}\,\} \mid \lambda x.M \mid M\,M \\ & \mid \mathsf{in}\ M \mid \mathsf{mit}\ M \mid \mathsf{mrec}\ M \mid \mathsf{mcvit}\ M \mid \mathsf{mcvrec}\ M \mid \mathsf{msfit}\ M \\ & \mid \mathsf{out} \mid \mathsf{inv} \quad \text{-- these are transient objects cannot apppear in source code} \end{aligned}$$

$$Program ::= Decs; M$$

- This description is still at a level of a calculus
- More concrete syntax is being designed by trying out a prototype implementation

Type System

$$\text{in}\ \frac{\Gamma \vdash M : F\,\overline{T}\,(\mu(F\,\overline{T}))}{\Gamma \vdash \mathsf{in}\,M : \mu(F\,\overline{T})} \qquad \text{mit}\ \frac{\Gamma, X : \star \vdash M : (X \to T') \to F\,\overline{T}\,X \to T'}{\Gamma \vdash \mathsf{mit}\,M : \mu(F\,\overline{T}) \to T'}$$

$$\text{mrec}\ \frac{\Gamma, X : \star \vdash M : (X \to \mu(F\,\overline{T})) \to (X \to T') \to F\,\overline{T}\,X \to T'}{\Gamma \vdash \mathsf{mrec}\,M : \mu(F\,\overline{T}) \to T'}$$

$$\text{mcvit}\ \frac{\begin{array}{c}\Gamma, X : \star, \vdash M : (X \to F\,\overline{T}\,X) \to (X \to T') \to F\,\overline{T}\,X \to T' \\ F \text{ is positive}\end{array}}{\Gamma \vdash \mathsf{mcvit}\,M : \mu(F\,\overline{T}) \to T'}$$

$$\text{mcvrec}\ \frac{\begin{array}{c}\Gamma, X : \star, \vdash M : (X \to F\,\overline{T}\,X) \to (X \to F\,\overline{T}\,X) \to (X \to T') \to F\,\overline{T}\,X \to T' \\ F \text{ is positive}\end{array}}{\Gamma \vdash \mathsf{mcvrec}\,M : \mu(F\,\overline{T}) \to T'}$$

$$\text{msfit}\ \frac{\Gamma, X : \star \vdash M : (T' \to X) \to (X \to T') \to F\,\overline{T}\,X \to T'}{\Gamma \vdash \mathsf{msfit}\,M : \mu(F\,\overline{T}) \to T'}$$

Reduction

$$
\begin{aligned}
(\lambda x.M)M' &\longrightarrow M[M'/x] \\
\mathsf{mit}\,M\,(\mathsf{in}\,M') &\longrightarrow M\,(\mathsf{mit}\,M)\,M' \\
\mathsf{mrec}\,M\,(\mathsf{in}\,M') &\longrightarrow M\,(\lambda x.x)\,(\mathsf{mrec}\,M)\,M' \\
\mathsf{mcvit}\,M\,(\mathsf{in}\,M') &\longrightarrow M\,\mathsf{out}\,(\mathsf{mcvit}\,M)\,M' \\
\mathsf{mcvrec}\,M\,(\mathsf{in}\,M') &\longrightarrow M\,(\lambda x.x)\,\mathsf{out}\,(\mathsf{mcvrec}\,M)\,M' \\
\mathsf{out}\,(\mathsf{in}\,M') &\longrightarrow M' \\
\mathsf{msfit}\,M\,(\mathsf{in}\,M') &\longrightarrow M\,\mathsf{inv}\,(\mathsf{msfit}\,M)\,M' \\
\mathsf{msfit}\,M\,(\mathsf{inv}\,M') &\longrightarrow M'
\end{aligned}
$$

$$
\frac{M \longrightarrow M'}{E[M] \longrightarrow E[M']}
$$

$E ::= \ldots$

Outline

- Background & Motivation
- Preliminary Concepts
    - Recursive types (equi/iso, positive/negative)
    - Mendler style iteration/recursion
- Current design of Nax
- Future work

Future Work

- Try to write more interesting examples involving negative datatypes using msfiter
  (e.g., Normalization by Evaluation for a simple calculus)
- Extend the language from F rather than F_ω
  (non-regular datatypes, or datatypes of higher kinds)
- Add indexed types (in spirit of GADTs) and user defined kinds lifted from user defined types
- Concrete syntax and type checking/inference

- Dependent types and Induction principle?
  (i.e., dependent version of recursion combinators)
- Explore more iteration/recursion combinators

Current status of Nax

```
data Tag = E | O -- values of 1st order type can be lifted to index

flip E = O
flip O = E

gadt P : (Tag -> Nat -> *) -> Tag -> Nat -> * where
  Base : P r {E} {zero}
  StepO : r {O} {i} -> P r {E} {succ i}
  StepE : r {E} {i} -> P r {O} {succ i}

type Proof t n = Mu (Tag -> Nat -> *) P t n

type Even n = Proof {E} n
base = In (Tag -> Nat -> *) Base
stepO x = In (Tag -> Nat -> *) (StepO x)

type Odd n  = Proof {O} n
stepE x = In (Tag -> Nat -> *) (StepE x)

-- stepProof : Proof {t} {i} -> Proof {flip t} {succ i}
stepProof pf = miter {t i . Proof {flip t} {succ i}} pf
                 where   phi  Base       = stepE base
                         phi  (StepO p) = stepE(phi p)
                         phi  (StepE p) = stepO(phi p)

-- evenORodd : Vec a {n} -> Either (Even {n}) (Odd {n})
```

Mendler style Induction for positive datatypes

$$\text{mprec} :: (\forall r.\ (r \to \text{Mu } f) \to (r \to a) \to f\ r \to a) \to \text{Mu } f \to a$$
$$\text{mprec}\ \varphi\ (\text{In } x) \quad = \varphi\ \text{id}\ \ (\text{mprec}\ \varphi)\ x$$

$$\text{mind} :: (\ \forall r.\ (\text{cast}: r \to \text{Mu } f) \to ((x{:}r) \to a\ (\text{cast } x))$$
$$\to (y{:}f\ r) \to a\ (\text{fmap cast } y)\ ) \to (z{:}\text{Mu } f) \to a\ z$$
$$\text{mind}\ \varphi\ (\text{In } x) = \varphi\ \text{id}\ \ (\text{mprec}\ \varphi)\ x$$

- Just as the conventional style, we can define induction as a dependent version of primitive recursion on positive datatypes (note the use of fmap)
- We don't know yet how to formulate induction for negative datatypes

Conclusion

- We want a core language with both normalization and arbitrary recursive types
- We know that this is possible by discovering a a new family of Mendler style iteration combinator $\mathrm{msfiter}$
- We are designing Nax to realize this idea