Mendler style Recursion Combinators in Dependently Typed Languages

Ki Yung Ahn

Thesis Proposal
2011-06-09

Thesis

A type based approach (in particular, the Mendler-style approach), in contrast to the value based approach (structural recursion), to ensure termination in dependently typed functional languages is both possible and practical.

It broadens the range of certified programming practices and provides an intuitive abstraction for expressing terminating recursion that generalizes naturally over many kinds of inductive datatypes.

Problem Setting

- Combining functional languages and proof systems
  - a promising approach for certified programming
- Termination is essential for logical consistency (since proofs must be finite)
  - if it was easier to prove more programs terminating, it would be easier to prove more properties
- The most widely used method for ensuring termination only works on certain kinds of programs
  - <u>need a more general and flexible method</u>

The most widely used method: Structural Recursion

- Value based approach
  - recurse on structurally smaller arguments

    (1) $xs < Cons\ x\ xs$, $x < (x,y)$

    (2) $f\ a < f$

    $$f = \{(a, f\ a), (b, f\ b), \ldots\}$$

    $$f\ a < (a, f\ a) \in f$$

details will be discussed later

- Ensures termination for only positive datatypes
  - including non-regular types & dependent types
  - NOT including negative datatypes

Where structural recursion falls short

- negative datatypes
  $data\ T = C\ (T \rightarrow ())$

- datatypes with impredicative polymorphism
  $data\ T = C\ (\forall a.\ a \rightarrow a)$
  it is possible to instantiate *a* with *T*

Why Mendler-style?

- Ensures termination of programs over wide range of datatypes
  - non-regular datatypes
  - negative datatypes
  - as well as positive & regular datatypes

- Studied in the context of Fω (higher-order polymorphic lambda calculus)
  - not studied in the context of dependent types
  - extension to dependent types is a research goal

Research Goals

1. Demonstrate Mendler style combinators are expressive, easy to use, and provide termination guarantees
2. Organize the Hierarchy of the Mendler style recursion combinators
3. Extend Mendler style recursion combinators to dependent types
4. Identify which features found in dependently typed languages
   - interact smoothly with or
   - conflict with the Mendler style approach

Research Methods:
Goal 1. expressivity and ease of use

- Wrote examples in Haskell
  - ease of use:
    code size is smaller than Squiggol style approach
    easier to understand
  - expressiveness:
    examples on wide variety of datatypes
- I will produce examples demonstrating the use of recursion combinators by writing them in a dependently typed language
- I will create larger and more sophisticated examples

Research Methods: Goal 2. hierarchical organization

- Organize combinators by
  - their termination behavior
  - kinds of datatypes they work on
  - the class of functions they can define

- Observe general principles
  - more expressive family of combinators have additional arguments to their combining function type
  - less expressive family of combinators can be implemented in terms of more expressive ones

- Validate the principles by deriving new combinators

Research Methods:
Goal 3. extension to dependent types

- Use a dependently typed language system
  - as a tool for writing examples
  - as a tool for constructing generic proofs of termination
  - as a framework for constructing metatheoretical properties
    - the language system must be sound
    - and logically consistent
- Either find a suitable dependently typed language or I will have to develop/extend one myself (Or both). Possible choices are
  - Coq
  - Trellys

Research Methods:
Goal 4. interaction with language features

- New features may be required to define the recursion combinators for dependent types
- Some features might affect termination behavior of the recursion combinators
- Some features might make it harder to use the recursion combinators while not affecting their termination behavior
- I expect the list of features will become evident while I creating examples and constructing proofs

Necessary concepts

- Datatypes
- Dependent types
- Methods for ensuring terminating recursion
  - Structural Recursion (value-based)
  - Recursion Combinators (type-based)
    - Squiggol style (or, conventional)
    - Mendler style

Datatypes - quick tour I

categorized by pattern of
- recursive occurrences
- result types of data constructors

- Regular datatypes
  data List a = Nil | Cons a (List a)

- Non-regular datatypes
  - nested datatypes
    data Powl a = NilP | ConsP a (Powl (a,a))
    data Bush a = NilB | ConsB a (Bush (Bush a))
  - indexed datatypes (GADTs)

Datatypes - quick tour I

categorized by pattern of
- recursive occurrences
- result types of data constructors

- Regular datatypes

```
data List a where
  Nil  :: List a
  Cons :: a → List a → List a
```

- Non-regular datatypes
  - nested datatypes

```
data Bush a where
  NilB  :: Bush a
  ConsB :: a→Bush (Bush a)→Bush a
```

  - indexed datatypes (GADTs)

```
data Vec a n where
  Nil  :: Vec a Z
  Cons :: a → Vec a n→ Vec a (S n)
```

Datatypes - quick tour 2

categorized by positivity

- Positive datatypes

  all recursive occurrences appear in positive position

  data T = C (Int→T)

- Negative datatypes

  some recursive occurrences appear in negative position

  data T = C (T→Int)

  data T = C (T→T)

  data Exp = Lam (Exp→Exp) | App Exp Exp

Negative Datatypes

can cause non-termination

- observed by Mendler

$$\mathbf{data}\ T = C\ (T \to ())$$

$$p :: T \to (T \to ())$$
$$p\ (C\ f) = f$$

$$w :: T \to ()$$
$$w\ x = (p\ x)\ x$$

$$\begin{aligned} & w\ (C\ w) \\ \leadsto\ & (p\ (C\ w))\ (C\ w)) \\ \leadsto\ & w\ (C\ w) \\ \leadsto\ & (p\ (C\ w))\ (C\ w)) \\ \leadsto\ & \cdots \end{aligned}$$

Necessary concepts

- Datatypes
- Dependent types
- Methods for ensuring terminating recursion
  - Structural Recursion (value-based)
  - Recursion Combinators (type-based)
    - Squiggol style (or, conventional)
    - Mendler style

Dependent Types

- Parametric Polymorphism - types depend on types
  data List (p:Type) where $Nil :: List\ p$
  $Cons :: p \rightarrow List\ p \rightarrow List\ p$
- Ad-hoc Polymorphism - values depend on types
  e.g., $min :: Ord\ a \Rightarrow a \rightarrow a \rightarrow Bool$ in Haskell
- True (value) dependency - <u>types depend on values</u>
  e.g., *data Even (n:Nat) where*
  $EvenZ : Even\ Z$
  $EvenS : Odd\ k \rightarrow Even\ (S\ k)$

can express detailed properties about values!

Dependent Types

- True (value) dependency - <u>value depending on types</u>

```
data Either a b = Left a | Right b
data Nat = Zero | Succ Nat

data Even (n:Nat) where
   EvenZ : Even Zero
   EvenS : Odd k → Even (Succ k)
data Odd (n:Nat) where
   OddS : Even k → Odd (Succ k)

evenOrOdd : (n:Nat) → Either (Even n) (Odd n)
evenOrOdd n = case n of ...
```

functions can have dependent types too

Necessary Concepts

- Datatypes
- Dependent types
- Methods for ensuring terminating recursion
  - Structural Recursion (value-based)
  - Recursion Combinators (type-based)
    - Squiggol style (or, conventional)
    - Mendler style

Termination methods preview

- Structural Recursion (value-based)
  pros - untyped axioms can apply to most all datatype (nested, indexed, dependent)
  cons - does not apply to negative datatypes, little reuse of termination proofs

- Recursion Combinators (type-based)
  - Squiggol (conventional) style
    Functional languages with Hindley-Milner type system
    Works for positive regular datatypes
    non-trivial to generalize further
    more restrictive than structural recursion
  - Mendler style
    NuPRL - a dependently typed interactive theorem prover
    Some combinators work for ANY datatype in Fω
    Not yet been studied in dependent types

Structural Recursion

- Value based approach
  - recurse on structurally smaller arguments

    xs < *Cons x xs*

  length Nil = *0*
  length (Cons x xs) = *1* + *length xs*

- Ensures termination for only positive datatypes
  - including non-regular types & dependent types

Structural recursion falls short on negative datatypes

data Exp = Num Int | Lam (Exp→Exp) | App Exp Exp

The following function *countLam* terminates but structural recursion can't prove its termination

countLam :: Exp → Int
countLam (Num n) = n
countLam (Lam f) = countLam (Num 1) -- not structurally smaller
countLam (App e e') = *countLam* e + *countLam* e'

Necessary Concepts

- Datatypes
- Dependent types
- Methods for ensuring terminating recursion
  - Structural Recursion (value-based)
  - Recursion Combinators (type-based)
    - Squiggol (conventional) style
    - Mendler style

Conventional Catamorphism

generalization of folds

```
data List p = Nil | Cons p List p
foldList :: a → (p→a→a) → List p → a
foldList v f Nil            = v
foldList v f (Cons x xs) = f x (foldList f v xs)
```

Type of fold grows with the number and arity of the data constructors

```
data Tree p = Leaf p | Node (Tree p) (Tree p)
foldTree :: (p→a) → (a→a→a) → Tree p → a
foldTree fL fN (Leaf x)       = fL x
foldTree fL fN (Node tl tr) = fN (foldTree fL fN tl) (foldTree fL fN tr)
```

```
data Expr = VAL Int | BOP Op Expr Expr | IF Expr Expr Expr | ...
foldExpr :: (Int→a)→(Op→a→a→a)→(a→a→a→a)→...→Exp→a
```

Two level types

newtype μ0 *f = In0 (f (*μ0 *f))* -- datatype fixpoint operator

data List p = Nil | Cons p List p

data L p r = N | C p r -- base datatype
type List p = μ0 *(L p)* -- the fixpoint of the base
-- *Nil = In0 N , Cons x xs = In0 (C x xs)*

data Tree p = Leaf p | Node (Tree p) (Tree p)

data T p r = Lf p | Nd r r -- base datatype
type Tree p = μ0 *(T p)* -- the fixpoint of the base
-- *Leaf x = In0 (Lf x) , Node tl tr = In0 (Nd tl tr)*

Conventional Catamorphism

generalization of folds

$\mathit{data}\ L\ p\ r = N \mid C\ p\ r$

$\mathit{type}\ \mathit{List}\ p = \mu 0\ (L\ p)$

$\mathit{foldList} :: \underline{(L\ p\ a \to a)} \to \mathit{List}\ p \to a$

$\mathit{foldList}\ \varphi\ \mathit{Nil} \qquad\qquad = \varphi\ N$

$\mathit{foldList}\ \varphi\ (\mathit{Cons}\ x\ xs) = \varphi\ (C\ x\ (\mathit{foldList}\ \varphi\ xs))$

Two level types ameliorate the problem of verbose types of folds

$\mathit{data}\ T\ p\ r = \mathit{Lf}\ p \mid \mathit{Nd}\ r\ r$

$\mathit{type}\ \mathit{Tree}\ p = \mu 0\ (T\ p)$

$\mathit{foldTree} :: \underline{(T\ p\ a \to a)} \to \mathit{Tree}\ p \to a$

$\mathit{foldTree}\ \varphi\ (\mathit{Leaf}\ x) \qquad = \varphi\ (\mathit{Lf}\ x)$

$\mathit{foldTree}\ \varphi\ (\mathit{Node}\ tl\ tr) = \varphi\ (\mathit{Nd}\ (\mathit{foldTree}\ \varphi\ tl)\ (\mathit{foldTree}\ \varphi\ tr))$

Conventional Catamorphism

generalization of folds

```
data L p r = N | C p r
type List p = μ0 (L p)
foldList :: (L p a→a) → List p → a
foldList φ (In0 N)          = φ N
foldList φ (In0 (C x xs)) = φ (C x (foldList φ xs))

data T p r = Lf p | Nd r r
type Tree p = μ0 (T p)
foldTree :: (T p a→a) → Tree p → a
foldTree φ (In0 (Lf x))       = φ (Lf x)
foldTree φ (In0 (Nd tl tr)) = φ (Nd (foldTree φ tl) (foldTree φ tr))
```

Data constructors also encoded in two level

Conventional Catamorphism

generalization of folds

data L p r = N | C p r
type List p = μ0 (L p)

foldList :: (L p a→a) → List p → a
foldList φ (In0 N) = φ N
foldList φ (In0 (C x xs)) = φ (C x (foldList φ xs))

len :: List p → Int
len = foldList φ where
φ N = 0
φ (C _ ans) = 1 + ans

Intuition for termination:

One *In0* is removed each time *foldList* gets invoked

Side by side comparison

```
data L p r = N | C p r
type List p = μ0 (L p)

len :: List p → Int
len = foldList φ where
  φ N         = 0
  φ (C _ ans) = 1 + ans
-- conventional version
```

```
data L p r = N | C p r
type List p = μ0 (L p)

mcata0 :: (∀r. (r→a)→f r→a)→μ0 f→a
mcata0 φ (In0 x) = φ (mcata0 φ) x

len :: List p → Int
len = mcata0 φ where
  φ len' N         = 0
  φ len' (C _ xs) = 1 + len' xs
-- Mendler style version
```

```
-- general recursive version
len :: List p → Int
len Nil          = 0
len (Cons _ xs) = 1 + len xs
```

Side by side comparison 2

```
data T p r = Lf | Nd p r
type Tree p = μ0 (T p)

mcata0 :: (∀r. (r→a)→f r→a)→μ0 f→a
mcata0 φ (In0 x) = φ (mcata0 φ) x
-- tree flattening function
flat :: Tree p → [p]
flat = mcata0 φ where
  φ flat’ (Lf x)       = [x]
  φ flat’ (Nd tl tr) = flat’ tl ++ flat’ tr
-- Mendler style version
```

```
-- general recursive version
flat :: Tree p → [p]
flat (Leaf x)       = [x]
flat (Node tr tl) = flat tl ++ flat tr
```

Mendler style summary

- Does not assume anything about datatypes
  - works for any datatype
- Definition of φ is syntactically similar to the general recursive version
- Since *r* is abstract, the abstract recursive caller *(len' :: r→a)* can only be applied to *(xs :: r)*
- Seamlessly generalize to datatypes of higher-kinds

```
data L p r = N | C p r
type List p = μ0 (L p)

mcata0 :: (∀r. (r→a)→f r→a)→μ0 f→a
mcata0 φ (In0 x) = φ (mcata0 φ) x
-- list length function
len :: List p → Int
len = mcata0 φ where
  φ len' N        = 0
  φ len' (C _ xs) = 1 + len' xs
-- Mendler style version
```

```
-- general recursive version
len :: List p → Int
len Nil           = 0
len (Cons _ xs) = 1 + len xs
```

Some Preliminary Work

- Negative datatypes:
  Formulated Fegaras-Sheard Mendler style catamorphism
  - is more expressive over negative datatypes than the plain catamorphism

- Different termination behaviors:
  Clarified that some Mendler style combinators (e.g., histomorhpism) do not ensure termination for negative datatypes by providing a concrete counterexample

- Identified several combinators from the literature:
  they are interesting but their properties are not well studied

Research Goals

1. Demonstrate Mendler style combinators are expressive, easy to use, and provide termination guarantees

2. Organize the Hierarchy of the Mendler style recursion combinators

3. <u>Extend Mendler style recursion combinators to dependent types</u>

4. Identify which features commonly found in dependently typed languages
   - interact smoothly with or
   - conflict with the Mendler style approach

Research Challenges:
Goal 3. extension to dependent types

```
data Either a b = Left a | Right b
data Nat = Zero | Succ Nat

data Even (n:Nat) where
  EvenZ : Even Zero
  EvenS : Odd k → Even (Succ k)
data Odd (n:Nat) where
  OddS : Even k → Odd (Succ k)

evenOrOdd : (n:Nat) → Either (Even n) (Odd n)
evenOrOdd Zero     = EvenZ
evenOrOdd (Succ n) = case evenOrOdd n of
                       Left p  → OddS p
                       Right p → EvenS p
```

Research Challenges:
Goal 3. extension to dependent types

data N r = Z | S r

type Nat = μ0 N -- *Zero = In0 Z , Succ n = In0 (S n)*

$$mcataD :: (\forall r.\ ((v{:}r) \to \overset{\boxed{a :: r \to *}}{a\ v}) \to (y{:}f\ r) \to \overset{\boxed{a :: f\ r \to *}}{a\ y}) \to (x{:}\mu 0\ f\ ) \to \overset{\boxed{a :: \mu 0\ f \to *}}{a\ x}$$

$$mcataD\ \varphi\ (In0\ x) = \varphi\ (mcataD\ \varphi)\ x$$

evenOrOdd : (n:Nat) → Either (Even n) (Odd n)

evenOrOdd = cataD φ where

  φ evenOrOdd' Z = EvenZ

  φ evenOrOdd' (S n) = case evenOrOdd' n of

    Left p → OddS p

    Right p → EvenS p

Research Challenges: Goal 3. extension to dependent types

$$mcataD :: (\forall r.\, ((v{:}r) \to \overset{\boxed{a\,::\,r\to *}}{a}\ v) \to (y{:}f\ r) \to \overset{\boxed{a\,::\,f\ r\to *}}{a}\ y) \to (x{:}\mu 0\ f) \to \overset{\boxed{a\,::\,\mu 0\ f\to *}}{a}\ x$$

$$mcataD\ \varphi\ (In0\ x) = \varphi\ (mcataD\ \varphi)\ x$$

- Mendler style approach works by hiding the details of inductive datatype ($\mu 0\ f$) as an abstract type (r)
- Value dependency makes it hard to hide the details since the answer type (a) expects a value index whose type is the very inductive datatype ($\mu 0\ f$), which we wanted to hide
- So, what we need is a way to hide it at the value level but reveal it only at the type level
  - analogous to the translucent types to describe modules

Tentative Approach:
Goal 3. extension to dependent types

$\boxed{tr : r \to \mu 0\ f}$ $\boxed{tfr : f\ r \to \mu 0\ f}$ $\boxed{a : \mu 0\ f \to *}$

$$mcataD :: (\forall r.\ ((v{:}r) \to a(tr\ v)) \to (y{:}f\ r) \to a(tfr\ y)) \to (x{:}\mu 0\ f) \to a\ x$$

$$mcataD\ \varphi\ (In0\ x) = \ldots$$

- Seems that we need type coercion functions
  - *tr* coerces abstract values (*r*)
  - *trf* coerces structure containg abstract values (*f r*)

  to inductive values ($\mu 0\ f$)
- Challenge: make the coercion only happen at type level only

Tentative Approach:
Goal 3. extension to dependent types

$$\boxed{a : \mu 0\ f \to *}$$

$$mcataD :: (\forall r.\ [tr : r \to \mu 0\ f\,] \to [tfr : f\ r \to \mu 0\ f\,] \to$$
$$((v{:}r) \to a(tr\ v)) \to (y{:}f\ r) \to a(tfr\ y)\ ) \to (x{:}\mu 0\ f\,) \to a\ x$$
$$mcataD\ \varphi\ (In0\ x) = \varphi\ id\ In0\ (mcataD\ \varphi)\ x$$

- <u>Erasable</u> type coercion functions
  - *tr* coerces abstract values (*r*)
  - *trf* coerces structure containing abstract values (*f r*)

  to inductive values ($\mu 0\ f$)

- <u>Challenge: make the coercion only happen at type level only</u>

- One more thing ...

Tentative Approach:
Goal 3. extension to dependent types

$$mcataD :: (\forall r.\ [tr : r \to \mu 0\ f\,] \to [tfr : f\ r \to \mu 0\ f\,] \to$$
$$[trEq : tr = id] \to [tfrEq : tfr = In0] \to$$
$$((v{:}r) \to a(tr\ v)) \to (y{:}f\ r) \to a(tfr\ y)\ ) \to (x{:}\mu 0\ f\,) \to a\ x$$

$\boxed{a : \mu 0\ f \to *}$

$$mcataD\ \varphi\ (In0\ x) = \varphi\ id\ In0\ join\ join\ (mcataD\ \varphi)\ x$$

- Erasable type coercion functions
  - *tr* coerces abstract values (*r*)
  - *trf* coerces structure containg abstract values (*f r*)

  to inductive values ($\mu 0\ f$)

- Heterogeneous equality proofs that specify the properties of the coercion functions
  - needed when type checking the function body of φ definition

Summary

- There is a large amount of literature on Mendler style recursion combinators
- We studied and categorized many of the combinators
- We discovered general principles during our study
- We discovered new combinators by applying the general principles
- Applying the principles to dependently typed settings creates many new challenges

Thesis

A type based approach (in particular, the Mendler-style approach), in contrast to the value based approach (structural recursion), to ensure termination in dependently typed functional languages is both possible and practical.

It broadens the range of certified programming practices and provides an intuitive abstraction for expressing terminating recursion that generalizes naturally over many kinds of inductive datatypes.

Mendler style Catamorphism

seamlessly generalize to datatypes of higher-kinds

(e.g., nested, indexed datatypes)

$$
\begin{array}{l}
\textbf{data}\ Z \\
\textbf{data}\ S\ i \\
\textbf{data}\ \mathit{Vec}\ p\ i\ \textbf{where} \\
\quad N_V :: \mathit{Vec}\ p\ Z \\
\quad C_V :: p \to \mathit{Vec}\ p\ i \to \mathit{Vec}\ p\ (S\ i)
\end{array}
$$

$$
\begin{array}{l}
\mathit{copy} :: \mathit{Vec}\ p\ i \to \mathit{Vec}\ p\ i \\
\mathit{copy}\ N_V \qquad\quad = N_V \\
\mathit{copy}\ (C_V\ x\ xs) = C_V\ x\ (\mathit{copy}\ xs)
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ V\ p\ r\ i\ \textbf{where} \\
\quad N_V :: V\ p\ r\ Z \\
\quad C_V :: p \to r\ i \to V\ p\ r\ (S\ i) \\
\textbf{type}\ \mathit{Vec}\ p\ i = \mu_1\ (V\ p)\ i \\
\mathit{nil}_V \qquad\quad = \mathsf{In}_1\ N_V \\
\mathit{cons}_V\ x\ xs = \mathsf{In}_1\ (C_V\ x\ xs) \\
\mathit{copy} :: \mathit{Vec}\ p\ i \to \mathit{Vec}\ p\ i \\
\mathit{copy} = \mathit{mcata}_1\ \varphi\ \textbf{where} \\
\quad \varphi :: (\forall i. r\ i \to \mathit{Vec}\ p\ i) \to V\ p\ r\ i \to \mathit{Vec}\ p\ i \\
\quad \varphi\ \mathit{cp}\ N_V \qquad\quad = \mathit{nil}_V \\
\quad \varphi\ \mathit{cp}\ (C_V\ x\ xs) = \mathit{cons}_V\ x\ (\mathit{cp}\ xs)
\end{array}
$$

newtype $\mu 1\ (f :: (* \to *) \to (* \to *))\ (i :: *) = \mathit{In1}\ \{\ \mathit{out1} :: f\ (\mu 1\ f)\ i\ \}$

mcata1 $:: (\forall r. (r\ i \to a\ i) \to f\ r\ i \to a\ i) \to \mu 1\ f\ i \to a\ i$

mcata1 $\varphi\ (\mathit{In1}\ x) = \varphi\ (\mathit{mcata1}\ \varphi)\ x$

newtype $\mu 0\ (f :: * \to *) = \mathit{In0}\ \{\ \mathit{out0} :: f\ (\mu 0\ f)\ \}$

mcata0 $:: (\forall r. (r \to a) \to f\ r \to a) \to \mu 0\ f \to a$

mcata0 $\varphi\ (\mathit{In0}\ x) = \varphi\ (\mathit{mcata0}\ \varphi)\ x$

Datatypes - quick tour 1

categorized by pattern of
- recursive occurrences
- result types of data constructors

- Regular datatypes
- Non-regular datatypes
  - nested datatypes
  - indexed datatypes (GADTs)

data List p where
Nil :: List p
Cons :: p → List p → List p

data Bush i where
NilB :: Bush i
ConsB :: i → Bush (Bush i) → Bush i

data Vec p i where
Nil :: Vec p Z
Cons :: p → Vec p i → Vec p (S i)

convention
p: type parameter
i : type index