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

Aaron Stump     Garrin Kimmell
Harley D. Eades III      Peng Fu
Tim Sheard
Ki Yung Ahn     Nathan Collins
Stephanie Weirich
Chris Casinghino    Vilhelm Sjöberg
Aaron Stump     Garrin Kimmell
Harley D. Eades III      Peng Fu
# Characteristics of programming languages and reasoning systems

<table>
<thead>
<tr>
<th>Typed Functional Programming Languages (e.g., ML, Haskell)</th>
<th>Typed Logical Reasoning Systems (e.g., Coq, HOL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Can define arbitrary recursive types definition</td>
<td>- Can NOT define arbitrary recursive types (only positive ones)</td>
</tr>
<tr>
<td>- No guarantee for normalization</td>
<td>- Guaranteed to normalize</td>
</tr>
</tbody>
</table>
Nax: a middle ground taking good properties of the both worlds

<table>
<thead>
<tr>
<th>Typed Functional Programming Languages (e.g., ML, Haskell)</th>
<th>Typed Logical Reasoning Systems (e.g., Coq, HOL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Can define arbitrary recursive types definition</td>
<td>• Can NOT define arbitrary recursive types (only positive ones)</td>
</tr>
<tr>
<td>• No guarantee for normalization</td>
<td>• Guaranteed to normalize</td>
</tr>
</tbody>
</table>

Spoiler: Aren’t there well known calculi with such properties? Like System F and $F\omega$ ...

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

- Named after Nax P. Mendler
- An extension of System F (or, $F_\omega$)
- 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
    \[ \text{Prog (programmatic sublanguage): } \Gamma \vdash_{\text{Prog}} e : t \]
  • and, a sound logical reasoning system
    \[ \text{Log (logical sublanguage): } \Gamma \vdash_{\text{Log}} e : t \]

• What are the minimal requirements of Log?
  • 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 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 \( \mu F \) such that \( \mu F = F(\mu F) \)
  - \( N(X) = 1 + X \) natural number type: \( \mu N \)
  - \( L(X) = 1 + (A \times X) \) list type: \( \mu L \)
  - \( T(X) = A + (X \times X) \) binary tree type: \( \mu T \)
Two styles of recursive types

- **Equi-recursive (implicit conversion)**

  \[
  G |- e : \mu F \\
  \mu{-}\text{elim} \\
  G |- e : F(\mu F)
  \]

- **Iso-recursive (explicit conversion)**

  \[
  G |- e : \mu F \\
  \mu{-}\text{elim} \\
  G |- \text{unIn } e : F(\mu F)
  \]

  \[
  G |- e : F(\mu F) \\
  \mu{-}\text{intro} \\
  G |- e : \mu F
  \]
Two styles of recursive types

- **Equi-recursive (implicit conversion)**
  
  ```haskell
  type X = Either () X
  data Either a b = Left a | Right b
  ```

- **Iso-recursive (explicit conversion)**
  
  ```haskell
  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
  ```

This is only an analogy … cyclic type synonym is a type error in Haskell
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 \((\text{Nat}, \text{zero}, \text{succ})\) of the two-level implementation that hides more primitive constructs, that is, the recursion operator \((\text{Mu}, \text{In}, \text{out})\) and the base structure \((N, Z, S)\)

```haskell
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 \(\mu\)

unIn (In x) = x -- recall the reduction rule
```
Exercise on two-level types

• Natural numbers
  data Nat = Zero | Succ Nat

• Lists
  data List a = Nil | Cons a (List a)

• Trees
  data Tree a = Leaf a | Node (Tree a) (Tree a)
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 \((\lambda x.xxx) (\lambda x.xxx)\)

```
data T = C (T -> ())
p :: T -> (T -> ())
p (C f) = f
w :: T -> ()
w x = (p x) x
```

\[
\begin{align*}
& w (C w) \\
\Rightarrow & (p (C w)) (C w) \\
\Rightarrow & w (C w) \\
\Rightarrow & (p (C w)) (C w) \\
\Rightarrow & \ldots
\end{align*}
\]

-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**

\[
\begin{align*}
G |- F : * \rightarrow * \\
\text{----------------- } \mu\text{-form} \\
G |- \mu F : *
\end{align*}
\]

- **Restrict Elimination rule**

\[
\begin{align*}
G |- e : \mu F \\
\text{----------------- } \mu\text{-elim} \\
G |- \text{unIn } e : F(\mu F)
\end{align*}
\]

\[
\begin{align*}
G |- e : F(\mu F) \\
\text{----------------- } \mu\text{-intro} \\
G |- \text{In } e : \mu F
\end{align*}
\]

\[
\text{unIn (In } e \text{) } \rightarrow e
\]
Positive vs. Negative occurrences in recursive types

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

- data $A = C \ ((A \rightarrow \text{Bool}) \rightarrow \text{Bool})$
  - Positive since $A$ is in doubly negated position, but not strictly positive since $A$ appears inside the left hand side of the top level $ightarrow$
  - 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 $\text{SN} = \text{SN} \ (\forall b. \ b \rightarrow (\text{SN} \rightarrow b) \rightarrow b)$
    Scott Numerals - an encoding of natural numbers
Diverging computation using Negative recursive types

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

```haskell
data T = C (T -> ())
p :: T -> (T -> ())
p (C f) = f
w :: T -> ()
w x = (p x) x
```

- 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
  • \text{Red}\{\text{Nat}\}(M) \text{ iff } M \text{ reduce to canonical form of Nat}
  • \text{Red}\{A \rightarrow B\}(M) \text{ iff for all } N, \text{Red}\{A\}(N) \text{ implies } \text{Red}\{B\}(M \, N)

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

\[
\text{Inductive Red: ty } \rightarrow \text{ exp } \rightarrow \text{ Prop}
\]
\[
:= \text{RedN : forall } n, \text{Const } n \rightarrow \text{Red } \text{nat } n
\]
\[
| \, \text{RedA : forall } e A B, (\text{forall } A \, e', \text{Red } A \, e' \rightarrow \text{Red } B \, (e \, e'))
\]
\[
\rightarrow \text{Red } (A \rightarrow B)
\]
Why care about negative datatypes? (Example 2: HOAS)

- HOAS for untyped lambda calculus (in Haskell)
  
data Exp = Lam (Exp -> Exp) | App Exp Exp

- Since `Exp` models the untyped lambda calculus, its eval function `eval :: Exp -> Exp` is partial

- But, there can be many useful total functions over `Exp`, such as `showExp :: Exp -> 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

<table>
<thead>
<tr>
<th>Squiggol style</th>
<th>Mendler style</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developed in the context of Functional Programming Languages with Type Inference (Hindley-Milner type system)</td>
<td>Developed in the context of Nuprl, an interactive theorem prover for constructive math, with a very powerful type system (dependent types, higher rank polymorphism)</td>
</tr>
<tr>
<td>● No wonder why this style has been more popular in functional programming</td>
<td>Mendler didn’t notice this himself, later discovered by others</td>
</tr>
<tr>
<td>● Iteration/Recursion well-defined for positive datatypes, but not for negative datatypes</td>
<td>● Iteration/Recursion well-defined for arbitrary datatypes including negative datatypes</td>
</tr>
<tr>
<td>● Defined for regular datatypes, but not easy to generalize to non-regular datatypes (e.g. nested datatypes, GADTs)</td>
<td>● Naturally generalize to non-regular datatypes (more generally to type constructors of arbitrary higher kinds)</td>
</tr>
</tbody>
</table>
Exercise on two-level types

- **Natural numbers**
  ```ml
  data Nat = Zero | Succ Nat
  ```

- **Lists**
  ```ml
  data List a = Nil | Cons a (List a)
  ```

- **Trees**
  ```ml
  data Tree a = Leaf a | Node (Tree a) (Tree a)
  ```
Iteration (a.k.a. catamorhpism) in Squiggol style

**Generalization of folds, expressed using 2-level types**

Recursion is captured by \( \text{iter} \) at term level, and \( \text{Mu} \) at type level (non-recursive elsewhere)

- \( \text{fmap} \) guides where to invoke recursive calls
- User supplies \( \phi \) \( :: \, f \, a \rightarrow a \), which defines how to process the base structure containing answers of the already processed subcomponents

\[
\text{iter} :: \text{Functor } f \Rightarrow (f \, a \rightarrow a) \rightarrow \text{Mu } f \rightarrow a
\]

```
iter \( \phi \) (\text{In } x) = \phi \left( \text{fmap} \left( \text{iter} \phi \right) x \right)
```

```
instance \text{Functor} (L \, x) \text{ where}
-- \text{fmap} :: (a \rightarrow b) \rightarrow L \, x \, a \rightarrow L \, x \, b
fmap \, h \, N = N
fmap \, h \, (C \, x \, a) = C \, x \, (h \, a)
```

```
\text{lenList} = \text{Mu} (L \, x) \rightarrow \text{Int}
\text{lenList} = \text{iter} \, \phi
where
  \phi :: L \, x \, \text{Int} \rightarrow \text{Int}
  \phi \, N = 0
  \phi \, (C \, x \, \text{xlen}) = 1 + \text{xlen}
```
Iteration (a.k.a. catamorhispism) in Mendler style

- Key idea: $\varphi$ 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 ($\forall r. \ldots$) enforce recursive subcomponents in the base structure ($f r$) be abstract inside $\varphi$, 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.

```haskell
mite :: (\forall r. (r -> a) -> f r -> a) -> Mu f -> a
mite \varphi (In x) = \varphi (mite \varphi) x

lenList = Mu (L x) -> Int
lenList = miter phi
where
phi :: \forall r. (r -> Int) -> L x r -> Int
phi len N = 0
phi len (C x xs) = 1 + len xs
```
Does miter really normalize?

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

\[
\text{miter} :: (\ (\text{Mu } f \to a) \to f (\text{Mu } f) \to a) \to \text{Mu } f \to a \quad -- \text{Naïve type}
\]

\[
\text{miter} :: (\forall r. (r \to a) \to f \ r \to a) \to \text{Mu } f \to a \quad -- \text{Mendler’s type}
\]

\[
\text{cons} :: x \to \text{Mu} (\text{L } x) \to \text{Mu} (\text{L } x)
\]

\[
\text{lenListBad} = \text{mcata} \phi
\]

where

\[
\phi \text{ len } N = 0
\]

\[
\phi \text{ len } (\text{C } x \text{ xs}) = 1 + \text{len} (\text{cons } x \text{ xs})
\]

- \text{lenListBad} will type check with the naïve type of \text{miter}
- \text{len} (\text{cons } x \text{ xs}) is a type error with Mendler’s type
- \text{cons} expects its 2\text{nd} arg of type \text{Mu} (\text{L} x) but \text{xs} :: r, where r is parametric (or, abstract)
- \text{len} :: (r \to a) expects an arg of abstract type r but the result type of \text{cons} is \text{Mu} (\text{L} x)
Does \texttt{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 $F_\omega$ (i.e., one reduction step involving \texttt{miter} is simulated by one or more constantly bound reduction steps in $F_\omega$)

\begin{align*}
\text{newtype} & \text{ Mu } f = \text{ In } (f \text{ (Mu } f)) \\
\text{mcata} :: (\forall r.(r \to a) \to f \ r \to a) \to \text{ Mu } f \\
& \to a \\
\text{mcata } \varphi \ (\text{In } x) = \varphi \ (\text{mcata } \varphi) \ x
\end{align*}

\begin{itemize}
  \item \{- lambda abstraction, application, and non recursive base structures have trivial embeddings into $F_\omega$ -\}
\end{itemize}
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 subcomponent values, it really isn’t. (Try to define factorial if you are in doubt.)

\[
\text{lenList} = \text{Mu} \ (L \ x) \rightarrow \text{Int} \\
\text{lenList} = \text{miter} \ \phi \\
\text{where} \\
\phi :: \forall r. (r \rightarrow \text{Int}) \rightarrow L \ x \ \text{Int} \rightarrow \text{Int} \\
\phi \ \text{len} \ N = 0 \\
\phi \ \text{len} \ (C \ x \ xs) = 1 + \text{len} \ xs
\]
**Mendler style Primitive Recursion**

\[
	ext{miter} :: (\forall r. (r \to a) \to f r \to a) \to \text{Mu } f \to a
\]

\[
\text{miter } \varphi \text{ (In } x) = \varphi \text{ (miter } \varphi) \text{ } x
\]

\[
\text{mprec} :: (\forall r. (r \to \text{Mu } f) \to (r \to a) \to f r \to a) \to \text{Mu } f \to a
\]

\[
\text{mprec } \varphi \text{ (In } x) = \varphi \text{ id } (\text{mprec } \varphi) \text{ } x
\]

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

\[
\text{fact} = \text{Mu } N \to \text{Nat}
\]

\[
\text{fact} = \text{mprec } \phi \text{ where } \phi :: \forall r. (r \to \text{Mu } N) \to (r \to \text{Nat}) \to N r \to \text{Nat}
\]

\[
\phi \text{ cast } \text{fac } Z = 0
\]

\[
\phi \text{ cast } \text{fac } (S \text{ n}) = \text{succ}(\text{cast } n) \times \text{fac } n
\]

- \(\varphi\) for \text{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 \text{mprec}
A Hierarchy of Mendler style Iteration/Recursion Combinators

- miter, msfiter can be embedded into $F_\omega$
- mprec embeds into $\text{Fix}_\omega$ (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 $\text{Mu}$ and related combinators are index by kinds $\text{Mu}_{\ast}, \text{In}_{\ast}, \text{miter}_{\ast}, \ldots$
$\text{Mu}_{\ast \rightarrow \ast}, \text{In}_{\ast \rightarrow \ast}, \text{miter}_{\ast \rightarrow \ast}, \ldots$
Mendler style

course of values Iteration

\[
\begin{align*}
\text{miter} &:: (\forall r. (r \rightarrow a) \rightarrow f \ r \rightarrow a) \rightarrow \text{Mu} \ f \rightarrow a \\
\text{miter} \ \phi \ (\text{In} \ x) &= \phi \ (\text{miter} \ \phi) \ x
\end{align*}
\]

\[
\begin{align*}
\text{mcviter} &:: (\forall r. (r \rightarrow f \ r) \rightarrow (r \rightarrow a) \rightarrow f \ r \rightarrow a) \rightarrow \text{Mu} \ f \rightarrow a \\
\text{mcviter} \ \phi \ (\text{In} \ x) &= \phi \ \text{unIn} \ (\text{mcviter} \ \phi) \ x
\end{align*}
\]

\[
(+) :: \text{Mu} \ N \rightarrow \text{Mu} \ N \rightarrow \text{Nat}
\]

\[
\text{fibo} = \text{Mu} \ N \rightarrow \text{Nat}
\]

\[
\text{fibo} = \text{mcviter} \ \phi \ \text{where} \ \phi :: (r \rightarrow N \ r) \rightarrow (r \rightarrow \text{Nat}) \rightarrow N \ r \rightarrow \text{Nat}
\]

\[
\begin{align*}
\phi \ \text{out} \ \text{fib} \ Z &= \text{succ} \ \text{zero} \\
\phi \ \text{out} \ \text{fib} \ (S \ n) &= \text{case} \ \text{out} \ n \ \text{of} \ Z \rightarrow \text{succ} \ \text{zero} \\
& \quad S \ n' \rightarrow \text{fib} \ n + \text{fib} \ n'
\end{align*}
\]

- \( \phi \) for \( \text{mcviter} \) expects yet another argument, which is an abstract eliminator \( \text{out} :: r \rightarrow f \ r \) passing around \( \text{unIn} :: \text{Mu} \ f \rightarrow f \ (\text{Mu} \ f) \), giving the ability to abstractly eliminate (i.e., pattern match away) \( \text{In} \) constructor of \( \text{Mu} \)
**Mendler style**

**Sheard-Fegaras Iteration**

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

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

```haskell
data E r = A r r | L (r → r)  -- base structure for HOAS
type Exp = \forall a. Mu' E a
countVar :: Exp → Int  -- count the no. of variable use. (\lambda x.xx) is 2, (\lambda x.\lambda y.x) is 1
countVar = msfiter phi  where  phi :: \forall 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 \((\text{inv} 1 :: r \text{ Int})\) from an expected result value \((1 :: \text{Int})\) so that you can supply it to a function \((g :: r \text{ Int} → r \text{ Int})\) expecting an abstract recursive value.

See our ICFP11 paper for further details.
A Hierarchy of Mendler style
Iteration/Recursion Combinators

- miter, msfiter can be embedded into $F_\omega$
- mprec embeds into $\text{Fix}_\omega$
  (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 $\text{Mu}$ and related combinators are index by kinds $\text{Mu}_{\ast}, \text{In}_{\ast}, \text{miter}_{\ast}, ...$
  $\text{Mu}_{\ast\rightarrow\ast}, \text{In}_{\ast\rightarrow\ast}, \text{miter}_{\ast\rightarrow\ast}, ...$
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 $F\omega$ or $\text{Fix}_\omega$? Why not just use $F\omega$ or $\text{Fix}_\omega$?

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

- Embedding into $F\omega$ or $\text{Fix}_\omega$ is only a tool for showing normalization
- Encoding datatypes in $F\omega$ or $\text{Fix}_\omega$ 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

\[
\text{data } \text{Mu}' \ f \ a = \text{In}' \ (f \ (\text{Mu}' \ f \ a)) \mid \text{Inverse } a \quad \text{-- Mu with syntactic inverse}
\]

\[
\text{msfiter} :: (\forall r. (a \to r \ a) \to (r \ a \to a) \to f \ (r \ a) \to a) \to (\forall a. \text{Mu}' \ f \ a) \to a
\]
\[
\text{msfiter } \varphi \ (\text{In}' \ x) = \varphi \ \text{Inverse} \ (\text{msfiter } \varphi) \ x
\]
\[
\text{msfiter } \varphi \ (\text{Inverse } x) = x
\]

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

\[
\text{msfiter} :: (\forall r. (a \to r) \to (r \to a) \to f \ r \to a) \to \text{Mu } f \to a
\]
\[
\text{msfiter } \varphi \ (\text{In}' \ x) \to \varphi \ \text{Inverse} \ (\text{msfiter } \varphi) \ x
\]
\[
\text{msfiter } \varphi \ (\text{Inverse } x) \to x
\]

\text{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

\[ Dec ::= F \overline{X} X.\{ \overline{C} T \} \]

\[ Decs ::= \cdot | Dec; Decs \]

\[ T ::= F \overline{TT} | T \rightarrow T | X | \forall X.T | \mu(F \overline{T}) \]

\[ M ::= x | C | \text{case } M \{ \overline{C} \overline{x}.M \} | \lambda x.M | MM \]

\[ | \text{in } M | \text{mit } M | mrec M | mcvit M | mcvrec M | msfit M \]

\[ | \text{out} | \text{inv} \quad \text{-- these are transient objects cannot appear in source code} \]

\[ Program ::= Decs; M \]

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

\[
\frac{\Gamma \vdash M : F\overline{T} (\mu(F\overline{T}))}{\Gamma \vdash \text{in} \ M : \mu(F\overline{T})}
\]

\[
\frac{\Gamma, X : * \vdash M : (X \to \mu(F\overline{T})) \to (X \to T') \to F\overline{T} X \to T'}{\Gamma \vdash \text{mit} \ M : \mu(F\overline{T}) \to T'}
\]

\[
\frac{\Gamma, X : * \vdash M : (X \to \mu(F\overline{T})) \to (X \to T') \to F\overline{T} X \to T'}{\Gamma \vdash \text{mrec} \ M : \mu(F\overline{T}) \to T'}
\]

\[
\frac{\Gamma, X : * \vdash M : (X \to F\overline{T} X) \to (X \to T') \to F\overline{T} X \to T'}{\Gamma \vdash \text{mcvit} \ M : \mu(F\overline{T}) \to T'}
\]

\[
\frac{\Gamma, X : * \vdash M : (X \to F\overline{T} X) \to (X \to F\overline{T} X) \to (X \to T') \to F\overline{T} X \to T'}{\Gamma \vdash \text{mcvrec} \ M : \mu(F\overline{T}) \to T'}
\]

\[
\frac{\Gamma, X : * \vdash M : (T' \to X) \to (X \to T') \to F\overline{T} X \to T'}{\Gamma \vdash \text{msfit} \ M : \mu(F\overline{T}) \to T'}
\]
Reduction

\((\lambda x. M) M' \longrightarrow M[M'/x]\)

mit \(M\) (in \(M'\)) \(\longrightarrow\) \(M\) (mit \(M\)) \(M'\)

mrec \(M\) (in \(M'\)) \(\longrightarrow\) \(M\) (\(\lambda x.x\)) \(mrec M\) \(M'\)

mcvit \(M\) (in \(M'\)) \(\longrightarrow\) \(M\) \(\text{out}(\text{mcvit} M)\) \(M'\)

mcvrec \(M\) (in \(M'\)) \(\longrightarrow\) \(M\) (\(\lambda x.x\)) \(\text{out}(\text{mcvrec} M)\) \(M'\)

\(\text{out} \ (\text{in} \ M') \longrightarrow M'\)

msfit \(M\) (in \(M'\)) \(\longrightarrow\) \(M\) \(\text{inv}(\text{msfit} M)\) \(M'\)

msfit \(M\) (inv \(M'\)) \(\longrightarrow\) \(M'\)

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

\(E ::= \ldots\)
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 $F_\omega$ (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

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

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

- 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 new family of Mendler style iteration combinator \texttt{msfiter}.
- We are designing Nax to realize this idea.