Inductiveness of types and Normalization of terms

Ki Yung Ahn <kya@cs.pdx.edu>

Thesis Proposal talk on 2011-09-28

Portland State University

My View on the problem space of Formal Reasoning Systems

	normalizing terms	possibly non-normalizing terms
Inductive types	**IND**	**$\mathrm{IND}_{\perp}$**
recursive types (possibly non-inductive)	**REC**	**$\mathrm{REC}_{\perp}$**

My View on the problem space of Formal Reasoning Systems

	normalizing terms	possibly non-normalizing terms
Inductive types	**IND** Typed Proof Assistants (e.g. HOL, Coq, Agda, ...)	**$\mathbf{IND}_{\perp}$** (First-order) General purpose prog langs
recursive types (possibly non-inductive)	System F, Fw, ... **REC**	General purpose prog langs **$\mathbf{REC}_{\perp}$** LCF (Scotts' domain theory) based proof assistants (e.g. Edinburgh LCF, HOLCF)

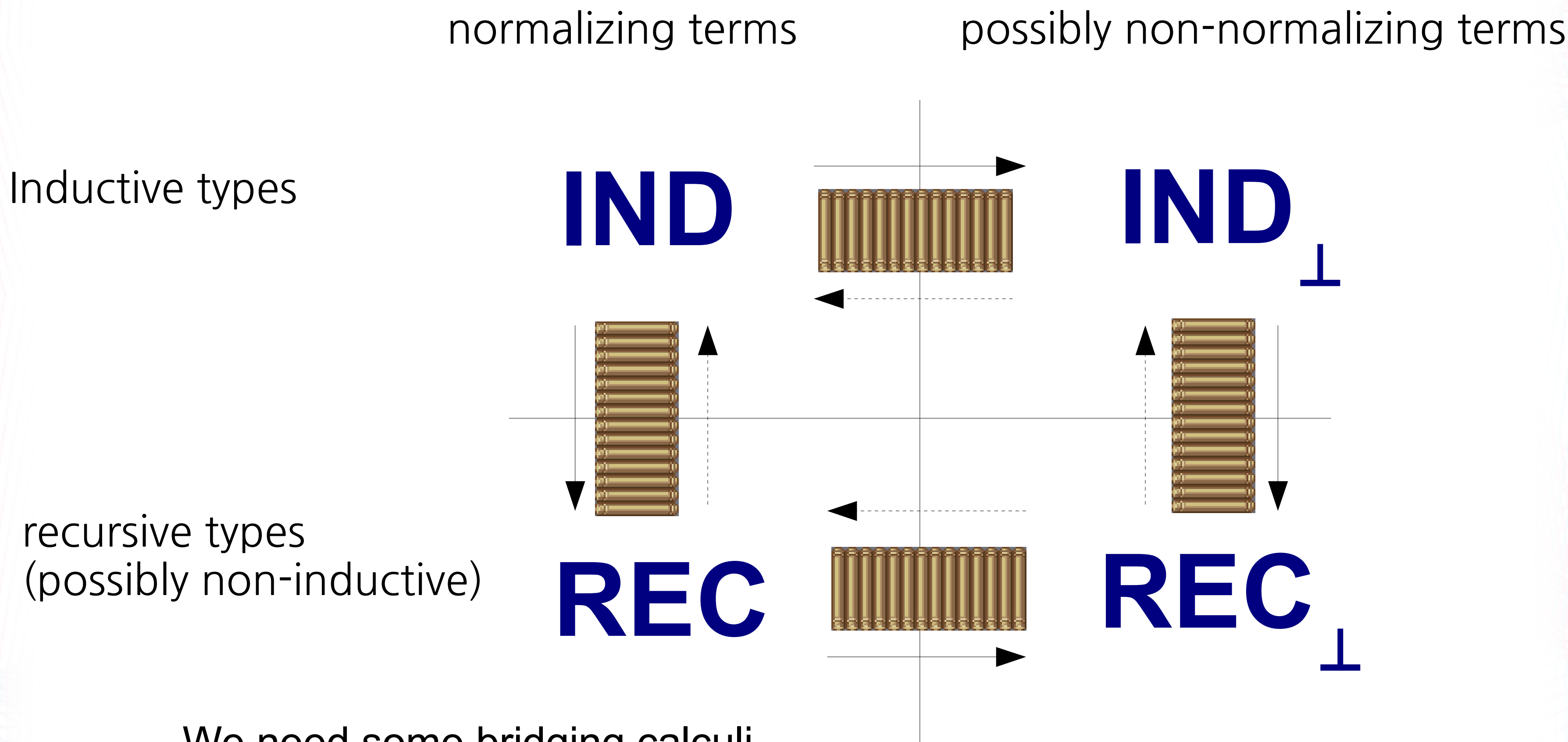
Calculi here are well-known, but
often neglected as a design space
of formal reasoning systems

My View on the problem space of Formal Reasoning Systems

	normalizing terms		possibly non-normalizing terms
Inductive types	**IND**	$\subset$	**$\mathbf{IND}_\perp$**
	$\cap$		$\cap$
recursive types (possibly non-inductive)	**REC**	$\subset$	**$\mathbf{REC}_\perp$**

Conceptually, the four fragment are related by inclusion relations

My View on the problem space of Formal Reasoning Systems



We need some bridging calculi.
There are problem statements in a narrower fragment (e.g. IND) whose solution is easier to express in a broader fragment (e.g. REC) such as Normalization by Evaluation

Proposed Thesis

- Normalization of terms and Inductiveness types are separate concerns (as illustrated in the previous diagrams)
- Language design properly separating these two concerns can lead to more expressive or more usable formal reasoning systems

Why do we care about REC?

Interesting and useful examples of non-inductive recursive types exist

- Reducibility (a unary logical relation) in normalization proofs of typed lambda calculi
  - Most naturally written as a non-inductive type
  - In systems like Coq, users need to employ more complicated tricks to avoid this natural encoding
- Higher-Order Abstract Syntax (HOAS)
  - Classical example in theory
  - Used to implement interpreters and type preserving transformations in compilers (papers and even a thesis on this topic)

Why do we care about REC?

Interesting and useful examples of non-inductive recursive types exist

- Reducibility (a unary logical relation) in normalization proofs of typed lambda calculi
  - Most naturally written as a non-inductive type
  - In systems like Coq, users need to employ more complicated tricks to avoid this natural encoding
- Higher-Order Abstract Syntax (HOAS)
  - Classical example in theory
  - Used to implement interpreters and type preserving transformations in compilers (papers and even a thesis on this topic)

Why do we care about REC? (Example 1: Reducibility)

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

- In proof assistants like Coq, this 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 do we care about REC? (Example 2: HOAS)

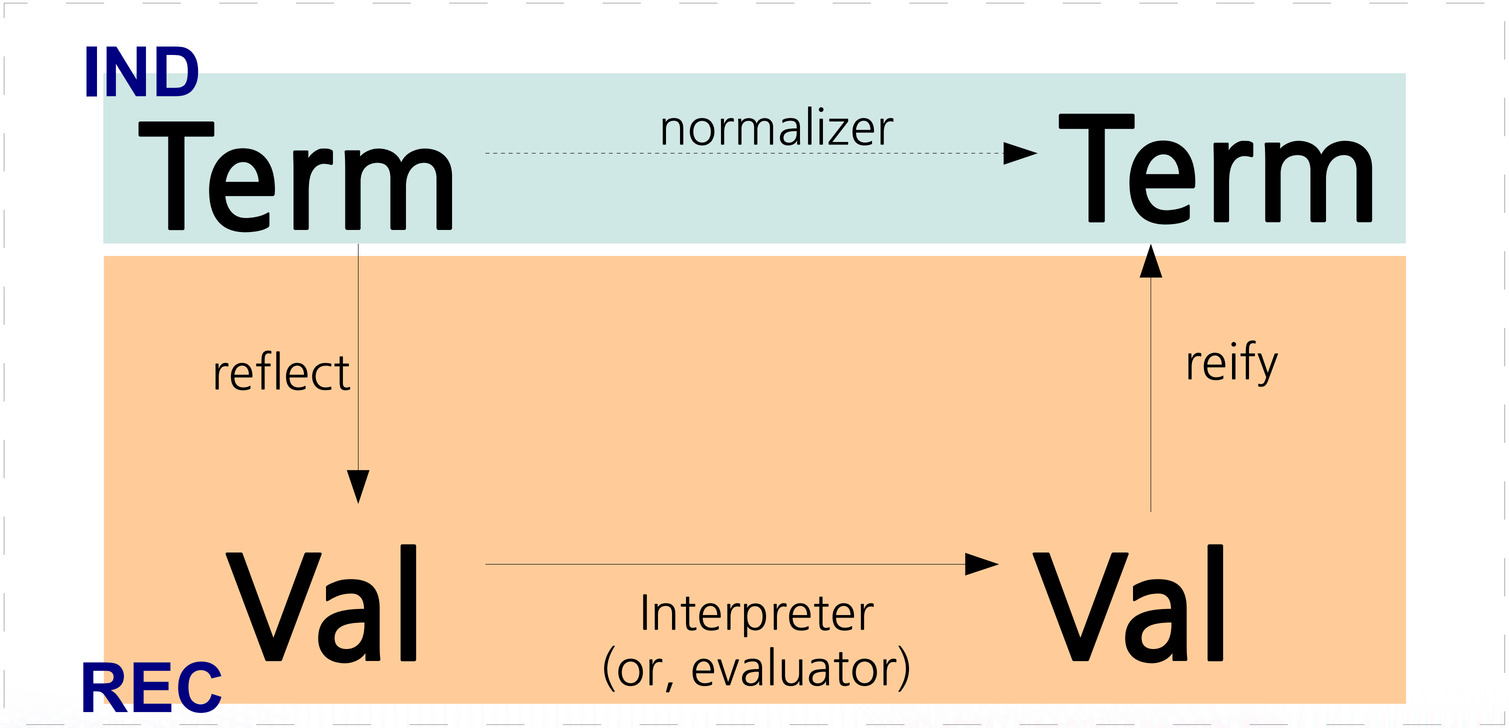
- HOAS for untyped lambda calculus (in Haskell)

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

  - 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 do we care about bridging between REC and IND?

- Example: Normalization by Evaluation
  - Define normalization of terms (inductive type) using evaluation of values (non-inductive type)



Why do we care about bridging between REC and IND?

- Example: Normalization by Evaluation
  - Define normalization of terms (inductive type) using evaluation of values (non-inductive type)
- More generally

$$f = g \circ f' \circ h$$

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "h"'] & B \\
A' \arrow[r, "f'"] & B' \arrow[u, "g"']
\end{tikzcd}$$

$$\frac{\Gamma \vdash_{\mathsf{REC}} f : A \to B \qquad \Gamma \vdash_{\mathsf{IND}} A : \star \qquad \Gamma \vdash_{\mathsf{IND}} B : \star}{\Gamma \vdash_{\mathsf{IND}} f : A \to B}$$

Why do we care about bridging between REC and IND?

- Example: Normalization by Evaluation
  - Define normalization of terms (inductive type) using evaluation of values (non-inductive type)
- More generally

$$\frac{\Gamma \vdash_{\mathsf{REC}} f : A \to B \qquad \Gamma \vdash_{\mathsf{IND}} A : \star \qquad \Gamma \vdash_{\mathsf{IND}} B : \star}{\Gamma \vdash_{\mathsf{IND}} f : A \to B}$$

- Even more generally

$$\textsf{REC-IND}\ \frac{\Gamma \vdash_{\mathsf{REC}} e : T \qquad \Gamma \vdash_{\mathsf{IND}} T : \star}{\Gamma \vdash_{\mathsf{IND}} e : T}$$

Motivation

- Want to consider functions defined in a broader fragment (e.g. REC) as if they were in a narrower fragment (e.g. IND), under certain conditions
  - For REC and IND, we believe the only condition we need is when the type of a term is inductive
  - For other cases, we may need to track other conditions such as totality or termination
- We want to be able to do this because the broader fragment (e.g. REC) is more expressive than the narrower fragment (e.g. IND)
  - may be easier to implement
  - more efficient implementation may exist
  - can reuse existing functional language code

Some Important Questions

- What do “inductive type” and “recursive type” mean?
- When do recursive types coincide with inductive types?
  - syntactic
  - Strictly positive datatypes
  - semantic
  - Monotonicity (by Matthes)
  - somewhere in between
- How do we ensure or prove normalization?
  - Inductive types and positive types
    : usually rely on principled recursion
    (e.g. structural recursion, primitive recursion)
  - Recursive types including negative datatypes
    : can use Mendler style iteration
- Language design?
  - Many open issues (e.g. dependent types) here

Two Paradigms on Type Systems

- Recursive type paradigm (programming langs)
  - Types are safety properties (i.e., preserved during program execution)
  - Syntactically correct type definitions are valid
- Inductive type paradigm (proof assistants)
  - Martin-Löf's Intuitionistic Type Theory
  - Types are propositions and programs are proofs
  - Since types are propositions, they must have well understood interpretations (e.g. sets)
  - Therefore, not all recursive types are inductive!

Inductive types

- Bootstrap from finite types
- Build more complex types using well-understood connectives (e.g. Π, Σ, W)
  - Types are defined as set of canonical forms
  - Compute non-canonical forms into canonical forms using primitive recursion
  - Equality
- All types have well-behaved (i.e. set theoretic) interpretation by construction
- $\bot$ (divergence) is NOT an instance of any type!!!

Recursive types

- Example: Natural Numbers

  $\mu X . 1 + X$ denotes a solution for $X = 1 + X$

- Equi-recursive (implicit conversion both ways)

$$\frac{G \mid\!- \; n : \mu X.1+X}{G \mid\!- \; n : \mu X.1+(\mu X.1+X)} \qquad \frac{G \mid\!- \; n : \mu X.1+(\mu X.1+X)}{G \mid\!- \; n : \mu X.1+X}$$

- Iso-recursive (explicit conversion each way)

$$\frac{G \mid\!- \; n : \mu X.1+X}{G \mid\!- \; \mathrm{unroll}\ n : \mu X.1+(\mu X.1+X)} \qquad \frac{G \mid\!- \; n : \mu X.1+(\mu X.1+X)}{G \mid\!- \; \mathrm{roll}\ n : \mu X.1+X}$$

$$\mathrm{unroll}\ (\mathrm{roll}\ e) \rightarrow e$$

Recursive types

- Example: Natural Numbers

  $\mu X . 1 + X$ denotes a solution for $X = 1 + X$

- Equi-recursive (implicit conversion both ways)

```
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 each way)

```
data N x = Z | S x                  -- like 1 + X part
type Nat = Mu N                     -- μ X . 1 + X
zero    = Roll Z
succ n  = Roll (S n)
newtype Mu f = Roll (f (Mu f))   -- definition of μ
unRoll (Roll x) = x         -- recall the reduction rule
```

Two-level types

- Usual one-level recursive type definition of 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, Roll, unRoll) and the base structure (N, Z, S)

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

```
data N x = Z | S x                  -- like 1 + X part
type Nat = Mu N                     -- μ X . 1 + X
zero     = Roll Z
succ n   = Roll (S n)
newtype Mu f = Roll (f (Mu f))   -- definition of μ
unRoll (Roll x) = x         -- recall the reduction rule
```

Some Important Questions

- What do “inductive type” and “recursive type” mean?
- When do recursive types coincide with inductive types?

syntactic

  - Strictly positive datatypes

semantic

  - Monotonicity (by Matthes)

somewhere in between

- How do we ensure or prove normalization?
  - Inductive types and positive types
    : usually rely on principled recursion
    (e.g. structural recursion, primitive recursion)
  - Recursive types including negative datatypes
    : can use Mendler style iteration
- Language design
  - Many open issues (e.g. dependent types) here

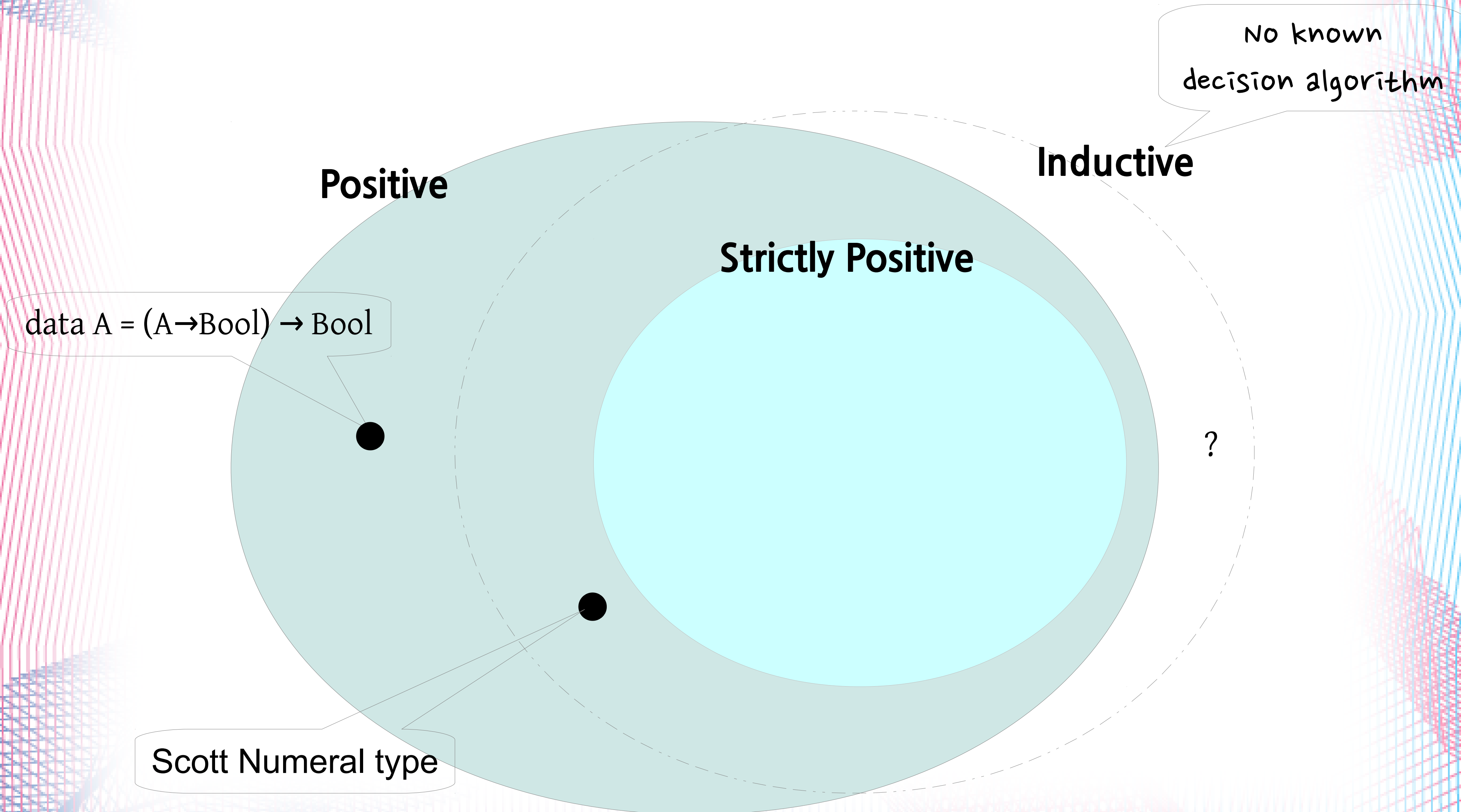
Positive vs. Negative occurrences in recursive types

- Interpreting (A→B) logically as implication, which is equivalent to (¬A∧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 $A = (A \to Bool) \to Bool$
  - 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 non-inductive 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 are inductive
- Some positive, but not strictly positive, types CAN be considered inductive
  - data $SN = SN\ (\forall b.\ b \to (SN \to b) \to b)$
    Scott Numerals encode of natural numbers

Strictly Positive vs. Positive

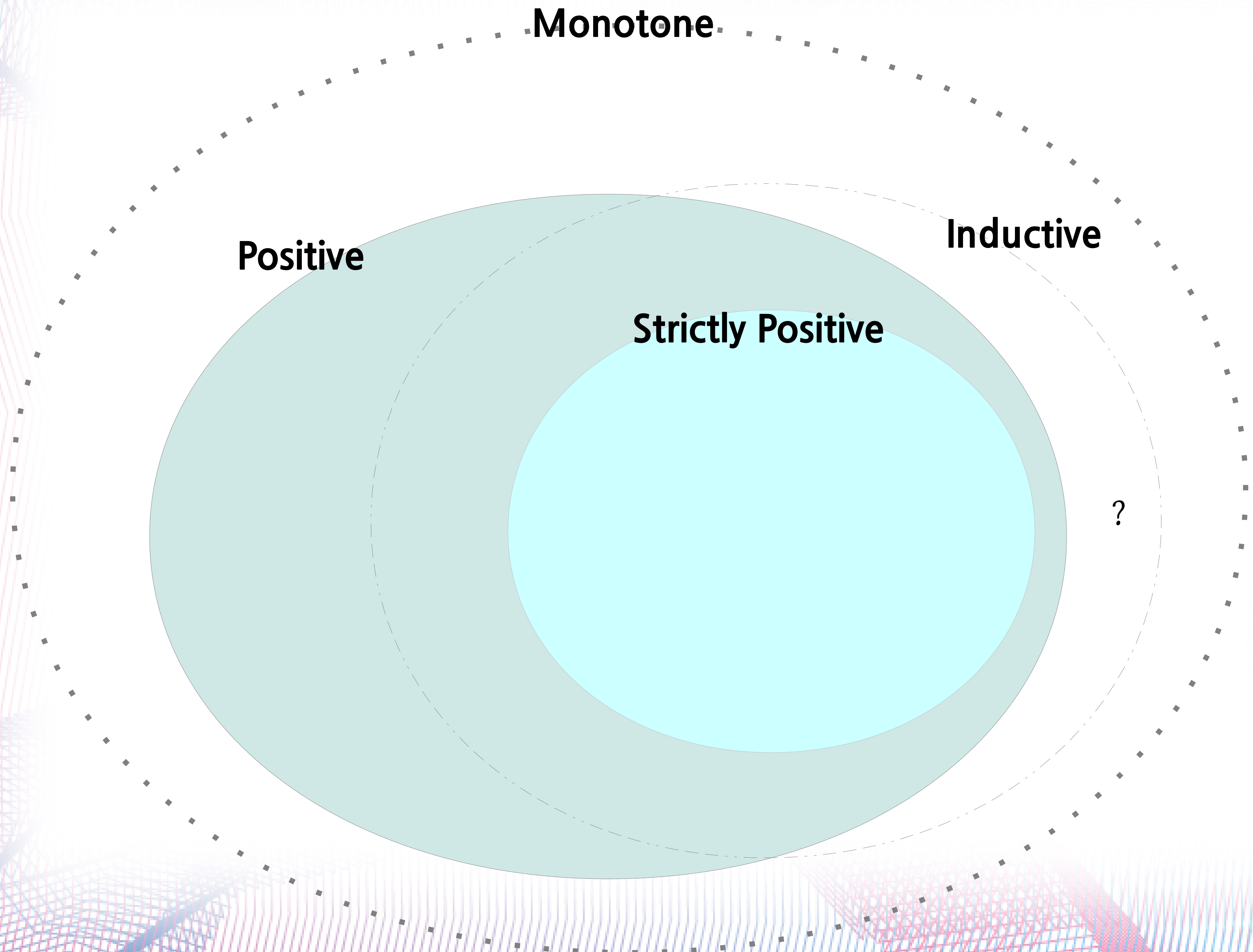


Strictly Positive vs. Positive

- Strict positivity is a syntactic criteria that conservatively approximates inductiveness
  - All strictly positive types are inductive
  - Not all positive, but non-strictly positive, types are inductive (some are, some aren't)
- All positive types are known to share the same normalization property under primitive recursion
  - Regardless of whether they are strictly positive or inductive
  - This again implies that **normalization is a separate concern from inductiveness**

Monotonicity vs. Positivity

- A recursive type $\mu\alpha.T$ is monotone when there exists a term of type $\forall\alpha.\forall\beta.(\alpha\to\beta)\to T\to T[\beta/\alpha]$, which is called a monotonicity witness
  - Monotonicity is a semantic characterization that generalizes positivity
  - It is reported that some negative types are monotone
  - All monotone types share the same normalization property, which hold for positive types
  - Again, not all monotone types are inductive
    (not all positive types are inductive, nor negative types are)
- Emphasizing again: **Normalization is a separate concern from Inductiveness**


Monotone
Positive
Inductive
Strictly Positive
?

Strict Positivity ~ Inductiveness Positivity ~ Normalization property

In my dissertation, I will just stick to the syntactic approximation for the sake of simplicity (since there are many other research topics to work on)

That is, I will use the syntactic approximation

- Strict Positivity for Inductiveness
- Positivity for Normalization properties (under primitive recursion)

Some Important Questions

- What do "inductive type" and "recursive type" mean?
- When do recursive types coincide with inductive types?

syntactic

  - Strictly positive datatypes

somewhere in between

semantic

  - Monotonicity (by Matthes)
- How do we ensure or prove normalization?
  - Inductive types and positive types
    : usually rely on principled recursion
    (e.g. structural recursion, primitive recursion)
  - Recursive types including negative datatypes
    : can use Mendler style iteration
- Language design
  - Many open issues (e.g. dependent types) here

(Primitive) Recursion vs. Iteration

$$\text{Pr-0}\;\frac{}{\mathsf{Pr}\ 0\ e_0\ e_2 \to e_0}$$

$$\text{Pr-s}\;\frac{}{\mathsf{Pr}\ (\mathsf{S}\ n)\ e_0\ e_2 \to e_2\ n\ (\mathsf{Pr}\ n\ e_0\ e_2)}$$

$$\text{Pr-ctx}\;\frac{e \to e'}{\mathsf{Pr}\ e\ e_0\ e_2 \to \mathsf{Pr}\ e'\ e_0\ e_2}$$

$$\text{It-0}\;\frac{}{\mathsf{It}\ 0\ e_0\ e_1 \to e_0}$$

$$\text{It-s}\;\frac{}{\mathsf{It}\ (\mathsf{S}\ n)\ e_0\ e_1 \to e_1\ (\mathsf{It}\ n\ e_0\ e_1)}$$

$$\text{It-ctx}\;\frac{e \to e'}{\mathsf{It}\ e\ e_0\ e_1 \to \mathsf{It}\ e'\ e_0\ e_1}$$

Primitive Recursion

- Have access to both the predecessor (n) and the answer (Pr n e0 e2) for the recursive call to the predecessor
- Constant time predecessor is definable (let e2 be λn.λa.n)

Iteration

- Have access to only the answer (It n e0 e1) for the recursive call to the predecessor
- Constant time predecessor is not known to be definable

(Primitive) Recursion vs. Iteration

- **Pr** has <u>*the ability to access recursive subcomponents*</u>, not only the result of the computation over the recursive subcomponents
- For natural numbers, and more generally for positive datatypes, primitive recursion and iteration have the same computability
  - **Pr** can be defined in terms if **It** and vice versa
  - Efficiency (computational complexity) may differ
- For negative datatypes, computability differs for primitive recursion and iteration
  - Iteration only express terminating computation
  - Primitive recursion can express diverging computation

Negative datatypes can cause diverging computation

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

$\mathbf{data}\ T = C\ (T \to ())$	$w\ (C\ w)$
$p :: T \to (T \to ())$	$\leadsto (p\ (C\ w))\ (C\ w))$
$p\ (C\ f) = f$	$\leadsto w\ (C\ w)$
$w :: T \to ()$	$\leadsto (p\ (C\ w))\ (C\ w))$
$w\ x = (p\ x)\ x$	$\leadsto \cdots$

- Can express diverging computation even without any use of term-level recursion
  - **Ability to access the recursive subcomponent** is enough to cause diverging computation

In 2-level types, unlmited use of **unRoll**

don't even need recursively computed answer

BUT, Negative datatype need not automatically imply divergence

- Principled use of recursion (e.g. folds on lists, primitive recursion, structural recursion) can guarantee terminating computation for positive datatypes only
- Question: Does there exist any principle X s.t. Principled use of such X can guarantee terminating computation for all datatypes, including negative datatypes?
- One answer for X is Mendler style iteration

Type Formation and Use (Inductive types)

- Only principled use guarantees normalization for inductive types when we have general recursion
- Inductive type formation (or, definition) does not guarantee normalization

```
data Nat = Zero | Succ n -- inductive type
loop n = loop (Succ n)     -- loop is partial
f Zero       = Succ Zero   -- f is total
f (Succ n) = Succ (f n)
```

- Principled use is the key for normalization in both inductive and recursive types paradigm

Type Formation and Use (Recursive types)

- Recursive type formation (or, definition) does not automatically imply divergence
- Principled use of the terms of recursive types can guarantee normalization
  - For some recursive types, which coincide with inductive types, the same principled recursion (e.g. structural recursion) can be used
  - More generally, including non-inductive recursive types, Mendler style **iteration** can guarantee normalization

Iteration ≈ catamorphism ≈ fold

- **Iteration**, in other context, called catamorphism
- Catamorphism is a generalization of folds
- Conventional (or, Squiggol style) catamorphism
  - well-defined only for covariant functors (≈ inductive types ≈ positive datatypes), but
  - not for contravariant or mixed variant functors (≈ non-inductive types ≈ negative datatypes)
- **Mendler style catamorphism** (Nax P. Mendler)
  - well-defined for ANY datatype, and
  - even for type constructors of higher rank (i.e. nested datatypes, GADTs)

Conventional vs. Mendler style Catamorphism

- Conventional (or, Squiggol style) catamorphism
  - Studied in the context of Hindely-Milner languages (automatic type inference)
  - Work for positive functors (≈ positive datatypes)
  - Do not generalize well to other datatypes
  - Motivates discussion of Mendler style
- Mendler style catamorhpism
  - Studied in the context (Nuprl) of interactive theorem proving (type check with manual intervention)
  - Work for all datatypes
  - generalize well for type constructors of higher rank
  - Requires higher-rank polymorphism

Exercise on two level types (warm-up for catamorhpism)

- Natural numbers

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

- Lists

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

- Trees

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

Conventional Catamorphism

```
cata :: Functor f ⇒
        (f a → a) → Mu f → a
cata φ (Roll x) =
  φ (fmap (cata φ) x)
```

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

phi :: L x Int → Int
phi N            = 0
phi (C x xslen) = 1 + xslen

lenList = Mu (L x) → Int
lenList = cata phi
```

- Generalization of folds using two level types
- All recursion is captured in **cata** at the term-level, and in **Mu** at the type level.
  (non-recursive everywhere else)
- **fmap** guides where to invoke the recursive call
- **phi** defines how to process the base structure containing the answers of the already processed subcomponents

Mendler style Catamorphism

$\text{mcata} :: (\forall r.(r \to a) \to f\ r \to a) \to$
$\quad \text{Mu}\ f \to a$
$\text{mcata}\ \varphi\ (\text{Roll}\ x) = \varphi\ (\text{mcata}\ \varphi)\ x$

$\text{phi} :: \forall r.(r \to \text{Int}) \to L\ x\ \text{Int} \to \text{Int}$
$\text{phi len N} \quad = 0$
$\text{phi len (C x xs)} = 1 + \text{len xs}$

$\text{lenList} = \text{Mu}\ (L\ x) \to \text{Int}$
$\text{lenList} = \text{mcata phi}$

- Key idea: **phi** has additional argument
- No more requirement on the base structure being a positive functor
- Higher rank polymorphism ($\forall r.$ …) enforce recursive subcomponents in the base structure ($f\ r$) be abstract inside **phi**
- That is, $\text{len} :: r \to \text{Int}$ can only be applied to $xs :: r$
- Guarantee termination for negative datatypes too!
  (intuition: r cannot escape **phi**)

from Conventional to Mendler style – key changes

```
cata φ (Roll x) =
  φ (fmap (cata φ) x)
-- Conventional

lenList = cata phi
  where
    phi N            = 0
    phi (C x xslen) = 1 + xslen
```

```
mcata φ (Roll x) =
  φ (mcata φ) x
-- Mendler style

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

- Instead of **fmap**, let programmer handle where recursive call happens inside **phi**
- Enable this by generalizing **phi**, which is under the programmer control
- i.e., **phi** becomes a function of two arguments

from Conventional to Mendler style – is this change safe?

```
cons :: p → Mu (L x) → Mu (L x)
cons x xs = Roll (C x xs)

-- Uh-oh, is Mendler style safe?
lenList = mcata phi
  where
    phi len N        = 0
    phi len (C x xs) = 1 + len (cons x xs)
```

```
mcata φ (Roll x) = φ (mcata φ) x
-- Okay, this terminates
lenList = mcata phi
  where
    phi len N        = 0
    phi len (C x xs) = 1 + len xs
```

- Does **mcata** guarantee termination?
- What if the programmer try to invoke recursive call on non-decreasing values in **phi** ?

from Conventional to Mendler style – Mendler's trick

```
mcata :: ((Mu f → a) → f (Mu f) → a) →
             Mu f → a

-- Naive type ... bad
lenList = mcata phi  where
   phi :: (Mu (L x) → Int) → L x (Mu (L x)) → Int
   phi len N          = 0
   phi len (C x xs) = 1 + len (cons x xs)
```

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

-- Mendler's type
lenList = mcata phi  where
   phi :: ∀r.(r → Int) → L x r → Int
   phi len N          = 0
   phi len (C x xs) = 1 + len xs
```

- len (cons x xs) is a type error with Mendler's type
  - cons expects its 2nd arg to be 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 of cons is Mu (L x)

Can't do cons with xs

Won't work anyway even if you could

from Conventional to Mendler style – Mendler's trick

```
mcata :: ((Mu f → a) → f (Mu f) → a) →
          Mu f → a

-- Naive type ... bad
lenList = mcata phi  where
   phi :: (Mu (L x) → Int) → L x (Mu (L x)) → Int
   phi len N          = 0
   phi len (C x xs) = 1 + len (cons x xs)
```

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

-- Mendler's type
lenList = mcata phi  where
   phi :: ∀r.(r → Int) → L x r → Int
   phi len N          = 0
   phi len (C x xs) = 1 + len xs
```

- len (cons x xs) is a type error with Mendler's type
  - cons expects its 2nd arg to be 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 of cons is Mu (L x)

Can't do cons with xs

Won't work anyway even if you could

Impredicative Encodings of Recursive types in System F

- Encodings of non-recursive types
  - $0 \equiv \forall a.\ a$ -- void
  - $1 \equiv \forall a.\ a \to a$ -- unit
  - $A \times B \equiv \forall a.\ A \to B \to a$ -- pair
  - $A + B \equiv \forall a.\ (A \to a) \to (B \to a) \to a$ -- sum
- Encodings of recursive types
  - $\mu X.1+X \equiv \forall a.\ a \to (a \to a) \to a$ -- nat
- In a richer calculus like Fw (System F extended with type level functions), we can encode the recursive operator (μ)

Normalization Proof for Mendler style Catamorphism

```
newtype Mu f = Roll (f (Mu f))

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

- Normalization proof done by embedding into Fw
  - **Mu** and **Roll** can be defined in Fw
  - **mcata** can be defined in Fw
  - Fw is normalizing

Q.E.D.

(details in the ICFP paper)

- Note, **unRoll** is not embeddable into Fw
  - This is expected since unrestricted use of **unRoll** is problematic

Ability to freely access recursive subcomponents

Other Mendler style iteration/recursion combinators

- **msfcata**: a more expressive catamorphism (especially for negative datatypes)
  - Concept studied in conventional style
  - Our contribution: formulated in Mendler style and proof of normalization by embedding into Fw
- **mhist**: course of values recursion combinator
  - Known to work for positive datatypes (generalized proof for monotone type constructors are still an open question)
  - Our contribution: counterexample, showing that it does not work for negative datatypes

Motivating example for msfcata: Count Lambda's in HOAS

- A total function, but not structurally recursive
- cannot easily be defined with **mcata**

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

countLam :: Exp → Int

countLam (Lam f)     = countLam (f  (MAGIC 1))
countLam (App e e’) = countLam e + countLam e’
```

Motivating example for msfcata: Count Lambda's in HOAS

- A total function, but not structurally recursive
- cannot easily be defined with **mcata**

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

countLam :: Exp → Int

countLam (Lam f)     = countLam (f  (MAGIC 1))
countLam (App e e’) = countLam e + countLam e’
```

Motivating example for msfcata: Count Lambda's in HOAS

- A total function, but not structurally recursive
- MAGIC is a syntactic inverse from Int to Exp

```
data Exp = Lam (Exp → Exp) | App Exp Exp | MAGIC  Int

countLam :: Exp → Int
countLam (MAGIC n) = n
countLam (Lam f)      = countLam (f  (MAGIC 1))
countLam (App e e') = countLam e + countLam e'
```

- But what if we want to write another function from Exp to String?

Capturing the common pattern: add syntactic inverse to datatypes

- Inverse for a specific one level type

```
data Exp a = Lam (Exp → Exp) | App Exp Exp | Inverse a
```

- Generic Inverse for every two-level type
  - factored the Inverse into the datatype fixpoint

```
data Mu’ f a = Roll’ (f (Mu’ f)) | Inverse a
```

Capturing the common pattern: add syntactic inverse to datatypes

- Inverse for a specific one level type

```
data Exp a = Lam (Exp → Exp) | App Exp Exp | Inverse a
```

- Generic Inverse for every two-level type
  - factored the Inverse into the datatype fixpoint

```
data Mu' f a = Roll' (f (Mu' f)) | Inverse a
```

Defintion of msfcata

```
data Mu' f a = Roll' (f (Mu' f)) | Inverse a
unRoll' (Roll' e) = e

msfcata :: (∀r. (a→r a) → (r a→a) → f (r a) → a) →
              (∀a. Mu' f a) → a
msfcata φ (Roll' x)          =  φ  Inverse  (msfcata φ)  x
msfcata φ (Inverse ans) =  ans
```

- Yet another argument (abstract inverse) for phi
  - Phi in **msfcata** has 3 arguments, which is one more than the phi of **mcata**
- Termination proof done by embedding into Fw (details in our ICFP paper)

Formating HOAS into String

```
data ExpF r = A r r | L (r → r)
type Exp = forall a . Mu' f a

showExp :: Exp -> String
showExp e = msfcata  phi e vars
  where
    Phi :: (([String]→String) → r)  →  (r → ([String]→String)) →
           ExpF r → ([String]→String)
    phi inv show' (A x y) = \vs      →  "("++ show' x vs ++" "
                                         ++ show' y vs ++")"
    phi inv show' (L z)     = \(v:vs)→  "(\\" ++ v ++ "->"
                                     ++ show' (z (inv (const v))) vs
                                     ++ ")"
```

Mendler style generalize naturally to type constructors of higher rank

- What are type constructors of higher-rank?
  - Non-regular datatypes (e.g. powerlist, bush)
  - Indexed datatypes (or, GADTs)
- **Mu** and the combinators are indexed by kind
  - What we have seen in this talk is for kind $*$ only
    - $\mathbf{mcata}_*$ on $\mathbf{Mu}_*$, $\mathbf{mcata}_{*\to*}$ on $\mathbf{Mu}_{*\to*}$, …
    - $\mathbf{msfcata}_*$ on $\mathbf{Mu}_*$, $\mathbf{msfcata}_{*\to*}$ on $\mathbf{Mu}_{*\to*}$, …
  - For the same family of combinators, their definitions are exactly the same, but only their type signatures become more complex
  - See our ICFP paper for details

Summary on Mendler style

- Mendler style is very expressive
  - catamorhpism is well-defined for any datatype
- Mendler style generalizes well
  - naturally for nested datatypes and GADTs
  - discover new combinators by adding args to **phi**
- Some Mendler style recursion combinators have well known termination properties

  - We proved it for our new **msfcata** combinator, and
  - found counterexample for **mhist** on negative datatypes
- Could be a practical tool (if the language implementation supports higher-rank polymorhpism) for building generic programming libraries over non-regular datatypes (nested datatypes, GADTs)

Related Work

- Catamorphism for datatypes with embedded functions including negative datatypes has been studied in conventional setting by
  - Meijer & Hutton (FPCA 1995)
  - Fegaras & Sheard (ICFP 1996)
  - Washburn & Weirich (ICFP 2003)
- Matthes, Uustalu, and others
  - discovered that Mendler style works for negative datatypes (Mendler himself didn't notice it)
  - case studies of **mcata** on nested datatypes
- Despeyroux, Pfenning, and others
  - Primitive recursion on HOAS in a modal λ-calculus
- Induction over HOAS as induction over context

Future Work

- More powerful Mendler style recursion that guarantee termination
  - e.g. Work of Pfenning et. al. can express parallel reduction, which I conjecture somewhat refined version of mhist.
    The "Boxes go Bananas" paper has an Fw encoding of Pfenning et. al., so I should try whether it can be formulated in Mendler style
- Language (calculi) design
  - Track termination behaviors in the presence of negative datatypes
  - Extending Mendler style to dependent types

Some Important Questions

- What do “inductive type” and “recursive type” mean?
- When do recursive types coincide with inductive types?

syntactic

  - Strictly positive datatypes

semantic

  - Monotonicity (by Matthes)

somewhere in between

- How do we ensure or prove normalization?
  - Inductive types and positive types
    : usually rely on principled recursion
    (e.g. structural recursion, primitive recursion)
  - Recursive types including negative datatypes
    : can use Mendler style iteration
- Language design
  - Many open issues (e.g. dependent types) here

Language Design: needs

In the context of Trellys project, we want to have

- All the programs in typed functional languages (called ***programming fragment*** in Trellys)
- All the propositions & proofs in proof assistants (called ***logical fragment*** in Trellys)
- Abilities to
  - Construct proofs that refer to programs
  - Write programs that compute over values containing proofs
- Dependent types
  - For programming (e.g. generic programming, datatypes with static constraints like GADTs, ...)
  - And, for rich logic (indexed propositions)
- Erasable arguments
  - for efficient compilation (especially for proofs)

Recursive types

Inductive types

the bridging calculi

Language Design: challenges

- Design of the bridging calculi
  - Designing new calculi always have challenges
  - I plan to look into libraries/systems for reasoning in the REC $\perp$ fragment, which is build on top of systems of the IND fragment. (one way bridge, not easy to cross back)
- Dependent types can conflict with parametric polymorphism
  - Mendler style combinators rely on parametricity to restrict the recursive subcomponents be abstract in the combining function **phi**
  - Value dependency on the answer type (T v) to the recursive argument (v) can conflict with such use of parametricity
  - I have some preliminary thoughts that Erasable arguments can help us resolve this
- More powerful terminating recursion combinators are always better
  - Hoping to discover new Mendler style recursion combinators

Research Plan

- Part I (introduction & background) [Nov 2012]
  - Motivation and Literature search
- Part II (positive datatypes) [Dec 2012]
  - Literature search focused on properties of positive datatypes (especially on termination properties)
  - Leads the discussion of Part III
- Part III (negative datatypes) [March 2012]
  - Literature search and our work on Mendler style iteration
  - Hoping to discover more powerful Mendler style iteration/recursion combinators ensuring termination
- Part IV (language design) [May 2012]
  - Develop the bridging calculi
  - Issues related to Dependent types and Erasable arguments
  - Case study of examples that work over more than one fragment