The Nax programming language

subtitle

Ki Yung Ahn   Tim Sheard[1]
Marcelo Fiore   Andrew M. Pitts[2]

[1]Department of Computer Science
Portland State University

[2]Computer Laboratory
University of Cambridge

the 24th Symposium on Implementation and Application of Functional Languages (IFL 2012)

Outline

1. Introduction
   - Motivation: Lightweight Approach - can we make it better?
   - Outline of our approach
   - Previous Work

2. The Nax programming language
   - Main Results
   - Code example

Outline

1. Introduction
   - Motivation: Lightweight Approach - can we make it better?
   - Outline of our approach
   - Previous Work
2. The Nax programming language
   - Main Results
   - Code example

Lightweight Approach

programming with lightweight dependent types in programming languages

- Maintain fine-grained program properties by rather moderate extension to the type system of functional programming languages (e.g. GADT extension in Haskell)
- May need less effort than verifying program properties with formal proof assistants based on full-spectrum dependent types
- Has gained popularity over the past decade but there are some shortcomings in practice

Lightweight Approach

programming with lightweight dependent types in programming languages

- Maintain fine-grained program properties by rather moderate extension to the type system of functional programming languages (e.g. GADT extension in Haskell)
- May need less effort than verifying program properties with formal proof assistants based on full-spectrum dependent types
- Has gained popularity over the past decade but there are some shortcomings in practice

Lightweight Approach

programming with lightweight dependent types in programming languages

- Maintain fine-grained program properties by rather moderate extension to the type system of functional programming languages (e.g. GADT extension in Haskell)
- May need less effort than verifying program properties with formal proof assistants based on full-spectrum dependent types
- Has gained popularity over the past decade but there are some shortcomings in practice

Shortcomings of Lightweight Approach

- No (formal) correctness guarantee
  but only (semi-formal) confidence of program properties
  - programming language type systems were not designed for logical consistency
  - must rely on belief that inconsistent features were not involved in reasoning about program properties
- Faked-term indices in implementations (until recently)
  - duplication of the term structure at type level

```
data Zero; data Succ n -- cannot prevent (Succ Bool)
data Vec a n where { Nil ::Zero;
                     Cons::a -> Vec a i -> Vec Succ i }
```

- Type inference
  - papers on GADT type inference being published every year
  - Do need annotation obliviously, but *how much* and *where*?

Shortcomings of Lightweight Approach

- No (formal) correctness guarantee
  but only (semi-formal) confidence of program properties
  - programming language type systems were not designed for logical consistency
  - must rely on belief that inconsistent features were not involved in reasoning about program properties
- Faked-term indices in implementations (until recently)
  - duplication of the term structure at type level

```
data Zero; data Succ n -- cannot prevent (Succ Bool)
data Vec a n where { Nil ::Zero;
                     Cons::a -> Vec a i -> Vec Succ i }
```

- Type inference
  - papers on GADT type inference being published every year
  - Do need annotation obliviously, but *how much* and *where*?

Shortcomings of Lightweight Approach

- No (formal) correctness guarantee
  but only (semi-formal) confidence of program properties
  - programming language type systems were not designed for logical consistency
  - must rely on belief that inconsistent features were not involved in reasoning about program properties
- Faked-term indices in implementations (until recently)
  - duplication of the term structure at type level

```
data Zero; data Succ n -- cannot prevent (Succ Bool)
data Vec a n where { Nil ::Zero;
                     Cons::a -> Vec a i -> Vec Succ i }
```

- Type inference
  - papers on GADT type inference being published every year
  - Do need annotation obliviously, but *how much* and *where*?

Outline

1. Introduction
   - Motivation: Lightweight Approach - can we make it better?
   - Outline of our approach
   - Previous Work
2. The Nax programming language
   - Main Results
   - Code example

Outline of our approach

- Formalize the idea of erasable term-indices by extending well-studied polymorphic lambda calculi
  - terms can appear in types but with phase distinction (i.e. not dependent on variables from term abstraction)
- Study on datatypes and their principled recursion schemes convenient for programming as well as logical reasoning
  - Mendler style
- Design a programming language based on the theory with adequate abstraction level (e.g. support datatypes and recursion schemes) and usability (e.g. type inference)
  - design a strongly normalizing and logically consistent language
  - later add non-logical features safely (ideas from Trellys project)

Outline

1. Introduction
   - Motivation: Lightweight Approach - can we make it better?
   - Outline of our approach
   - Previous Work
2. The Nax programming language
   - Main Results
   - Code example

Mendler-style recursion combinators

comparison of two styles – iteration (a.k.a catamorphism)

newtype μ_* (f :: $* \to *$) = In { unIn :: f (μ_*f) }

Oxford style

iter :: Functor f $\Rightarrow$ (f a $\to$ a)
$\to \mu_*$ f $\to$ a
iter $\varphi = \varphi$ (fmap (iter φ)) $\circ$ unIn

Mendler style

miter :: ($\forall$r. (r $\to$ a) $\to$ f r $\to$ a)
$\to \mu_*$ f $\to$ a
miter $\varphi = \varphi$ (miter φ) $\circ$ unIn

- Polytypic definition –
  fmap defined for each functor f
- Normalization relies on meta-properties of functor being positive, etc.
- Generalization to higher-kinds needs some thinking (e.g., gfold, efold for nested datatypes)

- Polymorphic definition – requires higher-rank polymorphism
- Type system guarantees normalization – miter can be embedded into System F_ω
- Immediately generalizes to higher-kinds ($* \to *$, $\texttt{Nat} \to *$, ...)

Mendler-style recursion combinators

comparison of two styles – iteration (a.k.a catamorphism)

newtype μ_* (f :: $* \to *$) = In { unIn :: f (μ_*f) }

Oxford style

iter :: Functor f $\Rightarrow$ (f a $\to$ a)
$\quad\to \mu_*$ f $\to$ a
iter $\varphi = \varphi$ (fmap (iter φ)) $\circ$ unIn

Mendler style

miter :: ($\forall$r. (r $\to$ a) $\to$ f r $\to$ a)
$\quad\to \mu_*$ f $\to$ a
miter $\varphi = \varphi$ (miter φ) $\circ$ unIn

- Polytypic definition – fmap defined for each functor f
- Normalization relies on meta-properties of functor being positive, etc.
- Generalization to higher-kinds needs some thinking (e.g., gfold, efold for nested datatypes)

- Polymorphic definition – requires higher-rank polymorphism
- Type system guarantees normalization – miter can be embedded into System F_ω
- Immediately generalizes to higher-kinds ($* \to *$, $\texttt{Nat} \to *$, ...)

Mendler-style recursion combinators

several families of combinators

types of abstract operations

$\mathrm{CALL} \triangleq (\mathrm{r} \to \mathrm{a})$ $\mathrm{CAST} \triangleq (\mathrm{r} \to \mu \mathrm{f})$ $\mathrm{OUT} \triangleq (\mathrm{r} \to \mathrm{f\ r})$ $\mathrm{INV} \triangleq (\mathrm{a} \to \mathrm{r})$

Each family supports a different set of abstract operations

- miter len (Cons x xs) = len xs + 1
- mprim fac (S n) = fac n * (S n)
- mcvit fib (S(S n)) = fib (S n) * fib n
- mcvpr luc (S(S n)) = luc (S n) * luc n + n
- msfit printing HOAS into string (Ahn & Sheard 2011)

$\vdots$

"cv" stands for course-of-values "sf" stands for Sheard & Fegaras

Mendler-style recursion combinators

several families of combinators

types of abstract operations

$\mathsf{CALL} \triangleq (r \to a)$ $\mathsf{CAST} \triangleq (r \to \mu f)$ $\mathsf{OUT} \triangleq (r \to f\ r)$ $\mathsf{INV} \triangleq (a \to r)$

Each family supports a different set of abstract operations

- miter :: $(\forall r.$ $\mathsf{CALL} \to f\ r \to a) \to \mu f \to a$
- mprim fac (S n) = fac n * (S n)
- mcvit fib (S(S n)) = fib (S n) * fib n
- mcvpr luc (S(S n)) = luc (S n) * luc n + n
- msfit printing HOAS into string (Ahn & Sheard 2011)

⋮

"cv" stands for course-of-values "sf" stands for Sheard & Fegaras

Mendler-style recursion combinators

several families of combinators

types of abstract operations

$\mathsf{CALL} \triangleq (r \to a)$ $\quad \mathsf{CAST} \triangleq (r \to \mu f)$ $\quad \mathsf{OUT} \triangleq (r \to f\ r)$ $\quad \mathsf{INV} \triangleq (a \to r)$

Each family supports a different set of abstract operations

- miter :: $(\forall r.\quad \mathsf{CALL} \to f\ r \to a) \to \mu f \to a$
- mprim :: $(\forall r.\quad \mathsf{CAST} \to \mathsf{CALL} \to f\ r \to a) \to \mu f \to a$
- mcvit fib (S(S n)) = fib (S n) * fib n
- mcvpr luc (S(S n)) = luc (S n) * luc n + n
- msfit printing HOAS into string (Ahn & Sheard 2011)

⋮

"cv" stands for course-of-values

"sf" stands for Sheard & Fegaras

Mendler-style recursion combinators

several families of combinators

types of abstract operations

$\mathrm{CALL} \triangleq (r \to a)$ $\mathrm{CAST} \triangleq (r \to \mu f)$ $\mathrm{OUT} \triangleq (r \to f\ r)$ $\mathrm{INV} \triangleq (a \to r)$

Each family supports a different set of abstract operations

- miter :: $(\forall r.\ \mathrm{CALL} \to f\ r \to a) \to \mu f \to a$
- mprim :: $(\forall r.\ \mathrm{CAST} \to \mathrm{CALL} \to f\ r \to a) \to \mu f \to a$
- mcvit :: $(\forall r.\ \mathrm{OUT} \to \mathrm{CALL} \to f\ r \to a) \to \mu f \to a$
- mcvpr luc (S(S n)) = luc (S n) * luc n + n
- msfit printing HOAS into string (Ahn & Sheard 2011)

⋮

"cv" stands for course-of-values "sf" stands for Sheard & Fegaras

Mendler-style recursion combinators

several families of combinators

types of abstract operations

$\mathsf{CALL} \triangleq (\mathsf{r} \to \mathsf{a})$ $\quad \mathsf{CAST} \triangleq (\mathsf{r} \to \mu\mathsf{f})$ $\quad \mathsf{OUT} \triangleq (\mathsf{r} \to \mathsf{f\ r})$ $\quad \mathsf{INV} \triangleq (\mathsf{a} \to \mathsf{r})$

Each family supports a different set of abstract operations

- miter :: $(\forall \mathsf{r}.\ \mathsf{CALL} \to \mathsf{f\ r} \to \mathsf{a}) \to \mu\mathsf{f} \to \mathsf{a}$
- mprim :: $(\forall \mathsf{r}.\ \mathsf{CAST} \to \mathsf{CALL} \to \mathsf{f\ r} \to \mathsf{a}) \to \mu\mathsf{f} \to \mathsf{a}$
- mcvit :: $(\forall \mathsf{r}.\ \mathsf{OUT} \to \mathsf{CALL} \to \mathsf{f\ r} \to \mathsf{a}) \to \mu\mathsf{f} \to \mathsf{a}$
- mcvpr :: $(\forall \mathsf{r}.\ \mathsf{OUT} \to \mathsf{CAST} \to \mathsf{CALL} \to \mathsf{f\ r} \to \mathsf{a}) \to \mu\mathsf{f} \to \mathsf{a}$
- msfit printing HOAS into string (Ahn & Sheard 2011)

$\vdots$

"cv" stands for course-of-values

"sf" stands for Sheard & Fegaras

Mendler-style recursion combinators

several families of combinators

types of abstract operations

$\mathsf{CALL} \triangleq (r \to a) \quad \mathsf{CAST} \triangleq (r \to \mu f) \quad \mathsf{OUT} \triangleq (r \to f\ r) \quad \mathsf{INV} \triangleq (a \to r)$

Each family supports a different set of abstract operations

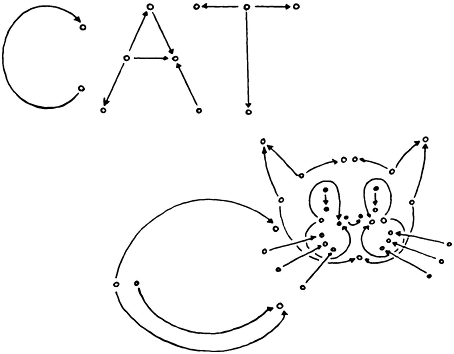
- miter :: $(\forall r.\ \mathsf{CALL} \to f\ r \to a) \to \mu f \to a$
- mprim :: $(\forall r.\ \mathsf{CAST} \to \mathsf{CALL} \to f\ r \to a) \to \mu f \to a$
- mcvit :: $(\forall r.\ \mathsf{OUT} \to \mathsf{CALL} \to f\ r \to a) \to \mu f \to a$
- mcvpr :: $(\forall r.\ \mathsf{OUT} \to \mathsf{CAST} \to \mathsf{CALL} \to f\ r \to a) \to \mu f \to a$
- msfit :: $(\forall r.\ \mathsf{INV} \to \mathsf{CALL} \to f\ r \to a) \to \mu f \to a$

miter, msfit normalize for arbitrary f – embeddable into System F_ω
mprim normalize for positive f – embeddable into System Fix_ω
mcv$**$ normalize for positive f – should be embeddable into $F_\omega, \mathit{Fix}_\omega$

Mendler-style recursion combinators

why do I care about embeddings into polymorphic lambda calculi?

I have problems dealing with



(img from The Joy of Cats)

But, I have faith in things like

What the F_ω?

(Var) $\Gamma, x : \tau \vdash x : \tau$

(Abs) $\dfrac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x. M) : \sigma \to \tau}$

(App) $\dfrac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}$

(Gen) $\dfrac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall \alpha \sigma}$ $(\alpha \notin \mathrm{FV}(\Gamma))$

(Inst) $\dfrac{\Gamma \vdash M : \forall \alpha^\kappa \sigma}{\Gamma \vdash M : \sigma[\alpha := \varphi]}$ (φ is of kind κ)

(Conv) $\dfrac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \tau}$ $(\sigma =_\beta \tau)$

(img from http://bartdesmet.net/blogs/bart/)

Lambda Calculi with Erasable Term Indices

- Minimal extensions to higher-order polymorphic lambda calculi to support true term indices

System F_i extended from System F_ω

$$\kappa \;::=\; * \mid \kappa \to \kappa \mid A \to \kappa \qquad \boxed{\vdash \kappa}$$

$$F, G, A, B \;::=\; \cdots \mid \lambda i^A.F \mid F\{s\} \mid \forall i^A.F \qquad \boxed{\Delta \vdash F : \kappa}$$

$$t, s \;::=\; x \mid \lambda x.t \mid t\ s \qquad \boxed{\Delta;\Gamma \vdash t : A}$$

- Strongly normalizing (by index erasure into F_ω) and Logically consistent ($\because$ subset of logically consistent subset of ICC)
- Further details in
  - poster in ACM-SRC hosted by ICFP 2012
  - draft submitted to POPL 2013
- can similarly extend System Fix_ω into System Fix_i

Outline

1. Introduction
   - Motivation: Lightweight Approach - can we make it better?
   - Outline of our approach
   - Previous Work
2. The Nax programming language
   - Main Results
   - Code example

The Nax programming language – Overview

We have background theories (e.g. System F_i) that

- formalizes the idea of erasable term indices
- strongly normalizing and logically consistent
- can embed datatypes and Mendler-style recursion schemes

The Nax programming language (named after after Nax P. Mendler)

- supports datatypes and Mendler-style recursion schemes
- semantics understood by embedding into background theory
- clear design on *how much* and *where* annotations are needed
  - no annotations for regular datatypes (exactly the same as HM)
  - annotations for non-regular datatypes at declaration (using GADT-like syntax) and elimination (i.e., pattern matching)
  - kind annotation for **Mu** and **In**

Outline

```
data Tag = E | O

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} → P r {O} {succ i}

synonym Proof t n = Mu(Tag→Nat→*) P t n

synonym Even n = Proof {E} n
base    = In(Tag→Nat→*) Base
stepO x = In(Tag→Nat→*) (StepO x)

synonym 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   f Base      = stepE base
                          f (StepO p) = stepE (f p)
                          f (StepE p) = stepO (f p)

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

```
data Tag = E | O

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} → P r {O} {succ i}
  deriving fixpoint Proof
-- synonym Proof t n = Mu(Tag → Nat → *) P t n

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

synonym 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   f Base       = stepE base
                          f (StepO p) = stepE (f p)
                          f (StepE p) = stepO (f p)

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

Summary

- We know how to extend higher-order polymorphic lambda calculi with erasable term-indices, which maintain desirable properties
- Mendler-style recursion schemes over indexed datatypes are embeddable these calculi – only need to believe in those calclui
- Nax is a programming language built on top of the theories above – programming with GADTs with
    - real term indices – no code duplication at type and term level
    - formal correctness guarantee
    - clearly understandable type reconstruction (or, partial type inference)

Outlook

- Ongoing work
  - writing down embedding of course-of-values iteration & recursion and try some examples over higher-kinded datatypes
  - write down typing rules & type inference algorithm of Nax and prove their correspondence
  - Fix_ω is strongly normalizing but is there a superset logically consistent calculi?
- Future work
  - try to reduce annotations further
    (e.g., kind annotations on Mu and In may be inferable)
  - let polymorphism for kinds may not be harmful
    (since HM is STLC by let-inlining)
  - More large eliminations in Nax?
    (e.g., `{ x . if x then Nat else (Nat→Nat) }`)