Towards Higher-Order Abstract Syntax in Cedille

Work in Progress

Aaron Stump
Computer Science
The University of Iowa
Iowa City, Iowa

LFMTP 2019

HOAS, the long road

- From higher-order constructs for quantification [Church 1940]
- To second-order rewrite rules [Huet and Lang 1978],
- To identification of HOAS [Pfenning and Elliot 1988]
- Edinburgh LF [Harper, Honsell, Plotkin 1993]
- Systems like
  - Twelf, λProlog, Beluga/Cocon, Abella, Dedukti
  - Definitional approaches (Hybrid, Nominal Isabelle)
- Benchmarks like POPLmark, ORBI [Felty et al. 2015]

It would be so great to have HOAS in a proof assistant!

For this, we seek HOAS with an induction principle

A beautiful wish

Isn't there hope of HOAS in a pure dependent type theory?

A beautiful wish

Isn't there hope of HOAS in a pure dependent type theory?

After all, we can Church encode lambda terms (in $\lambda 2$):

$$\mathit{Trm} \;:=\; \forall X : \star.\, ((X \to X) \to X) \to (X \to X \to X) \to X$$

A beautiful wish

Isn't there hope of HOAS in a pure dependent type theory?

After all, we can Church encode lambda terms (in $\lambda 2$):

$$Trm \;:=\; \forall X : \star.\; ((X \to X) \to X) \to (X \to X \to X) \to X$$

E.g., represent (object-language) $\lambda x.\; x\; x$ as

$$\lambda l.\; \lambda a.\; l\; (\lambda x.\; (a\; x\; x))$$

Similarly to Church-encoding 2 as

$$\lambda s.\; \lambda z.\; s\; (s\; z)$$

The problem: constructors

For a polynomial datatype, like

$$Nat \;:=\; \forall X : \star.\ (X \to X) \to X \to X$$

constructors are easily defined:

$$\begin{array}{lclcl} Zero & : & Nat & := & \lambda s.\ \lambda z.\ z \\ Zero & : & Nat \to Nat & := & \lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z) \end{array}$$

The problem: constructors

For a polynomial datatype, like

$$Nat \ := \ \forall X : \star.\ (X \to X) \to X \to X$$

constructors are easily defined:

$$\begin{array}{lclcl} Zero & : & Nat & := & \lambda s.\ \lambda z.\ z \\ Zero & : & Nat \to Nat & := & \lambda n.\ \lambda s.\ \lambda z.\ s\,(n\ s\ z) \end{array}$$

Not so for *Trm*:

$$\begin{array}{lclcl} App & : & Trm \to Trm \to Trm & := & \lambda t.\ \lambda t'.\ \lambda l.\ \lambda a.\ a\,(t\ l\ a)\,(t'\ l\ a) \\ Lam & : & (Trm \to Trm) \to Trm & := & ? \end{array}$$

Constructors?

Washburn and Weirich [2008] give an encoding, maybe a constructor?

$$\begin{array}{lcll} \mathit{Trma} & : & \star \to \star & := \quad \lambda X : \star.\, ((X \to X) \to X) \to (X \to X \to X) \to X \\ \mathit{lam} & : & \forall X : \star.(\mathit{Trma}\, X \to \mathit{Trma}\, X) \to \mathit{Trma}\, X & := \ldots \end{array}$$

Sadly, no

For <u>inductive</u> encodings, foundation is initial algebras

Sadly, no

For <u>inductive</u> encodings, foundation is initial algebras

Given an endofunctor *F* on category *C*,

define category with algebras (A,m) as objects:

$$F\ A \xrightarrow{m} A$$

$$F\ A' \xrightarrow{m'} A'$$

Sadly, no

For inductive encodings, foundation is initial algebras

Given an endofunctor *F* on category *C*,

define category with algebras (A,m) as objects:

$$F\ A \xrightarrow{m} A$$

$$F\ A' \xrightarrow{m'} A'$$

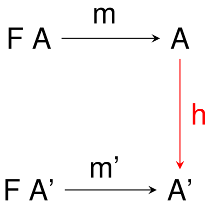
and algebra homomorphisms *h* as morphisms.

Sadly, no

For <u>inductive</u> encodings, foundation is initial algebras

Given an endofunctor *F* on category *C*,

define category with algebras (A,m) as objects:



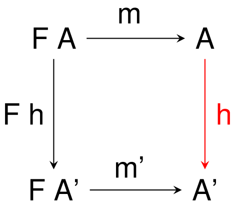
and algebra homomorphisms *h* as morphisms.

Sadly, no

For inductive encodings, foundation is initial algebras

Given an endofunctor *F* on category *C*,

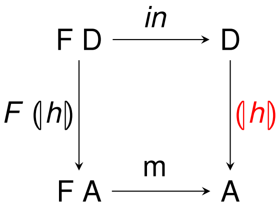
define category with algebras (A,m) as objects:



and algebra homomorphisms *h* as morphisms.

Initial algebras

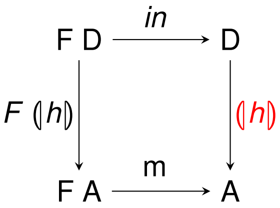
An initial object (D, in) in the category of algebras



So we need *in* of type $F\ D \rightarrow D$

Initial algebras

An initial object (D, in) in the category of algebras



So we need *in* of type $F\ D \to D$

For *Trm*, with $F\ X = X \to X$ (*eliding application*), need

$$in\ :\ F\ Trm \to Trm$$

Initial algebras

An initial object (D, in) in the category of algebras

$$\begin{array}{ccc} F\ D & \xrightarrow{in} & D \\ \downarrow{\scriptstyle F\,(\!|h|\!)} & & \downarrow{\scriptstyle (\!|h|\!)} \\ F\ A & \xrightarrow{m} & A \end{array}$$

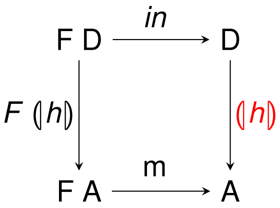
So we need *in* of type $F\ D \to D$

For *Trm*, with $F\ X = X \to X$ (*eliding application*), need

$$in\ :\ (Trm \to Trm) \to Trm$$

Initial algebras

An initial object (D, in) in the category of algebras



So we need *in* of type $F\ D \to D$

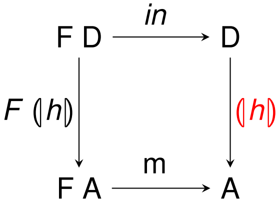
For *Trm*, with $F\ X = X \to X$ (*eliding application*), need

$$in\ :\ (Trm \to Trm) \to Trm$$

So the Washburn-Weirich definition will not work...

Initial algebras

An initial object (D, in) in the category of algebras



So we need *in* of type $F\ D \to D$

For *Trm*, with $F\ X = X \to X$ (*eliding application*), need

$$in\ :\ (Trm \to Trm) \to Trm$$

So the Washburn-Weirich definition will not work...
... but their idea of using polymorphism can

Changing the notion of algebra

We saw so far:

$$Alg \;:=\; \lambda X : \star.(X \to X) \to X$$

$$Trm \;:=\; \forall X : \star.\; Alg\; X \to X$$

Let us try to find an alternative definition of *Alg*

Changing the notion of algebra

We saw so far:

$$\mathit{Alg} \; := \; \lambda X : \star.(X \to X) \to X$$

$$\mathit{Trm} \; := \; \forall X : \star.\ \mathit{Alg}\ X \to X$$

Let us try to find an alternative definition of *Alg*

A useful tool: positive-recursive types; e.g. Scott-encoded nats:

$$\mathit{SNat} = \forall X : \star.(\mathit{SNat} \to X) \to X \to X$$

Adjoining indeterminates

Drawing inspiration from [Selinger 2002],

think of λ as introducing a new constructor, for the bound var.

$$\mathit{Trmga} \;:=\; \lambda \mathit{Alg} : \star \to \star.\ \lambda Y : \star.\ (\mathit{Alg}\ Y \to Y) \to Y$$

Adjoining indeterminates

Drawing inspiration from [Selinger 2002],

think of λ as introducing a new constructor, for the bound var.

$$\mathit{Trmga} \;:=\; \lambda \mathit{Alg} : \star \to \star.\ \lambda Y : \star.\ (\mathit{Alg}\ Y \to Y) \to Y$$

$$\mathit{Alg} \;=\; \lambda X : \star.\ (\forall\, Y{:}\star.\ Y \to \mathit{Trmga}\ \mathit{Alg}\ Y) \to X$$

An algebra takes in a subterm for the body,
which may use an addition input of abstracted type Y

Adjoining indeterminates

Drawing inspiration from [Selinger 2002],

think of λ as introducing a new constructor, for the bound var.

$$Trmga \;:=\; \lambda Alg : \star \to \star.\ \lambda Y : \star.\ (Alg\ Y \to Y) \to Y$$

$$Alg \;=\; \lambda X : \star.\ (\forall\, Y{:}\star.\ Y \to Trmga\ Alg\ Y) \to X$$

An algebra takes in a subterm for the body,
which may use an addition input of abstracted type Y

But: definition of *Alg* is negative-recursive!

We will fix this shortly...

Problem: building up data incrementally

With what we have so far:

The bound variable of a λ-abstraction is over a new type Y

Nested abstractions like $\lambda x.\ \lambda y.\ x$ cannot be built incrementally

- ▷ Body of $\lambda y.\ x$ must be over *second* abstracted type

Going under a λ is like entering a new world...

Problem: building up data incrementally

With what we have so far:

The bound variable of a λ-abstraction is over a new type Y

Nested abstractions like $\lambda x.\ \lambda y.\ x$ cannot be built incrementally

▷ Body of $\lambda y.\ x$ must be over *second* abstracted type

Going under a λ is like entering a new world...

But one reachable from the current one

Kripke function spaces

We need to relate old and new worlds

The new (Y) must be reachable from the old (X): $X \to Y$

$$\begin{aligned} \mathit{Trmga} &:= \lambda\, \mathit{Alg}{:}\star \to \star.\, \lambda\, X{:}\star.\, \mathit{Alg}\, X \to X \\ \mathit{Alg} &= (\forall\, Y{:}\star.\, (X \to Y) \to Y \to \mathit{Trmga}\ \mathit{Alg}\ Y) \to X \end{aligned}$$

Kripke function spaces

We need to relate old and new worlds

The new (Y) must be reachable from the old (X): $X \to Y$

$$\begin{aligned} \mathit{Trmga} &:= \lambda\, \mathit{Alg} : \star \to \star.\, \lambda\, X : \star.\, \mathit{Alg}\, X \to X \\ \mathit{Alg} &= (\forall\, Y : \star.\, (X \to Y) \to Y \to \mathit{Trmga}\, \mathit{Alg}\, Y) \to X \end{aligned}$$

Not the final encoding, because no iteration

- ▷ Like a Scott encoding
- ▷ Amazing recent result: recursion for Scott encoding!
- ▷ Parigot, communicated in [Lepigre, Raffalli 2017]
- ▷ We will not try that here...

Final definition of *Alg*

Want the algebra to accept a copy of itself, for recursion

And let us eliminate that negative-recursion!

Can use Mendler's technique of abstracting negative occurrences:

$$\begin{aligned} \mathit{Alg} \;=\; & \forall\, \mathit{Alga}{:}\star \to \star.\ (\forall\, Y{:}\star.\,(X \to Y) \to Y \to \mathit{Trmga}\ \mathit{Alga}\ Y) \\ & (\forall X : \star.\ \mathit{Alg}\ X \to \mathit{Alga}\ X) \to \\ & \mathit{Alga}\ X \to \\ & X \end{aligned}$$

It is legal to hide the type of an *Alg*

Proceed, in Haskell

All we need is recursive types + impredicative polymorphism

```
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE ExplicitForAll #-}
{-# LANGUAGE RankNTypes #-}

type Trmga alg x = alg x -> x

newtype Alg x =
  MkAlg
   { unfoldAlg ::
       forall (alga :: * -> *) .
       (forall (y :: *) . (x -> y) -> y -> Trmga alga y) ->
       (forall (z :: *) . Alg z -> alga z) ->
       alga x ->
       x}

newtype Trm =
  MkTrm { unfoldTrm :: forall (x :: *) . Alg x -> x}
```

Finally, a weakly initial algebra!

```
lamAlg :: Alg Trm
lamAlg = MkAlg (\ f embed talg ->
            MkTrm (\ alg ->
               unfoldAlg alg
                 (\ mx ->
                    f (\ t -> mx (unfoldTrm t alg)))
                 embed
                (embed alg)))
```

In the body:

```
f       ::  forall (y :: *) . (x -> y) -> y -> Trmga alga y
embed   ::  forall (z :: *) . Alg z -> alga z
talg    ::  alga x
```

`lamAlg` switches the algebra from `talg` (itself) to `alg`

Example encoded term: $\lambda x.\ \lambda y.\ x$

```
place :: forall (x :: *) . x -> Trmga Alg x
place = \ x -> \ alg -> x

test :: Trm
test = MkTrm (lam (\ mo x ->
               lam (\ mx y -> place (mx x))))
```

A size function

```
size :: Trm -> Int
size = \ t -> unfoldTrm t
                (MkAlg (\ f embed alg -> 1 + f id 1 alg))
```

Can check with `ghci`:

```
*WeaklyInitialHoas> size test
3
```

Conversion to de Bruijn notation

```
data Dbtrm = Lam Dbtrm | Var Int deriving Show

toDebruijn :: Trm -> Int -> Dbtrm
toDebruijn t =
  unfoldTrm t (MkAlg (\ f embed alg -> \ v ->
                 let v' = v + 1 in
                   Lam (f id (\ n -> Var (n - v')) alg v')))
```

With `ghci`:

```
*WeaklyInitialHoas> toDebruijn test 0
Lam (Lam (Var 1))
```

Converting `Trm` to `String`

```
vars :: Int -> [String]
vars n = ("x" ++ show n) : vars (n + 1)

printTrmH :: Trm -> [String] -> String
printTrmH t =
  unfoldTrm t (MkAlg (\ f embed alg vars ->
                 let x = head vars in
                   "\\ " ++ x ++ ". " ++
                   f id (\ vars -> x) alg (tail vars)))

printTrm :: Trm -> String
printTrm t = printTrmH t (vars 1)
```

With `ghci`:

```
*WeaklyInitialHoas> putStrLn $ printTrm test
\ x1. \ x2. x1
```

Back in Cedille...

- A notion of algebra homomorphism:

```
h (alg1 f alg1) ≃ alg2 (λ mx . f (λ a . mx (h a))) alg2
```

- Proven

```
foldTrm ◁ ∀ X : ★ . Alg · X → Trm → X =
  Λ X . λ alg . λ t . t alg.

foldHom : ∀ X : ★ . ∀ alg : Alg · X .
          IsHomomorphism · Trm lamAlg · X alg (foldTrm alg) =
```

Back in Cedille...

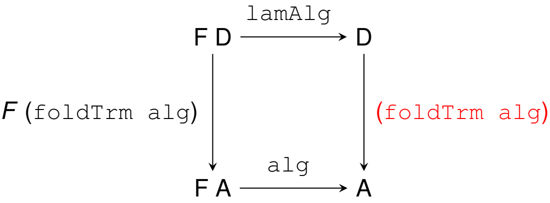
- A notion of algebra homomorphism:

```
h (alg1 f alg1) ≃ alg2 (λ mx . f (λ a . mx (h a))) alg2
```

- Proven

```
foldTrm ◁ ∀ X : ⋆ . Alg · X → Trm → X =
  Λ X . λ alg . λ t . t alg.

foldHom : ∀ X : ⋆ . ∀ alg : Alg · X .
          IsHomomorphism · Trm lamAlg · X alg (foldTrm alg) =
```



Conclusion

- Work in progress towards HOAS in Cedille
- Weakly initial algebra for HOAS
- Use parametric polymorphism, Kripke function spaces for
  - Bound variables as indeterminates
  - Incrementally constructed data
- Next step: induction via parametricity!

Acknowledgments:
Ernesto Copello
NSF 1524519, DoD FA9550-16-1-0082

AMDG