An Algebra of Dependent Data Types

TYPES 2006

Tyng-Ruey Chuang

Joint work with Jan-Li Lin

Institute of Information Science, Academia Sinica
Nangang, Taipei 115, Taiwan

trc@iis.sinica.edu.tw

List in Coq

```
Inductive List (A: Set): nat -> Set :=
  Nil:  List A O
| Cons: A -> forall (n: nat), List A n -> List A (1+n).

Fixpoint concat (A: Set) (m: nat) (p: List A m)
  (n: nat) (q: List A n) {struct q}: List A (n+m) :=
match q in List _ i return List _ (i+m) with
      Nil => p
    | Cons a n' q' => Cons A a (n'+m)  (concat A m p n' q')
end.
```

```
concat: forall (A : Set) (m : nat), List A m ->
        forall n : nat, List A n -> List A (n + m)
```

List in O'Caml

```
let rec concat p q =
    match q with
          [] -> p
        | h::t -> h::(concat p t)
```

```
concat : 'a list -> 'a list -> 'a list
```

Motivations

Given two data types S and T, we want a way to express

- (some) dependencies of S on T,
- dependencies in (some) functions from S to T.

We address only "regular" data types however:

- unit, sum, product, polynomial, and fixed point
- taking an initial F-algebra approach
- $\mathrm{List}_A = \mu X.F_A(X)$ where $F_A(X) = 1 + A \times X$

The Plan

- Use the arrow category to express dependencies between data types.
- Use natural transformations to characterize natural dependencies between data types.
- Define the dependency component in the inductive step for an inductive computation between data types.
- Formulate the above as abstract as possible; use initial $\mathcal{F}_\eta$-algebra.
- Recast the above to O'Caml to allow for generic and efficient run-time calculation of dependencies in inductive computations.

Initial F-algebra for Induction

$$\begin{tikzcd}
S \arrow[d, "(\!|f|\!)"'] & FS \arrow[l, "\alpha"'] \arrow[d, "F(\!|f|\!)"] \\
X & FX \arrow[l, "f"]
\end{tikzcd}$$

An F-algebra is called an initial F-algebra if it has an initial object (α, S). For any object (f, X), one uses the notation $(\!|f|\!)$ to denote the *unique* arrow $S \to X$ satisfying $(\!|f|\!) \circ \alpha = f \circ F(\!|f|\!)$.

Note that α is an isomorphism between S and FS. That is, we view a regular data type S as the fixed point of a polynomial endofunctor F.

Programming Initial F-algebra in O'Caml

```
type ('a, 'b) t = Nil | Cons of 'a * 'b
let map f t = match t with Nil -> Nil | Cons (a, b) -> Cons (a, f b)

type 'a list = Rec of ('a, 'a list) t
let rec fold f (Rec t) = f (map (fold f) t)

let concat p q =
    let f t = match t with Nil -> p | Cons (a, b) -> Rec (Cons (a, b))
 in
    fold f q
```

```
val map : ('a -> 'b) -> ('c, 'a) t -> ('c, 'b) t = <fun>
val fold : (('a, 'b) t -> 'b) -> 'a list -> 'b = <fun>
val concat : 'a list -> 'a list -> 'a list = <fun>
```

Arrow Category for Dependencies

objects:

$$\begin{tikzcd} X \arrow[d, "\varphi"'] \\ A \end{tikzcd}$$

arrows:

$$\begin{tikzcd} X \arrow[r, "h"] \arrow[d, "\varphi"'] & Y \arrow[d, "\psi"] \\ A \arrow[r, "k"'] & B \end{tikzcd}$$

Let $\mathbf{C}$ be a category, the *arrow category* of $\mathbf{C}$ is denoted as $\mathbf{C}^{\rightarrow}$. It has families $\varphi : X \rightarrow A$ as objects. For two objects $\varphi : X \rightarrow A$ and $\psi : Y \rightarrow B$, the arrows of $\mathbf{C}^{\rightarrow}$ from $\varphi : X \rightarrow A$ to $\psi : Y \rightarrow B$ are of the form (h, k), where h is a arrow of $\mathbf{C}$ from X to Y and h is a arrow of $\mathbf{C}$ from A to B, with the property that $k \circ \varphi = \psi \circ h$.

Dependency from Natural Transformation

$\mathcal{F}_\eta(\varphi)$:

$$\begin{tikzcd}
 & FX \arrow[dl, "F\varphi"'] \arrow[dr, "\eta_X"] \arrow[dd, dotted, "\mathcal{F}_\eta(\varphi)"] & \\
FA \arrow[dr, "\eta_A"'] & & GX \arrow[dl, "G\varphi"] \\
 & GA &
\end{tikzcd}$$

$\mathcal{F}_\eta(h,k)$:

$$\begin{tikzcd}
FX \arrow[r, "Fh"] \arrow[d, "\mathcal{F}_\eta(\varphi)"'] & FY \arrow[d, "\mathcal{F}_\eta(\psi)"] \\
GA \arrow[r, "Gk"'] & GB
\end{tikzcd}$$

For two endofunctors $F, G : \mathbf{C} \to \mathbf{C}$, and a natural transformation $\eta : F \to G$, we derive an endofunctor $\mathcal{F}_\eta : (\mathbf{C}^{\to}) \to (\mathbf{C}^{\to})$ as follows.

- For an object $\varphi : X \to A$, let
  $\mathcal{F}_\eta(\varphi) : FX \to GA = \eta_A \circ F\varphi = G\varphi \circ \eta_X$.
- For an arrow $(h,k) : \varphi \to \psi$, define $\mathcal{F}_\eta(h,k) = (Fh, Gk)$.

$\mathcal{F}_\eta$-algebra

objects:

$$\begin{tikzcd}
X \arrow[d, "\varphi"'] & FX \arrow[l, "p"'] \arrow[d, "\mathcal{F}_\eta(\varphi)"] \\
A & GA \arrow[l, "q"']
\end{tikzcd}$$

arrows:

$$\begin{tikzcd}
X \arrow[rrr, "p"'] \arrow[dr, "\varphi"] \arrow[ddd, dotted, "\mu"'] & & & FX \arrow[dl, "\mathcal{F}_\eta(\varphi)"'] \arrow[ddd, dotted, "F\mu"] \\
& A \arrow[d, dotted, "\nu"'] & GA \arrow[l, "q"'] \arrow[d, dotted, "G\nu"] & \\
& B & GB \arrow[l, "h"] & \\
Y \arrow[ur, "\psi"'] & & & FY \arrow[ul, "\mathcal{F}_\eta(\psi)"] \arrow[lll, "k"]
\end{tikzcd}$$

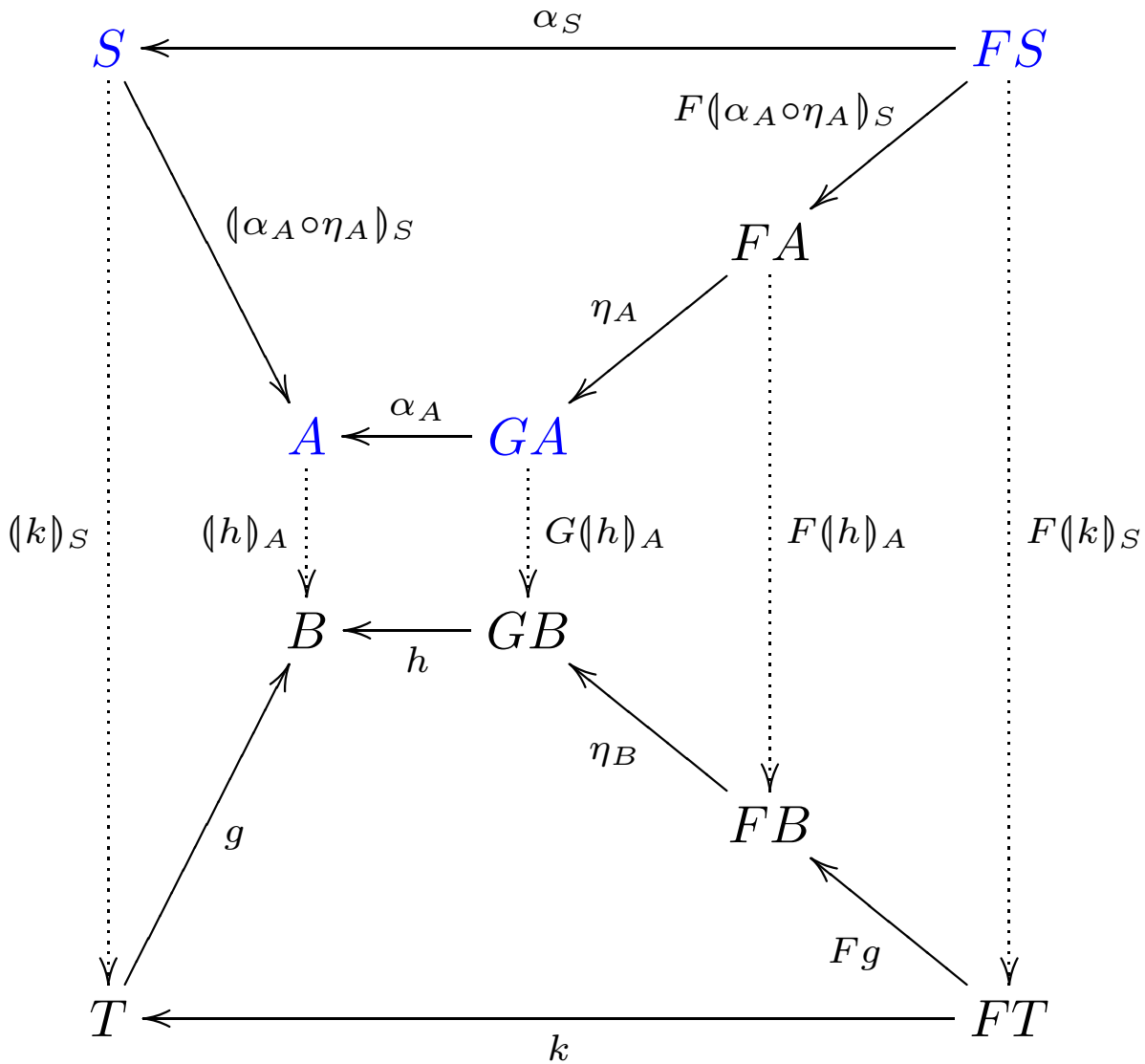
Let $\eta : F \to G$ be natural transformation between two endofunctors F and G. The category of $\mathcal{F}_\eta$-algebra is described above.

Initial $\mathcal{F}_\eta$-algebra for Inductive Dependency

$$
\begin{tikzcd}
S \arrow[d, "(\!|\alpha_A \circ \eta_A|\!)"'] & FS \arrow[l, "\alpha_S"'] \arrow[d, "\eta_A \circ F(\!|\alpha_A \circ \eta_A|\!)"] \\
A & GA \arrow[l, "\alpha_A"]
\end{tikzcd}
$$

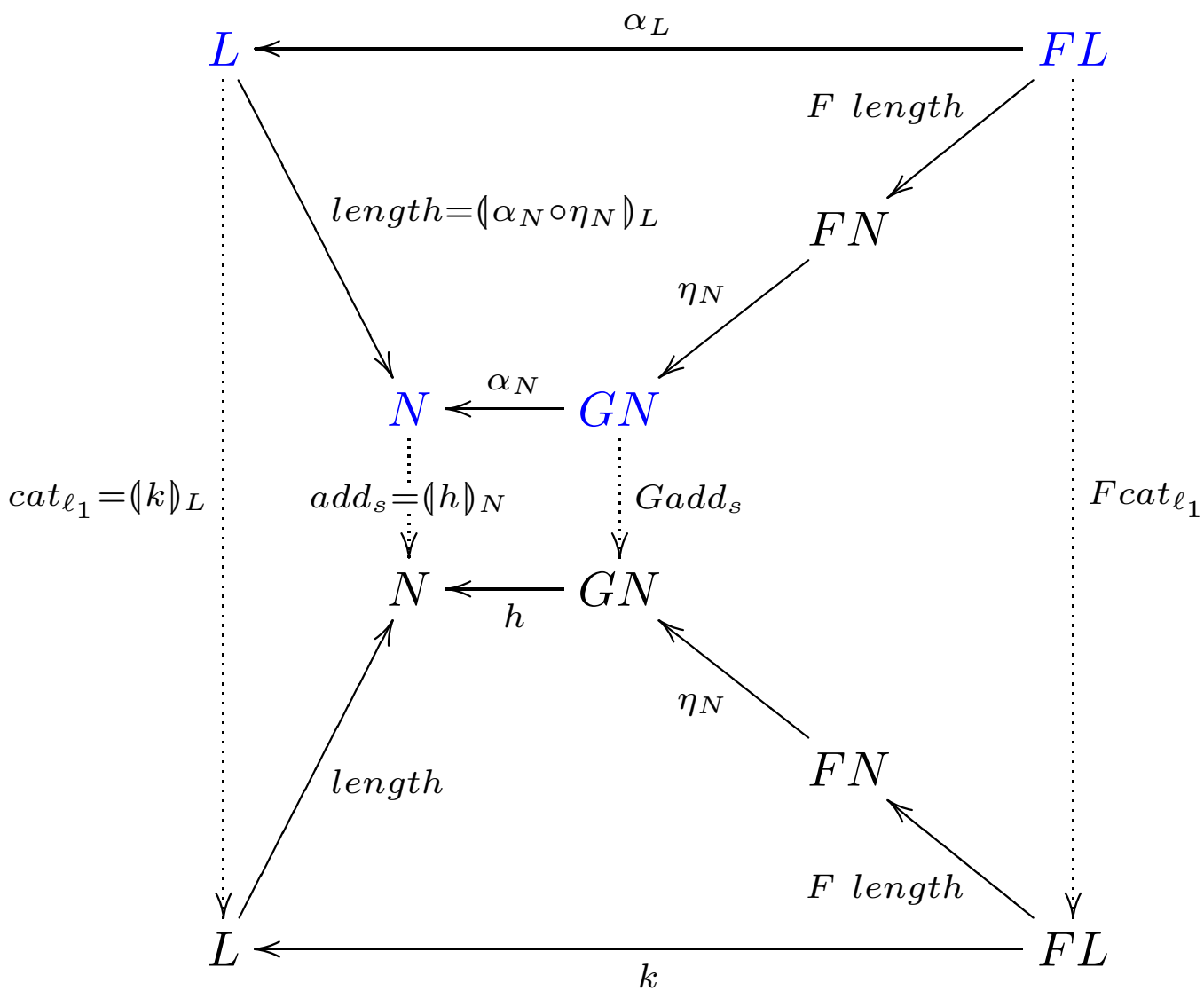
Proposition. Let (α_S, S) and (α_A, A) be the initial object of an F-algebra and a G-algebra, respectively. Let $\eta : F \to G$ be a natural transformation, then the above diagram is the initial object of the $\mathcal{F}_\eta$-algebra.

Note: Both S and A are regular data types. The above object describes a *natural* dependency of S on A. The initiality of this object can be used to derive other dependencies.



The dependency of S on B is described by $g \circ (\!|k|\!)_S = (\!|h|\!)_A \circ (\!|\alpha_A \circ \eta_A|\!)_S$. By fusion law, both sides equal to $(\!|h \circ \eta_B|\!)_S$.

Function `concat` Re-visited



where $\ell_1 :\ list\ \alpha\ m$ **and** $cat_{\ell_1} :\ \text{forall}\ n : nat,\ list\ \alpha\ n \rightarrow list\ \alpha\ (n + m)$

Function concat Re-visited, in O'Caml

```
type ('a, 'b) t = Nil | Cons of 'a * 'b
let mapF f t = match t with Nil -> Nil | Cons (a, b) -> Cons (a, f b)

type 'a list = Rec of ('a, 'a list) t

let rec fold (k, h) t = ...

let concat p q =
    let k t = match t with Nil -> p | Cons (a, b) -> Rec (Cons (a, b))
 in let h s = match s with O -> length p | S n -> 1 + n
 in
    fold (k, h) q
```

Programming $\mathcal{F}_\eta$-algebra in O'Caml, Modularly

- Layers of categorical constructions are systematically mapped to layers of ML modules.
  - objects are types; arrows are typed functions;
  - functors become type constructors and the map functions;
  - natural transformations become polymorphic functions;
  - fixpoints and dependencies are generated by parameterized modules, *etc.*
- The type constructors are of fixed arities (-).
- The modules are highly parameterized (+).

Programming $\mathcal{F}_\eta$-algebra in O'Caml, 1/7

```
module type CAT = sig
  type 'a t
end

module type FUN = sig
  type ('a, 'b) t
  val map: ('b -> 'c) -> ('a, 'b) t -> ('a, 'c) t
end

module type NAT = sig
  module S: FUN
  module T: FUN

  val eta: ('a, 'b) S.t -> ('a, 'b) T.t
end
```

Programming $\mathcal{F}_\eta$-algebra in O'Caml, 2/7

```
module type FIX = sig
  module Base: FUN
  type 'a t
  val up:   ('a, 'a t) Base.t -> 'a t
  val down: 'a t -> ('a, 'a t) Base.t
end

module type MU = functor (B: FUN) -> FIX with module Base = B

module Mu: MU = functor (B: FUN) ->
struct
  module Base  = B
  type 'a t = Rec of ('a, 'a t) Base.t
  let up        a  = Rec a
  let down (Rec a) =     a
end
```

Programming $\mathcal{F}_\eta$-algebra in O'Caml, 3/7

```
module type DEP =
sig
  module S: CAT
  module A: CAT
  val index: 'a S.t -> 'a A.t
end

module type NAT'DEP =
  functor (S: FIX) ->
  functor (A: FIX) ->
  functor (N: NAT with module S = S.Base and module T = A.Base) ->
  DEP with module S = S and module A = A
```

Programming $\mathcal{F}_\eta$-algebra in O'Caml, 4/7

```
module type DEP'FOLD = functor (S: FIX) -> functor (A: FIX) ->
  functor (N: NAT with module S = S.Base and module T = A.Base) ->
  functor (D: DEP) ->
sig
  val f: (('a, 'a D.S.t) S.Base.t -> 'a D.S.t) *
         (('a, 'a D.A.t) A.Base.t -> 'a D.A.t) ->
         ('a S.t -> 'a D.S.t) * ('a S.t -> 'a D.A.t)
end

module Fold: DEP'FOLD = functor (S: FIX) -> functor (A: FIX) ->
  functor (N: NAT with module S = S.Base and module T = A.Base) ->
  functor (D: DEP) ->
struct
  let f (k, h) =
      let rec s2t s = (k $         S.Base.map s2t $ S.down) s
   in let rec s2b s = (h $ N.eta $ S.Base.map s2b $ S.down) s
   in
      (s2t, s2b)
end
```

Programming $\mathcal{F}_\eta$-algebra in O'Caml, 5/7

```
module FNat = struct
  type ('a, 'b) t = O | S of 'b
  let map f t = match t with O -> O | S a -> S (f a)
end

module FList = struct
  type ('a, 'b) t = Nil | Cons of 'a * 'b
  let map f t = match t with Nil -> Nil | Cons (a, b) -> Cons (a, f b)
end

module Nat  = Mu (FNat)
module List = Mu (FList)

module List2Nat = struct
  module S = FList
  module T = FNat
  let eta t = match t with Nil -> O | Cons (_, b) -> S b
end
```

Programming $\mathcal{F}_\eta$-algebra in O'Caml, 6/7

```
module ListNatDep    = Dep  (List) (Nat) (List2Nat)
module NewListNatDep = Fold (List) (Nat) (List2Nat) (ListNatDep)

let k p   q   = match q   with Nil -> p   | _ -> List.up q
let h p_i q_i = match q_i with O   -> p_i | _ ->  Nat.up q_i

let list2 = List.up (Cons (true, List.up (Cons (true, List.up Nil))))
            (* [true; true] *)

let nat2  = Nat.up (S (Nat.up (S (Nat.up O))))
            (* 2 *)

let (cat, cat_i) = NewListNatDep.f (k list2, h nat2)
```

Programming $\mathcal{F}_\eta$-algebra in O'Caml, 7/7

```
let list3 = List.up (Cons (false, List.up (Cons (false,
            List.up (Cons (false, List.up Nil))))))
            (* [false; false; false] *)

let list5 = cat   list3  (* [false; false; false; true; true] *)

let nat5  = cat_i list3  (* 5 *)
```

```
cat : bool List.t -> bool List.t
cat_i : '_a List.t -> '_a Nat.t
```

Conclusion

- We have proposed an algebra of dependent data types.
- Category theory is helpful. (Sometimes.)
- O'Caml and Coq are fun! (Most of the time.)
- Natural transformations may be too restrictive in the specifications of dependencies. However, as long as the necessary diagrams are commutative, the results will still apply.