中央研究院
資訊科學研究所
Institute of Information Science, Academia Sinica • Taipei, Taiwan, ROC

TR-IIS-06-012

An Algebra of Dependent Data Types

Tyng-Ruey Chuang, Jan-Li Lin

September 12, 2006 || Technical Report No. TR-IIS-06-012

http://www.iis.sinica.edu.tw/LIB/TechReport/tr2006/tr06.html

An Algebra of Dependent Data Types*

Tyng-Ruey Chuang and Jan-Li Lin**

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

Abstract. We extend the standard categorical approach to algebraic data types to dependent algebraic data types, so that dependency between two algebraic data types has natural semantics. Specifically, for two inductive data types S and A characterized by two F-algebra F and G, any natural transformation $\eta : F \to G$ gives rise to a dependency of S on A. This natural dependency is the initial object of what we call a $\mathcal{F}_\eta$-algebra. The initiality further allows us to describe certain dependencies in functions that both involve S and A. We have used Objective Caml to write functional programs where dependencies among data types (and in the relevant functions) are made explicit. This is done by a systematic mapping of layers of categorical constructions to layers of Objective Caml modules.

1 Motivation

Types have been used in programming languages to specify computational values and to check, at compile-time or at run-time, whether the computed values will meet the specifications. As dependent types are types that depend on values, dependent types allow for more expressive specifications, and can be used to better describe and ensure program properties. Research along this direction has produced Cayenne [1] and Dependent ML [14], among others, which are programming languages with dependent types. The proof assistant system Coq [5], which is based on the Calculus of Inductive Construction, can be viewed as a functional languages with dependent types as well. Coq as a functional language further possess the properties that dependently typed programs are statically type-checked and the programs themselves are terminating.

---

* This result was first announced without review at the *2nd Taiwanese-French Conference on Information Technologies* (TFIT 2005; `http://iir.csie.ncku.edu.tw/TFIT2005/`) and at the *2006 Conference of the Types Project* (TYPES 2006; `http://www.cs.nott.ac.uk/types06/`). This paper now appears as technical report TR-IIS-06-012 at the Institute of Information Science, Academia Sinica, Taiwan. A digital copy of this paper is available from the Institute's website at <http://www.iis.sinica.edu.tw>, or from the authors by e-mail (trc@iis.sinica.edu.tw).

** Jan-Li Lin is now in the PhD program at the Department of Mathematics, University of Indiana, Bloomington, Indiana 47405, USA.

```
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.


Definition list2: List bool 2 := Cons bool true 1 (Cons bool true 0 (Nil bool)).

Definition list3: List bool 3 := Cons bool false 2 (Cons bool false 1
                                 (Cons bool false 0 (Nil  bool))).

Definition list5: List bool 5 := concat bool 2 list2 3 list3.
```

Fig. 1. Definitions of `List` and `concat` in Coq.

Figure 1 shows the definition in Coq of a data type `List` in which each of its values carrying with it its length. Using the terminology of functional languages, we may call `List` a type constructor with A being the type variable (i.e., `List` is polymorphic). The data type $\texttt{List}_A$ is dependent on data type `nat`. Each natural number n gives rise to a type $\texttt{List}_A\ n$: The type for all the $\texttt{List}_A$ values whose length is exactly n. Furthermore, one understands that

$$\texttt{List}_A = \sum_n (\texttt{List}_A\ n).$$

That is, there is a mapping from $\texttt{List}_A$ to `nat` and this mapping partitions $\texttt{List}_A$. Data type `nat`, in this case, is often called the index set for data type $\texttt{List}_A$. The mapping from $\texttt{List}_A$ to `nat` is called the indexing function which in this case is the length function.

Figure 1 further defines a function `concat` for list concatenation. In the definition of `concat`, we take care to prove that the length of the resulting list is the summation of those of the two input lists. This is done by a structural induction on `q` (. . . `{struct q}` . . .). Coq checks that such a proof is correct, with the help of the inductive step we provide (. . . `q in List _ i return List _ (i+m) with` . . .). Coq confirms that function `concat` does have the required property by returning a type signature that precisely describes this dependency:

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

Figure 2 shows the definitions in O'Caml of data type `list` and function `concat`. As O'Caml does not support dependent data types in the language, we

```
type 'a list = Nil | Cons of 'a * 'a list

let rec concat p q = match q with
                          Nil         -> p
                        | Cons (h, t) -> Cons (h, concat p t)


let list2 = Cons (true, Cons (true, Nil))

let list3 = Cons (false, Cons (false, Cons (false, Nil)))

let list5 = concat list2 list3
```

Fig. 2. Definitions of `list` and `concat` in O'Caml.

do not have a way to define a type for lists of a certain length. O'Caml can only infer the following type signature for `concat` which do not capture the length dependency between the result and the two input arguments:

```
val concat : 'a list -> 'a list -> 'a list = <fun>
```

This paper is an attempt to understand dependency purely as a property between data types, and between typed values. Specifically, for two data types, we seek to capture certain dependencies between the two based on their algebraic definitions. We look for ways to carry over such dependencies to inductive computations involving the two data types. Take as an example the indexing function *length*: $\texttt{List}_A \rightarrow \texttt{nat}$ that characterizes the dependency of $\texttt{List}_A$ on `nat`, we ask the following questions:

- How is this dependency defined? What are other dependencies and how shall they be defined?
- How is this dependency incorporated in computations involving $\texttt{List}_A$? How is it used to specify properties of the computations?

As our approach is not based on a particular calculus or a particular type system for the definitions of dependent types and dependently-typed computations, the results can be used in languages that do not support dependent types. We simply view dependencies as properties between typed values, and we aim to show that such properties can be calculated mechanically at run-time. This approach will guide us, for example, to define in O'Caml an indexing function from `list` to `int` in a natural way. Further, this indexing function can then be passed along to inductive functions like `concat` so that the intended dependencies between the result and the arguments are computed and made explicit if necessary.

The rest of this paper is organized as the following. Section 2 extends the standard categorical approach to algebraic data types to dependent algebraic data types, so that dependency between two algebraic data types has natural semantics. Section 3 uses the `concat` function as an example to illustrate how the

categorical semantics developed in Section 2 can be applied. Section 4 shows a systematic mapping of layers of categorical constructions to layers of Objective Caml modules, so that the program modules can be readily used to describe dependencies among various algebraic data types. Section 5 describes future works and concludes this paper.

2 A Categorical Semantics for Dependent Data Types

There is a well-established semantics based on category theory to explain algebraic types. A type is interpreted as an object of a category, usually the category of **Set** or **Cpo**. The product and sum of two types correspond to the product and coproduct of two objects in a category. The unit type corresponds to the terminal object in a category. An inductive type, such as lists or binary trees, can be interpreted as the least fixed point of a functor. In this section, we propose a way to extend this categorical semantics so that it can explain functions with dependent data types.

First let us review the categorical interpretation of algebraic types. The algebraic types are those built from the unit type with product, sum, and the least fixed point operators. As we have described the unit, products and sums in the last paragraph, we now review the correspondence of least fixed points in category theory and recursive data types. We assume some basic knowledge of category theory. Readers not familiar with category theory can consult Bird and de Moor [3], Hagino [9], or Mac Lane [12].

Let $\mathbf{C}$ be a category, and $F : \mathbf{C} \to \mathbf{C}$ be an endofunctor of $\mathbf{C}$. The category of F-algebra has objects of the form (f, X), where $f : FX \to X$ is an arrow. Its arrows are of the form $(\mu) : (f, X) \to (g, Y)$, where $\mu : X \to Y$ is an arrow satisfying $\mu \circ f = g \circ F\mu$. That is, it ensures the commutativity of the following diagram.

$$\begin{array}{ccc} X & \xleftarrow{\ f\ } & FX \\ {\scriptstyle \mu}\downarrow & & \downarrow{\scriptstyle F\mu} \\ Y & \xleftarrow[\ g\ ]{} & FY \end{array}$$

2.1 Initial F-algebra

If the category of F-algebra has an initial object (α, S), then it is called an initial F-algebra. For any object (f, X), one uses the notation $(\!| f |\!)$ to denote the *unique* arrow $S \to X$ with $(\!| f |\!) \circ \alpha = f \circ F(\!| f |\!)$. In such case, it is well known that α is an isomorphism between S and FS. It is in this sense that we view a recursive data type S as the fixed point of an endofunctor F. A fundamental result for initial F-algebra is the fusion law.

Lemma 1. (fusion law) *If the two arrows* $g : FY \to Y$ *and* $h : X \to Y$ *satisfies* $h \circ f = g \circ Fh$*, then* $(\!| g |\!) = h \circ (\!| f |\!)$*.*

```
type ('a, 'b) tF = 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) tF
let rec foldF f (Rec t) = f (mapF (foldF f) t)

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

Fig. 3. Initial F-algebra definitions of `list` and `concat` in O'Caml.

We mention that the proof of the fusion law is based on the following commutative diagram. In the diagram, by definitions, both the upper and the lower square are commutative, hence one proves that the entire rectangle is also commutative.

$$\begin{array}{ccc} S & \xleftarrow{\alpha} & FS \\ {\scriptstyle (\!|f|\!)}\downarrow & & \downarrow{\scriptstyle F(\!|f|\!)} \\ X & \xleftarrow[f]{} & FX \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle Fh} \\ Y & \xleftarrow[g]{} & FY \end{array}$$

The fusion law will be used later to prove our result. For more information about initial F-algebra and its use in interpreting recursive types and fold functions, please see Bird and de Moor [3] or Manes and Arbib [13].

The O'Caml code in Figure 3 illustrates how to define the list data type and the associated fold function with this initial F-algebra point-of-view. The initial F-algebra semantics of the list data type:

$$List_A = \mu X.F_A(X), \quad \text{where } F_A(X) = 1 + A \times X$$

is used literally for the definition in O'Caml the type constructor `'a list`. Note that we define function `contact` with the fold function. O'Caml infers from the definitions the following type signatures, which are exactly what ones have expected:

```
val mapF : ('x -> 'y) -> ('a, 'x) tF -> ('a, 'y) tF = <fun>
val foldF : (('a, 'b) tF -> 'b) -> 'a list -> 'b = <fun>
val concat : 'a list -> 'a list -> 'a list = <fun>
```

2.2 Arrow Category and $\mathcal{F}_\eta$ Algebra

As described in Section 1, the dependent data type $\texttt{List}_A$ can be thought as a partition derived from an indexing function from $\texttt{List}_A$ to $\texttt{nat}$:

$$\texttt{List}_A = \sum_n (\texttt{List}_A\ n).$$

This motivate the use of *arrow category*, where indexing function like *length*: $\texttt{List}_A \to \texttt{nat}$ is considered as an object in the arrow category.

Let $\mathbf{C}$ be a category, the arrow category of $\mathbf{C}$, which we denote by $\mathbf{C}^{\to}$, is as the following. It has families $(\varphi : X \to A)$ as objects. Thus, arrows in $\mathbf{C}$ become objects in $\mathbf{C}^{\to}$. For two objects $\varphi : X \to A$ and $\psi : Y \to B$, the arrows of $\mathbf{C}^{\to}$ from $\varphi : X \to A$ to $\psi : Y \to B$ are of the form (h, k), where h is a arrow of $\mathbf{C}$ from X to Y and k is a arrow of $\mathbf{C}$ from A to B, with the property that $k \circ \varphi = \psi \circ h$. The objects and arrows of the arrow category $\mathbf{C}^{\to}$ can be visualized as:

$$\text{objects:}\quad \begin{array}{c} X \\ \downarrow{\scriptstyle \varphi} \\ A \end{array} \qquad \text{arrows:}\quad \begin{array}{ccc} X & \xrightarrow{h} & Y \\ {\scriptstyle \varphi}\downarrow & & \downarrow{\scriptstyle \psi} \\ A & \xrightarrow[k]{} & B \end{array}$$

For two endofunctors $F, G : \mathbf{C} \to \mathbf{C}$ of $\mathbf{C}$, and a natural transformation $\eta : F \to G$, we can construct a functor $\mathcal{F}_\eta : (\mathbf{C}^{\to}) \to (\mathbf{C}^{\to})$ as follows. For an object $\varphi : X \to A$, let

$$\mathcal{F}_\eta(\varphi) \;=\; \eta_A \circ (F\varphi) \;=\; (G\varphi) \circ \eta_X : FX \to GA$$

Note that the second equality holds because η is a natural transformation. Moreover, for an arrow $(h, k) : \varphi \to \psi$, define

$$\mathcal{F}_\eta(h, k) = (Fh, Gk)$$

This definition of the functor $\mathcal{F}_\eta : (\mathbf{C}^{\to}) \to (\mathbf{C}^{\to})$ can be visualized as:

$$\mathcal{F}_\eta(\varphi) : \quad \begin{array}{ccccc} & & FX & & \\ & {\scriptstyle F\varphi}\swarrow & \vdots & \searrow{\scriptstyle \eta_X} & \\ FA & & {\scriptstyle \mathcal{F}_\eta(\varphi)} & & GX \\ & {\scriptstyle \eta_A}\searrow & \downarrow & \swarrow{\scriptstyle G\varphi} & \\ & & GA & & \end{array} \qquad \mathcal{F}_\eta(h,k) : \quad \begin{array}{ccc} FX & \xrightarrow{Fh} & FY \\ {\scriptstyle \mathcal{F}_\eta(\varphi)}\downarrow & & \downarrow{\scriptstyle \mathcal{F}_\eta(\psi)} \\ GA & \xrightarrow[Gk]{} & GB \end{array}$$

Given two endofunctors F and G, the above describes how to derive a dependency $\mathcal{F}_\eta(\varphi) : FX \to GA$ if there are already a dependency $\varphi : X \to A$ as well

as a natural transformation $\eta : F \to G$. We now lift the definition of F-algebra to form the definition for $\mathcal{F}_\eta$-algebra. $\mathcal{F}_\eta$-algebra can be thought as an F-algebra operating on the arrow category, the category of dependencies.

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

objects:

$$\begin{array}{ccc} X & \xleftarrow{\;p\;} & FX \\ {\scriptstyle\varphi}\downarrow & & \downarrow{\scriptstyle\mathcal{F}_\eta(\varphi)} \\ A & \xleftarrow{\;q\;} & GA \end{array}$$

arrows:

$$\begin{array}{ccccccc} X & & & \xleftarrow{\quad p \quad} & & & FX \\ & \searrow^{\varphi} & & & & {}^{\mathcal{F}_\eta(\varphi)}\swarrow & \\ & & A & \xleftarrow{\;q\;} & GA & & \\ {\scriptstyle\mu}\vdots & & {\scriptstyle\nu}\vdots & & \vdots{\scriptstyle G\nu} & & \vdots{\scriptstyle F\mu} \\ & & B & \xleftarrow[\;h\;]{} & GB & & \\ & \nearrow^{\psi} & & & & {}^{\mathcal{F}_\eta(\psi)}\nwarrow & \\ Y & & & \xleftarrow[\quad k \quad]{} & & & FY \end{array}$$

Intuitively, an object in the $\mathcal{F}_\eta$-algebra is a pair (p, q), where p an object in the category of F-algebra and q an object in the category of G-algebra, such that they relate the two dependencies $\varphi : X \to A$ and $\mathcal{F}_\eta(\varphi) : FX \to GA$, where $\eta : F \to G$ is a natural transformation. An arrow in the $\mathcal{F}_\eta$-algebra is a pair (μ, ν), where μ a dependency of X on Y and ν a dependency of A on B, such that the two dependencies relate the two existing dependencies established by the natural transformation $\eta : F \to G$.

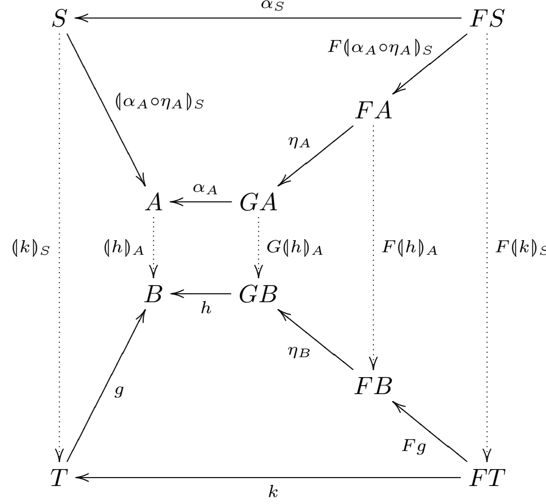
2.3 Initial $\mathcal{F}_\eta$-algebra

We are now ready for the main result. Let (α_S, S) and (α_A, A) be the initial object of an F-algebra and a G-algebra, respectively. And $\eta : F \to G$ be a natural transformation, then the following diagram illustrates an object of the $\mathcal{F}_\eta$-algebra:

$$\begin{array}{ccc} S & \xleftarrow{\;\alpha_S\;} & FS \\ {\scriptstyle (\!|\alpha_A \circ \eta_A|\!)}\downarrow & & \downarrow{\scriptstyle \eta_A \circ F(\!|\alpha_A \circ \eta_A|\!)} \\ A & \xleftarrow[\;\alpha_A\;]{} & GA \end{array}$$

Proposition 1. *The above object is the initial $\mathcal{F}_\eta$-algebra.*

Proof. The proof of the proposition is based on the following diagram:



If we can show that all parts of the diagram is commutative, then the proposition is proved because the uniqueness part comes directly from the uniqueness of $(\!|k|\!)_S$ and $(\!|h|\!)_A$. First, by the definition of an $\mathcal{F}_\eta$-algebra, we have

$$(\!|\alpha_A \circ \eta_A|\!)_S \circ \alpha_S = \alpha_A \circ \eta_A \circ F(\!|\alpha_A \circ \eta_A|\!)_S$$
$$g \circ k = h \circ \eta_B \circ Fg.$$

Next, since (α_S, S) and (α_A, A) are the initial F-algebra and G-algebra, respectively, we know that

$$(\!|k|\!)_S \circ \alpha_S = k \circ F(\!|k|\!)_S$$
$$(\!|h|\!)_A \circ \alpha_A = h \circ G(\!|h|\!)_A.$$

Furthermore, since η is a natural transformation, the equality

$$G(\!|h|\!)_A \circ \eta_A = \eta_B \circ F(\!|h|\!)_A$$

also holds. Now, it remains to show that

$$g \circ (\!|k|\!)_S = (\!|h|\!)_A \circ (\!|\alpha_A \circ \eta_A|\!)_S.$$

By the fusion law, both sides are equal to $(\!|h \circ \eta_B|\!)_S$. $\square$

3 Function concat Re-visited

Let us revisit function concat from Section 1. Let

$$p : \texttt{List}_A\ m$$

be a list of length m, then the partially applied function ($concat\ p$) has type

$$concat\ p: \ \ (\text{forall } n : \texttt{nat})\ \texttt{List}_A\ n \to \texttt{List}_A\ (n+m)$$

where A is type of the list elements.

To interpret this type of $concat\ p$, we can draw the following diagram:

$$\begin{array}{ccc}
L & \xrightarrow{length} & N \\
{\scriptstyle concat\ p}\Big\downarrow & & \Big\downarrow{\scriptstyle add\ m} \\
L & \xrightarrow[length]{} & N
\end{array}$$

Where L is the object for type $\texttt{List}_A$, and N is the object for type $\texttt{nat}$. Here the function $add\ m : N \to N$ is defined by $(add\ m)\ n\ = n + m$.

Let F and G be two functors defined by $FX = 1 + A \times X$ and $GX = 1 + X$, respectively. It is well known that L is the least fixed point of F, and N is the least fixed point of G. Note that it is not hard to write down two functions $k : FL \to L$ and $h : GN \to N$ such that $concat\ p\ =\ (\!|k|\!)_L$ and $add\ m\ =\ (\!|h|\!)_N$. (See, for example, functions k and h in Figure 6 in Section 4.) Moreover, we can define a natural transformation $\eta : F \to G$ by

$$\eta_X = id_1 + \pi_X$$

which means $\eta_X(p)\ =\ p$ if $p \in 1$ is the unique element in the unit type, and $\eta_X(a, x)\ =\ x$ for $(a, x) \in A \times X$. We can check that $length\ =\ (\!|\alpha_N \circ \eta_N|\!)$.

The following diagram then describes the length dependency in function *concat*.

$$\begin{array}{ccccc}
L & & \xleftarrow{\qquad\qquad \alpha_L \qquad\qquad} & & FL \\
\vdots & \searrow^{length=(\!|\alpha_N\circ\eta_N|\!)_L} & & \swarrow^{F\ length} & \vdots \\
\vdots & & & FN & \vdots \\
\vdots & & & \swarrow^{\eta_N} & \vdots \\
\vdots & N & \xleftarrow{\alpha_N} & GN & \vdots \\
{\scriptstyle concat\ p\ =\ (\!|k|\!)_L}\ \vdots & \vdots\ {\scriptstyle add\ m\ =\ (\!|h|\!)_N} & & \vdots\ {\scriptstyle G\ (add\ m)} & \vdots\ {\scriptstyle F\ (concat\ p)} \\
\vdots & N & \xleftarrow[h]{} & GN & \vdots \\
\vdots & & & \nwarrow^{\eta_N} & \vdots \\
\vdots & \nearrow^{length} & & FN & \vdots \\
\downarrow & & & \nwarrow^{F\ length} & \downarrow \\
L & & \xleftarrow[k]{\qquad\qquad\qquad\qquad} & & FL
\end{array}$$

In particularly, we can infer that

$$length \circ (concat\ p)\ \ =\ \ (add\ m) \circ length$$

That is, the length of the result list can be computed in two ways —- complete the concat operation first then compute the length of the result, or compute the length of the second list argument to concat then add up with the length of the first list argument — and they produce the same result. Furthermore, both can be fused and are equal to the following fold function

$$(\!|h \circ \eta_N|\!)_N$$

This shows how to compute the length of the resulting list, i.e., the dependency of $\texttt{List}_A$ on `nat`, at the same time while completing the concat operation. Note that the diagram above is just a specialized version of the diagram used in the proof of proposition 1.

4 Programming Initial $\mathcal{F}_\eta$-algebra, Modularly

In this section we use the module facility in O'Caml to realize the categorical semantics of dependent data types in Section 2. We have followed the categorical constructions as faithful as possible so as to produce a layered structure of O'Caml modules. The O'Caml modules are quite abstract but are very general. They can be used to specify dependencies between various algebraic data types.

We have used the following principles in mapping categorical constructions to O'Caml modules:

- objects are mapped to O'Caml types (actually, unary type constructors);
- arrows are mapped to typed functions;
- functors become type constructors, with the associated map functions;
- natural transformations become polymorphic functions;
- fixpoints and dependencies are built with parameterized modules.

Currently the module library is rather restrictive as we only deal with type constructors of a fixed arity. The modules are highly parameterized, however. Module applications are used to achieve a high-level of code re-use.

Figure 4 includes some basic definitions. Module type `CAT` is the interface for categories. Not much is required except that there must be an unary type constructor `t` for objects. Module type `FUN` is the interface for functors. It is required that there is a binary type constructor to build algebraic data types (objects) from other types (objects), as well as a map function for defining functions (arrows) between these types. Module type `NAT` is the interface for natural transformations. Natural transformations must be polymorphic functions. Module type `FIX` is the interface for fixed points. The least fixed points are derived by applying module `Mu` to a module of type `FUN`. Dependencies are simply viewed as indexing functions in module type `DEP`. Module `Dep` shows how to to generate a dependency whenever given a natural transformation between two functors. Function `index` in module `Dep` is the indexing function that characterizes a dependency of the algebraic data type `S` on the algebraic data type `A`. Note that the definition of `index` is exactly that same as the initial object in the $\mathcal{F}_\eta$-algebra as proved in Proposition 1.

Figure 5 shows the module definitions necessary for folding dependent data types. Module type `DEP'FOLD` is the interface for module `Fold`. The type signature for function f in `DEP'FOLD` looks rather complex. Let's review it together with the diagram used in Proposition 1. What f asks is a pair of functions k and h, where k is the inductive step for folding an S-typed value to a T-typed value, and h is the inductive step necessary to index the resulting T-typed value onto type B. When given k and h, function f then returns two functions that fold, respectively, from S to T, and from S to the index set B. We can do the following new-naming (too bad O'Caml does not allow this): module `T` for module `D.S`, module `B` for module `D.A`, module `F` for module `S.Base`, and module `G` for module `A.Base`. We then arrive at the following type signature for f

```
val f: (('a, 'a T.t) F.t -> 'a T.t) *
       (('a, 'a B.t) G.t -> 'a B.t) ->
       ('a S.t -> 'a T.t) *
       ('a S.t -> 'a B.t)
```

The above is exactly what is described in the diagram in Proposition 1. Note also that the implementation in module `Fold` follows the diagram as well. Module `ListNatDep` defines an indexing function from `List` to `Nat` deriving from natural transformation `List2Nat`. This indexing function is in fact the length function. Using this dependency, module `NewListNatDep` will define function f which not only folds a list but also computes the index of the result.

Figure 6 reconstructs the concat example. Function k is the inductive step for computing list concatenation, and function h is the corresponding step for calculate the index of the returned result. When both are passed to function `NewListNatDep.f`, it returns two functions: One that computes the result of the concatenation, and one that computes the result's index. Apparently, functions k and h much be properly matched and they must ensure that, together with the natural transformation `List2Nat`, they make the diagram commutative. This is the responsibility of the programmer. Other than that, module `NewListNatDep` will mechanically generate efficient functions for computing the result and its index.

Note that, in the concat case, module `Fold` takes up again the dependency `ListNatDep` (as its fourth argument) for describing the data type of the final result (`List`) and and its dependency on `Nat` (length). One can plug in other dependency as well. As an example, we may only be interested in knowing whether the resulting list is a null list or not. In such a case, the fourth argument will be `ListBoolDep`. When used together with the following function h'

```
let h' p_i q_i = match q_i with
                     true  -> p_i
                  |  false -> Bool.up false
```

as the new inductive step for calculating the index, function `NewListNatDep.f` will return the (usual) function for list concatenation but with a new indexing function telling whether the resulting list is null or not.

5 Conclusion and Future Work

We have extended the usual F-algebra interpretation of algebraic data types using arrow category, so that fold functions for dependent data types are given sound semantics. The extended algebra provides us with a better understanding of dependent data types, hence lead ways to further research.

Right now we only handle regular data types where the algebraic data types under investigation appears as a fixed point in a uniform way in the type expression. We have not yet considered dependencies for irregular data types or nested data types.

Also, dependency between algebraic data types currently must be generated by a natural transformation. This can be too restrictive. A closer examination of the diagram in Proposition 1 shows that the natural transformation is only used for two instances A and B, where both are the index sets. It may well be the case that one can prove the necessary diagram is commutative using properties of A and B, and property of the inductive indexing function h, without resorting to a natural transformation.

We are also re-casting $\mathcal{F}_\eta$-algebra in Coq so that properties about specific dependent data types and inductive computations can be proved. As an example, we may want to prove that the pair of functions (k, h) that is passed to the fold function is properly matched up: h is in deed the corresponding indexing step for k. In O'Caml, this is the responsibility of the programmer. Using Coq, the system can be used to check that it is so.

References

1. L. Augustsson. Cayenne: A Language with Dependent Types. In *Proceedings of ICFP 1998*, pp. 239–250. ACM Press, 1998.
2. Henk Barendregt and Herman Geuvers. Proof-assistants using Dependent Type Systems. In *Handbook of Artificial Reasoning*, Volume II, Chapter 18 (2001), pp. 1149–1240.
3. Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.
4. N. G. de Bruijn. A survey of the project AUTOMATH. In Sheldin and Hindley, editors. *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press Limited, 1980. pp. 579–606.
5. The Coq proof assistant. **URL:** `http://coq.inria.fr` .
6. R. M. Amadio and P.-L. Curien. *Domains and Lambda-Calculi*. Cambridge Tracts in Theoretical Computer Science, 1998.
7. Herman Geuvers. Induction Is Not Derivable in Second Order Dependent Type Theory. In S. Abramsky, editor, *Proceedings of Typed Lambda Calculus and Applications (TLCA 2001)*, Krakow, Poland, May 2001. LNCS 2044, pp. 166–181, 2001.
8. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7, Cambridge University Press, 1989.
9. Tatsuya Hagino. *A Categorical Programming Language*. Ph.D. thesis. University of Edinburgh, 1987.
10. R. Harper, F. Honsell, and F. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

11. G. Longo and E. Moggi, Constructive Natural Deduction and its "Modest" Interpretation. Technical Report CMU-CS-88-131.
12. S. Mac Lane, *Categories for the Working Mathematician*, second edition, Graduate Texts in Mathematics, no. 5, Springer-Verlag, 1998.
13. E.G. Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics*. Text and Monographs in Computer Science. Springer Verlag, 1986.
14. Hongwei Xi and Frank Pfenning, Dependent Types in Practical Programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '99)*, pp. 214–227, San Antonio, January 1999.

```
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

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

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

module Dep: NAT'DEP =
  functor (S: FIX) ->
  functor (A: FIX) ->
  functor (N: NAT with module S = S.Base and module T = A.Base) ->
struct
  module S = S
  module A = A
  let rec index s = A.up (N.eta (S.Base.map index (S.down s)))
end
```

Fig. 4. A modular definition of dependencies between two data types (`Dep`).

```
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

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

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

Fig. 5. A modular definition of fold for dependent data types (`Fold`).

```
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))))
let nat2  = ListNatDep.index list2

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

let list3 = List.up (Cons (false, List.up (Cons (false,
            List.up (Cons (false, List.up Nil))))))

let list5 = cat   list3
let nat5  = cat_i list3
```

Fig. 6. The concat example re-constructed.