Initial Algebra, Final Coalgebra and Datatype

Masuka Yeasin

August 08, 2011

1 Introduction

Bart Jacobs and Jan Rutten published "A Tutorial on (Co) Algebras and (Co) Induction" in which they provided a brief introduction to initial algebras and final coalgebras[1]. Induction is used both as a definition principle, and as a proof principle for algebraic structures. But there are also important dual "coalgebraic" structures. Spaces of infinite data (infinite lists, non-well founded sets) are generally of this kind and "coinduction" is used as a definition principle and as a proof principle for such structures. Algebra is a well-established part of mathematics, dealing with sets with operations satisfying certain properties, like groups, rings, etc. Algebraic methods are used in computer science, in a branch called algebraic specification or abstract data type theory. Initial algebras turn out to be inductive data types. Dually final coalgebras entail a notion of behaviour types. Both of these structures, referred to as categorical data types, may be directly used in programming. In programming semantics, a recursive type is understood as a solution of a recursive type equation [3]. Least fixpoints of covariant recursive type equations correspond to the so-called *inductive types* which contain only finite elements, like natural numbers, lists or trees. On the other hand, infinite data structures, such as streams, are given by the greatest fixpoints and are called *co-inductive types.* From an algebraic and categorical perspective least/greatest solutions of recursive type equations correspond to the notions of initial algebras/final coalgebras. The data types used by computer scientists are often generated from a given collection of (constructor) operations, and it is for this reason that "initiality" of algebras plays such an important role. In order to fully appreciate the underlying duality between algebra and induction on the one hand, and coalgebra and coinduction on the other, some elementary notions from category theory are needed, especially the notions of functor, and of initial and final object in a category.

2 Initial and Final Objects

An object is initial if there exists a unique morphism from the object to every object in the category, and an object is final if there exists a unique morphism from every object in the category to the object [2]. Let C be a category and

$S \in \mathsf{Ob}(C)$. Then, S is an *initial object* if for all $X \in \mathsf{Ob(C)}$, there exists a unique $f \in \mathsf{hom(S,X)}$. That is $S \overset{f}{\dashrightarrow} X$ for each object X, where the dashed arrow denotes the uniqueness of the morphism. Dually, S is a *final object* if for all $X \in \mathsf{Ob(C)}$, there exists a unique morphism $g \in \mathsf{hom(X,S)}$.That is, $X \overset{g}{\dashrightarrow} S$ for each object X. The property of initial objects is called *initiality* and the property of final objects is called *finality.*

In the category of sets, the initial object is Ø, since there exists a unique morphism $0 : \emptyset \to X$; the final object is the set 1, since there exists a unique morphism $1 : X \to 1$, for each set X.

3 Algebraic and Coalgebraic phenomena

The distinction between algebra and coalgebra may be described as *construction* versus *observation* [1]. As a typical example of algebra, consider for a fixed data set A, the set $A^* = list(A)$ of finite sequences (lists) of elements of A. One can inductively define a length function $len : A^* \to N$ by the two clauses: $len(nil) = 0$ and $len(a.\sigma) = 1 + len(\sigma)$ for all $a \in A$ and $\sigma \in A^*$. Here $nil \in A^*$ for the empty list and $a.\sigma$ for the list obtained from $\sigma \in A^*$ by prefixing $a \in A$.

A typical induction proof that a predicate $P \subseteq A^*$ holds for all lists requires us to prove the induction assumptions.$P(nil)$ and $P(\sigma) \Rightarrow P(a.\sigma)$ for all $a \in A$ and $\sigma \in A^*$. For example, in this way one can prove that $len(a.\sigma) = 1 + len(\sigma)$ by taking $P = \{\sigma \in A^* \mid \forall a \in A.\ len(\sigma.a) = 1 + len(\sigma)\}$. In this algebraic setting we make essential use of the fact that all finite lists of elements of A can be constructed from the two operations $nil \in A^*$ and $cons : A \times A^* \to A^*$.

Consider an example of coalgebraic phenomena, suppose there is a black box machine which has a button and a light. If the button is pressed then the machine performs a certain action and the light goes on only if the machine stops operating. A client cannot directly observe the internal state of the machine, he can only observe its behaviour via the button and the light. In this simple situation, all that can be observed directly about a particular state of the machine is whether the light is on or not. In this situation, a user can observe how many times he has to press the button to make the light go on. This may be zero times, $n \epsilon N$ times, or infinitely many times. If we describe in terms of a set X, then we define a function $button : X \to \{*\} \cup X$.

In a particular state $s \in X$, applying the function *button*- which corresponds to pressing the button-has two possible outcomes: either $button(s) = *$, meaning that the machine stops operating and that the light goes on, or $button(s) \in X$. In the latter case the machine has moved to a next state as a result of the button being pressed. The above pair $(X, button : X \to \{*\} \cup X)$ is an example of a coalgebra.

4 Inductive and coinductive definitions

“Constructor” and “Destructor” operations play an important role for algebras and coalgebras, respectively. Constructors tell us how to generate our (algebraic) data elements: the empty list constructor **nil** and the prefix operation **cons** generate all finite lists. And destructors tell us what we can observe about our data elements: the head and tail operations tell us about infinite lists: head gives a direct observation, and tail returns a next state [1].

In an *inductive definition* of a function f, one defines the value of f on all constructors.In an *coinductive definition* of a function f, one defines the values of all destructors on each outcome $f(x)$. We shall illustrate inductive and coinductive definitions in some examples involving finite lists (with constructors nil and cons) and infinite lists (with destructors head and tail). Consider an example: the function **len** from finite lists to natural numbers giving the length: $len(nil) = 0$ and $len(cons(a.\sigma)) = 1 + len(\sigma)$

Consider an example of coinductive definitions on infinite lists. If we have a function $f : A \to A$, then we would like to define an extension $ext(f)$ of f mapping an infinite list to an infinite list by applying f componentwise: $head(ext(f)(\sigma)) = f(head(\sigma))$ and $tail(ext(f)(\sigma)) = ext(f)(tail(\sigma))$

5 Algebras and induction

Induction is used both as a definition principle and as a proof principle [1]. The first real step is to reformulate ordinary induction in a more abstract way, using initiality. More precisely, using initiality for “algebras of a functor”. The abstract reformulation of induction that can be defined as:

$$\boxed{\text{induction} = \text{use of initiality for algebras}}$$

An algebra is initial if for an arbitrary algebra there is a unique homomorphism from initial algebra to arbitrary algebra. Initiality involves unique existence, which has two aspects:(i) Existence and (ii) Uniqueness. In uniqueness proofs, one shows that two functions acting on an initial algebra are the same by showing that they are both homomorphisms.

In this section, we will describe structures intended to store data elements in particular configurations. It turns out that the types of such configurations are suitably described by functors and the structures themselves arise as algebras for such functors. At first we can start by considering the polynomial functors. Polynomial functors are inductively defined as the least collection of functors containing the identity Id and constant functors for all objects K in the category, closed by functor composition and finite application of product and coproduct functors.

Let T be the polynomial functor $T(X) = 1+X+(X \times X)$ and consider for a set U a function $a : T(U) \to U$. Such a map a may be identified with a 3-cotuple $[a_1, a_2, a_3]$of maps $a_1 : 1 \to U$, $a_2 : U \to U$, $a_3 : U \times U \to U$ giving us three seperate functions going into the set U They form an example of an algebra (of the functor T): a set together with a (cotupled) number of functions going into that set. For example, if a group G has unit element $e : 1 \to G$, inverse function $i : G \to G$ and multiplication function $m : G \times G \to G$, then these three maps form an algebra $[e, i, m] : T(G) \to G$.

Syntactically, a data structure is described by a set of operations which specify how its values to be produced. A sequence, for example, is either empty or built by adding an element to the front of a pre-existing sequence. A binary tree signature includes an empty constant and a node constructor whereby data and two other trees are aggregated to become the root node of a new tree. These two examples can be modeled by polynomial functors, which are basically n-ary sums of m-ary products. For example,

$$\begin{aligned}
\mathbf{T_{Nat}}X &= 1 + X \quad (natural \quad numbers) \\
\mathbf{T_{Seq}}X &= 1 + \mathit{Data} \times X \quad (sequences) \\
\mathbf{T_{Bin}}X &= 1 + \mathit{Data} \times X \times X \quad (binary \quad trees) \\
\mathbf{T_{Lef}}X &= \mathit{Data} + X \times X \quad (leaf \quad trees)
\end{aligned}$$

All constructors of a given type can be grouped together into a single operation. For example, the constructors of a sequence are

$$[\mathit{nil}, \mathit{cons}] : 1 + \mathit{Data} \times X \to X$$

In general, if the shape of one of these structures is specified by a functor T, the structure itself is given as a map

$$d : TD \to D$$

i.e., as a T-algebra. Concrete structures are, therefore, obtained by specifying both the carrier set D and map d. Formally, we define,

Definition 5.1. *Let T be a functor. An algebra of T is a pair consisting of a set U and a function $a : T(U) \to U$. The set U is the carrier of the algebra, and the function a is the algebra structure, or also the operation of the algebra.*

For example, the zero and successor functions $0 : 1 \to N$, $S : N \to N$ on the natural numbers form an algebra $[0, S] : 1+N \to N$ of the functor $T(X) = 1+X$. And the set of A-labeled finite binary trees $Tree(A)$ comes with functions $nil : 1 \to Tree(A)$ for the empty tree, and $node : Tree(A) \times A \times Tree(A) \to Tree(A)$ for constructing a tree out of two subtrees and a node label. Together, nil and node form an algebra $1 + (Tree(A) \times A \times Tree(A)) \to Tree(A)$ of the functor $S(X) = 1 + (X \times A \times X)$.

6 Algebras homomorphism

Definition 6.1. *Let T be a functor with algebras $a : T(U) \to U$ and $b : T(V) \to V$. A* homomorphism of algebras *from (U, a) to (V, b) is a function $f : U \to V$ between the carrier sets which commutes with the operations [1]: $f \circ a = b \circ T(f)$ in*

$$\begin{tikzcd}
T(U) \arrow[r, "a"] \arrow[d, "T(f)"'] & U \arrow[d, "f"] \\
T(V) \arrow[r, "b"'] & V
\end{tikzcd}$$

Suppose we have two algebras $l_1 : 1 \to U_1$, $c_1 : A \times U_1 \to U_1$ and $l_2 : 1 \to U_2$, $c_2 : A \times U_2 \to U_2$. A homomorphism of algebras from the first to the second consists of a function $f : U_1 \to U_2$ between the carriers with $f \circ c_1 = c_2 \circ (id \times f)$. In two diagrams:

$$\begin{tikzcd}
1 \arrow[r, "l_1"] \arrow[d, equal, "1"'] & U_1 \arrow[d, "f"] \\
1 \arrow[r, "l_2"'] & U_2
\end{tikzcd}
\qquad
\begin{tikzcd}
A \times U_1 \arrow[r, "c_1"] \arrow[d, "id \times f"'] & U_1 \arrow[d, "f"] \\
A \times U_2 \arrow[r, "c_2"'] & U_2
\end{tikzcd}$$

These two diagrams can be combined into a single diagram:

$$\begin{tikzcd}
1 + (A \times U_1) \arrow[r, "{[l_1, c_1]}"] \arrow[d, "id + id \times f"'] & U_1 \arrow[d, "f"] \\
1 + (A \times U_2) \arrow[r, "{[l_2, c_2]}"'] & U_2
\end{tikzcd}$$

i.e., for the list-functor $T(X) = 1 + (A \times X)$,

$$\begin{tikzcd}
T(U_1) \arrow[r, "{[l_1, c_1]}"] \arrow[d, "T(f)"'] & U_1 \arrow[d, "f"] \\
T(U_2) \arrow[r, "{[l_2, c_2]}"'] & U_2
\end{tikzcd}$$

Definition 6.2. *An algebra $a : T(U) \to U$ of a functor T is initial if for each algebra $b : T(V) \to V$ there is a unique homomorphism of algebras from (U, a)*

to (V, b) [1]. Diagrammatically we express this uniqueness by a dashed arrow, call it f, in

$$\begin{tikzcd}
T(U) \arrow[r, "a"] \arrow[d, dashed, "T(f)"'] & U \arrow[d, dashed, "f"] \\
T(V) \arrow[r, "b"'] & V
\end{tikzcd}$$

As a first example [1], we shall describe the set N of natural numbers as initial algebra. Consider the set N of natural number with its zero and successor function $0 : 1 \to \mathsf{N}$ and $S : \mathsf{N} \to \mathsf{N}$. These functions combine into a single function $[0, S] : 1 + \mathsf{N} \to \mathsf{N}$, forming an algebra of the functor $T(X) = 1 + X$. We will show that this map $[0, S] : 1+\mathsf{N} \to \mathsf{N}$ is the initial algebra of this functor.

To prove initiality, assume we have an arbitrary set U carrying a T-algebra structure $[u, h] : 1 + U \to U$. We have to deine a homomorphism $f : \mathsf{N} \to U$. We try iteration: $f(n) = h^{(n)}(u)$. That is, $f(0) = u$ and $f(n + 1) = h(f(n))$.

These two equations express that we have a commuting diagram

$$\begin{tikzcd}
1 + \mathbf{N} \arrow[r, "{[0,S]}"] \arrow[d, "id+f"'] & \mathsf{N} \arrow[d, "f"] \\
1 + U \arrow[r, "{[u,h]}"'] & U
\end{tikzcd}$$

making f a homomorphism of algebras. This can be verified easily by distinguishing for an arbitrary element $x \in 1+\mathsf{N}$ in the upper-left corner the two cases $x = (0, *)$=$k(*)$ and $x = (1, n) = k('(n))$, for $n \in \mathsf{N}$. In the first case $x = k(*)$ we get, $f([0, S](k(*))) = f(0) = u = [u, h](k(*)) = [u, h]((id + f)(k(*)))$.

In the second case [1] $x = k('(n))$ we similarly check: $f([0, S](k('(n))) = f(S(n)) = u(f(n)) = [u, h](k('(f(n))) = [u, h]((id + f)(k('(n)))$.

Hence we may conclude that $f([0, S](x)) = [u, h]((id + f)(x))$, for all $x \in 1 + \mathsf{N}$, i.e. that $f \circ [0, S] = [u, h] \circ (id + f)$.

We also have to show that f is the only map making the diagram commute. If $g : \mathsf{N} \to U$ also satisfies $g \circ [0, S] = [u, h] \circ (id + g)$, then $g(0) = u$ and$g(n + 1) = h(g(n))$, by the same line of reasoning followed above. Hence $g(n) = f(n)$ by induction on n, so that $g = f : \mathsf{N} \to U$.

7 Properties of Initial Algebras

Lemma 6.1

(i) Initial T-algebras, if they exist, are unique, up-to-isomorphism of algebras. (ii)The operation (constructor) of an initial algebras is an isomorphism: if $a : T(U) \to U$ *is initial algebra, then* a *has an inverse* $a^{-1} : U \to T(U)$.

The first point tells us about the initial algebra of a functor T. And the second point - which is due to Lambek - says that an initial algebra $T(U) \to U$ is a fixed point $T(U) \cong U$ of the functor T [2].

Proof. (i) Suppose both $a : T(U) \to U$ and $a' : T(U') \to U'$ are initial algebras of the functor T. By initiality of a there is a unique algebra map $f : U \to U'$. Similarly, by initiality of a' there is a unique algebra map $f' : U' \to U$ in the other direction:

$$\begin{tikzcd}
T(U) \arrow[r, "a"] \arrow[d, dotted, "T(f)"'] & U \arrow[d, dotted, "f"] \\
T(U') \arrow[r, "a'"'] & U'
\end{tikzcd}
\qquad
\begin{tikzcd}
T(U') \arrow[r, "a'"] \arrow[d, dotted, "T(f')"'] & U' \arrow[d, dotted, "f'"] \\
T(U) \arrow[r, "a"'] & U
\end{tikzcd}$$

Here, we use the existence part of initiality. The uniqueness part gives us that the two resulting algebra maps $(U, a) \to (U, a)$, namely $f \circ f'$ and id in:

$$\begin{tikzcd}
T(U) \arrow[r, "a"] \arrow[d, "T(f)"'] & U \arrow[d, "f"] \\
T(U') \arrow[r, "a'"] \arrow[d, "T(f')"'] & U' \arrow[d, "f'"] \\
T(U) \arrow[r, "a"'] & U
\end{tikzcd}
\qquad
\begin{tikzcd}
T(U) \arrow[r, "a"] \arrow[d, "T(id)"'] & U \arrow[d, "id"] \\
T(U) \arrow[r, "a"'] & U
\end{tikzcd}$$

must be equal, i.e.,that $f' \circ f = id$. Uniqueness of algebra maps $(U', a') \to (U', a')$ similarly yields $f \circ f' = id$. Hence f is an isomorphism of algebras.

(ii) Let $a : T(U) \to U$ be initial T-algebra. In order to show that the function a is an isomorphism, we have to produce an inverse function $U \to T(U)$. Initiality of (U, a) can be used to define functions out of U to arbitrary algebras. Since we seek a function $U \to T(U)$, we have to put an algebra structure on the set $T(U)$. By applying the functor T to the function a, we get,

$T(a) : T(T(U)) \to T(U)$. This function $T(a)$ gives by initiality of $a : T(U) \to U$ rise to a function $a' : U \to T(U)$ with $T(a) \circ T(a') = a' \circ a$ in:

$$\begin{tikzcd}
T(U) \arrow[r, "a"] \arrow[d, dotted, "T(a')"'] & U \arrow[d, dotted, "a'"] \\
T(T(U)) \arrow[r, "T(a)"'] & T(U)
\end{tikzcd}$$

The function $a \circ a' : U \to U$ is an algebra map $(U, a) \to (U, a)$:

$$\begin{tikzcd}
T(U) \arrow[r, "a"] \arrow[d, "T(a')"'] & U \arrow[d, "a'"] \\
T(T(U)) \arrow[r, "T(a)"] \arrow[d, "T(a)"'] & T(U) \arrow[d, "a"] \\
T(U) \arrow[r, "a"'] & U
\end{tikzcd}$$

so that $a \circ a' = id$ by uniqueness of algebra maps $(U, a) \to (U, a)$. But then

$$\begin{aligned}
a' \circ a &= T(a) \circ T(a') \\
&= T(a \circ a') \\
&= T(id) \\
&= id
\end{aligned}$$

Hence $a : T(U) \to U$ is an isomorphism with a' as its inverse.

□

8 Strong Category and strong functor

If X is a Cartesian category (i.e. has a binary product and a terminal object), an X **strong category** Y is a cartesian category Y with a bifunctor, called an X action [4].

$$- \oslash - : Y \times X \to Y$$

that is compatible with the coherent natural isomorphism
$ass : Y \oslash (X_1 \times X_2) \to (Y \oslash X_1 \oslash X_2)$
$rid : (Y \oslash 1) \to Y$
$rep : Y \oslash X \to Y \times (1 \oslash X)$

X **strong functors** are functors between X-strong categories $\mathbf{C}$ and $\mathbf{D}$ equipped with a natural transformations [4].

$$\theta^F_{A,X} : F(A) \times X \to F(A \times X)$$

called an **X strength** such that the following two diagrams commute.

$$\begin{tikzcd}
F(A) \times X \arrow[r, "\theta^F_{A,X}"] \arrow[dr, "\pi_0"'] & F(A) \times X \arrow[d, "F(\pi_0)"] \\
 & F(A)
\end{tikzcd}$$

$$\begin{tikzcd}
F(A) \times (B \times C) \arrow[r, "ass"] \arrow[dd, "\theta^F_{A,B\times C}"'] & (F(A) \times B) \times C \arrow[d, "\theta^F_{A,B} \times I"] \\
 & F(A \times B) \times C \arrow[d, "\theta^F_{A\times B,C}"] \\
F((A \times B) \times C) \arrow[r, "F(ass)"'] & F((A \times B) \times C)
\end{tikzcd}$$

9 Strong Data Types and Initial Algebras

The category - theoretic explanation of recursive types is based on the idea that types constitute objects of a category C, and type constructors are functors on C. Given a recursive type definition in a functional language, it is always possible to derive an endofunctor $F : C \to C$ that captures the recursive shape(or signature) of the datatype. Thus, the recursive type turns out to be an object T such that $T \cong FT$, i.e. a fixpoint of F [3]. Least fixpoints of covariant functors give rise to inductive types which can be understood as initial functor-algebras, a generalization of the usual notion of term algebras over a given signature. An inductive type together with an interpretation of its constructors is given by an initial F algebra, i.e. an initial object in the category of algebras. For instance, given a datatype for natural numbers, $Nat = Zero \mid Succ\ Nat$ its signature is captured by a functor $N : C \to C$ such that $NU = 1 + U$ and $Nf = id_1 + f$. It turns out that any N - algebra is a case analysis morphism $[u, h] : 1 + U \to U$, with $u : 1 \to U$ and $h : U \to U$. In particular, the initial algebra is given by $[Zero, Succ] : 1 + \mathbf{N} \to \mathbf{N}$ with $Zero : 1 \to \mathbf{N}$ and $Succ : \mathbf{N} \to \mathbf{N}$. Thus, a data type can be viewed as an algebra whose operations are given by the data type constructors [3]. The initial datatype for natural numbers satisfy the following universal properties:

$$\begin{tikzcd}
1+\mathbf{N} \arrow[r, "{[Zero,Succ]}"] \arrow[d, dotted, "id+f"'] & \mathbf{N} \arrow[d, dotted, "f"] \\
1+U \arrow[r, "{[u,h]}"'] & U
\end{tikzcd}
\qquad
\begin{tikzcd}
1 \arrow[r, "Zero"] \arrow[d, equal, "1"'] & \mathbf{N} \arrow[d, dotted, "f"'] & \mathbf{N} \arrow[l, "Succ"'] \arrow[d, dotted, "f"] \\
1 \arrow[r, "u"'] & U & U \arrow[l, "h"]
\end{tikzcd}$$

Cockett and Spencer showed one can achieve a definition of fold for any inductive type that is $s$trong, i.e. that is given by an algebra which is initial with parameters. Indeed, $f$old may be regarded as a 'catamorphism with parameters'[3]. It is common to find recursive functions on datatypes that require extra parameters for their computation such as in a higher-order language currying is used. A recursive function that requires parameters may be given as a curried definition by recursion on the datatype that yields a function on the parameters as result. In categorical terms, a cartesian closed category is used in a higher-order language for every pair of objects A and B, there is an exponential object $[A \to B]$ satisfying an universal property. Consider functions on the natural numbers such as given a constant $u : 1 \to U$ and a function $h : U \to U$ the natural numbers catamorphism is the unique function $f : \mathbf{N} \to U$ such that $f\ Zero = u$ and $f(Succ\ n) = h(f\ n)$. Thus, by a higher-order catamorphism of type $N \to [X \to U]$ a recursive function can be presented that admits an object of parameters X and returs a value of type U, for certain $u : [X \to U]$ and $h : [X \to U] \to [X \to U]$. This is the case for addition $add : \mathbf{Nat} \to [\mathbf{Nat} \to \mathbf{Nat}]$ and the curried recursive definition is: $add\ Zero\ n = n$ and $add\ (Succ\ m)n = Succ(add\ m\ n)$.

On the contrary, in the first order language [3], the absence of the currying feature leads to the necessity of modifying the definition of catamorphism in order to explicitly manage parameters. The new functional-called $f$old is then introduced as a morphism of two variables $f : \mathbf{N} \times X \to U$ by recursion on $\mathbf{N}$ (the inductive type) that uses the second variable as parameter. Such as, given the functions $u : X \to U$ and $h : U \times X \to U$, the natural number fold is the unique function $f : \mathbf{N} \times X \to U$ such that: $f(Zero, x) = h(x)$ and $f(Succ\, n, x) = h(f(n, x), x)$. Now addition is given by $add : \mathbf{Nat} \times \mathbf{Nat} \to \mathbf{Nat}$. So the traditional number-theoretic definition is $add(Zero, n) = n$ and $add(Succ\ m, n) = Succ(add(m, n))$.

$$\begin{tikzcd}
\mathbf{Nat} \times \mathbf{1} \arrow[r, "{\langle \pi_0, \pi_1 0 \rangle}"] \arrow[d, equal] & \mathbf{Nat} \times \mathbf{Nat} \arrow[d, dotted, "add"'] & \mathbf{Nat} \times \mathbf{Nat} \arrow[l, "{\langle \pi_0, \pi_1 s \rangle}"'] \arrow[d, dotted, "{\langle \pi_0, add \rangle}"] \\
\mathbf{Nat} \times \mathbf{1} \arrow[r, "\pi_0"'] & \mathbf{Nat} & \mathbf{Nat} \times \mathbf{Nat} \arrow[l, "\pi_1 s"]
\end{tikzcd}$$

Cockett and Spencer addressed the definition of fold in a not necessarily closed category by introducing the concept of strong initiality which is based on the concept of a strong functor. In fact, initiality with parameters can be stated only for those datatypes whose signature is captured by a so-called strong functor [3].

Tatsuya Hagino's thesis introduced two types of datatype declarations: initial datatype and final datatypes for categorical programming language. The initial datatype is declared as follows [4]:

$$data \;\; L(A) \to C = c_1 : E_1(A, C) \to C| \;\; ...| \;\; c_n : E_n(A, C) \to C.$$

The canonical maps associated with $L(A)$ is $c_i : E_i(A, L(A)) \to L(A)$ which is called constructors. The initial datatypes satisfy the following universal property:

$$\begin{tikzcd}
E_i(A, L(A)) \arrow[r, "c_i"] \arrow[d, dotted, "E_i(A, f_L(h))"'] & L(A) \arrow[d, dotted, "f_L(h))"] \\
E_i(A, C) \arrow[r, "h_i"'] & C
\end{tikzcd}$$

The diagram for a strong initial datatype is

$$\begin{tikzcd}
E_i(A, L(A)) \times X \arrow[r, "c_i \times id_X"] \arrow[d, "\langle \theta_2^{E_i}, p_1 \rangle"'] & L(A) \times X \arrow[dd, dotted, "fold\{hs\}"] \\
E_i(A, L(A) \times X) \times X \arrow[d, dotted, "E_i(A, fold\{hs\}) \times id_X"'] & \\
E_i(A, C) \times X \arrow[r, "h_i"'] & C
\end{tikzcd}$$

Here L is strong functor and (L, c) is strong datatype. For natural numbers the diagram for strong initiality is as follows:

$$\begin{tikzcd}
(1 + \mathbf{N}) \times X \arrow[r, "{[zero, succ] \times id_X}"] \arrow[d, "\langle \theta^{1+-}, p_1 \rangle"'] & \mathbf{N} \times X \arrow[dd, dotted, "fold"] \\
(1 + (X \times \mathbf{N})) \times X \arrow[d, dotted, "(1 + fold) \times id_X"'] & \\
(1 + U) \times X \arrow[r, "{[u, h]}"'] & U
\end{tikzcd}$$

In a cartesian closed category ordinary datatypes are necessarily strong datatypes. If (L, c) is an ordinary initial datatype then it satisfies the simple algebra initiality property. Suppose $h_i : E_i(A, C) \times X \to C$ is the collection of programmer-chosen maps [4]. Using the h_i^* (particular transpose) for the ordinary initial datatype L, it is possible to construct the diagram below to obtain a unique arrow $a : L(A) \to X \Rightarrow C$ and $fold\{hs\} = (a \times id_X); eval$

$$\begin{tikzcd}
E_i(A, L(A)) \times X \arrow[r, "c_i \times id_X"] \arrow[d, swap, "E_i(id_A, a) \times id_X"] & L(A) \times X \arrow[d, "a \times id_X"] \\
E_i(A, X \Rightarrow C) \times X \arrow[r, "h_i^* \times id_X"] \arrow[d, swap, "\langle \theta_2^{E_i}, p_1 \rangle"] & (X \Rightarrow C) \times X \arrow[dd, "eval"] \\
E_i(A, (X \Rightarrow C) \times X) \times X \arrow[d, swap, "E_i(id_A, eval) \times id_X"] & \\
E_i(A, C) \times X \arrow[r, swap, "h_i"] & C
\end{tikzcd}$$

$$\begin{tikzcd}
(1 + \mathbf{N}) \times X \arrow[r, "{[zero, succ] \times id_X}"] \arrow[d, swap, "(1+a) \times id_X"] & \mathbf{N} \times X \arrow[d, "a \times id_X"] \\
(1 + (X \Rightarrow U)) \times X \arrow[r, "h_i^* \times id_X"] \arrow[d, swap, "\langle \theta^{1+-}, p_1 \rangle"] & (X \Rightarrow U) \times X \arrow[dd, "eval"] \\
(1 + ((X \Rightarrow U) \times X)) \times X \arrow[d, swap, "(1+eval) \times id_X"] & \\
(1 + U) \times X \arrow[r, swap, "{[u, h]}"] & U
\end{tikzcd}$$

Here $N \times X \to U$, $a : \mathbf{N} \to X \Rightarrow U$ and $[u, h] = h_i : (1 + U) \times X \to U$, $h_i^* : 1 + (X \Rightarrow U) \to X \Rightarrow U$.

Consider the initial algebra for the list datatype which is defined as $[nil, cons] : 1 + A \times List(A) \to List(A)$ where $nil : 1 \to List(A)$ and $cons : A \times List(A) \to$

$List(A)$. The initial datatype for list satisfy the following universal property:

$$\begin{tikzcd}
1+(A\times List(A)) \arrow[r, "{[nil,cons]}"] \arrow[d, dotted, "id+(id\times f)"'] & List(A) \arrow[d, dotted, "f"] \\
1+(A\times C) \arrow[r, "{[u,h]}"'] & C
\end{tikzcd}
\qquad
\begin{tikzcd}
1 \arrow[r, "nil"] \arrow[d, equal, "1"'] & List(A) \arrow[d, dotted, "f"'] & A\times List(A) \arrow[l, "cons"'] \arrow[d, dotted, "id\times f"] \\
1 \arrow[r, "u"'] & C & A\times C \arrow[l, "h"]
\end{tikzcd}$$

For list datatype the diagram for strong initiality is as follows:

$$\begin{tikzcd}
(1+(A\times List(A)))\times X \arrow[r, "{[nil,cons]\times id_X}"] \arrow[d, "{\langle \theta^{1+A^*-},p_1\rangle}"'] & List(A)\times X \arrow[dd, dotted, "fold"] \\
(1+(A\times(X\times List(A))))\times X \arrow[d, dotted, "(1+A\times fold)\times id_X"'] & \\
(1+(A\times C))\times X \arrow[r, "{[u,h]}"'] & C
\end{tikzcd}$$

As the ordinary datatypes in a cartesian closed category are necessarily strong datatypes, so the list datatype satisfies the following algebra initiality property:

$$\begin{tikzcd}
(1+(A\times List(A)))\times X \arrow[r, "{[nil,cons]\times id_X}"] \arrow[d, "(1+(A\times a))\times id_X"'] & List(A)\times X \arrow[d, "a\times id_X"] \\
(1+(A\times(X\Rightarrow C)))\times X \arrow[r, "h_i^*\times id_X"] \arrow[d, "{\langle \theta^{1+A^*-},p_1\rangle}"'] & (X\Rightarrow C)\times X \arrow[dd, "eval"] \\
(1+(A\times((X\Rightarrow C)\times X)))\times X \arrow[d, "(1+(A\times eval))\times id_X"'] & \\
(1+(A\times C))\times X \arrow[r, "{[u,h]}"'] & C
\end{tikzcd}$$

Here $X\times List(A)\rightarrow C$, $a : List(A)\rightarrow X\Rightarrow C$ and $[u,h]=h_i : (1+(A\times C))\times X\rightarrow C$, $h_i^* : 1+(A\times(X\Rightarrow C))\rightarrow X\Rightarrow C$.

9.1 Associativity of appending lists

From the definition of append [5]

$$append(x, y) \equiv \left\{ nil : () \mapsto y \quad \middle| cons : (a, c) \mapsto cons(a, c) \right\} (x)$$

To establish the associativity property we need to show that $append(x, append(y, z)) = append(append(x, y), z))$. Now we can define a factorizer form $append(, append(y, z)) = \left\{ nil : () \mapsto append(y, z) \quad \middle| cons : (a, c) \mapsto cons(a, c) \right\} (x)$. It can be denoted as pgm_l. Abbreviating $List(A)$ as $L(A)$, we have

$$\begin{tikzcd}
1 \times (L(A) \times L(A)) \arrow[r, "nil \times id"] \arrow[d, dotted, "id"'] & L(A) \times (L(A) \times L(A)) \arrow[d, dotted, "pgm_l"'] & (A \times L(A)) \times (L(A) \times L(A)) \arrow[l, "cons \times id"'] \arrow[d, dotted, "\langle ass^{-1};id \times pgm_l, p_1"] \\
1 \times (L(A) \times L(A)) \arrow[r, "p_1;append(-,-)"'] & L(A) & (A \times L(A)) \times (L(A) \times L(A)) \arrow[l, "p_0, cons"]
\end{tikzcd}$$

Using fold uniqueness rule for List(A), let $t(-) = append(append(,y), z)$ and thereby create the following subproblems:

$(i) append(append(nil(), y), z) = append(y, z)$

$(ii) append(append(cons(a, l), y), z) = cons(a, append(append(l, y), z))$

The proof of (i) is immediate that requires a single factorizer reduction and the proof of (ii) requires two reductions:

$$\begin{aligned}
append(append(cons(a, l), y), z) &= append(cons(a, append(l, y)), z) \\
&= cons(a, append(append(l, y), z))
\end{aligned}$$

Thus, appending lists established associativity property.

10 Coalgebra

In the semantics of programming [1], finite data types such as finite lists, have traditionally been modelled by initial algebras. Later final coalgebras were used in order to deal with infinite data types. Coalgebras, which are the dual of algebras, turned out to be suited, moreover, as models for certain types of automata and more generally, for (transition and dynamical) systems. An important property of initial algebras is that they satisfy the familiar principle of induction. Such a principle was missing for coalgebras until the work of Aczel (1988) on a theory of non-wellfounded sets, in which he introduced a proof principle nowadays called coinduction. It was formulated in terms of bisimulation, a notion originally stemming from the world of concurrent programming languages (Milner, 1980; Park, 1981). Using the notion of coalgebra homomorphism, the definition of bisimulation on coalgebras can be shown to be formally dual to that of congruence on algebras (Aczel and Mendler, 1989). Thus the three basic notions of universal algebra: algebra, homomorphism of algebras, and congruence, turn out to correspond to: coalgebra, homomorphism of coalgebras, and bisimulation, respectively.

Definition 10.1. *Let T be a functor. An coalgebra of T is a pair consisting of a set U and a function $c : U \to T(U)$. The set U is the carrier of the coalgebra, and the function c is the coalgebra structure, or also the operation of the coalgebra.*

Consider for example the functor $T(X) = A \times X$, where A is a fixed set. A coalgebra $U \to T(U)$ consists of two funcions $U \to A$ and $U \to U$. we can call $value : U \to A$ and $next : U \to U$. With these operations we can do two things, given an element $u \in U$:

1. produce an element in A, namely $value(u)$;

2. produce a next element in U, namely $next(u)$.

Now we can repeat 1. and 2. and form another element in A, namely $value(next(u))$. By proceeding in this way we can get for each element $u \in U$ an infinite sequence $(a_1, a_2, ...) \in A^{\mathbf{N}}$ of elements of $a_i = value(next^(n)(u)) \in A$. This sequence of elements that u gives rise to is what we can observe about u. Two elements $u_1, u_2 \in U$ may well give rise to the same sequence of elements of A, without actually being equal as elements of U. In such a case one calls u_1 and u_2 observationally indistinguishable, or bisimilar.

11 Homomorphism of coalgebras

Definition 11.1. *Let $T(X) = A \times X$ be a the "infinite list" functor with coalgebras $\langle h_1, t_1 \rangle : U_1 \to A \times U_1$ and $\langle h_2, t_2 \rangle : U_2 \to A \times U_2$. A* homomorphism of coalgebras *from the first to the second consists of a function $f : U_1 \to U_2$ between the carrier sets which commutes with the operations:$h_2 \circ f = h_1$ and $t_2 \circ f = f \circ t_1$ in [1]*

$$\begin{tikzcd}
U_1 \arrow[r, "f"'] \arrow[d, "h_1"'] & U_2 \arrow[d, "h_2"] \\
A \arrow[r, equal, "1"'] & A
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
U_1 \arrow[r, "f"] \arrow[d, "t_1"'] & U_2 \arrow[d, "t_2"] \\
U_1 \arrow[r, "f"'] & U_2
\end{tikzcd}$$

These two diagrams can be combined into a single diagram:

$$\begin{tikzcd}
U_1 \arrow[r, "f"] \arrow[d, "\langle h_1{,}t_1\rangle"'] & U_2 \arrow[d, "\langle h_2{,}t_2\rangle"] & \textit{i.e. into} & U_1 \arrow[r, "f"] \arrow[d, "\langle h_1{,}t_1\rangle"'] & U_2 \arrow[d, "\langle h_2{,}t_2\rangle"] \\
A \times U_1 \arrow[r, "id\times f"'] & A \times U_2 & & T(U_1)(f) \arrow[r, "T"'] & T(U_2)
\end{tikzcd}$$

Definition 11.2. *(i)A* homomorphism of coalgebras *from a $T-coalgebra$ $c_1 : U_1 \rightarrow T(U_1)$ to another $T-coalgebra$ $c_2 : U_2 \rightarrow T(U_2)$ consists of a function $f : U_1 \rightarrow U_2$ between the carrier sets which commutes with the operations: $c_2 \circ f = T(f) \circ c_1$ as expressed by the following diagram.*

$$\begin{tikzcd}
U_1 \arrow[r, "f"] \arrow[d, "c_1"'] & U_2 \arrow[d, "c_2"] \\
T(U_1)(f) \arrow[r, "T"'] & T(U_2)
\end{tikzcd}$$

(ii)A final coalgebra $d : W \rightarrow T(W)$ is a coalgebra such that for every coalgebra $c : U \rightarrow T(U)$ there is a unique map of coalgebras (U, c) to (W, d).

Now that we have seen the definition of initiality and finalty. At an informal level we can explain the similarities between the two [1]. A typical initiality diagram may be drawn as:

$$\begin{tikzcd}
T(U) \arrow[r, dashed, "T(f)"] \arrow[d, "initial\quad algebra"'] & T(V) \arrow[d, "base\ step\ plus\ next\ step"] \\
U \arrow[r, dotted, "``and-so-forth''"'] & V
\end{tikzcd}$$

The map "and-so-forth" that is defined in this diagram applies the "next step" operations repeatedly to the " base step". The pattern in a finality diagram is similar:

$$\begin{tikzcd}
V \arrow[r, dashed, "``and-so-forth''"] \arrow[d, "observe\ plus\ next\ step"'] & U \arrow[d, "final\quad coalgebra"] \\
T(V) \arrow[r, dotted, "T(f)"'] & T(U)
\end{tikzcd}$$

In this case the "and-so-forth" map captures the observations that arise by repeatedly applying the "next step" operation. This captures the observable behaviour. The technique for defining a function $f : V \rightarrow U$ by finality is thus: describe the direct observations together with the single next steps of f as a coalgebra structure on V. The function f then arises by repetition. Hence a coinductive definition of f does not determine f “at once”, but “step-by-step”.

Lemma 10.1

(i) Final coalgebras, if they exist, are uniquely determined (up-to-isomorphism).
(ii)A final coalgebra $W \to T(W)$ is a fixed point $W \cong T(W)$ of the functor T

12 Coinductive Datatypes

Coinductive types represent infinite structures.The general form of the coinductive datatype is as follows:

$$data \;\; C \to R(A) = d_1 : C \to E_1(A,C)| \;\; ...| \;\; d_n : C \to E_n(A,C).$$

This definition delivers to the system the type $R(A)$ and destructors $d_i, i = 1, ...n$[6]. An element of datatype $R(A)$ is broken down by destructors with type: $d_i : R(A) \to E_i(A, R(A))$. With respect to this map the coinductive datatype satisfy the following universal property:

$$\begin{tikzcd}
C \arrow[r, "g_i"] \arrow[d, dotted, "f_R(g)"'] & E_i(A,C) \arrow[d, dotted, "E_i(A,f_R(g))"] \\
R(A) \arrow[r, "d_i"'] & E_i(A,R(A))
\end{tikzcd}$$

For a strong coinductive datatype the diagram is

$$\begin{tikzcd}
C \times X \arrow[r, "\langle g_i,p_1\rangle"] \arrow[dd, dotted, "unfold\{gs\}"'] & E_i(A,C) \times X \arrow[d, "\theta_2^{E_i}"] \\
 & E_i(A,C \times X) \arrow[d, dotted, "E_i(A,unfold\{gs\})"] \\
R(A) \arrow[r, "d_i"'] & E_i(A,R(A))
\end{tikzcd}$$

Here the collection of functions are represented by $g_1, ...g_n$ as “gs”.

References

[1] Bart Jacobs, Jan Rutten, *A tutorial on (Co) Algebra and (Co) Induction*, Bulletin of the Europian Association of Theoretical Computer Science (1997).

[2] Anseok Joo, *Categorical ideas as expressed in the programming language charity.*

[3] Alberto Pardo, *A calculational approach to strong datatypes*, Technische Hochschule Darmstadt, Germany.

[4] J.Robin B. Cockett and Dwight Spencer, *Strong Categorical Datatypes-I*, *Canadian Mathematical Society Proceedings*, Vol. 13, AMS Montreal (1992).

[5] J.Robin B. Cockett and Dwight Spencer, *Strong Categorical Datatypes-II: A term logic for categorical programming*, Theoretical Computer Science, Vol 139 (1995).

[6] Dale Barry Yee, *Implementing the charity abstract machine*, Masters Thesis, University of Calgary (1995).