Coalgebras in Dependent Type Theory – The Saga Continues

Anton Setzer
Swansea University
Swansea, UK

September 8, 2010
1. Coalgebras as Defined By Elimination Rules

2. Using Destructors: Destructor Patterns, Objects

3. Codata and $\sim$

4. $\infty \ A$

5. Understanding Nested Algebras and Coalgebras

5. Model
Algebraic Data Types

Algebraic data types one of the main ingredients of Agda.

```
data List (A : Set) : Set where
  []     : List A
  _ :: _ : A → List A → List A
```

**Notation:**

```
[]' + A ::' X
```

stands for the labelled disjoint union, i.e. the set $B$ containing elements $[]'$ and $a ::' x$ for $a : A$ and $x : X$.

Let

$$F_A : \text{Set} \to \text{Set}$$

$$F_A X = []' + A ::' X$$
1. Coalgebras as Defined By Elimination Rules

Algebraic Data Types

\[ F_A X = \text{[]} + A ::' X \]

Then the following is essentially equivalent to the definition of List \( A \):

\[
\text{data List } (A : \text{Set}) : \text{Set} \text{ where } \\
\text{ intro : } F (\text{List } A) \rightarrow \text{List } A \\
\text{ where } \\
\text{ [] } = \text{ intro } [\text{]}' \\
\text{ a :: l } = \text{ intro } (a ::' l)
\]
Algebraic Data Types

The introduction elimination and equality rules for algebraic data types follow then from the diagram for initial $F$-algebras (denoted by $\mu F$)

$F (\mu F) \xrightarrow{\text{intro}} \mu F$

$F g \xrightarrow{\exists! g} \mu F$

$F A \xrightarrow{f} A$

One writes $\mu X.t$ for $\mu (\lambda X.t)$ e.g.

List $A = \mu X.[]' + A ::' X$
Final Coalgebras

Final Coalgebras $\nu F$ are obtained by reversing the arrows:

$$
\begin{array}{c}
A \xrightarrow{f} FA \\
\exists! g \\
\nu F \xleftarrow{\text{unfold}} F (\nu F)
\end{array}
$$

Again we write $\nu X.t$ for $\nu (\lambda X.t)$.

In weakly final coalgebras the uniqueness of $g$ is omitted.

Coalgebras can be used to model interactive programs and objects from object-oriented programming in dependent type theory.
Suggested Notation

\[
\text{coalg coList (A : Set) : Set where}
\]
\[
\text{unfold : coList A → [] + A ::' coList A}
\]

- To an element of \(\text{coList A}\) as above we can apply \(\text{unfold}\) as above.
- Furthermore from the finality we can derive the principle of guarded recursion:
  We can define \(f : B → \text{coList A}\) by saying what \(\text{unfold (f b)}\) is:
    - \([]'\)
    - \(a ::' l\) for some \(a : A, l : \text{coList A}\)
    - \(a ::' f \ b'\) for some \(a : A, b' : B\).
Example

\[\text{inclist} : \mathbb{N} \rightarrow \text{coList} \mathbb{N}\]
\[\text{unfold} (\text{inclist } n) = n ::' (\text{inclist } (n + 1))\]
1. Coalgebras as Defined By Elimination Rules

2. Using Destructors: Destructor Patterns, Objects

3. Codata and ∼

4. ∞ A

5. Understanding Nested Algebras and Coalgebras

5. Model
Using Several Destructors

When using *data* we had several constructors. Similarly we can allow for *coalg* several destructors. Example:

\[
\text{coalg Stream } (A : \text{Set}) : \text{Set} \text{ where}
\]
\[
\begin{align*}
\text{head} & : \text{Stream } A \rightarrow A \\
\text{tail} & : \text{Stream } A \rightarrow \text{Stream } A
\end{align*}
\]

\[
\text{inc} : \mathbb{N} \rightarrow \text{Stream } \mathbb{N}
\]
\[
\begin{align*}
\text{head } (\text{inc } n) & = n \\
\text{tail } (\text{inc } n) & = \text{inc } (n + 1)
\end{align*}
\]
Nested Destructor Patterns

We can even define nested destructor patterns (Andreas Abel):

\[
\begin{align*}
\text{inc'} &: \mathbb{N} \rightarrow \text{Stream } \mathbb{N} \\
\text{head } (\text{inc'} n) &= n \\
\text{head } (\text{tail } (\text{inc'} n)) &= n + 1 \\
\text{tail } (\text{tail } (\text{inc'} n)) &= \text{inc'} (n + 2)
\end{align*}
\]
Bisimulation

\[
\text{coalg } \_ \approx \_ \{ A : \text{Set} \} : \text{Stream } A \rightarrow \text{Stream } A \rightarrow \text{Set} \quad \text{where}
\]
\[
\text{headeq} : \{ l / l' : \text{Stream } A \} \rightarrow l \approx l' \rightarrow \text{head } l \cong \text{head } l'
\]
\[
\text{taileq} : \{ l / l' : \text{Stream } A \} \rightarrow l \approx l' \rightarrow \text{tail } l \cong \text{tail } l'
\]
Example Proof

lemma : (n : ℕ) → inc n ≈ inc' n
headeq (lemma n) = refl
headeq (taileq (lemma n)) = refl
taileq (taileq (lemma n)) = lemma (n + 2)

(Slide improved after some comments during the talk).
Fibonacci

\[
\begin{align*}
\text{fib : Stream } \mathbb{N} \\
\text{head fib} &= 1 \\
\text{head (tail fib)} &= 1 \\
\text{tail (tail fib)} &= \text{addStream fib (tail fib)}
\end{align*}
\]

Not guarded recursion but can be justified by sized types.

Or (not very useful but the result of unfolding the sized version(?)�):

\[
\begin{align*}
\text{fib : } \mathbb{N} \rightarrow \text{Stream } \mathbb{N} \\
\text{head (fib zero)} &= 1 \\
\text{head (fib (suc zero))} &= 1 \\
\text{head (fib (suc (suc n)))} &= \text{head (fib n)} + \text{head (fib (suc n))} \\
\text{tail (fib n)} &= \text{fib (n + 1)}
\end{align*}
\]
We can combine constructor and destructor patterns:

\[
\begin{align*}
\text{inc}'' & : \mathbb{N} \rightarrow \text{Stream } \mathbb{N} \\
\text{head } (\text{inc}'' \text{ zero}) & = 0 \\
\text{head } (\text{tail } (\text{inc}'' \text{ zero})) & = 1 \\
\text{tail } (\text{tail } (\text{inc}'' \text{ zero})) & = \text{inc}'' 2 \\
\text{head } (\text{inc}'' (\text{suc } n)) & = \text{suc } n \\
\text{tail } (\text{inc}'' (\text{suc } n)) & = \text{inc}'' (n + 1)
\end{align*}
\]
Objects

\[ \text{coalg Stack (} A : \text{Set}) : \mathbb{N} \rightarrow \text{Set where} \]
\[
\text{top} : \{n : \mathbb{N}\} \rightarrow \text{Stack (suc } n) \rightarrow A \\
\text{pop} : \{n : \mathbb{N}\} \rightarrow \text{Stack (suc } n) \rightarrow \text{Stack } n
\]

\[
\text{push} : \{n : \mathbb{N}\} \rightarrow A \rightarrow \text{Stack } A \ n \rightarrow \text{Stack } A \ (n + 1) \\
\text{top (push } a \ l) = a \\
\text{pop (push } a \ l) = l
\]
Objects

calg Stack \((A : \text{Set}) : \mathbb{N} \rightarrow \text{Set}\) where

\[
\begin{align*}
\text{top} & : \{n : \mathbb{N}\} \rightarrow \text{Stack} (\text{suc } n) \rightarrow A \\
\text{pop} & : \{n : \mathbb{N}\} \rightarrow \text{Stack} (\text{suc } n) \rightarrow \text{Stack } n
\end{align*}
\]

The empty stack is introduced as follows:

\[
\text{emptystack} : \{A : \text{Set}\} \rightarrow \text{Stack } A \text{ zero}
\]

\[
() \quad \text{-- no destructor applies}
\]

Note that the coalgebra Stack zero has no destructors and contains exactly one element up to bisimularity.

(Slide improved after comments during the talk)
Question

- Can we get a good notion of a heap?
- Can we use this to define the class of queues efficiently?
1. Coalgebras as Defined By Elimination Rules

2. Using Destructors: Destructor Patterns, Objects

3. Codata and ~

4. $\infty A$

5. Understanding Nested Algebras and Coalgebras

5. Model
Complications with Coalgebras

Several constructors for `data` corresponds to disjoint unions of the argument types.

```
data List (A : Set) : Set where
  []   : List A
  _ :: _ : A → List A → List A
```
corresponds to

```
data List (A : Set) : Set where
  intro : 1 + (A × List A) → List A
```
Complications with Coalgebras

Several destructors for `coalg` corresponds to the product of the argument types:

```haskell
data Stream (A : Set) : Set where
  head  :  Stream A → A
  tail  :  Stream A → Stream A
```
corresponds to

```haskell
data Stream (A : Set) : Set where
  unfold :  Stream A → A × Stream A
```
Complications with Coalgebras

If we dualise data types introduced by several constructors, we obtain types which are more complicated to describe:

List looks nice:

```
data List (A : Set) : Set where
  []   : List A
  _ :: _ : A → List A → List A
```

whereas colists don’t look nice

```
coalg coList (A : Set) : Set where
  unfold : coList A → [] + A ::' coList A
```
Codata types seem to solve this problem:

```haskell
codata coList (A : Set) : Set where
  [] : coList A
  _ :: _ : A → coList A → coList A
```

So elements of `coList A` are now introduced by introduction rules which allows to define the disjoint union nicely.

Idea is that elements of `coList A` are infinitary lists:

- `n_1 :: n_2 :: n_3 :: · · ·`
- `n_1 :: n_2 :: n_3 :: · · · :: n_k :: []`
3. Codata and ~

Problem of codata

- No normalisation, e.g.

\[
\text{inc } 0 = 0 :: 1 :: 2 :: \ldots
\]

- Undecidability of equality.

\[
f 0 :: f 1 :: \ldots = g 0 :: g 1 :: \ldots \iff \forall n. f \ n = g \ n
\]

In case of coalgebras

- Elements of coalgebras are not expanded indefinitely. They are only expanded if unfold is applied to them.

- In case of weakly final coalgebras equality of elements of the coalgebras is equality of the underlying algorithms.
Pseudo-Constructors

If we have

\[
\text{coalg coList (}A : \text{Set)} : \text{Set where }
\text{unfold : coList A} \rightarrow \text{[]' + A ::' coList A}
\]

we can define by guarded recursion

\[
\text{[]} : \text{coList A} \\
\text{unfold [] = []'}
\]

\[
_ :: _ : A : \text{Set} \rightarrow A \rightarrow \text{coList A} \rightarrow \text{coList A} \\
\text{unfold (}a :: l) = a ::' l
\]
However we do not have

\[ \text{unfold } l = a :: l' \implies l = a :: l \]

So elements of coList A are not of the form \([\] \) or \(a :: l\).

But behave like \([\] \) or \(a :: l\).
codata coList (A : Set) : Set where
  [] : coList A
  _ :: _ : A → coList A → coList A

is an abbreviation for

calg coList (A : Set) : Set where
  unfold : coList A → []' + A ::' coList A

[] : A : Set → coList A
unfold [] = []'

_ :: _ : A : Set → A → coList A → coList A
unfold (a :: l) = a ::' l
Furthermore let

\[ s \sim t \iff \text{unfold } s = \text{unfold } t \]

Then

\[
\begin{align*}
\text{unfold } s &= []' \iff s \sim [] \\
\text{unfold } s &= a ::' l \iff s \sim a :: l
\end{align*}
\]

so there is no need to write []' or _ ::' _ or unfold.

Unfortunately \( \sim \) was replaced by = which misled the users.
1. Coalgebras as Defined By Elimination Rules

2. Using Destructors: Destructor Patterns, Objects

3. Codata and ∼

4. ∞ A

5. Understanding Nested Algebras and Coalgebras

5. Model
Nils Danielsson’s ∞

Nils Danielsson and Thorsten Altenkirch suggested to have the following

\[
\begin{align*}
\infty & : \text{Set} \rightarrow \text{Set} \\
♭ & : \{B : \text{Set}\} \rightarrow B \rightarrow \infty B \\
♮ & : \{B : \text{Set}\} \rightarrow \infty B \rightarrow B
\end{align*}
\]

∞ B denote coalgebraic arguments in a definition (which can be “expanded infinitely”) and one defines \text{coList} A as

\[
\text{data coList} (A : \text{Set}) : \text{Set} \text{ where} \\
[] & : \text{coList} A \\
\_ :: \_ & : A \rightarrow \infty (\text{coList} A) \rightarrow \text{coList} A
\]
What is $\infty B$?

$\infty B$ cannot mean $\nu X. B$ since $\nu X. B$ is as a (non-weakly) final coalgebra isomorphic to $B$: With $F X = B$ we get

$$
\begin{array}{c}
X \xrightarrow{f} F X = B \\
\downarrow \exists! g \\
B \xrightarrow{id} F B = B
\end{array}
$$

$$
F g = \text{id}
$$
Underlying reason

\(\nu\) gives something real only if applied to a functor. Applied to a set (or \(\lambda X. A\) for a set \(A\)) it is essentially the identity. So \(\infty\) must be something like \((\text{Set} \rightarrow \text{Set}) \rightarrow \text{Set}\).
What is $\infty A$?

What is meant by it is, that if $A$ is defined as an algebraic data type, $\infty A$ is defined mutually coalgebraically:

```haskell
data coList (A : Set) : Set where
  []      : coList A
  _ :: _  : A → \infty (coList A) → coList A
```

stands for

```haskell
data coList (A : Set) : Set where
  []      : coList A
  _ :: _  : A → \infty (coList A) → coList A

coalg \infty (coList _) (A : Set) : Set where
  \_ : \infty (coList A) → coList A
```
Order between data/codata

data coList (A : Set) : Set where
  [] : coList A
  _ :: _ : A → ∞ (coList A) → coList A

coalg ∞ (coList_) (A : Set) : Set where
  ♭ : ∞ (coList A) → coList A

But there are two interpretations of the above:

1. 

\[
\begin{align*}
F(X, Y) & = [] + A :: Y \\
G(X, Y) & = X \\
F'(Y) & = \mu X.F(X, Y) = \mu X.[] + A :: Y \\
& \Rightarrow [] + A :: Y \\
\infty (coList A) & = \nu Y.G(F'(Y), Y) = \nu Y.F'(Y) \\
& \Rightarrow \nu Y.[] + A :: Y \\
coList A & = F'(\infty (coList A)) \\
& = [] + A :: (\infty (coList A))
\end{align*}
\]
Order between data/codata

\[ \text{data coList } (A : \text{Set}) : \text{Set where} \]
\[ [] : \text{coList } A \]
\[ _ :: _ : A \to \infty (\text{coList } A) \to \text{coList } A \]

\[ \text{coalg } \infty (\text{coList}_-) (A : \text{Set}) : \text{Set where} \]
\[ \vdash : \infty (\text{coList } A) \to \text{coList } A \]

2.

\[ G(X, Y) = X \]
\[ F(X, Y) = [] + A :: Y \]
\[ G'(X) = \nu Y. G(X, Y) = \nu Y. X \]
\[ \equiv X \]

\[ \text{coList } A = \mu X. F(X, G'(X)) \cong \mu X. F(X, X) \]
\[ = \mu X. [] + A :: X \]

\[ \infty (\text{coList } A) = G'(\text{coList } A) \]
\[ \equiv \text{coList } A \]
First solution gives the desired result.

Origin of problem:

- If we have two functors $F(X, Y)$, and $G(X, Y)$ and if we want to minimize $X$ and maximize $Y$ there are two solutions:
  - Minimize $X$ as a functor depending on $Y$. Then maximize $Y$.
  - Maximize $Y$ as a functor depending on $X$. Then minimize $X$.

- With mutual data types this problem didn’t occur since if we minimize both $X$ and $Y$, the order doesn’t matter.
1. Coalgebras as Defined By Elimination Rules

2. Using Destructors: Destructor Patterns, Objects

3. Codata and \sim

4. \infty A

5. Understanding Nested Algebras and Coalgebras

5. Model
In general we want to be able to form arbitrary combinations of $\mu$ and $\nu$. Idea: minimize and maximize in the order of occurrence.
data $A$ : Set where
  intro$_0$ : $F(A, B, C, D) \to A$
coalg $B$ : Set where
  unfold$_0$ : $B \to G(A, B, C, D)$
data $C$ : Set where
  intro$_1$ : $H(A, B, C, D) \to C$
coalg $D$ : Set where
  unfold$_1$ : $D \to K(A, B, C, D)$

to be interpreted as:

\[
\begin{align*}
  F_0(Y, Z, Z') &= \mu X. F(X, Y, Z, Z') & \text{A in terms of } Y, Z, Z' \\
  G_1(Z, Z') &= \nu Y. G(F'(Y, Z, Z'), Y, Z, Z') & \text{B in terms of } Z, Z' \\
  F_1(Z, Z') &= F_0(G_1(Z, Z'), Z, Z') & \text{A in terms of } Z, Z' \\
  H_2(Z') &= \mu Z. H(F_1(Z, Z'), G_1(Z, Z'), Z, Z') & \text{C in terms of } Z' \\
  G_2(Z') &= G_1(H_2(Z'), Z') & \text{B in terms of } Z' \\
  F_2(Z') &= F_1(H_2(Z'), Z') & \text{A in terms of } Z' \\
  D &= \nu Z'. K(F_2(Z'), G_2(Z'), H_2(Z'), Z') & \text{Final Value of } D \\
  C &= H_2(D) & \text{Final Value of } C \\
  B &= G_2(D) & \text{Final Value of } B \\
  A &= F_2(D) & \text{Final Value of } A
\end{align*}
\]
Example: \textit{coList }A

data coList (A : Set) : Set where

\[
\emptyset : \text{coList }A
\]

_ :: _ : A \rightarrow \infty (\text{coList }A) \rightarrow \text{coList }A

stands for

data coList (A : Set) : Set where

\[
\emptyset : \text{coList }A
\]

_ :: _ : A \rightarrow \infty (\text{coList }A) \rightarrow \text{coList }A

coalg \infty (\text{coList}_-) (A : \text{Set}) : \text{Set where}

\[\natural : \infty (\text{coList }A) \rightarrow \text{coList }A\]
inclist

inclist : \( \mathbb{N} \to \infty \) (coList \( \mathbb{N} \))

\[ \vdash (\text{inclist } n) = n :: \text{inclist } (n + 1) \]

or

inclist \( n \sim \vdash (n :: \text{inclist } (n + 1)) \)

With

\[ s \triangleright t : \iff \vdash s = t \]

we get

inclist \( n \triangleright n :: \text{inclist } (n + 1) \)
1. Coalgebras as Defined By Elimination Rules

2. Using Destructors: Destructor Patterns, Objects

3. Codata and ∼

4. ∞ A

5. Understanding Nested Algebras and Coalgebras

5. Model
Model

Form a term model with reduction rules corresponding to the equalities stated.
E.g. inclist is a function symbol with equality rule

\[
\text{unfold} (\text{inclist } n) = n :: (\text{inclist } (n + 1))
\]

Interpretation of \( \mu X. F(X) \):

\[
\llbracket \mu X. F(X) \rrbracket = \bigcap \{ X \subseteq \text{Term} \mid \text{intro}[\llbracket F(X) \rrbracket] \subseteq X \}
\]

Interpretation of \( \nu X. F(X) \):

\[
\llbracket \nu X. F(X) \rrbracket = \bigcup \{ X \subseteq \text{Term} \mid \text{unfold}[X] \subseteq \llbracket F(X) \rrbracket \}
\]
Conclusion

- Design decisions should be done by referring to the notion of coalgebras.
- Coalgebras with constructor/destructor patterns looks very neat.
- Other solutions $\sim$, $\infty$ don’t look very elegant at the moment and need a proper semantic treatment.
  - $\sim$ was a reasonable good abbreviation mechanism.
- If $A$ is a data type referring to $\infty A$, then $\infty A$ gets is meaning as a coalgebra defined implicitly mutually after the definition of $A$.
- Order of algebras coalgebras matters.
- Suggestion by Peter Hancock: Why not use $\mu$ and $\nu$?
  - Not really necessary, since we can built up expressions of nested $\mu$, $\nu$ using mutual algebras and coalgebras understood in our way.