Coalgebras in Dependent Type Theory – The Saga Continues

Anton Setzer
Swansea University
Swanesa, UK

September 8, 2010

Algebraic Data Types

Algebraic data types one of the main ingredients of Agda.

$$\begin{array}{lll} \multicolumn{3}{l}{\text{data List } (A : \text{Set}) : \text{Set where}} \\ \quad [] & : & \text{List } A \\ \quad \_ :: \_ & : & A \rightarrow \text{List } A \rightarrow \text{List } A \end{array}$$

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

$$\begin{array}{l} F_A : \text{Set} \rightarrow \text{Set} \\ F_A\, X = []' + A ::' X \end{array}$$

Algebraic Data Types

$F_A\ X = []' + A ::' X$

Then the following is essentially equivalent to the definition of $\mathrm{List}\ A$:

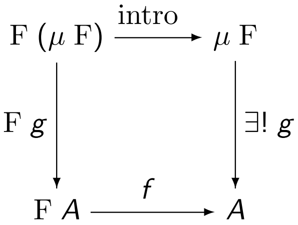
$$\begin{array}{l} \mathrm{data\ List}\ (A : \mathrm{Set}) : \mathrm{Set\ where} \\ \quad \mathrm{intro} \quad : \quad F\ (\mathrm{List}\ A) \to \mathrm{List}\ A \end{array}$$

where

$$\begin{array}{lcl} [] & = & \mathrm{intro}\ []' \\ a :: l & = & \mathrm{intro}\ (a ::' l) \end{array}$$

Algebraic Data Types

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



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

$$\mathrm{List}\ A = \mu X.[]' + A ::' X$$

Final Coalgebras

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

$$\begin{array}{ccc} A & \xrightarrow{\ \ f\ \ } & \mathrm{F}\ A \\ {\scriptstyle \exists ! g}\downarrow & & \downarrow{\scriptstyle \mathrm{F}\ g} \\ \nu\ \mathrm{F} & \xrightarrow{\ \mathrm{unfold}\ } & \mathrm{F}\ (\nu\ \mathrm{F}) \end{array}$$

Again we write $\nu X.t$ for $\nu\ (\lambda X.t)$.
In weakly final coalgbras 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

$$\begin{array}{l} \mathrm{coalg}\ \mathrm{coList}\ (A : \mathrm{Set}) : \mathrm{Set}\ \ \mathrm{where} \\ \quad \mathrm{unfold} : \mathrm{coList}\ A \rightarrow [\,] + A ::' \mathrm{coList}\ A \end{array}$$

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

Example

$$\text{inclist} : \mathbb{N} \to \text{coList}\ \mathbb{N}$$
$$\text{unfold}\ (\text{inclist}\ n) = n ::' (\text{inclist}\ (n+1))$$

Using Several Destructors

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

$$\begin{array}{l} \mathrm{coalg\ Stream}\ (A : \mathrm{Set}) : \mathrm{Set\ where} \\ \quad \begin{array}{lcl} \mathrm{head} & : & \mathrm{Stream}\ A \to A \\ \mathrm{tail} & : & \mathrm{Stream}\ A \to \mathrm{Stream}\ A \end{array} \end{array}$$

$$\begin{array}{l} \mathrm{inc} : \mathbb{N} \to \mathrm{Stream}\ \mathbb{N} \\ \begin{array}{lcl} \mathrm{head}\ (\mathrm{inc}\ n) & = & n \\ \mathrm{tail}\ \ (\mathrm{inc}\ n) & = & \mathrm{inc}\ (n+1) \end{array} \end{array}$$

Nested Destructor Patterns

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

$$\begin{array}{lcl} \mathrm{inc}' : \mathbb{N} \to \mathrm{Stream}\ \mathbb{N} & & \\ \mathrm{head}\ (\mathrm{inc}'\ n) & = & n \\ \mathrm{head}\ (\mathrm{tail}\ (\mathrm{inc}'\ n)) & = & n + 1 \\ \mathrm{tail}\ \ (\mathrm{tail}\ (\mathrm{inc}'\ n)) & = & \mathrm{inc}'\ (n + 2) \end{array}$$

Bisimulation

$$
\begin{array}{l}
\text{coalg } \_ \approx \_\{A : \text{Set}\} : \text{Stream } A \to \text{Stream } A \to \text{Set where} \\
\quad \text{headeq} \quad : \quad \{l\ l' : \text{Stream } A\} \to l \approx l' \to \text{head } l == \text{head } l' \\
\quad \text{taileq} \quad : \quad \{l\ l' : \text{Stream } A\} \to l \approx l' \to \text{tail } l \approx \text{tail } l'
\end{array}
$$

Example Proof

$$\begin{array}{lcl} \text{lemma} : (n : \mathbb{N}) \to \text{inc}\ n \approx \text{inc}'\ n & & \\ \text{headeq}\ (\text{lemma}\ n) & = & \text{refl} \\ \text{headeq}\ (\text{taileq}\ (\text{lemma}\ n)) & = & \text{refl} \\ \text{taileq}\ \ (\text{taileq}\ (\text{lemma}\ n)) & = & \text{lemma}\ (n+2) \end{array}$$

(Slide improved after some comments during the talk).

Fibonacci

$$
\begin{array}{lcl}
\mathrm{fib} : \mathrm{Stream}\ \mathbb{N} & & \\
\mathrm{head}\ \mathrm{fib} & = & 1 \\
\mathrm{head}\ (\mathrm{tail}\ \mathrm{fib}) & = & 1 \\
\mathrm{tail}\ \ (\mathrm{tail}\ \mathrm{fib}) & = & \mathrm{addStream}\ \mathrm{fib}\ (\mathrm{tail}\ \mathrm{fib})
\end{array}
$$

Not guarded recursion but can be justified by sized types.
Or (not very useful but the result of unfolding the sized version(?)):

$$
\begin{array}{lcl}
\mathrm{fib} : \mathbb{N} \to \mathrm{Stream}\ \mathbb{N} & & \\
\mathrm{head}\ (\mathrm{fib}\ \mathrm{zero}) & = & 1 \\
\mathrm{head}\ (\mathrm{fib}\ (\mathrm{suc}\ \mathrm{zero})) & = & 1 \\
\mathrm{head}\ (\mathrm{fib}\ (\mathrm{suc}\ (\mathrm{suc}\ n))) & = & \mathrm{head}\ (\mathrm{fib}\ n) + \mathrm{head}\ (\mathrm{fib}\ (\mathrm{suc}\ n)) \\
\mathrm{tail}\ \ (\mathrm{fib}\ n) & = & \mathrm{fib}\ (n+1)
\end{array}
$$

Combining Constructor and Destructor Patterns

We can combine constructor and destructor patterns:

$$
\begin{array}{lll}
\mathrm{inc}'' : \mathbb{N} \to \mathrm{Stream}\ \mathbb{N} & & \\
\mathrm{head}\ (\mathrm{inc}''\ \mathrm{zero}) & = & 0 \\
\mathrm{head}\ (\mathrm{tail}\ (\mathrm{inc}''\ \mathrm{zero})) & = & 1 \\
\mathrm{tail}\ \ (\mathrm{tail}\ (\mathrm{inc}''\ \mathrm{zero})) & = & \mathrm{inc}''\ 2 \\
\mathrm{head}\ (\mathrm{inc}''\ (\mathrm{suc}\ n)) & = & \mathrm{suc}\ n \\
\mathrm{tail}\ \ (\mathrm{inc}''\ (\mathrm{suc}\ n)) & = & \mathrm{inc}''\ (n+1)
\end{array}
$$

Objects

$$
\begin{array}{l}
\text{coalg Stack } (A : \text{Set}) : \mathbb{N} \to \text{Set where} \\
\quad \text{top} \quad : \quad \{n : \mathbb{N}\} \to \text{Stack } (\text{suc } n) \to A \\
\quad \text{pop} \quad : \quad \{n : \mathbb{N}\} \to \text{Stack } (\text{suc } n) \to \text{Stack } n
\end{array}
$$

$$
\begin{array}{l}
\text{push} : \{n : \mathbb{N}\} \to A \to \text{Stack } A\ n \to \text{Stack } A\ (n+1) \\
\text{top } (\text{push } a\ l) \quad = \quad a \\
\text{pop } (\text{push } a\ l) \quad = \quad l
\end{array}
$$

Objects

$$
\begin{array}{l}
\mathrm{coalg}\ \mathrm{Stack}\ (A : \mathrm{Set}) : \mathbb{N} \to \mathrm{Set}\ \mathrm{where} \\
\quad \mathrm{top} \quad : \quad \{n : \mathbb{N}\} \to \mathrm{Stack}\ (\mathrm{suc}\ n) \to A \\
\quad \mathrm{pop} \quad : \quad \{n : \mathbb{N}\} \to \mathrm{Stack}\ (\mathrm{suc}\ n) \to \mathrm{Stack}\ n
\end{array}
$$

The empty stack is introduced as follows:

$$
\begin{array}{l}
\mathrm{emptystack} : \{A : \mathrm{Set}\} \to \mathrm{Stack}\ A\ \mathrm{zero} \\
() \qquad \text{-- no destructor applies}
\end{array}
$$

Note that the coalgebra $\mathrm{Stack\ zero}$ 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?

Complications with Coalgebras

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

$$
\begin{array}{l}
\text{data List } (A : \text{Set}) : \text{Set where} \\
\quad \begin{array}{lcl}
[] & : & \text{List } A \\
\_ :: \_ & : & A \rightarrow \text{List } A \rightarrow \text{List } A
\end{array}
\end{array}
$$

corresponds to

$$
\begin{array}{l}
\text{data List } (A : \text{Set}) : \text{Set where} \\
\quad \text{intro} \quad : \quad \mathbf{1} + (A \times \text{List } A) \rightarrow \text{List } A
\end{array}
$$

Complications with Coalgebras

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

$$
\begin{array}{l}
\text{data Stream } (A : \text{Set}) : \text{Set where} \\
\quad \begin{array}{lcl} \text{head} & : & \text{Stream } A \to A \\ \text{tail} & : & \text{Stream } A \to \text{Stream } A \end{array} \qquad :
\end{array}
$$

corresponds to

$$
\begin{array}{l}
\text{data Stream } (A : \text{Set}) : \text{Set where} \\
\quad \text{unfold} \quad : \quad \text{Stream } A \to A \times \text{Stream } A
\end{array}
$$

Complications with Coalgebras

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

$$
\begin{array}{l}
\mathrm{data}\ \mathrm{List}\ (A : \mathrm{Set}) : \mathrm{Set}\ \ \mathrm{where} \\
\quad \begin{array}{lcl}
[] & : & \mathrm{List}\ A \\
\_ :: \_ & : & A \to \mathrm{List}\ A \to \mathrm{List}\ A
\end{array}
\end{array}
$$

whereas colists don't look nice

$$
\begin{array}{l}
\mathrm{coalg}\ \mathrm{coList}\ (A : \mathrm{Set}) : \mathrm{Set}\ \ \mathrm{where} \\
\quad \mathrm{unfold} : \mathrm{coList}\ A \to [] + A ::' \mathrm{coList}\ A
\end{array}
$$

Codata

Codata types seem to solve this problem:

$$\begin{array}{lll} \multicolumn{3}{l}{\mathrm{codata}\ \mathrm{coList}\ (A : \mathrm{Set}) : \mathrm{Set}\ \mathrm{where}} \\ \quad [] & : & \mathrm{coList}\ A \\ \quad \_ :: \_ & : & A \to \mathrm{coList}\ A \to \mathrm{coList}\ A \end{array}$$

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

Idea is that elements of $\mathrm{coList}\ A$ are infinitary lists:

- $n_1 :: n_2 :: n_3 \ :: \cdots$
- $n_1 :: n_2 :: n_3 \ :: \cdots :: n_k :: []$

Problem of codata

- No normalisation, e.g.

$$\mathrm{inc}\ 0 = 0 :: 1 :: 2 :: \cdots$$

- Undecidability of equality.

$$f\ 0 :: f\ 1 :: \cdots = g\ 0 :: g\ 1 :: \cdots \Leftrightarrow \forall n. f\ \ n = g\ \ n$$

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

$$\begin{array}{l} \text{coalg coList } (A : \text{Set}) : \text{Set where} \\ \quad \text{unfold} : \text{coList } A \to []' + A ::' \text{coList } A \end{array}$$

we can define by guarded recursion

$$\begin{array}{l} [] : \text{coList } A \\ \text{unfold } [] = []' \end{array}$$

$$\begin{array}{l} \_ :: \_ : A : \text{Set} \to A \to \text{coList } A \to \text{coList } A \\ \text{unfold } (a :: l) = a ::' l \end{array}$$

Pseudo-Constructors

However we do not have

$$\mathrm{unfold}\ l = a ::' l' \quad \text{implies} \quad l = a :: l$$

So elements of $\mathrm{coList}\ A$ are not of the form [] or $a :: l$.

But behave like [] or $a :: l$.

∼-Notation (Nils Danielsson)

$$
\begin{array}{l}
\text{codata coList } (A : \text{Set}) : \text{Set where} \\
\quad [] \quad : \quad \text{coList } A \\
\quad \_ :: \_ \quad : \quad A \to \text{coList } A \to \text{coList } A
\end{array}
$$

is an abbreviation for

$$
\begin{array}{l}
\text{coalg coList } (A : \text{Set}) : \text{Set where} \\
\quad \text{unfold} : \text{coList } A \to []' + A ::' \text{coList } A
\end{array}
$$

$$
\begin{array}{l}
[] : A : \text{Set} \to \text{coList } A \\
\text{unfold } [] = []'
\end{array}
$$

$$
\begin{array}{l}
\_ :: \_ : A : \text{Set} \to A \to \text{coList } A \to \text{coList } A \\
\text{unfold } (a :: l) = a ::' l
\end{array}
$$

$\sim$-Notation (Nils Danielsson)

Furthermore let

$$s \sim t \Leftrightarrow \text{unfold } s = \text{unfold } t$$

Then

$$\begin{array}{lcl} \text{unfold } s = []' & \Leftrightarrow & s \sim [] \\ \text{unfold } s = a ::' l & \Leftrightarrow & s \sim a :: l \end{array}$$

so there is no need to write $[]'$ or $\_ ::' \_$ or unfold.
Unfortunaly $\sim$ was replaced by $=$ which misled the users.

Nils Danielsson's ∞

Nils Danielsson and Thorsten Altenkirch suggested to have the following

$$\begin{array}{lcl} \infty & : & \mathrm{Set} \to \mathrm{Set} \\ \flat & : & \{B : \mathrm{Set}\} \to B \to \infty\ B \\ \natural & : & \{B : \mathrm{Set}\} \to \infty\ B \to B \end{array}$$

$\infty\ B$ denote coalgebraic arguments in a definition (which can be "expanded infinitely") and one defines $\mathrm{coList}\ A$ as

$$\begin{array}{l} \mathrm{data\ coList}\ (A : \mathrm{Set}) : \mathrm{Set\ \ where} \\ \quad \begin{array}{lcl} [] & : & \mathrm{coList}\ A \\ \_ :: \_ & : & A \to \infty\ (\mathrm{coList}\ A) \to \mathrm{coList}\ A \end{array} \end{array}$$

What is ∞ B?

∞ B cannot mean

$$\nu X.B$$

since $\nu X.B$ is as a (non-weakly) final coalgebra isomorphic to B: With $\mathrm{F}\ X = B$ we get

$$\begin{array}{ccc} X & \xrightarrow{\ f\ } & \mathrm{F}\ X = B \\ {\scriptstyle \exists! g}\downarrow & & \downarrow{\scriptstyle \mathrm{F}\ g = \mathrm{id}} \\ B & \xrightarrow{\ \mathrm{id}\ } & \mathrm{F}\ B = B \end{array}$$

Underlying reason

ν 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 ∞ must be something like $(\mathrm{Set} \to \mathrm{Set}) \to \mathrm{Set}$.

What is ∞ A?

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

$$
\begin{array}{l}
\text{data coList } (A : \text{Set}) : \text{Set where} \\
\quad [] \quad : \quad \text{coList } A \\
\quad \_ :: \_ \quad : \quad A \to \infty\, (\text{coList } A) \to \text{coList } A
\end{array}
$$

stands for

$$
\begin{array}{l}
\text{data coList } (A : \text{Set}) : \text{Set where} \\
\quad [] \quad : \quad \text{coList } A \\
\quad \_ :: \_ \quad : \quad A \to \infty\, (\text{coList } A) \to \text{coList } A
\end{array}
$$

$$
\begin{array}{l}
\text{coalg } \infty\, (\text{coList}\_)\ (A : \text{Set}) : \text{Set where} \\
\quad \natural : \infty\, (\text{coList } A) \to \text{coList } A
\end{array}
$$

Order between data/codata

$$
\begin{array}{l}
\text{data coList } (A : \text{Set}) : \text{Set where} \\
\quad [] \quad : \quad \text{coList } A \\
\quad \_ :: \_ \quad : \quad A \to \infty\, (\text{coList } A) \to \text{coList } A \\
\text{coalg } \infty\, (\text{coList}\_)\ (A : \text{Set}) : \text{Set where} \\
\quad \natural : \infty\, (\text{coList } A) \to \text{coList } A
\end{array}
$$

But there are two interpretations of the above:

1.

$$
\begin{array}{lcl}
F(X, Y) & = & [] + A :: Y \\
G(X, Y) & = & X \\
F'(Y) & = & \mu X.F(X, Y) = \mu X.[] + A :: Y \\
 & \cong & [] + A :: Y \\
\infty\, (\text{coList } A) & = & \nu Y.G(F'(Y), Y) = \nu Y.F'(Y) \\
 & \cong & \nu Y.[] + A :: Y \\
\text{coList } A & = & F'(\infty\, (\text{coList } A)) \\
 & = & [] + A :: (\infty\, (\text{coList } A))
\end{array}
$$

Order between data/codata

$$
\begin{array}{l}
\text{data coList } (A : \text{Set}) : \text{Set where} \\
\quad [] \quad : \quad \text{coList } A \\
\quad \_ :: \_ \quad : \quad A \to \infty\ (\text{coList } A) \to \text{coList } A \\
\text{coalg } \infty\ (\text{coList}\_)\ (A : \text{Set}) : \text{Set where} \\
\quad \natural : \infty\ (\text{coList } A) \to \text{coList } A
\end{array}
$$

2.

$$
\begin{array}{lcl}
G(X, Y) & = & X \\
F(X, Y) & = & [] + A :: Y \\
G'(X) & = & \nu Y.G(X, Y) = \nu Y.X \\
 & \cong & X \\
\text{coList } A & = & \mu X.F(X, G'(X)) \cong \mu X.F(X, X) \\
 & = & \mu X.[] + A :: X \\
\infty\ (\text{coList } A) & = & G'(\text{coList } A) \\
 & \cong & \text{coList } A
\end{array}
$$

Order between data/codata

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.

Generality

In general we want to be able to form arbitrary combinations of μ and ν.
Idea: minimize and maximize in the order of occurrence.

$$
\begin{array}{l}
\text{data } A : \text{Set where} \\
\quad \text{intro}_0 : F(A, B, C, D) \to A \\
\text{coalg } B : \text{Set where} \\
\quad \text{unfold}_0 : B \to G(A, B, C, D) \\
\text{data } C : \text{Set where} \\
\quad \text{intro}_1 : H(A, B, C, D) \to C \\
\text{coalg } D : \text{Set where} \\
\quad \text{unfold}_1 : D \to K(A, B, C, D)
\end{array}
$$

to be interpreted as:

$$
\begin{array}{lcll}
F_0(Y, Z, Z') & = & \mu X.F(X, Y, Z, Z') & A \text{ in terms of } Y, Z, Z' \\
G_1(Z, Z') & = & \nu Y.G(F'(Y, Z, Z'), Y, Z, Z') & B \text{ in terms of } Z, Z' \\
F_1(Z, Z') & = & F_0(G_1(Z, Z'), Z, Z') & A \text{ in terms of } Z, Z' \\
H_2(Z') & = & \mu Z.H(F_1(Z, Z'), G_1(Z, Z'), Z, Z') & C \text{ in terms of } Z' \\
G_2(Z') & = & G_1(H_2(Z'), Z') & B \text{ in terms of } Z' \\
F_2(Z') & = & F_1(H_2(Z'), Z') & A \text{ in terms of } Z' \\
D & = & \nu Z'.K(F_2(Z'), G_2(Z'), H_2(Z'), Z') & \text{Final Value of } D \\
C & = & H_2(D) & \text{Final Value of } C \\
B & = & G_2(D) & \text{Final Value of } B \\
A & = & F_2(D) & \text{Final Value of } A
\end{array}
$$

Example: $\text{coList}\ A$

$$\begin{array}{l}
\text{data coList } (A : \text{Set}) : \text{Set where} \\
\quad [] \qquad : \quad \text{coList } A \\
\quad _ :: _ \quad : \quad A \to \infty\,(\text{coList } A) \to \text{coList } A
\end{array}$$

stands for

$$\begin{array}{l}
\text{data coList } (A : \text{Set}) : \text{Set where} \\
\quad [] \qquad : \quad \text{coList } A \\
\quad _ :: _ \quad : \quad A \to \infty\,(\text{coList } A) \to \text{coList } A
\end{array}$$

$$\begin{array}{l}
\text{coalg } \infty\,(\text{coList}_)\ (A : \text{Set}) : \text{Set where} \\
\quad \natural : \infty\,(\text{coList } A) \to \text{coList } A
\end{array}$$

inclist

$$\mathrm{inclist} : \mathbb{N} \to \infty\,(\mathrm{coList}\ \mathbb{N})$$
$$\natural\,(\mathrm{inclist}\ n) = n :: \mathrm{inclist}\ (n+1)$$
or
$$\mathrm{inclist}\ n \sim \flat\,(n :: \mathrm{inclist}\ (n+1))$$

With

$$s \rhd t :\Leftrightarrow \natural s = t$$

we get

$$\mathrm{inclist}\ n \rhd n :: \mathrm{inclist}\ (n+1)$$

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 $\mu X.F(X)$:

$$[\![\,\mu X.F(X)\,]\!] = \bigcap\{X \subseteq \text{Term} \mid \text{intro}[[\![\,F(X)\,]\!]] \subseteq X\}$$

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

$$[\![\,\nu X.F(X)\,]\!] = \bigcup\{X \subseteq \text{Term} \mid \text{unfold}[X] \subseteq [\![\,F(X)\,]\!]\}$$

Conclusion

- Design decisions should be done by referring to the notion of coalgebras.
- Coalgebras with constructor/destructor patterns looks very neat.
- Other solutions $\sim$, ∞ 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 ∞ A,
  then ∞ 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 μ and ν?
  - Not really necessary, since we can built up expressions of nested μ, ν using mutual algebras and coalgebras understood in our way.