Simulating Codata Types using Coalgebras

Anton Setzer
Swansea University, Swansea UK
Types 2018, Braga, Portugal

18 June 2018

Coalgebras in Type Theory

Musical Notation In Agda Reduced to Coalgebras

Suggested Extension

Conclusion

Codata Types

- Original way of defining infinite (in general non-well-founded) structures in functional programming are codata types:

$$\begin{array}{l} \text{codata coList} : \text{Set where} \\ \quad \begin{array}{lcl} \text{nil} & : & \text{coList} \\ \text{cons} & : & \mathbb{N} \rightarrow \text{coList} \rightarrow \text{coList} \end{array} \end{array}$$

- Define

$$\begin{array}{l} \text{from} : \mathbb{N} \rightarrow \text{coList} \\ \text{from } n = \text{cons } n \text{ (from } (n+1)) \end{array}$$

- Problem
  - Literally taken **non-normalising**.
  - Restriction of evaluation to lazy evaluation or similar led to **subject reduction problem** in Coq and early versions of Agda.
  - Proof in [4] that there is **no decidable equality** on codata types **which would allow pattern matching**.

Coalgebras

- Solution define infinite structures by elimination rules or by their observations.
- Replace Pattern matching by copattern matching [2].
- Example Streams (syntax is desired syntax – Agda uses record types instead):

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

- Define the stream $n, n+1, n+2, \ldots$ by copattern matching

$$\begin{array}{llcl} \multicolumn{4}{l}{\mathrm{from} : \mathbb{N} \to \mathrm{Stream}} \\ \mathrm{head} & (\mathrm{from}\ n) & = & n \\ \mathrm{tail} & (\mathrm{from}\ n) & = & \mathrm{from}\ (n+1) \end{array}$$

Colists

- When applying the above to colists one need to have an observation which determines for every colist whether it is nil or cons.
- Most easily done by using a simultaneous inductive-coinductive definition.

$$
\begin{array}{l}
\mathrm{mutual} \\
\quad \mathrm{coalg}\ \infty\mathrm{coList} : \mathrm{Set\ where} \\
\quad\quad \flat : \infty\mathrm{coList} \to \mathrm{coList} \\
\\
\quad \mathrm{data}\ \mathrm{coList} : \mathrm{Setwhere} \\
\quad\quad \begin{array}{lcl}
\mathrm{nil} & : & \mathrm{coList} \\
\mathrm{cons} & : & \mathbb{N} \to \infty\mathrm{coList} \to \mathrm{coList}
\end{array}
\end{array}
$$

Examples of Coalgebras in Agda

- We have developed lots of coalgebras in Agda:
  - IO monad in Agda.
  - Formalisation of CSP in Agda.
  - Objects (as in object-based programming) in Agda
  - GUIs in Agda.
  - Business processes in Agda
  - Many variants of the above
- In some examples definition by several eliminators is the right thing.
- In some example one has the pattern as above but needs to add to the type corresponding to $\infty\mathrm{coList}$ extra components.
- But most examples follow exactly the **same pattern** as above.
- **Musical notation** modified by Danielsson [5] was an abbreviation mechanism in Agda for having the above. (Currently abandoned).
- Suggestion: Reduce **musical notation** to coalgebras.
- Musical notation as a **syntactic sugar** for coalgebra approach.

Functions for Introduction Codata like Coalgebras

- When defining functions into a codata like coalgebra one defines mutually two functions:

$$\begin{array}{l} \sharp\mathrm{f} : A \to \infty\mathrm{coList} \\ \flat\,(\sharp\mathrm{f}\ a) = f\ a \\ \\ f : A \to \mathrm{coList} \\ f\ a = \cdots \quad \text{(referring to } \sharp\mathrm{f}\text{)} \end{array}$$

- Example from musical notation documentation in Agda:

$$\begin{array}{l} \sharp\mathrm{map} : (\mathbb{N} \to \mathbb{N}) \to \mathrm{coList} \to \infty\mathrm{coList} \\ \flat\,(\sharp\mathrm{map}\ f\ l) = \mathrm{map}\ f\ l \\ \\ \mathrm{map} : (\mathbb{N} \to \mathbb{N}) \to \mathrm{coList} \to \mathrm{coList} \end{array}$$

$$\begin{array}{lcl} \mathrm{map}\ f\ \mathrm{nil} & = & \mathrm{nil} \\ \mathrm{map}\ f\ (\mathrm{cons}\ n\ l) & = & \mathrm{cons}\ (f\ n)\ (\sharp\mathrm{map}\ f\ l) \end{array}$$

Coalgebras in Type Theory

Musical Notation In Agda Reduced to Coalgebras

Suggested Extension

Conclusion

∞ as Syntactic Sugar

- Whenever one defines a new constant

$$C : (x_1 : A_1)\,(x_2 : A_2)\ \cdots\ (x_n : A_n) \to Set$$

one defines simultaneously with any definition involving C

$$\begin{array}{l} \mathrm{coalg}\ \infty C\ (x_1 : A_1)\,(x_2 : A_2)\ \cdots\ (x_n : A_n) : \mathrm{Set\ where} \\ \quad \flat : \infty C\ x_1\ \cdots\ x_n \to C\ x_1\ \cdots\ x_n \end{array}$$

- Whenever one defines a new constant

$$f : (x_1 : A_1)\,(x_2 : A_2)\ \cdots\ (x_n : A_n) \to C\ t$$

where C is a constant one defines

$$\begin{array}{l} \sharp\mathrm{f} : (x_1 : A_1)\,(x_2 : A_2)\ \cdots\ (x_n : A_n) \to \infty C\ t \\ \flat\ (\sharp\mathrm{f}\ x_1\ \cdots\ x_n) = f\ x_1\ \cdots\ x_n \end{array}$$

∞ as Syntactic Sugar

- Whether the following is a good notation needs to be discussed.
- However it allows to give an interpretation of Altenkirch et. al.'s boxed operator [3], where $\sharp\ t$ is the type of delayed computations.
- If $C\ s_1\ \cdots\ s_n$ is an expression where C is a constant define

$$\infty\,(C\ s_1\ \cdots\ s_n) := \infty C\ s_1\ \cdots\ s_n$$

- If $f\ s_1\ \cdots\ s_n$ is an expression where f is a constant define

$$\sharp\,(f\ s_1\ \cdots\ s_n) := \sharp f\ s_1\ \cdots\ s_n$$

- In the above $C\ s_1\ \cdots\ s_n$ and $f\ s_1\ \cdots\ s_n$ are not evaluated, the right hand side is evaluated.
- Note that $\infty\ x$ and $\sharp\ x$ for variables x is not defined.

Syntactic Sugar

- Note that the above is just syntactic sugar.
- Type checking and Term checking relies on type and termination checking of the desugared version.
- Should the above definitions not be suitable, the user has access to the standard coalgebra definitions.

Example: coList, map and from

$$
\begin{array}{l}
\mathrm{data\ coList} : \mathrm{Setwhere} \\
\quad \begin{array}{lcl}
\mathrm{nil} & : & \mathrm{coList} \\
\mathrm{cons} & : & \mathbb{N} \to \infty\ \mathrm{coList} \to \mathrm{coList}
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{map} : (\mathbb{N} \to \mathbb{N}) \to \mathrm{coList} \to \mathrm{coList} \\
\begin{array}{lcl}
\mathrm{map}\ f\ \mathrm{nil} & = & \mathrm{nil} \\
\mathrm{map}\ f\ (\mathrm{cons}\ n\ l) & = & \mathrm{cons}\ (f\ n)\ (\sharp\ (\mathrm{map}\ f\ l))
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{from} : \mathbb{N} \to \mathrm{coList} \\
\mathrm{from}\ n \quad = \quad \mathrm{cons}\ n\ (\sharp\ (\mathrm{from}\ (n+1)))
\end{array}
$$

Sized Types

- For coalgebras one needs except for very simple examples sized types.
- In case of coList the definition is as follows:

$$
\begin{array}{l}
\mathrm{mutual} \\
\quad \mathrm{coalg}\ \infty\mathrm{coList}\ \{i : \mathrm{Size}\} : \mathrm{Set\ where} \\
\quad\quad \flat : \{j : \mathrm{Size} < \ i\} \to \infty\mathrm{coList}\ \{i\} \to \mathrm{coList}\{j\} \\
\\
\quad \mathrm{data\ coList}\ \{i : \mathrm{Size}\} : \mathrm{Set\ where} \\
\quad\quad \mathrm{nil} \quad\ : \quad \mathrm{coList}\ \{i\} \\
\quad\quad \mathrm{cons} \quad : \quad \mathbb{N} \to \infty\mathrm{coList}\ \{i\} \to \mathrm{coList}\ \{i\}
\end{array}
$$

∞, $\sharp$ with Sizes

- Whenever one defines a new constant

$$C : \{i : \mathrm{Size}\}\ (x_1 : A_1)\ (x_2 : A_2)\ \cdots\ (x_n : A_n) \to Set$$

one defines simultaneously with any definition involving C

$$\begin{array}{l} \mathrm{coalg}\ \infty C\ \{i : \mathrm{Size}\}\ (x_1 : A_1)\ (x_2 : A_2)\ \cdots\ (x_n : A_n) : \mathrm{Set\ where} \\ \quad \flat : \{j : \mathrm{Size} < i\} \to \infty C\ \{i\}\ x_1\ \cdots\ x_n \to C\ \{j\}\ x_1\ \cdots\ x_n \end{array}$$

- Whenever one defines a new constant

$$f : \{i : \mathrm{Size}\}\ (x_1 : A_1)\ (x_2 : A_2)\ \cdots\ (x_n : A_n) \to C\ t$$

where C is a constant one defines

$$\begin{array}{l} \sharp\mathrm{f} : \{i : \mathrm{Size}\}\ (x_1 : A_1)\ (x_2 : A_2)\ \cdots\ (x_n : A_n) \to \infty C\ t \\ \flat\ (\sharp\mathrm{f}\ \{i\}\ x_1\ \cdots\ x_n)\ \{j\} = f\ \{j\}\ x_1\ \cdots\ x_n \end{array}$$

IO Interface

$$
\begin{array}{l}
\text{data Command} : \text{Set where} \\
\quad \begin{array}{lcl}
\text{getStr} & : & \text{Command} \\
\text{putStr} & : & \mathsf{String} \to \text{Command}
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\text{Response} : \text{Command} \to \text{Set} \\
\begin{array}{llcl}
\text{Response} & \text{getStr} & = & \mathsf{String} \\
\text{Response} & (\text{putStr } s) & = & \top
\end{array}
\end{array}
$$

Example IO

$$
\begin{array}{l}
\text{data IO } \{i : \text{Size}\}\ (A : \text{Set}) : \text{Set where} \\
\quad \begin{array}{lcl}
\text{return} & : & A \to \text{IO } \{i\}\ A \\
\text{exec} & : & (c : \text{Command}) \\
 & & (p : \text{Response } c \to \infty\ (\text{IO } \{i\}\ A)) \\
 & & \to \text{IO } \{i\}\ A
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\text{copycat} : \{i : \text{Size}\} \to \text{IO } \{i\}\ \bot \\
\text{copycat} = \text{exec getStr } \lambda\ s \to \\
\qquad\qquad \sharp\ (\text{exec } (\text{putStr } s)\ \lambda\ \_ \to \\
\qquad\qquad\quad \sharp\ \text{copycat})
\end{array}
$$

Object Interface for Cell

$$
\begin{array}{l}
\mathrm{data}\ \mathrm{Method} : \mathrm{Set}\ \mathrm{where} \\
\quad \begin{array}{lcl}
\mathrm{get} & : & \mathrm{Method} \\
\mathrm{put} & : & \mathbb{N} \rightarrow \mathrm{Method}
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{Result} : \mathrm{Method} \rightarrow \mathrm{Set} \\
\begin{array}{llcl}
\mathrm{Result} & \mathrm{get} & = & \mathbb{N} \\
\mathrm{Result} & (\mathrm{put}\ n) & = & \top
\end{array}
\end{array}
$$

Example Object [1]

$$\text{Object} : \{i : \text{Size}\} \to \text{Set}$$
$$\text{Object}\{i\} = (m : \text{Method}) \to \text{IO}\ \infty\ (\text{Result}\ m \times \sharp\ (\text{Object}\{i\}))$$

$$\begin{array}{lcl}
\multicolumn{3}{l}{\text{cell} : \{i : \text{Size}\} \to \mathbb{N} \to \text{Object}\ \{i\}} \\
\text{cell}\ n\ \text{get} & = & \text{exec}\ (\text{putStr}\ \text{``get invoked''})\ \lambda\ _ \to \\
 & & \sharp\ (\text{return}\ (n\ ,\ \sharp\ (\text{cell}\ n))) \\
\text{cell}\ n\ (\text{put}\ m) & = & \text{exec}\ (\text{putStr}\ \text{``put invoked''})\ \lambda\ _ \to \\
 & & \sharp\ (\text{return}\ (_\ ,\ \sharp\ (\text{cell}\ m)))
\end{array}$$

Conclusion

- Definition of coalgebras by their elimination as a clean approach for defining “infinite data types” (more precisely well-founded).
- Musical notation as syntactic sugar which reduces to coalgebras.
- Delayed computations can be interpreted in this setting.
- Resulting code is very close to codata types.

Bibliography I

A. Abel, S. Adelsberger, and A. Setzer.
Interactive programming in Agda – Objects and graphical user interfaces.
Journal of Functional Programming, 27, Jan 2017.
doi 10.1017/S0956796816000319.

A. Abel, B. Pientka, A. Setzer, and D. Thibodeau.
Copatterns: Programming infinite structures by observations.
In R. Giacobazzi and R. Cousot, editors, Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '13, pages 27–38, New York, NY, USA, 2013. ACM.

Bibliography II

T. Altenkirch, N. Danielsson, A. Löh, and N. Oury.
ΠΣ: Dependent types without the sugar.
In M. Blume, N. Kobayashi, and G. Vidal, editors, Functional and Logic Programming, volume 6009 of Lecture Notes in Computer Science, pages 40–55. Springer Berlin / Heidelberg, 2010.
http://dx.doi.org/10.1007/978-3-642-12251-4_5.

U. Berger and A. Setzer.
Undecidability of equality for codata types, 2018.
To appear in proceedings of CMCS'18, available from
http://www.cs.swan.ac.uk/~csetzer/articles/CMCS2018/bergerSetzerProceedingsCMCS18.pdf.

Bibliography III

N. A. Danielsson.
Changes to coinduction, 17 March 2009.
Message posted on gmane.comp.lang.agda, available from
http://article.gmane.org/gmane.comp.lang.agda/763/.