Simulating Codata Types using Coalgebras

Anton Setzer

Dept. of Computer Science, Swansea University, Swansea, UK
a.g.setzer@swan.ac.uk

Abstract

We show how the so called musical notation in Agda for codata types can be considered as syntactic sugar for using codata types in a coalgebraic setting. This allows to simulate codata types using coalgebras while avoiding subject reduction and undecidability problems of codata types. This restricted form of codata types can be added to the coalgebraic setting allowing to shorten proofs and programs involving codata types.

The idea of a codata type is that it is like an algebraic data type (as introduced by the keyword data in Agda/Haskell), but one allows infinite, or more generally non-wellfounded, applications of constructors. An example is the type of colists which in Agda style would be defined as

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

and which contains finite lists as well as infinite lists such as the list of numbers greater than n, defined as enum $n :=$ cons n (cons $(n+1)$ (cons $(n+2)$ $\cdots$)).
The problems are that when implementing it first in Coq and Agda a subject reduction problem occurred. In our coauthored article [4] we showed that the implicit assumption when pattern matching on codata types, namely that every element of a codata type is introduced by a constructor, results in an undecidable equality.
In order to repair this, in our coauthored article [1] coalgebras and copatterns were proposed for replacing codata types, giving a cleaner theory. They have since been implemented in Agda, In that approach coalgebraic types are defined by their observations. An example is the set of streams which in our desired notation (Agda uses record types instead) would be defined as:

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

We can define colists using coalgebras as follows:

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

The type of colists is called ∞coList which has observation $\flat$, which determines for a colist whether it is of the form nil or (cons n s). coList is the type of colist shapes having these elements. Danielsson [5] pointed out that a key example for codata types is the map function, which can get in variants of codata types quite long definitions. In coalgebras it can be defined as follows:

$$\begin{array}{l}\text{map} : (\mathbb{N} \to \mathbb{N}) \to \text{coList} \to \text{coList}\\ \text{map } f\ (\text{cons } n\ l) \quad = \quad \text{cons } (f\ n)\ (\sharp\text{map } f\ (\flat\ l))\\ \text{map } f\ \text{nil} \qquad\quad = \quad \text{nil}\end{array} \qquad \begin{array}{l}\sharp\text{map} : (\mathbb{N} \to \mathbb{N}) \to \text{coList} \to \infty\text{coList}\\ \flat\ (\sharp\text{map } f\ l) \quad = \quad \text{map } f\ l\end{array}$$

In Agda there exists a (currently no longer maintained) variant of codata types, using the so called "Musical Notation" [2], which is a termination checked version of [3]. There one has a

type former $\infty : \mathrm{Set} \to \mathrm{Set}$, which defines a generic coalgebra $\infty\, A$ from A, and an operation $\sharp : A \to \infty\, A$, lifting elements from A to $\infty\, A$. It can be considered as being defined as follows:

$$\begin{array}{ll} \text{coalg } (\infty\, A) : \mathrm{Set} \text{ where} & \sharp : A \to \infty\, A \\ \quad \flat \ : \ \infty\, A \to A & \flat\ (\sharp\ a) = a \end{array}$$

Then colists and the map function are defined as

$$\begin{array}{ll} \text{data coList} : \mathrm{Set} \text{ where} & \text{map} : (\mathbb{N} \to \mathbb{N}) \to \text{coList} \to \text{coList} \\ \quad \text{cons} \ : \ \mathbb{N} \to \infty\, \text{coList} \to \text{coList} & \text{map } f\ (\text{cons } n\ l) \ = \ \text{cons } (f\ n)\ (\sharp\ (\text{map } f\ (\flat\ l))) \\ \quad \text{nil} \ : \ \text{coList} & \text{map } f\ \text{nil} \ = \ \text{nil} \end{array}$$

The problem is that the definition of map is, if followed to the letter, non normalising. Furthermore it is not clear, what the right notion of equality is for elements $\sharp\ (\text{map } f\ l)$ and $\sharp\ (\text{map } f'\ l')$. In addition, $\infty\, A$ cannot defined generically in advance, it needs to be a coalgebra defined simultaneously with A. These problems can be clarified, by understanding them as definitions using coalgebras, and introducing the following notations:

- When introducing a new constant $A : (\vec{x} : \vec{A}) \to \mathrm{Set}$ we define automatically a new constant ∞A of the same type, and when introducing a new function $f : (\vec{y} : \vec{B}) \to A\ \vec{t}$ we define a constant $\sharp f : (\vec{y} : \vec{B}) \to \infty A\ \vec{t}$ with definitions (which are simultaneously defined with the definitions of A and f):

$$\begin{array}{ll} \text{coalg } \infty A\ (\vec{x} : \vec{A}) : \mathrm{Set} \text{ where} & \sharp f : (\vec{y} : \vec{B}) \to \infty A\ \vec{t} \\ \quad \flat \ : \ \infty A\ \vec{x} \to A\ \vec{x} & \flat\ (\sharp f\ \vec{y}) = f\ \vec{y} \end{array}$$

- If A, f are constants, then $\infty\ (A\ \vec{t})$ denotes $\infty A\ \vec{t}$ and $\sharp\ (f\ \vec{t})$ denotes $\sharp f\ \vec{t}$.

With this the above musical definition of coList and map is the same as the previously introduced direct simulation of coList using coalgebras, and the musical notation can be considered as syntactic sugar for simulating codata types in coalgebras. It could live alongside the coalgebraic version, shortening proofs and programs involving codata types.
We will discuss in our talk how to modify this definition to accommodate sized types. One should note that Agda seems to treat mutual coinductive-inductive definitions as $\nu X.\mu Y$ definitions without giving the option of defining them as $\mu Y.\nu X$. Experiments show that this seems to be the case both for the coalgebraic version and the version with musical notation.

References

[1] A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: Programming infinite structures by observations. In R. Giacobazzi and R. Cousot, editors, *Proceedings of POPL '13*, pages 27–38, New York, NY, USA, 2013. ACM. https://doi.org/10.1145/2429069.2429075.

[2] Agda Wiki. Coinductive data types, 1 January 2011. http://wiki.portal.chalmers.se/agda/pmwiki.php?n=ReferenceManual.Codatatypes.

[3] 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.

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

[5] 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/.