Dependent Inductive and Coinductive Types Through Dialgebras in Fibrations

Henning Basold[1,2] and Herman Geuvers[1,3]

[1] Radboud University, iCIS, Intelligent Systems
[2] CWI, Amsterdam, The Netherlands
[3] Technical University Eindhoven, The Netherlands

It is well-known that dependent type theories with fixed type constructors can be interpreted over certain fibrations, see, for example, [Jac99]. On the other hand, Hagino [Hag87] showed how a language with constructors for inductive and coinductive types *without* term dependencies can be treated uniformly through the use of special dialgebras. However, at least to our knowledge, these techniques have not been combined, yet. Hence, the goal of the present work is to interpret dependent inductive and coinductive data types using certain dialgebras in fibrations.

The reason for the choice of dialgebras is that we can interpret data types without having to require the existence of (co)products separately, see below. An example of such a data type are vectors, that is, lists combined with their length, which could be declared in Agda by

$$\begin{aligned}&\textbf{data}\ \mathrm{Vec}\ (\mathrm{A} : \mathbf{Set}) : \mathbb{N} \to \mathbf{Set}\ \textbf{where}\\ &\quad \mathrm{nil}\ : \mathbf{1} \to \mathrm{Vec}\ \mathrm{A}\ 0\\ &\quad \mathrm{cons}\ : (\mathrm{k} : \mathbb{N}) \to \mathrm{A} \times \mathrm{Vec}\ \mathrm{A}\ \mathrm{k} \to \mathrm{Vec}\ \mathrm{A}\ (\mathrm{k}+1)\end{aligned}$$

where $\mathbf{1}$ is the one-element type and $\mathbb{N}$ the natural numbers. The important observation is that Vec has two constructors, nil and cons, with the following properties

1. nil has no *local dependencies*, whereas cons introduces locally a fresh variable $k : \mathbb{N}$.
2. nil has a trivial argument of type $\mathbf{1}$, whereas cons has as argument an element of A and an element of $(\mathrm{Vec}\, A\, n)[k/n]$ in context $k : \mathbb{N}$.
3. nil constructs an element of $(\mathrm{Vec}\, A\, n)[0/n]$, whereas cons constructs an element of the type $(\mathrm{Vec}\, A\, n)[k+1/n]$ in context $k : \mathbb{N}$.

These properties allow us to interpret the constructors as a dialgebra.

The semantics of these constructors can be explained in the category $\mathbf{Set}^{\mathbb{N}}$. Given a set I, the category $\mathbf{Set}^I$ has set families $X = \{X_i\}_{i\in I}$ as objects and families $\{f_i : X_i \to Y_i\}_{i\in I}$ of functions as morphisms $f : X \to Y$. Moreover, for every function $g : I \to J$ there is a functor $g^* : \mathbf{Set}^J \to \mathbf{Set}^I$, the *reindexing* or *substitution* along g, given by $g^*(X) = \{X_{g(i)}\}_{i\in I}$. The categories $\mathbf{Set}^I$ and the reindexing functors can be organised into the families fibration $\mathrm{Fam}(\mathbf{Set})$, see for example [Jac99].

The example of vectors is represented in $\mathrm{Fam}(\mathbf{Set})$ as follows. We define the family $\mathrm{Vec}\, A = \{A^n\}_{n\in\mathbb{N}}$, which lives in $\mathbf{Set}^{\mathbb{N}}$. The constructors, however, live in different categories, namely nil is given by the singleton family $\{\mathrm{nil}_* : \mathbf{1} \to \mathrm{Vec}\, A\, 0\}_{*\in\mathbf{1}}$ in $\mathbf{Set}^{\mathbf{1}}$, and the constructor cons is given by the morphism $\mathrm{cons} = \{\mathrm{cons}_k : A \times \mathrm{Vec}\, A\, k \to \mathrm{Vec}\, A\, (k+1)\}_{k\in\mathbb{N}}$ in $\mathbf{Set}^{\mathbb{N}}$. We can see these constructors as one morphism $(\mathrm{nil}, \mathrm{cons})$ in the product category $\mathbf{Set}^{\mathbf{1}} \times \mathbf{Set}^{\mathbb{N}}$.

To understand the nature of these constructors, we draw on dialgebras, as pioneered by Hagino [Hag87]. We define two functors $F, G : \mathbf{Set}^{\mathbb{N}} \to \mathbf{Set}^{\mathbf{1}} \times \mathbf{Set}^{\mathbb{N}}$, such that F captures the domain and G the codomain of the constructors:

$$F(X) = (\{\mathbf{1}\}, \{A \times X_k\}_{k\in\mathbb{N}}) \qquad G(X) = (\{X_0\}, \{X_{k+1}\}_{k\in\mathbb{N}})$$

with the obvious action on morphisms. The constructors for Vec form then an (F, G)-dialgebra, that is, $(\mathrm{nil}, \mathrm{cons}) : F(\mathrm{Vec}\, A) \to G(\mathrm{Vec}\, A)$. In fact, $(\mathrm{nil}, \mathrm{cons})$ is an *initial* (F, G)-dialgebra.

The final step is to note the special shape of G. On the natural numbers, we have the two maps $z : \mathbf{1} \to \mathbb{N}$ and $s : \mathbb{N} \to \mathbb{N}$ given by $z(*) = 0$ and $s(n) = n + 1$, respectively. From these we get the two reindexing functors $z^* : \mathbf{Set}^{\mathbb{N}} \to \mathbf{Set}^{\mathbf{1}}$ and $s^* : \mathbf{Set}^{\mathbb{N}} \to \mathbf{Set}^{\mathbb{N}}$, which we can combine, using pairing of functors, into $\langle z^*, s^* \rangle$. Looking closely at the definitions, we find that this is exactly G.

Using this analysis, we can define what (non-nested) data types in Fam(**Set**) are. We call a pair of functors (F, G) with $F, G : \mathbf{Set}^I \to \prod_{i=1,\dots,n} \mathbf{Set}^{J_i}$ and $G = \langle f_1^*, \dots, f_n^* \rangle$ a *data type* signature. An *inductive* data type is then an initial (F, G)-dialgebra and a *coinductive* data type is a final (G, F)-dialgebra (note the order).

An example of a coinductive data type are partial streams, which are streams indexed by their (potentially infinite) length. These are given by the following pseudo-Agda declaration.

$$\begin{aligned} &\textbf{codata}\ \mathrm{PStr}\ (\mathrm{A} : \mathbf{Set}) : \mathbb{N}^\infty \to \mathbf{Set}\ \textbf{where} \\ &\quad \mathrm{hd} : (\mathrm{n} : \mathbb{N}^\infty) \to \mathrm{PStr}\ \mathrm{A}\ (s_\infty\ \mathrm{n}) \to \mathrm{A} \\ &\quad \mathrm{tl} : (\mathrm{n} : \mathbb{N}^\infty) \to \mathrm{PStr}\ \mathrm{A}\ (s_\infty\ \mathrm{n}) \to \mathrm{PStr}\ \mathrm{A}\ \mathrm{n} \end{aligned}$$

where $\mathbb{N}^\infty$ is the coinductive type of natural numbers extend by infinity and $s_\infty : \mathbb{N}^\infty \to \mathbb{N}^\infty$ the definable successor. The type PStr has two *destructors*, which can only be applied to partial streams that contain at least one element. They are the final $(\langle s_\infty^*, s_\infty^* \rangle, H)$-dialgebra, where $H : \mathbf{Set}^{\mathbb{N}^\infty} \to \mathbf{Set}^{\mathbb{N}^\infty} \times \mathbf{Set}^{\mathbb{N}^\infty}$ is given by $H(X) = (w(A), X)$ and $w : \mathbf{Set}^{\mathbf{1}} \to \mathbf{Set}^{\mathbb{N}^\infty}$ is the weakening functor given by reindexing along $!_{\mathbb{N}^\infty} : \mathbb{N}^\infty \to \mathbf{1}$.

It is noteworthy that coinductive data types are perfectly dual to inductive types. They are also an extension of the syntax for record types in Agda, in the sense that we are allowed to refine the domain of destructors using substitutions.

Another interesting example are products along arbitrary maps. In our pseudo-Agda notation, these are given by

$$\begin{aligned} &\textbf{codata}\ \textstyle\prod_{\_}\ (\mathrm{f} : \mathrm{I} \to \mathrm{J})\ (\mathrm{B} : \mathrm{I} \to \mathbf{Set}) : \mathrm{J} \to \mathbf{Set}\ \textbf{where} \\ &\quad \mathrm{app} : (\mathrm{i} : \mathrm{I}) \to \textstyle\prod_f\ \mathrm{B}\ (\mathrm{f\ i}) \to \mathrm{B\ i} \end{aligned}$$

It turns out that, when interpreted in Fam(**Set**), $\prod_f$ is right-adjoint to f^*, which is the usual definition of the product. Coproducts (Σ-types) can be defined analogously as inductive types. Note though that the $\Sigma x : A.B$ is usually given in Agda as record-type with projections on A and B, which gives a strong sum elimination that we do not have in the present setting. We will discuss this in conjunction with induction.

The talk will consist of the following topics. We will generalise the idea of dependent inductive and coinductive data types, we outlined above, to allow for nested data types. Moreover, we will see how these data types relate to algebras and coalgebras, and we will discuss induction and coinduction in this setting. Finally, if time permits, and we will investigate a Beck-Chevalley condition for data types.

This work has been submitted to CALCO 2015.

References

[Hag87] Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In *Category Theory in Computer Science*, pages 140–157, 1987.

[Jac99] B. Jacobs. *Categorical Logic and Type Theory.* Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam, 1999.