Unfolding Nested Patterns and Copatterns

Anton Setzer[1*]

Department of Computer Science, Swansea University, Singleton Park, Swansea SA2 8PP, UK
a.g.setzer@swan.ac.uk

Because of the importance of interactive programs, which result in potentially infinite computation traces, coalgebraic data types play an important rôle in computer science. Coalgebraic data types are often represented in functional programming as codata types. Implicit in the formulation of codata types is that every element of the coalgebra is introduced by a constructor. Our first result in this talk is to show that this assumption results in an undecidable equality.

In order to have a decidable equality modifications were added to the rules in Agda and Coq. This results in lack of subject reduction in the theorem prover Coq and a formulation of coalgebraic types in Agda, which is severely restricted.

In our joint article [1] we demonstrated how, when following the well known fact in category theory that final coalgebras are the dual of initial algebras, we obtain a formulation of final and weakly final coalgebras which is completely symmetrical to that of initial or weakly initial algebras. Introduction rules for algebras are given by the constructors, whereas elimination rules correspond to recursive pattern matching. Elimination rules for coalgebras are given by destructors, whereas introduction rules are given by recursive copattern matching. The resulting theory was shown to fulfil subject reduction. The article [1] allowed nested pattern and copattern matching and even mixing of the two. That article allows as well full recursion and therefore is not normalising.

In the second part of our talk we will investigate how to represent codata types which are often given by having several constructors, in this coalgebraic setting using a suitable abbreviation mechanism. Functions can be almost written in the same way as using codata types, while maintaining the fact that there are no special restrictions on the reductions as needed when using codata types.

In the third part, we will show how to reduce nested copattern and pattern matching to simple (non-nested) pattern matching. We will extend the algorithm replacing nested pattern matching for algebras in [2]. Then we introduce two versions of (co)recursion operators. One is allows full (co)recursion (and could be replaced by (co)case distinction and the Y-combinator), and the other corresponds to primitive (co)recursion, which is essentially $F_{(co)rec}$ in [3]. All terms can now be translated using the full (co)recursion operators into combinatorial terms. Terms which allow the translation into primitive (co)recursion operators should be those which are to be passed by a termination checker in an implementation of the calculus in [1].

As an example the full and primitive (co)recursion operator for $\mathbb{N}$ and Stream are:

$$
\begin{array}{l}
\mathrm{P}_{\mathbb{N},A} : A \to (\mathbb{N} \to A \to A) \to \mathbb{N} \to A \\
\mathrm{P}_{\mathbb{N},A}\ \mathrm{step}_0\ \mathrm{step}_{\mathrm{suc}}\ 0 \quad = \quad \mathrm{step}_0 \\
\mathrm{P}_{\mathbb{N},A}\ \mathrm{step}_0\ \mathrm{step}_{\mathrm{suc}}\ (\mathrm{suc}\ n) \quad = \quad \mathrm{step}_{\mathrm{suc}}\ n\ (\mathrm{P}_{\mathbb{N},A}\ \mathrm{step}_0\ \mathrm{step}_{\mathrm{suc}}\ n)
\end{array}
$$

$$
\begin{array}{l}
\mathrm{R}_{\mathbb{N},A} : ((\mathbb{N} \to A) \to A) \to ((\mathbb{N} \to A) \to \mathbb{N} \to A) \to \mathbb{N} \to A \\
\mathrm{R}_{\mathbb{N},A}\ \mathrm{step}_0\ \mathrm{step}_{\mathrm{suc}}\ 0 \quad = \quad \mathrm{step}_0\ (\mathrm{R}_{\mathbb{N},A}\ \mathrm{step}_0\ \mathrm{step}_{\mathrm{suc}}) \\
\mathrm{R}_{\mathbb{N},A}\ \mathrm{step}_0\ \mathrm{step}_{\mathrm{suc}}\ (\mathrm{suc}\ n) \quad = \quad \mathrm{step}_{\mathrm{suc}}\ (\mathrm{R}_{\mathbb{N},A}\ \mathrm{step}_0\ \mathrm{step}_{\mathrm{suc}})\ n
\end{array}
$$

*Supported by EPSRC grant EP/G033374/1, theory and applications of induction-recursion. Part of this work was done while the second author was a visiting fellow of the Isaac Newton Institute for Mathematical Sciences, Cambridge, UK.

$$\begin{array}{l}\mathrm{coP}_{\mathrm{Stream},A} : (A \to \mathbb{N}) \to (A \to (\mathrm{Stream} + A)) \to A \to \mathrm{Stream}\\ \mathrm{head}\ (\mathrm{coP}_{\mathrm{Stream},A}\ \mathrm{step}_{\mathrm{head}}\ \mathrm{step}_{\mathrm{tail}}\ a) \;=\; \mathrm{step}_{\mathrm{head}}\ a\\ \mathrm{tail}\ \ (\mathrm{coP}_{\mathrm{Stream},A}\ \mathrm{step}_{\mathrm{head}}\ \mathrm{step}_{\mathrm{tail}}\ a) \;=\; \mathrm{case}_{\mathrm{Stream},A,\mathrm{Stream}}\ (\lambda\ s.s)\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathrm{coP}_{\mathrm{Stream},A}\ \mathrm{step}_{\mathrm{head}}\ \mathrm{step}_{\mathrm{tail}})\ (\mathrm{step}_{\mathrm{tail}}\ a)\end{array}$$

$$\begin{array}{l}\mathrm{case}_{A,B,C} : (A \to C) \to (B \to C) \to (A + B) \to C\\ \mathrm{case}_{A,B,C}\ \mathrm{step}_{\mathrm{inl}}\ \mathrm{step}_{\mathrm{inr}}\ \ (\mathrm{inl}\ a) \;=\; \mathrm{step}_{\mathrm{inl}}\ a\\ \mathrm{case}_{A,B,C}\ \mathrm{step}_{\mathrm{inl}}\ \mathrm{step}_{\mathrm{inr}}\ \ (\mathrm{inr}\ b) \;=\; \mathrm{step}_{\mathrm{inr}}\ b\end{array}$$

$$\begin{array}{l}\mathrm{coR}_{\mathrm{Stream},A} : ((A \to \mathrm{Stream}) \to A \to \mathbb{N}) \to ((A \to \mathrm{Stream}) \to A \to \mathrm{Stream}) \to \mathrm{Stream}\\ \mathrm{head}\ (\mathrm{coR}_{\mathrm{Stream},A}\ \mathrm{step}_{\mathrm{head}}\ \mathrm{step}_{\mathrm{tail}}\ a) \;=\; \mathrm{step}_{\mathrm{head}}\ (\mathrm{coR}_{\mathrm{Stream},A}\ \mathrm{step}_{\mathrm{head}}\ \mathrm{step}_{\mathrm{tail}})\ a\\ \mathrm{tail}\ (\mathrm{coR}_{\mathrm{Stream},A}\ \mathrm{step}_{\mathrm{head}}\ \mathrm{step}_{\mathrm{tail}}\ a) \;=\; \mathrm{step}_{\mathrm{tail}}\ (\mathrm{coR}_{\mathrm{Stream},A}\ \mathrm{step}_{\mathrm{head}}\ \mathrm{step}_{\mathrm{tail}})\ a\end{array}$$

As a simple example consider for some fixed $N : \mathbb{N}$ the stream (cycle n), which is informally written as $(n, n-1, \cdots, 0, N, N-1, N-2, \cdots 0, N, N-1, \cdots)$:

$$\begin{array}{lcl}\mathrm{cycle} : \mathbb{N} \to \mathrm{Stream}\\ \mathrm{head}\ (\mathrm{cycle}\ n) & = & n\\ \mathrm{tail}\ \ (\mathrm{cycle}\ 0) & = & \mathrm{cycle}\ N\\ \mathrm{tail}\ \ (\mathrm{cycle}\ (\mathrm{suc}\ n)) & = & \mathrm{cycle}\ n\end{array}$$

The algorithm for replacing it by non-nested (co)pattern matching yields:

$$\begin{array}{lcl}\mathrm{cycle} : \mathbb{N} \to \mathrm{Stream}\\ \mathrm{head}\ (\mathrm{cycle}\ n) & = & n\\ \mathrm{tail}\ \ (\mathrm{cycle}\ n) & = & \mathrm{cycle}_0\ n\end{array} \qquad \begin{array}{lcl}\mathrm{cycle}_0 : \mathbb{N} \to \mathrm{Stream}\\ \mathrm{cycle}_0\ 0 & = & \mathrm{cycle}\ N\\ \mathrm{cycle}_0\ (\mathrm{suc}\ n) & = & \mathrm{cycle}\ n\end{array}$$

which in this case can be replaced by primitive (co)recursion:

$$\begin{array}{l}\mathrm{cycle} : \mathbb{N} \to \mathrm{Stream}\\ \mathrm{cycle} = \mathrm{coP}_{\mathrm{Stream},\mathbb{N}}\ (\lambda n.n)\ \mathrm{cycle}_1\end{array} \qquad \begin{array}{l}\mathrm{cycle}_1 : \mathbb{N} \to (\mathrm{Stream} + \mathbb{N})\\ \mathrm{cycle}_1 = \mathrm{P}_{\mathbb{N},(\mathrm{Stream}+\mathbb{N})}\ (\mathrm{inr}\ N)\ (\lambda n, x.\mathrm{inr}\ n)\end{array}$$

By Mendler [4] and Geuvers [3] it follows that the restriction to primitive (co)recursion operators is fully normalising, which implies that a termination checked version of the calculus in [1] is normalising.

We would like to thank the anonymous referees for valuable comments on earlier versions of this abstract.

References

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 27–38, 2013.

[2] Chi Ming Chuang. *Extraction of Programs for Exact Real Number Computation using Agda*. PhD thesis, Dept. of Computer Science, Swansea University, Swansea SA2 8PP, UK, March 2011. Availabe from http://www.swan.ac.uk/∼csetzer/articlesFromOthers/index.html.

[3] Herman Geuvers. Inductive and coinductive types with iteration and recursion. In B. Nordström, K. Petersson, and G. Plotkin, editors, *Informal proceedings of the 1992 workshop on Types for Proofs and Programs, Bastad 1992, Sweden*, pages 183 – 207, 1992.

[4] N. P. Mendler. Inductive types and type constraints in second-order lambda calculus. In *Proceedings of the Second Symposium of Logic in Computer Science. Ithaca, N.Y.*, pages 30 – 36. IEEE, 1987.