Productive Infinite Objects via Copatterns

Andreas Abel

Department of Computer Science and Engineering
Chalmers and Gothenburg University
Gothenburg, Sweden
andreas.abel@gu.se

Inductive data such as lists and trees is modeled category-theoretically as *algebra* where *construction* is the primary concept and elimination is obtained by initiality. In a more practical setting, functions are programmed by *pattern matching* on inductive data. Dually, coinductive structures such as streams and processes are modeled as *coalgebras* where *destruction* (or transition) is primary and construction rests on finality [Hag87]. Due to the coincidence of least and greatest fixed-point types [SP82] in lazy languages such as Haskell, the distinction between inductive and coinductive types is blurred in partial functional programming. As a consequence, coinductive structures are treated just as infinitely deep (or, non-well-founded) trees, and pattern matching on coinductive data is the dominant programming style. In total functional programming, which is underlying the dependently-typed proof assistants Coq [INR12] and Agda [Nor07], the distinction between induction and coinduction is vital for the soundness, and pattern matching on coinductive data leads to the loss of subject reduction [Gim96]. Further, in terms of expressive power, the *productivity checker* for definitions by coinduction lacks behind the termination checker for inductively defined functions.

It is thus worth considering the alternative picture that a *coalgebraic approach* to coinductive structures might offer for total and, especially, for dependently-typed programming. The coalgebraic approach as pioneered by Hagino has been followed in the design of the language *Charity* [CF92] and advocated by Setzer for use in Type Theory [Set12]. Now, if "algebraic programming" amounts to defining functions by pattern matching, what is "coalgebraic programming"? Or, asked otherwise, what is the proper dualization of pattern matching, what is *copattern* matching?

While patterns match the introduction forms of finite data, copatterns match on elimination contexts for infinite objects, which are applications (eliminating functions) and destructors/projections (eliminating coalgebraic types = Hagino's codatatypes = Cockett's final datatypes). An infinite object such as a function or a stream can be defined by its behavior in all possible contexts. Thus, if we consider a set of copatterns covering all possible elimination contexts, plus the object's response for each of the copatterns, that object is defined uniquely. More concretely, a stream is determined by its head and its tail, thus, we can introduce a new stream object by giving two equations; one that specifies the value it produces if its head is demanded, and one for the case that the tail is demanded. Another covering set of copatterns consists of head, head of tail, and tail of tail. For instance, the stream of Fibonacci numbers can be given by the three equations, using a function $\mathsf{zipWith}\ f\ s\ t$ which pointwise applies the binary function f to the elements of streams s and t.

$$
\begin{aligned}
\mathsf{zipWith}\ f\ s\ t\ .\mathsf{head} &= f\ (s\ .\mathsf{head})\ (t\ .\mathsf{head}) \\
\mathsf{zipWith}\ f\ s\ t\ .\mathsf{tail} &= \mathsf{zipWith}\ f\ (s\ .\mathsf{tail})\ (t\ .\mathsf{tail}) \\
\\
\mathsf{fib}\ .\mathsf{head} &= 0 \\
\mathsf{fib}\ .\mathsf{tail}\ .\mathsf{head} &= 1 \\
\mathsf{fib}\ .\mathsf{tail}\ .\mathsf{tail} &= \mathsf{zipWith}\ (+)\ \mathsf{fib}\ (\mathsf{fib}\ .\mathsf{tail})
\end{aligned}
$$

Taking the above equations as left-to-right rewrite rules, we obtain a strongly normalizing system. This is in contrast to the conventional definition of fib in terms of the stream constructor $h :: t$ by

$$\mathsf{fib} = 0 :: 1 :: \mathsf{zipWith}\ (+)\ \mathsf{fib}\ (\mathsf{fib}\,.\mathsf{tail})$$

which, even if unfolded under destructors only, admits an infinite reduction sequence starting with $\mathsf{fib}\,.\mathsf{tail} \longrightarrow 1 :: \mathsf{zipWith}\ (+)\ \mathsf{fib}\ (\mathsf{fib}\,.\mathsf{tail}) \longrightarrow 1 :: \mathsf{zipWith}\ (+)\ \mathsf{fib}\ (1 :: \mathsf{zipWith}\ (+)\ \mathsf{fib}\ (\mathsf{fib}\,.\mathsf{tail}))$ $\longrightarrow \ldots$ The crucial difference is that $\mathsf{fib}\,.\mathsf{tail}$ does not reduce if we choose the definition by copatterns above, since the elimination .tail is not matched by any of the copatterns; only in contexts .head or .tail .head or .tail .tail it is that fib springs into action.

Using definitions by copattern matching, we reduce productivity to termination and productivity checking to termination checking. As termination of a function is usually proven by a measure on the size of the function arguments, we prove productivity by well-founded induction on the size of the elimination context. For instance, fib is productive because the recursive calls occur in smaller contexts: at least one tail-destructor is "consumed" and, equally important, zipWith does not add any more destructors. The number of eliminations (as well as the size of arguments) can be tracked by sized types [HPS96], reducing productivity (and termination) checking to type checking. For a polymorphic lambda-calculus with inductive and coinductive types and patterns and copatterns, this has been spelled out in joint work with Brigitte Pientka [AP13]. An introductory study of copatterns and covering sets thereof can be found in previous work [APTS13].

References

[AP13] Andreas Abel and Brigitte Pientka. Wellfounded recursion with copatterns: A unified approach to termination and productivity. In *International Conference on Functional Programming (ICFP 2013)*. ACM Press, 2013.

[APTS13] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming infinite structures by observations. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'13, Rome, Italy, January 23 - 25, 2013*, pages 27–38. ACM Press, 2013.

[CF92] Robin Cockett and Tom Fukushima. About Charity. Technical report, Department of Computer Science, The University of Calgary, 1992. Yellow Series Report No. 92/480/18.

[Gim96] Eduardo Giménez. *Un Calcul de Constructions Infinies et son application a la vérification de systèmes communicants*. PhD thesis, Ecole Normale Supérieure de Lyon, 1996. Thèse d'université.

[Hag87] Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 410–423, 1996.

[INR12] INRIA. *The Coq Proof Assistant Reference Manual*. INRIA, version 8.4 edition, 2012.

[Nor07] Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, 2007.

[Set12] Anton Setzer. Coalgebras as types determined by their elimination rules. In *Epistemology versus Ontology: Essays on the Philosophy and Foundations of Mathematics in Honour of Per Martin-Löf*, volume 27 of *Logic, Epistemology, and the Unity of Science*, pages 351–369. Springer-Verlag, 2012.

[SP82] Michael B. Smyth and Gordon D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4):761–783, 1982.