Coinduction in Agda
Using Copatterns and Sized Types

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. Understanding "algebraic programming" as defining functions by pattern matching, the dualization "coalgebraic programming" leads us to the notion of 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). 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.

$$\text{record Stream \{i : \text{Size}\} (A : \text{Set}) : \text{Set} \text{ where}\n$$
$$\text{coinductive}\n$$
$$\text{constructor } _\text{::}_ \text{field } \text{head} : A\n$$
$$\text{tail} : \forall \{j : \text{Size< i}\} \rightarrow \text{Stream \{j\} A}\n$$
$$\text{open Stream public}\n$$
$$\text{zipWith} : \forall \{i A B C\} (f : A \rightarrow B \rightarrow C) \rightarrow \text{Stream \{i\} A \rightarrow \text{Stream \{i\} B \rightarrow \text{Stream \{i\} C}\n$$
$$\text{head} \ (\text{zipWith} \ f \ s \ t) = f \ (\text{head} \ s) \ (\text{head} \ t)\n$$
$$\text{tail} \ (\text{zipWith} \ f \ s \ t) = \text{zipWith} \ f \ (\text{tail} \ s) \ (\text{tail} \ t)\n$$

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 $\text{zipWith} \ f \ s \ t$ which pointwise applies the binary function $f$ to the elements of streams $s$ and $t$. 
fib : \( \forall \{ i \} \rightarrow \text{Stream} \{ i \} \mathbb{N} \)

( (head fib)) = 0

(\(\text{head} \ (\text{tail} \ fib)\)) = 1

(\(\text{tail} \ (\text{tail} \ fib)\)) = \text{zipWith } _+\_ \ fib \ (\text{tail} \ fib)

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 \(\text{fib}\) in terms of the stream constructor \(h :: t\) by

\(\text{fib} = 0 :: 1 :: \text{zipWith } _+\_ \ fib \ (\text{tail} \ fib)\)

which, even if unfolded under destructors only, admits an infinite reduction sequence starting with \(\text{tail} \ fib\)

\(\rightarrow 1 :: \text{zipWith } _+\_ \ fib \ (\text{tail} \ fib)\) \(\rightarrow 1 :: \text{zipWith } _+\_ \ fib \ (1 :: \text{zipWith } _+\_ \ fib \ (\text{tail} \ fib))\) \(\rightarrow \ldots \)

The crucial difference is that \(\text{tail} \ fib\) does not reduce if we choose the definition by copatterns above, since

the elimination \(\text{tail}\) is not matched by any of the copatterns; only in contexts \(\text{head}\) or \(\text{head}\) of \(\text{tail}\) or \(\text{tail}\) of

\(\text{tail}\) it is that \(\text{fib}\) 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, \(\text{fib}\) is productive because the recursive calls occur in smaller

contexts: at least one \(\text{tail}\)-destructor is “consumed” and, equally important, \text{zipWith} does not add any

more destructors. The number of eliminations (as well as the size of arguments) can be tracked by sized

types \([\text{HPS96}]\), 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 \([\text{AP13}]\). An introductory study of copatterns and covering sets

thereof can be found in previous work \([\text{APTS13}]\).

A similar abstract has appeared under the title \textit{Productive Infinite Objects via Copatterns} in the informal pro-

ceedings of NWPT 2013 (Nordic Workshop of Programming Theory, Tallinn, Estonia, November 2013), and under

the title \textit{Programming and Reasoning with Infinite Structures Using Copatterns and Sized Types} in the proceedings

of ATPS 2014 (Arbeitstagung Programmiersprachen, Kiel, Germany, February 2014).

\[\text{References}\]


structures by observations. In Proc. of the 40th ACM Symp. on Principles of Programming Languages,


[Gim96] Eduardo Giménez. \textit{Un Calcul de Constructions Infinies et son application a la vérification de systèmes


[HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized

types. In Proc. of the 23rd ACM Symp. on Principles of Programming Languages, POPL’96, pages


[Nor07] Ulf Norell. \textit{Towards a Practical Programming Language Based on Dependent Type Theory}. PhD thesis,


[SP82] Michael B. Smyth and Gordon D. Plotkin. The category-theoretic solution of recursive domain equa-