Matching (Co)patterns with Cyclic Proofs

Lide Grotenhuis & Daniël Otten
University of Amsterdam

Overview. We investigate connections between cyclic proof theory and recursive functions defined by (co)pattern matching. Cyclic proof systems replace (co)induction rules with sound forms of circular reasoning. For example, by adding a cycle between the green nodes we get a cyclic proof in arithmetic:

$$\dfrac{\dfrac{}{0+0=0}\,{+_0} \quad \dfrac{\dfrac{}{0+\mathsf{suc}\,x'=\mathsf{suc}(0+x')}\,{+_{\mathsf{suc}}} \quad \dfrac{0+x'=x'}{\mathsf{suc}(0+x')=\mathsf{suc}\,x'}\,\mathrm{cong}_{\mathsf{suc}}}{0+\mathsf{suc}\,x'=\mathsf{suc}\,x'}\,\mathrm{trans}}{0+x=x}\,\mathrm{case}_x$$

This proof is sound because the variable x is decreased to its predecessor x' before we cycle back, and so it represents a proof by infinite descent. The advantage of these systems lies in proof search: to apply (co)induction we need to guess the right (co)induction hypothesis, whereas with cycles we can start generating the proof until our current goal matches one that we have seen before. Under the Curry-Howard correspondence, such cycles correspond to recursive function calls, while the soundness condition ensures that the function always terminates:

cyclic proof	recursive function
fixpoint formula	(co)inductive type
cycle	recursive function call
soundness condition	termination checking

In this way, recursive functions defined with dependent (co)pattern matching can be seen as a proof-relevant and dependent generalisation of cyclic proofs. Our goal is to explain these correspondences and use them to extend type-theoretic conservativity results.

(Co)pattern matching. An *inductive type* is a type I that is specified by a finite set of constructors of the form $c : A_0 \to \cdots \to A_{n-1} \to I$, where I occurs strictly positive in the A_i; one can view I as the type freely generated by these constructors, or as the least type with such functions. For example, the natural numbers $\mathbb{N}$ can be defined as the inductive type specified by the constructors $\mathsf{0} : \mathbb{N}$ and $\mathsf{suc} : \mathbb{N} \to \mathbb{N}$. Proof assistants like `Agda`, `Dedukti`, `Rocq`, and `Lean` allow the user to define functions on inductive types using *pattern matching*. For example, the predecessor function $\mathsf{pred} : \mathbb{N} \to \mathbb{N}$ can be defined as follows:

$$\mathsf{pred}\,n \coloneqq \mathsf{case}\,n \begin{cases} 0 \mapsto 0, \\ \mathsf{suc}\,n' \mapsto n'. \end{cases}$$

The idea here is that we define $f\,n$ via a case distinction on the form (or 'pattern') of n, that is, how n was obtained from the constructors of $\mathbb{N}$. Dually, functions with a *coinductive type* as codomain can be defined using *copattern matching*.

Definitions by (co)pattern matching can contain recursive calls. Addition for example:

$$\mathsf{add}\,m\,n \coloneqq \mathsf{case}\,m \begin{cases} 0 \mapsto m, \\ \mathsf{suc}\,m' \mapsto \mathsf{suc}\,(\mathsf{add}\,m'\,n). \end{cases}$$

Which function definitions are accepted differs per proof assistant, and depends in particular on which unification algorithm is employed. One point of interest here is whether the accepted functions could also have been implemented using primitive (co)inductive rules; such a conservativity result shows that any function accepted by the proof assistant has an interpretation in any model of the type theory. At the moment, the known conservativity results [GMM06, CDP16, Thi20] only cover pattern matching definitions satisfying structural recursion: there is one inductive input that is decreased in every recursive call (or one coinductive output that is productive before every recursive call). However, `Agda` and `Dedukti` allow more complex interleaving of recursive calls. To implement these complex functions using primitive (co)induction rules, we need to apply induction and coinduction multiple times, and often in a specific order. For example, for the Ackermann function we need a lexicographical order on inputs:

$$\begin{aligned}&\mathsf{ack} : \mathbb{N} \to \mathbb{N} \to \mathbb{N},\\ &\mathsf{ack}\, m\, n \coloneqq \mathsf{case}\, m \begin{cases} 0 \mapsto \mathsf{suc}\, n, \\ \mathsf{suc}\, m' \mapsto \mathsf{case}\, n \begin{cases} 0 \mapsto \mathsf{ack}\, m'\, 1, \\ \mathsf{suc}\, n' \mapsto \mathsf{ack}\, m'\, (\mathsf{ack}\, (\mathsf{suc}\, m')\, n'). \end{cases} \end{cases}\end{aligned}$$

As another example, the next function uses induction on both inputs simultaneously:

$$\begin{aligned}&\mathsf{aggr} : \mathbb{N} \to \mathbb{N} \to \mathbb{N},\\ &\mathsf{aggr}\, m\, n \coloneqq \mathsf{case}\, m \begin{cases} 0 \mapsto 0, \\ \mathsf{suc}\, m' \mapsto \mathsf{case}\, n \begin{cases} 0 \mapsto \mathsf{suc}\, 0, \\ \mathsf{suc}\, n' \mapsto \mathsf{aggr}\, m'\, m' + \mathsf{aggr}\, n'\, n'. \end{cases} \end{cases}\end{aligned}$$

In general, the acceptance condition employed in `Agda` is the *size-change termination principle*: for any infinite sequence of function calls that might occur, there should eventually be an input/output that we can track, where progress is made infinitely often [LJBA01, Wah07]. This can either be an inductive input that is decreased infinitely often, or a coinductive output that is productive infinitely often. This principle ensures termination, but it is no longer clear how the accepted functions can be implemented using primitive (co)induction rules.

Cyclic proofs. On the proof-theoretic side, the conservativity of pattern matching corresponds to the conservativity of cyclic proofs over finitary proofs with an explicit (co)induction rule, which has for example been studied for the modal μ-calculus [SD03] and for Peano and Heyting arithmetic [Sim17, Das20, BT17, Weh23]. Though the setting here is generally proof-irrelevant, both [SD03] and [Weh23] do try to preserve the 'content' of the cyclic proof when devising the corresponding inductive proof: namely, they unfold the cyclic proof until the structure of the proof tree matches the induction order of the cycles, and then obtain an inductive proof by replacing each cycle by an appropriate inductive argument. Crucial to this procedure is a well-defined induction order on the cycles; however, for any cyclic proof system fitting the abstract framework of [LW24], such an order exists for (an appropriate unfolding of) every cyclic proof.

Outlook. Inspired by the proof-theoretic picture, we hope to extend the current type-theoretic conservativity results by allowing more complex interleaving function calls. Indeed, we can view the definitions by (co)pattern matching satisfying the size-change termination principle as a cyclic proof system with a global soundness condition that fits the abstract framework specified in [LW24]. Following an approach similar to [SD03] and [Weh23], we hope to turn these cyclic proofs into inductive proofs in a way that preserves computational content.

References

[BT17] Stefano Berardi and Makoto Tatsuta. Equivalence of inductive definitions and cyclic proofs under arithmetic. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12. IEEE, 2017.

[CDP16] Jesper Cockx, Dominique Devriese, and Frank Piessens. Eliminating dependent pattern matching without k. *Journal of functional programming*, 26:e16, 2016.

[Das20] Anupam Das. On the logical complexity of cyclic arithmetic. *Logical Methods in Computer Science*, 16, 2020.

[GMM06] Healfdene Goguen, Conor McBride, and James McKinna. *Eliminating Dependent Pattern Matching*, pages 521–540. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. *SIGPLAN Not.*, 36(3):81–92, January 2001.

[LW24] Graham E. Leigh and Dominik Wehr. From gtc to image 1: Generating reset proof systems from cyclic proof systems. *Annals of Pure and Applied Logic*, 175(10):103485, 2024.

[SD03] Christoph Sprenger and Mads Dam. On the structure of inductive reasoning: Circular and tree-shaped proofs in the μ-calculus. In Andrew D. Gordon, editor, *Foundations of Software Science and Computation Structures*, pages 425–440, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[Sim17] Alex Simpson. Cyclic arithmetic is equivalent to peano arithmetic. In *International Conference on Foundations of Software Science and Computation Structures*, pages 283–300. Springer, 2017.

[Thi20] David Thibodeau. *An Intensional Type Theory of Coinduction Using Copatterns*. PhD thesis, McGill University Montréal, QC, Canada, 2020.

[Wah07] David Wahlstedt. *Dependent Type Theory with Parameterized First-Order Data Types and Well-Founded Recursion*. Chalmers Tekniska Hogskola (Sweden), 2007.

[Weh23] Dominik Wehr. Representation matters in cyclic proof theory. 2023.