Monsters: Programming and Reasoning

Venanzio Capretta[1] and Christopher Purdy[2]

[1] University of Nottingham, UK, venanzio.capretta@nottingham.ac.uk
[2] Cambridge University, UK, cp766@cam.ac.uk

A monadic stream (which we call a *monster*) is a potentially infinite sequence of values in which every element triggers a monadic action. Monsters are useful tools in functional programming: they can be instantiated to pure streams, lazy lists, finitely branching trees, interactive processes, state machines, and many other data structures.

A monster σ consists of an action for some monad M that, when executed, returns a head (first element) and a tail (continuation of the stream). This process is repeated in non-well-founded progression: monsters form a coinductive type.

Here is the formal type-theoretic definition of the set of streams with base monad M and elements of type A (*M-monsters*), in Haskell/Agda notation:

$$\begin{array}{l}\textbf{codata}\ \mathbb{S}_M\ A : \mathsf{Set}\\ \quad \mathsf{mcons}_M : M\ (A \times \mathbb{S}_M\ A) \to \mathbb{S}_M\ A\end{array}$$

A previous article [4] introduced monadic streams and proved that polymorphic discrete functions on them are always continuous. A slightly different definition of monadic stream functions have been studied previously by Perez, Bärenz and Nilsson [7] to model signal processors. The definition of M-monsters is very close to that of *cofree (or iterative) comonad*, which can be seen as the type of M-monsters with a pure leading value [2, 5].

The functor M needs not be a monad for the type to be well-defined, though it enjoys some convenient properties when it is. For example, when M is a monad, monadic stream functions (isomorphic to monsters with the underlying functor ReaderT M) are arrows [7]. However, for the coinductive definition of $\mathbb{S}_M$ to be sound, the functor $M(A\times-)$ must have a final coalgebra. This is the case, for example, if M is a container [1]. (Monadic containers, in particular, are related to universes closed under Σ-types [3].)

Some important data structures are obtained as instances of monsters. If we choose M to be the identity functor, we obtain *pure streams*, that is, infinite sequences of elements of A. If we choose M to be the Maybe monad, we obtain *lazy lists*: the Just constructor returns a head element and a tail; the Nothing constructor terminates the list; since the type is coinductive, lists may go on forever. If we choose M to be the List constructor, we obtain *finitely branching trees*: a node consists of a list of children, each comprising an element of A and a subtree; if the list is empty we have a leaf; since the type is coinductive, trees need not be well-founded. If we choose M to be the State monad, we obtain *state machines*: processes that output an infinite stream of values depending on an underlying mutable state. If we choose M to be the IO monad, we obtain *interactive processes*: every stage of the stream is an input-output action that returns an element and a new process.

We developed an extensive library of generic functions for monsters in Haskell, publicly available on GitHub at https://github.com/venanzio/monster.
It provides generalizations of many operations on lists, streams, trees, and state machines. They allow high-level programming of abstract algorithms that can be instantiated to those data structures and others.

We also defined instances of the type classes of Functor/Applicative/Monad for $\mathbb{S}_M$. However, these satisfy the corresponding class laws only under certain conditions. If M is a functor, it is straightforward to prove that $\mathbb{S}_M$ is a functor. If M is applicative, we proved that $\mathbb{S}_M$ is also applicative: it is surprisingly hard to establish this fact; the proof is complex and requires the definition of new operators and several intermediate technical lemmas. We are working on the verification of these results in Coq [9] and Agda [8]. We're exploring the possibility that a simpler proof could be derived from some abstract theorems on lax monoidal functors [6].

Finally, $\mathbb{S}_M$ is not in general a monad, even when M is: we showed this by a counterexample for State-monsters that violates the monadic laws. The monad class requires the definition of an operator $\mathsf{join} : \mathbb{S}_M\ (\mathbb{S}_M\ A) \to M\ A$ satisfying certain laws. We can see an element of $\mathbb{S}_M\ (\mathbb{S}_M\ A)$ as a *monster matrix*, a 2-dimensional array of elements of A in which both columns and rows emerge from M-actions. This must be compressed into a linear monster: it could only be done (generalizing the instantiation for pure streams) by travelling down the diagonal. However, there is no way, in a monster matrix, to step from one diagonal element to the next: we must always start from the outside of the matrix and choose the next column from there. Accordingly, the evaluation of each element on the diagonal activates the same M-actions repeatedly: this results in the failure of the monadic laws.

To ensure that $\mathbb{S}_M$ is a monad, M must satisfy additional requirements. We are looking for the minimal set of these, but we know it is sufficient for M to be a *representable* monad (which is the case for several common instances like Maybe and List).

In conclusion, we developed an extensive Haskell library on monadic streams (*monsters*) that provides many high-level operators that can be used on a wide range of important data structures. We also provide instances of the type classes Functor, Applicative and Monad, which hold valid under some assumption on the underlying type operator M.

References

[1] Michael Abott, Thorsten Altenkirch, and Neil Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.

[2] Peter Aczel, Jirí Adámek, Stefan Milius, and Jiri Velebil. Infinite trees and completely iterative theories: a coalgebraic view. *Theor. Comput. Sci.*, 300(1-3):1–45, 2003.

[3] Thorsten Altenkirch and Gun Pinyo. Monadic containers and Σ-universes. In *TYPES 2017 Abstracts*, pages 20–21, 2017.

[4] Venanzio Capretta and Jonathan Fowler. The continuity of monadic stream functions. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12. IEEE Computer Society, 2017.

[5] Venanzio Capretta, Tarmo Uustalu, and Varmo Vene. Recursive coalgebras from comonads. *Information and Computation*, 204(4):437–468, April 2006.

[6] Conor McBride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

[7] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. Functional reactive programming, refactored. In Geoffrey Mainland, editor, *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, pages 33–44. ACM, 2016.

[8] The Agda Team. *Agda User Manual*, 2021. Release 2.6.2.

[9] The Coq Development Team. *The Coq Proof Assistant. Reference Manual. Version 8.5.* INRIA, 2016. `http://coq.inria.fr/refman/index.html`.