

Random State Comonads encode Cellular Automata evaluation

Madalina I Sas

Centre for Complexity Science, Imperial College London, UK

Julian HJ Sutherland

Nethermind, UK

Cellular automata (CA) are quintessential ALife and ubiquitous in many studies of collective behaviour and emergence, from morphogenesis to social dynamics and even brain modelling. Recently, there has been an increased interest in formalising CA, theoretically through category theory and practically in terms of a functional programming paradigm. Unfortunately, these remain either in the realm of simple implementations lacking important practical features, or too abstract and conceptually inaccessible to be useful to the ALife community at large. In this paper, we present a brief and accessible introduction to a category-theoretical model of CA computation through a practical implementation in Haskell. We instantiate arrays as comonads with state and random generators, allowing stochastic behaviour not currently supported in other known implementations. We also emphasise the importance of functional implementations for complex systems: thanks to the Curry-Howard-Lambek isomorphism, functional programs facilitate a mapping between simulation, system rules or semantics, and categorical descriptions, which may advance our understanding and development of generalised theories of emergent behaviour. Using this implementation, we show case studies of four famous CA models: first Wolfram's CA in 1D, then Conway's game of life, Greenberg-Hastings excitable cells, and the stochastic Forest Fire model in 2D, and present directions for an extension to N dimensions. Finally, we suggest that the comonadic model can encode arbitrary topologies and propose future directions for a comonadic network.

I. INTRODUCTION

The origins of life remain mysterious. As mysterious is the ability of artificial life to imitate emergent life-like properties and behaviours, and to do so by computation. As such, ALife is intrinsically linked to computer science, as in the study of computation, and to complexity science, as in the study of emergence.

All life can be seen as a complex system, made up of a multitude of components which interact in simple but often non-linear ways, showing emergent properties or behaviours as the result of self-organisation (Jensen 2022). Emergence in complex systems is often studied by modelling these interactive processes and creating the simplest simulation that can reproduce the emergent behaviour at the system level. Phenomena reproducible in this way are often deemed *weakly* emergent (Bedau 1997): this method bypasses the explanatory gap of strong emergentist accounts, but also limits the phenomena explored to those that can be expressed as *computation*.

Cellular automata (CA) are an obvious example of such computational emergent behaviour, and can model a broad range of real-world phenomena. Being discrete in time and space, they allow the utter simplification of the interactive processes between parts. Emergence manifests even in a 1D model such as the one proposed by Wolfram (1983), where a trivial rule can produce pure unpredictable chaos, with such computational complexity it has puzzled cryptanalysts (Wolfram 2019), while maintaining spatial patterns uncannily similar to those seen in nature (see Fig 1).

Given this simplicity, it is tempting to try to express the rules underlying the interactions between cells into a

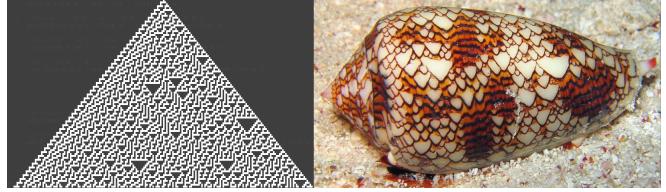


FIG. 1. Artificial versus natural emergent patterns.

(a) A Wolfram CA using rule 30 and an initial world with only one live cell in the middle. (b) The rule 30 CA shows very similar patterns to the *Conus textile* shell.

mathematical theory that describes their behaviour, but also into a practical implementation that is intrinsically linked to theory. This idea is, of course, not new: there is significant historical and recent work establishing links between category theory and automata theory, by using categorical structures to model, operationalise, and generalize automata and learning algorithms (Colcombet et al. 2020, Rine 1971, van Heerdt 2020). These methods allow us to identify the theoretic capabilities of a general interactive system based on descriptions of its input and output states, but depend on finding the right description which is often non-trivial. In the case of CA, a number of novel algebraic or categorical descriptions have been recently developed (Basold et al. 2025, Capobianco and Uustalu 2010, Widemann 2012, Widemann and Hauhs 2011).

Implementation-wise, the functional programming paradigm appears particularly well-suited: its emphasis on immutability and pure functions lacking side-effects maps to the deterministic and iterative aspects of dynamical systems, facilitates reproducibility, but also naturally aids parallelisation. Moreover, strongly-typed functional

languages, by design, enforce mathematical constraints at compilation, guaranteeing correct execution, and lend themselves towards direct correspondences to semantics. As such, the functional programming community has been taking an interest in this problem for more than a decade (Bhat and Garay 2019, Kmett 2015, Piponi 2006, Potts 2006), but in the scientific modelling of multi-agent systems most implementations remain procedural and array-based.

However, existing algebraic descriptions for CA are often too abstract to be yet of use to the ALife community at large; meanwhile, relevant practical aspects, such as the stochasticity required by the Forest Fire model, remain unaddressed in existing functional implementations.

The functional programming approach to CA could also expand our general understanding of emergent phenomena, by formalising a mapping between semantics, mathematical descriptions and simulations of complex systems. With this motivation, this work aims to reunite CA, category theory and a functional approach through an implementation in Haskell using random-comonadic arrays, and presents some case studies including stochastic CA.

II. PRELIMINARIES

At its core, a CA can be seen as executing repeated, context-dependent computations: each cell evolves based on its position in the grid, and the states of itself and its neighbours. Category theory provides abstract mathematical structures that can encode this evolution regardless of model specifics. In this section, we present the background to expressing CA both as categories and as functional programs.

A. Functional programming and type theory

Functional and procedural programming offer distinct paradigms in their approach to computation, following the two distinct conceptual frameworks supporting the Church-Turing thesis. Turing machines treat computation as a sequence of transitions between states expressed as symbol manipulations on a tape, emphasising *how* to compute (Turing 1936) – naturally expressing step-by-step procedural programming. Albeit initially less intuitive or practical, Church’s λ -calculus focuses on *what* to compute, representing computation as function application and composition (Church 1936) – inspiring declarative, function-centered programming.

In typed λ -calculus, each expression is further enriched by an associated *type*. The correctness of programs is ensured by enforcing strict rules on how types can be used. While this reduces the expressive power of λ -calculus, programs in typed λ -calculus can be mapped directly to mathematical structures (Mitchell 1990).

Furthermore, the *Curry-Howard isomorphism* allows mapping types to logical propositions, effectively providing a correspondence between type-checking and program verification (Hoare 1980), which allows for extra safety guarantees when compiling strongly-typed languages.

B. From types to categories

“Category theory takes a bird’s eye view of mathematics” (Leinster 2017), allowing very general descriptions of abstract mathematical structures and the relationships between them. Category theory can provide a unifying paradigm to model automata, enabling generic definitions and descriptions of their behaviour (Rine 1971).

A category \mathcal{C} consists of a collection of objects and relationships (*morphisms* or *arrows*) between them. Each arrow has a domain and codomain, extracted from the objects in \mathcal{C} . Morphisms can be chained using an associative composition operator \circ . An identity morphism maps each object A in \mathcal{C} to itself $\text{Id}_A : A \rightarrow A$.

Category theory provides a framework to express the semantics of typed λ -calculus: by modelling types as objects in a category, and functions as morphisms between objects. Then, compositions of morphisms corresponds to function composition, and identity morphisms correspond to identity functions. Through the above, the correspondence between programs and logic can also be expanded to categories; this is known as the *Curry-Howard-Lambeek isomorphism*.

A *functor* $F : \mathcal{C} \rightarrow \mathcal{D}$ is a higher-order concept: a mapping that preserves structure between two categories \mathcal{C} and \mathcal{D} , by assigning to each object and each morphism in \mathcal{C} an object and morphism in \mathcal{D} in a way that maintains composition and identity of morphisms. In particular, a functor $E : \mathcal{C} \rightarrow \mathcal{C}$ can map a category to itself; such *endofunctors* can be equipped with constraints and properties which can be used to intuitively encode certain kinds of computation.

A *monad* is an endofunctor $M : \mathcal{C} \rightarrow \mathcal{C}$ with two natural transformations: a unit element $e : I \rightarrow M$ and a multiplication $\mu : M \times M \rightarrow M$, satisfying identity and associativity laws analogous to those of a monoid.

A *comonad* is defined by ‘reversing the arrows’ in the monad: it is an endofunctor with a counit $\varepsilon : W \rightarrow I$ and comultiplication $\delta : W \rightarrow W \times W$.

Intuitively, a monad maps two elements to one, while a comonad maps an element to two. Monads and comonads have a *dual* relationship, such that the types of their fundamental operators are reversed. When thinking of computation, the former can act as an analogy for chaining functions sequentially, while the latter can encapsulate data extraction from a larger context (Uustalu and Vene 2008).

C. Implementation in Haskell

Haskell is a strongly-typed, functional programming language which implements many concepts in both λ -calculus and category theory.

Loosely, types correspond to objects in a category, while functions of type $a \rightarrow b$ correspond to morphisms between those objects of types a and b .

Endofunctors can be written using a data constructor, such as $f a$, for an endofunctor f and an arbitrary type a . Practically, functors can be seen as data structures that allow mapping a function (or morphism) to their elements. Lists, for example, are functors: `map` takes a function and a list and applies it to all elements, returning an updated list:

```
map :: (a → b) → [a] → [b]
fmap :: Functor f ⇒ (a → b) → f a → f b
```

The comonad can also express a kind of container or data structure for some arbitrary type. The comonadic operators $\epsilon : W \rightarrow I$ and $\delta : W \rightarrow W \times W$ map to the functions `extract` and `duplicate`:

```
extract :: Comonad w ⇒ w c → c
duplicate :: Comonad w ⇒ w c → w (w c)
```

III. A COMONADIC MODEL OF CELLULAR AUTOMATA

Let us now consider a cell of type c and a comonad w that represents a grid of cells. We compute the value of each cell in the next generation by performing a *local* computation based on its neighbours. A comonad allows us to *focus* on each individual cell in the grid, while the value of its neighbours and the rest of the grid are considered part of the *context*. A full category-theoretical definition of comonadic CA is available in [Capobianco and Uustalu \(2010\)](#). For simplicity, we will present the model using function types and data constructors in Haskell.

Most comonadic implementations of CA enhance the list functor with a context in the form of a list zipper data structure, which has a focused cell and two lists left and right for the remaining cells: $W [c] c [c]$. Checking the value of neighbours is done by context-switching: navigating the focus point to a neighbour – which can also make implementing periodic boundaries trivial.

However, lists are inefficient to index, making the expansion to multiple dimensions difficult. Taking inspiration from procedural implementations, we propose that an n -dimensional array, which is a kind of functor, features comonadic structure when considered with an index i to choose which element is currently in focus.

```
data W i c = W i (Array i c) deriving Functor
```

This can be shown by turning the array datatype into a comonad via an instance, which entails mapping the comonadic operators `extract` and `duplicate` to equivalent functions on arrays: `extract` simply returns the value of

the focused cell at index i from the array, while `duplicate` takes an array focused on element at index i , and creates a an array of arrays expressing a ‘grid-of-grids’ where each possible index is in focus.

```
instanceIx i ⇒ Comonad (W i) where
    extract (W i a) = a ! i
    duplicate (W i a) = U i (listArray ix grids)
    where
        ix = bounds a
        grids = [W i a | i ← range ix]
```

Suppose we then choose a rule by which a cell evolves according to its neighbours. The rule would take relevant neighbour states from the whole grid of type $w c$, but return only a cell of type c , with the same type as `extract`:

```
rule :: Comonad w ⇒ w c → c
```

In order to apply this rule everywhere in the grid, we need to apply it to each possible shift of the grid, for which we use `duplicate`. The `extend` function allows us to apply evolution rules r over the ‘duplicated’ system state s , effectively ‘reducing’ the nested grid-of-grids structure of type $w (w c)$ back to a grid of type $w c$ one generation later:

```
extend :: (w c → c) → w c → w c
extend r s = fmap r $ duplicate s
```

This elegant implementation is particularly general as there are no constraints on the structure of the index: the index type $lx i$ can be instantiated to either an integer, for 1D arrays, a pair for 2D arrays, or a larger tuple or a fixed-length list for arbitrary dimensions.

A. Transform to random

Being able to add random computations is crucial to many multi-agent simulations. To implement this in a pure languages, the side-effects of stochasticity can be contained in a monad, allowing reproducibility based on random seeds. We also use the random monad to create random initial conditions, by making a cell an instance of the `Random` class and using a random number generator.

To allow stochasticity at each iteration, two transformations need be applied to the comonad datatype. First, we use an environment comonad transformer, `EnvT`, which allows us to carry-on any model parameters as state, and the random monad transformer, `RandT`, which allows us to contain the side-effects of a random number generator with a seed g :

```
data RandGrid p i c =
    RandT StdGen (EnvT Params (U i)) c
```

Using this data structure, we can apply various array dimensions and indices, cell datatypes, or simulation parameters, allowing for a very versatile range of simulations, as shown by the case studies to follow.

Our full implementation is available online ([Sas and Sutherland 2024](#)) and uses the `Comonad` package ([Kmett 2014](#)).

IV. CASE STUDIES

Below, we present four case studies of CA of increasing semantic complexity, with simulations produced by the implementation described above.

A. Wolfram CA

The one-dimensional CA proposed by [Wolfram \(1983\)](#) is usually the starting point of all functional implementations of CA (such as [Piponi \(2006\)](#)). Cell values are binary, and its evolution rules are given by a function applied to value of the current cell and its two immediate neighbours, with some combinations producing unexpectedly complex outcomes. In this example we use rule 22, which produces the famous Sierpinski triangle.

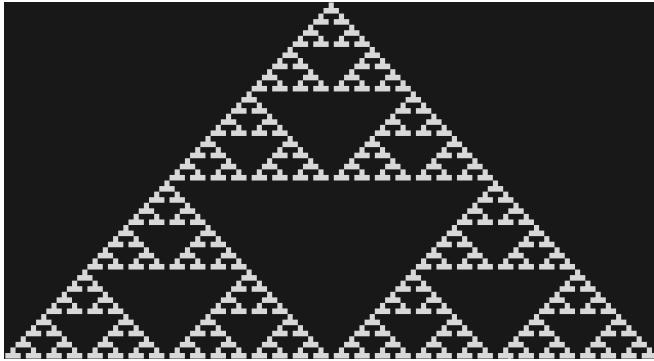


FIG. 2. Simulation of Wolfram rule 22. Consecutive snapshots from a 65 cell system, with initial conditions given by one living cell in the middle.

B. Conway's Game of Life

The Game of Life needs no introduction. We use the comonadic 2-dimensional array to encode encode the world, and a binary cell type. Local interactions are encoded using all 8 neighbours in 2D, with a cell surviving or being spawned if and only if it has two or three living neighbours.



FIG. 3. Simulation of Game of Life . Consecutive snapshots from a 27x27 system with random initial conditions (seed $s_1 = 4123$) and $p = 0.1$ of being alive. A ‘Traffic Light’ period-2 pattern is recognisable in the top-left corner.

C. Greenberg-Hasting excitable media

Another classic model, the excitable medium, introduces a cell type inspired by neural spiking and slime

moulds ([Greenberg and Hastings 1978](#)). The cell type has three possible states: Quiet, Spike, and Rest. Quiet cells will spike if there is a nearby spiking cell; after spiking, cells rest in a refractory state, during which they do not respond to stimulation; resting cells then return to the equilibrium quiet state.

In our simulation, we started with random initial configurations. If only spiking and quiet cells are included the grid quickly returns to the quiet equilibrium state. But when including at least one resting cell, travelling waves of activity emerge, with the system self-organising into periodic behaviour, which reminds of models of epilepsy ([Rabinovitch et al. 2024](#)) and atrial fibrillation ([Ciacci et al. 2020](#)).

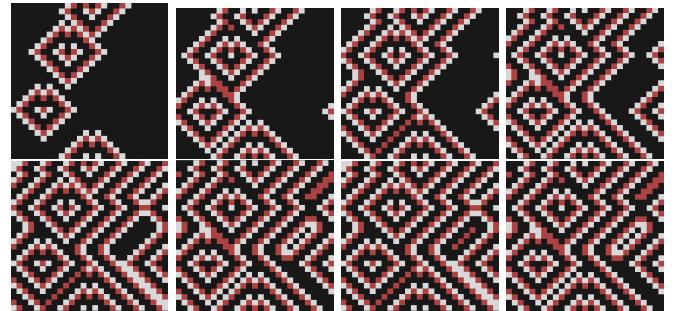


FIG. 4. Simulation of an excitable medium. Snapshots separated by $\Delta t = 2$ timesteps from a 27x27 system, using random initial conditions ($s = 37873$) with probability of 0.1 for cells to be spiking and 0.1 for cells to be resting. After some irregular behaviour for 19 timesteps, the system settles into a periodic configuration of spirals, which in this case shows period 3.

D. Forest Fire

The forest fire model is another famous CA which manifests self-organized criticality ([Bak et al. 1990](#)). The cells represent trees in a forest and, like excitable media, can take three states: Fire, Tree, and Empty. A tree ignites if there is a nearby burning tree and a burning cell becomes empty.

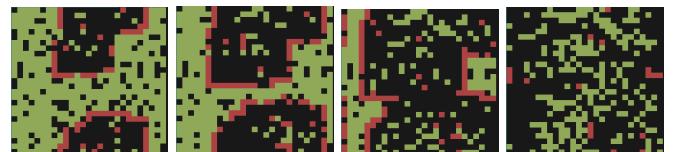


FIG. 5. Simulation of a stochastic Forest Fire model. Snapshots from a 27x27 system, separated by $\Delta t = 2$ timesteps, evolving randomly (seed $s_1 = 4123$) with firing rate $f = 0.0005$ and growing rate $p = 0.1$, and random initial conditions ($s_1 = 1978$) given by the same rates. The balance between the growth rate and firing rate maintains the model in the critical state, where neither all trees have died nor do they survive forever.

Unlike the CA presented so far, this model has a vital

stochastic component: a tree can spontaneously ignite with a small probability f , while an empty cell can grow a new tree with probability p . The ratio between f and p controls system behaviour; at certain parameter ranges, the model enters a critical state, as seen in the example simulation above.

V. FROM GRIDS TO NETWORKS

Not only CA, but complex systems in general require local interactions between components to manifest emergence. As such, the comonadic model can apply to much more complex simulations, as long as there is a distinct, fixed topology. One could show, by induction, that the comonadic array model in 1D and 2D can be easily expanded to arbitrary dimensions.

Moreover, it is tempting to extend the model to arbitrary topologies; while in the current implementation we only specified neighbourhoods as operations with indices on arrays, these could, alternatively, be expressed as neighbour lists on an underlying network. Such an implementation, alongside the support for environment variables and randomness, would allow modelling social networks or brain regions and emergent spreading phenomena like epidemics or information flow.

VI. CONCLUSION

We have presented an example comonadic implementation of cellular automata which, unlike existing implementations, uses an underlying array structure and monad and comonad transformers to add further environment variables and stochasticity in the comonadic evaluation.

This contribution is valuable as it offers an easily-parallelisable and extensible framework to simulate CA. Thanks to the Curry-Howard-Lambeck isomorphism, we may link this implementation to both a mathematical description of the model and its semantics.

This brings to mind Rine's categorical characterisation of automata, offering a means to identify what an arbitrary system can do given a categorical description of its input and output states (Rine 1971). In this way we can effectively provide a theory of the traits exhibited by a system. It will be interesting to consider the effect of this approach on theories of emergence and criteria for detecting emergence in general. In conjecture to recent descriptions of complex systems as computation structures (more specifically ϵ -machines) (Rosas et al. 2024), it is extremely timely to turn to categorically-theoretical descriptions of multi-agent systems and explore the isomorphism between semantics, mathematics, and simulations, towards a unifying mathematical theory of complex behaviour emerging by computation.

REFERENCES

- Bak, P., Chen, K., and Tang, C. (1990). A forest-fire model and some thoughts on turbulence. *Phys. Lett. A*, 147(5-6):297–300.
- Basold, H., Ford, C., and Pirée, L. (2025). An expressive coalgebraic modal logic for cellular automata. arXiv:2504.16735.
- Bedau, M. A. (1997). Weak emergence. *Noûs*, 31:375–399.
- Bhat, S. and Garay, F. (2019). A collection of Cellular Automata written in Haskell with Diagrams. github.com/bollu/cellularAutomata.
- Capobianco, S. and Uustalu, T. (2010). A categorical outlook on cellular automata. In *Proceedings of JAC 2010. Journées Automates Cellulaires*, page 88–99.
- Church, A. (1936). An Unsolvable Problem of Elementary Number Theory. *Am. J. Math.*, 58(2):345–363.
- Ciacci, A. et al. (2020). Understanding the transition from paroxysmal to persistent atrial fibrillation. *Phys. Rev. Res.*, 2(2).
- Colcombet, T., Petrişan, D., and Stabile, R. (2020). Learning automata and transducers: A categorical approach. arXiv:2010.13675.
- Greenberg, J. M. and Hastings, S. P. (1978). Spatial patterns for discrete models of diffusion in excitable media. *SIAM J. Appl. Math.*, 34(3):515–523.
- Hoare, W. P. (1980). The Correspondence Between Proofs and Programs. *Commun. ACM*, 23(7):417–422.
- Jensen, H. J. (2022). *Complexity Science*. Cambridge UP.
- Kmett, E. (2014). The Comonad Package. hackage.haskell.org.
- Kmett, E. (2015). Cellular automata. schoolofhaskell.com.
- Leinster, T. (2017). *Basic category theory*. Cambridge UP.
- Mitchell, J. C. (1990). Type systems for programming languages. In van Leeuwen, J., editor, *Formal Models and Semantics*, Handbook of Theoretical Computer Science, pages 365–458.
- Piponi, D. (2006). Evaluating cellular automata is comonadic. blog.sigfpe.com.
- Potts, P. R. (2006). From Bits to Cells: Simple Cellular Automata in Haskell, Part Two. praisecurseandrecuse.blogspot.com.
- Rabinovitch, A. et al. (2024). Ephaptic conduction in tonic-clonic seizures: A cellular automaton model. *Front. Neurol.*, 15:1477174.
- Rine, D. C. (1971). A categorical characterization of general automata. *Information and Control*, 19(1):30–40.
- Rosas, F. E. et al. (2024). Software in the natural world: A computational approach to emergence in complex multi-level systems. arXiv:2402.09090.
- Sas, M. I. and Sutherland, J. H. J. (2024). Hascell. github.com/mearlboro/hascell.
- Turing, A. M. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265.
- Uustalu, T. and Vene, V. (2008). Comonadic notions of computation. *Electron. Notes Theor. Comput. Sci.*, 203(5):263–284.
- van Heerdt, G. K. (2020). *The Categorical Automata Learning Framework*. PhD thesis, University College London.
- Widemann, B. T. (2012). Structural operational semantics for cellular automata. *Lec. Notes Comp. Sci.*, page 184–193.
- Widemann, B. T. and Hauhs, M. (2011). Distributive-law semantics for cellular automata and agent-based models. *Lec. Notes Comp. Sci.*, page 344–358.
- Wolfram, S. (1983). Statistical mechanics of cellular automata. *Rev. Mod. Phys.*, 55(3):601–644.
- Wolfram, S. (2019). *A New Kind of Science*. Champaign, IL Wolfram Media.