Toy Monads

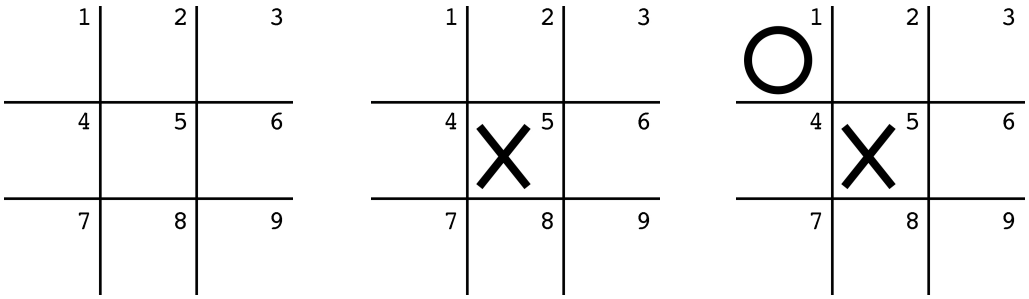
Eric Steinhart

ABSTRACT: Toy monads are little computations. Toy monads are useful for illustrating a variety of points about Leibnizian metaphysics. Two toy monads are developed here which play tic-tac-toe with each other. Each monad runs a program. Together, these two monads form a finite mechanical universe. Although they appear to causally interact, they are merely computationally harmonized. The different strategies for playing tic-tac-toe permit many possible universes in which these toy monads have counterparts.

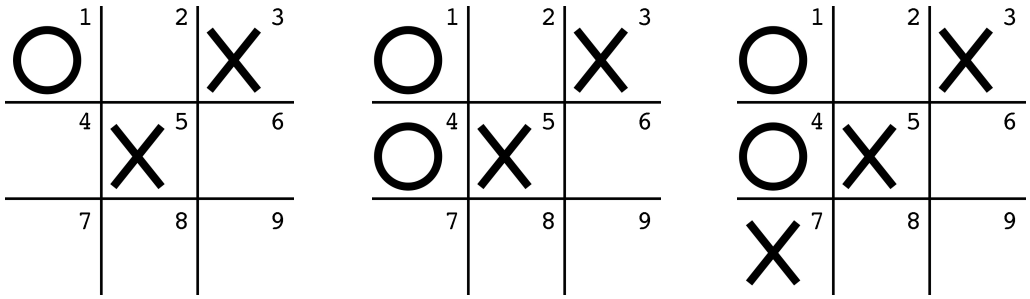
1. Tic-Tac-Toe

Toy monads are little computations. Tic-tac-toe is useful for illustrating some initial points about (toy) monads. Tic-tac-toe is played on a 3-by-3 grid. The squares of grid can be numbered from 1 through 9. Given such an enumeration, every tic-tac-toe game is equivalent to the cooperative production of a sequence of numbers.

Table 1 illustrates a game of tic-tac-toe. You can think of the game as played on a piece of paper – the players mark the squares with pencils. However, you can also think of the game as played *mentally*. Chess players sometimes play blindfold chess, in which they just mentally visualize the board. Tic-tac-toe players can do the same. At any time, each player has a *state*. The state is the player's mental image of the marks on the board. Each player sends some *output* to the other player – perhaps by saying "X moves to the center square". And, conversely, each player receives some *input* from the other player – just by listening to what the other player says. And each player follows a *strategy* – a system of rules that guides his or her moves. Tragically, the strategyof O is not very good.



The empty grid. X marks square 5. O marks square 1.



X marks square 3. O marks square 4. X marks 7 to win.

Table 1. A sample tic-tac-toe game.

2. Finite Programs

A *program* is a system of if-then rules that guides the activity of a machine. At this point we are discussing only finite programs. A finite program has only finitely many rules. A rule has this form:

if the machine gets some *input* while it is in its *current state*,
then it changes to a *new state* and produces an *output*.

Any machine that plays tic-tac-toe runs a program for playing tic-tac-toe. The program is its strategy. The rules must cover *all possible cases*. Otherwise, the machine might get stuck in some situation where it can't play at all. To define the rules, we have to first define the possible inputs, possible states, and possible outputs.

We start with the inputs. Any input indicates a mark (either X or O) placed on some square on the board. Since the squares are numbered, the inputs are just the numbers 1 through 9. There is also null input – the null input is used to start and end the game. This null input is the number 0. So the set of inputs is the set of numbers 0 through 9. Outputs are the same as inputs – the numbers 0 through 9.

What about the states? At any moment of time, the state of the game is the result of all the moves made on the board. Since every move marks a square, and the squares are numbered, each state is a series of numbers. The odd positions in the series are the moves by X while the even positions are the moves by O. The null mark 0 indicates that the board is blank (it can be ignored after the game is started). Table 2 shows the states corresponding to the moves in the game in Table 1.

Move	**State of the Game**
The empty grid.	0
X marks square 5	5

O marks square 1	51
X marks square 3	513
O marks square 4	5134
X marks square 7	51347

Table 2. States of tic-tac-toe are sequences of numbers.

Of course, in any real game of tic-tac-toe, somebody will win pretty quickly – often after only 5 moves. But, since we're covering all possibilities, we have to be *complete* – the set of states includes *every possible way* to mark the tic-tac-tmmmmoe board. So games can continue until all 9 squares are marked. The longest state is a series of 9 numbers. We'll gather all the states into the set S. Abstractly speaking, S is just the set of sequences of numbers that represent legal sequences of moves in tic-tac-toe.

How many states are there? There are 9 ways to mark an empty board with an X; then 8 ways to mark that board with an O; then 7 ways to mark with an X; and so on. So there are 9! possible states of the grid. And 9! = 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 = 362880. Adding the start state, that's 362881 possible states.

We're ready to look at some if-then rules in the tic-tac-toe program. These rules guide the behavior of any machine running the tic-tac-toe playing program. They tell the machine how to move. Every tic-tac-toe program has to be complete – it must have a rule for every possible input and state. Since there are 10 inputs, and 362881 states, that makes well over three million rules. To save some space, we'll only present two rules. The rules are taken directly from the game of tic-tac-toe in Table 1:

- Rule 1 says: "If you get the null input while your grid is blank, then change your state to a grid with an X in the center, and produce an X in the center as output." This rule is represented arithmetically as "If your input is 0 while you are in state 0, then change your state to 5 and your output to 5."

- Rule 2 says: "If you get an X in the center while your grid is blank, then change your state to a grid with an X in the center and an O in the upper left corner, and produce an O in the upper left corner as output." Arithmetically, "If your input is 5 while your state is 0, then change your state to 51 and your output to 1."

Every rule associates an (input, current state) pair with a (next state, output) pair. It says: if your present situation is (input, current state), then change to (next state, output). It's useful to break each if-then rule into two simpler associations. Technically, each association is a *function*. So we need to talk about two functions.

The first association is the *transition function*. This function is F. The transition function F associates every possible (input, current state) pair with some next state. It can be expressed as a table or spreadsheet with three columns. The first column is the input; the

second column is the current state; the third column is the next state. Each row thus has the form

(input i, current state s) → next state.

Thus F associates (input i, current state s) with the next state. The application of F to the pair (input i, current state s) is written F(i, s). Since there are over 3 million rules, there are over 3 million rows in this spreadsheet.

The second association is the *output function*. This function is G. The output function G associates every possible (input i, current state s) pair with an output. The output is G(i, s). It can be displayed just by adding a fourth column to our spreadsheet.

The spreadsheet with these four columns is a program. Table 3 shows part of the spreadsheet for player X in Table 1. Let's refer to this program as *Alpha*. Table 3 only shows the rules used by player X in the game in Table 1.

Input i	State s	Next State F(i, s)	Next Output G(i, s)
0	0	5	5
1	5	513	3
4	513	51347	7

Table 3. Part of the program Alpha for playing tic-tac-toe.

All games of tic-tac-toe share the same set of inputs, states, and outputs. So, for any tic-tac-toe playing programs, the sets I, S, and O are the same. From program to program, all that can vary are the functions F and G. So there are many different possible tic-tac-toe playing programs. The functions F and G encode the *strategy* of the tic-tac-toe program. Some versions of F and G are very bad tic-tac-toe playing strategies. Other versions are very good tic-tac-toe strategies. Better strategies win more frequently.

Our analysis shows that every tic-tac-toe program has five components: the list of possible inputs I, the list of possible states S, the list of possible outputs O, the transition function F, and the output function G. These five components are grouped into an ordered whole by listing them in parentheses like this: (I, S, O, F, G). Although we can write down any program as a list of rules, programs don't have to be written down. Abstractly speaking, programs are mathematical objects. They involve only numbers. More formally, a program is a quintuple (I, S, O, F, G), where I is a set of possible inputs; S is a set of possible states; O is a set of possible outputs; F associates each (input, state) pair with a state ; and G associates each (input, state) pair with an output.

3. Finite Machines

A *machine* is an object whose behavior is guided by a program. The *nature* of a machine is defined by its program. The program is the *essence* of the machine. It defines all the possible conditions and dispositions of the machine. So far, we've only been talking about finite programs; consequently, we're only talking about finite machines. They're also known as *finite state machines* (FSMs). Not all machines are finite machines – a machine can have infinitely many states, inputs, outputs, and rules.

To see how a program defines the nature or essence of a machine, let's look at its parts. We said that a program is a quintuple (I, S, O, F, G). The set of possible inputs I includes all the possible ways the environment can act on the machine. Each possible input is some way the machine can receive forces, energies, or other physical signals from other machines. The set of possible states S includes all the possible conditions of the machine. Each possible state encodes all the physical properties of the machine. For each physical property, the state encodes one of the possible values of that property. The set of possible outputs O includes all the possible ways the machine can act on its environment. Each possible output is some way the machine can send forces, energies, or other physical signals to other machines. The functions F and G encode all the dispositions of the machine. These are all the if-then rules that express the behavior of the machine. The dispositions express all the ways the machine can causally interact with other machines.

At every moment, every machine has four properties. Since every machine runs a program, its first property is its program p. Since every machine gets some input from its environment, its second property is its input i. Since every machine is in some state, its third property is its state s. Since every machine sends some output to its environment, its fourth property is its output o. You might want to say that a machine is a *substance* in which these four properties inhere in some way. However, it is simpler to just identify a machine with the bundle of its properties. What is a bundle? Let's formalize the notion of a bundle by identifying it with an ordered list. Hence a machine is a quadruple:

(program p, input i, state s, output o).

As time goes by, machines change. Whenever a machine changes, it changes into a new machine with new properties. This means that every machine exists only for an instant. It immediately changes into or becomes another machine. Machines are not temporally extended – they do not persist through time. However, monads are temporally extended. Eery monad persists. Monads are composed of machines like books are composed of pages or movies are composed of photographs. Every monad is a temporally ordered series of machines. Conversely, monads are stages of monads.

To see this, let's consider a monad that plays tic-tac-toe. This monad is *Xenophon*. Xenophon is a temporally extended series of machines (its stages). At any moment, all that exists of Xenophon is a stage. The initial stage of Xenophon is Xenophon-0. This stage contains the program Alpha (partly shown in Table 3). This stage is a quadruple:

(program Alpha, initial input i_0, initial state s_0, initial output o_0).

The stage Xenophon-0 changes into the next stage Xenophon-1. As time goes by, Xenophon-1 changes into Xenophon-2. The program of Xenophon always remains the same – it's the essence of Xenophon. Only the inputs, states, and outputs change.

4. Finite Mechanical Universes

A machine cannot exist by itself. A machine gets inputs from its environment and gives outputs to its environment. At the very least, the environment of a machine is another machine. Those two machines interact: the output of each is the input to the other. You can think of these two machines as sending signals to one another. They form a simple *network*. A network is a system of causally interacting machines. It is analogous to a system of interacting computers like the Internet. More formally, a network is a set of machines in which every machine interacts with every other machine.

A network is *closed* iff every machine in that network gets input from and gives output to another machine in that movie. A network is *open* otherwise. A *finitary universe* is a closed finite network of finite machines (FSMs). The simplest finitary universe is just two FSMs interacting with one another. One example of a finitary universe involves two tic-tac-toe monads playing tic-tac-toe. Suppose Xenophon is playing with Odessa. These two monads are interacting with and only with each other. They're playing tic-tac-toe – but there's no external tic-tac-toe grid. They each encode the grid in their states. And there's no space that contains them. They aren't like little planets sitting in some bigger space. On the contrary, they define the *entire content* of the universe.

Each moment of the universe consists of a stage for each monad. So the universe is a series of pairs of stages: (Xenophon-1, Odessa-1), (Xenophon-2, Odessa-2), and so on. All these two machines do is play tic-tac-toe – they *love* it, that's what they do. Each move in tic-tac-toe requires each player to do four things:

1. Apply your strategy to change your state and produce your output.
2. Send your output to the other player.
3. Wait while other player applies his or her strategy.
4. Get your input from the other player.

As moves are made, the roles of the players alternate. While one player applies his or her strategy, the other player waits. And while one player sends its output, the other player receives its input. Table 4 illustrates two moves from Table 1. Figure 1 shows the first part of the game in Table 1. Figure 2 shows the second part. Stages 13 to 16 are just resetting the game. As soon as one game is over, they switch X/O roles and they play another game. They'll continue forever into the future. For the sake of illustration, we started Xenophon and Odessa with initial stages. But really, they've been playing forever into the past. Note that stage 16 in Figure 2 is identical to stage 0 in Figure 1. The cycle just repeats.

Time	Xenophon	Odessa
0	Apply strategy	Wait
1	Send output	Get input
2	Wait	Apply strategy
3	Get input	Send output
4	Apply strategy	Wait
5	Send output	Get input
6	Wait	Apply strategy
7	Get input	Send output

Table 4. Two moves in the game in Table 1.

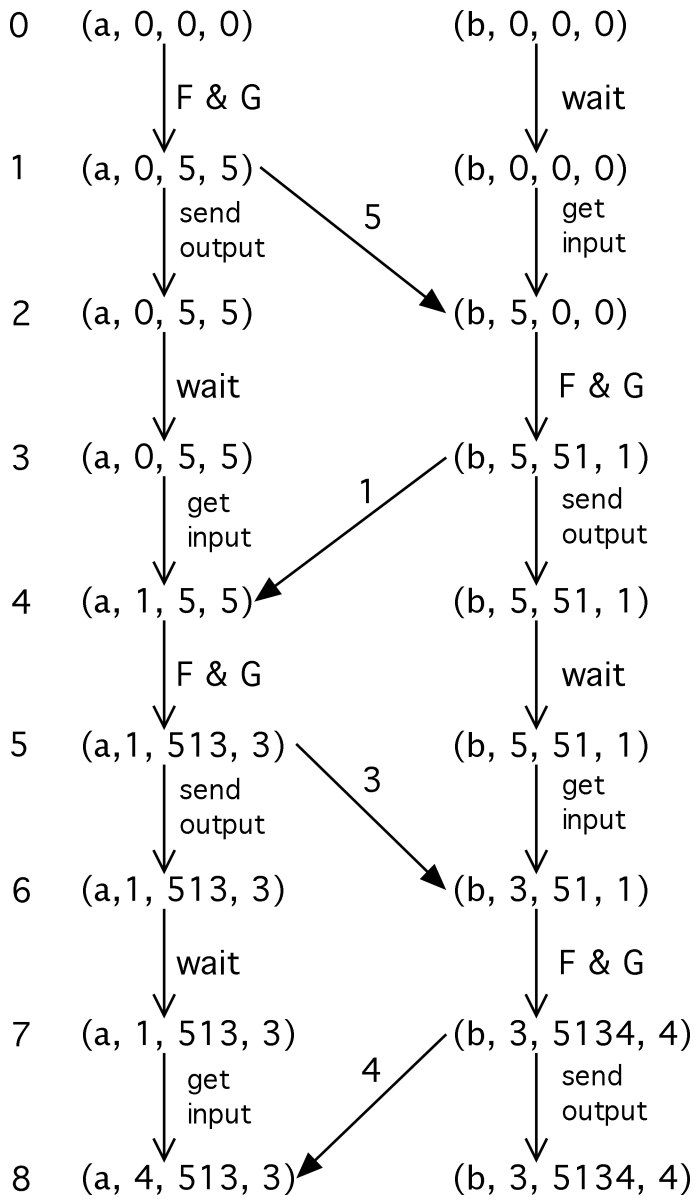


Figure 1. The first part of the game from Table 1.

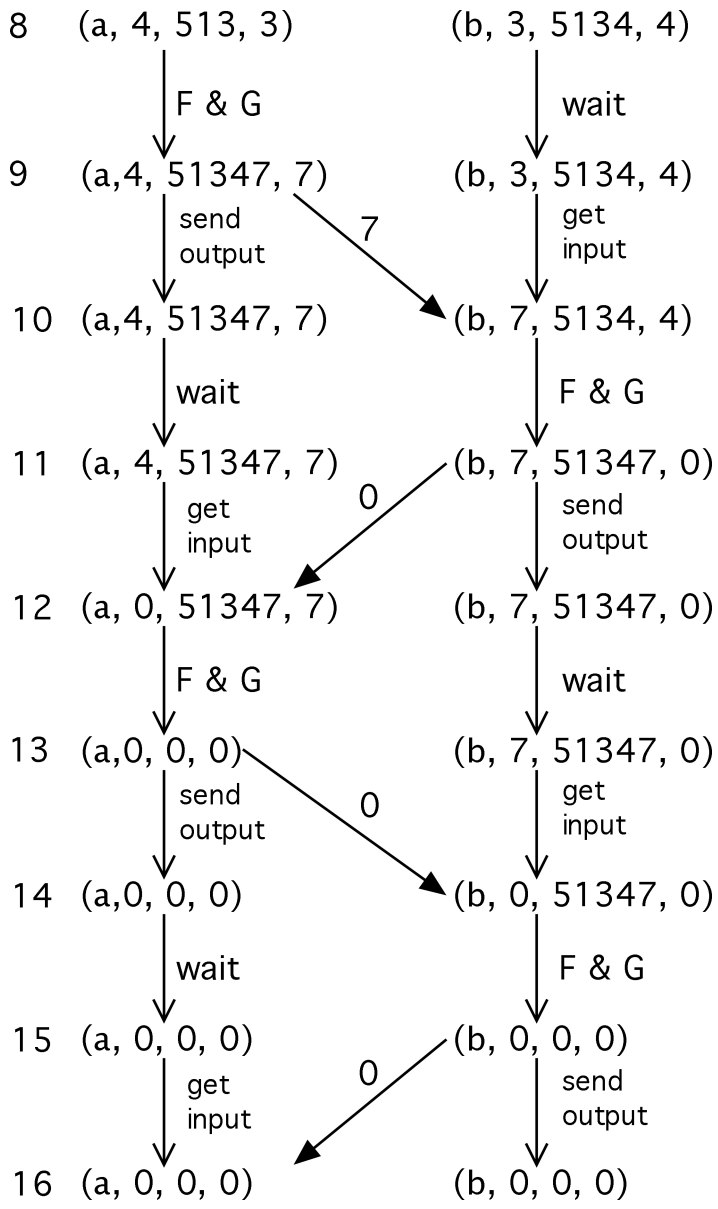


Figure 2. The second part of the game from Table 1.

5. The Independence of Monads

Figures 1 and 2 show the interactions of the players Xenophon and Odessa. The one sends its output to the other and the other receives its input from the one. They appear to exchange signals. This exchange seems to be causal: when Xenophon sends an output to Odessa, it seems like Xenophon is causing an effect in Odessa. But one of the stranger parts of the *Monadology* is that monads don't causally interact. They are merely synchronized, coordinated, or harmonized. Table 5 shows the stages of Xenophon. And Table 6 shows the parallel stages of Odessa. These two monads are coordinated – their stages can be aligned so that it looks as if they are interacting – it looks as if each is causing effects in the other. However, that appearance is an illusion. The two monads are merely synchronized, coordinated, or harmonized.

Time	Stage
0	(a, 0, 0, 0)
1	(a, 0, 5, 5)
2	(a, 0, 5, 5)
3	(a, 0, 5, 5)
4	(a, 1, 5, 5)
5	(a, 1, 513, 3)
6	(a, 1, 513, 3)
7	(a, 1, 513, 3)
8	(a, 4, 513, 3)
9	(a, 4, 51347, 7)
10	(a, 4, 51347, 7)
11	(a, 4, 51347, 7)
12	(a, 0, 51347, 7)
13	(a, 0, 0, 0)
14	(a, 0, 0, 0)
15	(a, 0, 0, 0)

Table 5. Xenophon by herself.

Time	Stage
0	(b, 0, 0, 0)
1	(b, 0, 0, 0)
2	(b, 5, 0, 0)
3	(b, 5, 51, 1)
4	(b, 5, 51, 1)
5	(b, 5, 51, 1)
6	(b, 3, 51, 1)
7	(b, 3, 5134, 4)
8	(b, 3, 5134, 4)
9	(b, 3, 5134, 4)
10	(b, 7, 5134, 4)
11	(b, 7, 51347, 0)
12	(b, 7, 51347, 0)
13	(b, 7, 51347, 0)
14	(b, 0, 51347, 0)
15	(b, 0, 0, 0)

Table 6. Odessa by himself.

6. On the Plurality of Mechanical Universes

One aspect of the Xenophon-Odessa universe is that it is a universe of eternal recurrence. Since their programs do not vary, these two players always behave the same way. Xenophon, for instance, will always open by marking the center square with an X. And Odessa will always respond with an O to the upper right corner. When Xenophon opens, it will always be in exactly one way. And Odessa will reply in exactly one way. Hence there are two games that are alternately repeated endlessly into the past and future. That's hideously dull. This single universe does not exhaust the tic-tac-toe playing abilities of either Xenophon or Odessa. Fortunately, there's no reason to say that this is the only universe. Other tic-tac-toe universes are possible.

There are many different programs for playing tic-tac-toe. Hence there are many possible partners for any player. Focus on Xenophon. In universe-0, Xenophon plays with Odessa. But in alternative universe-1, the counterpart of Xenophon plays with Orville. Orville does not play like Odessa plays. So the history of universe-1 is different from the history of universe-0. In alternate universe-2, the counterpart of Xenophon plays with yet another player with yet another strategy. And, in every possible universe, some counterpart of Xenophon plays with some player with some other strategy. Across all these many possible universes, all the possible ways that Xenophon can play are realized.

By scanning across all possible universes that contain counterparts of Xenophon, we can form the *game tree* for Xenophon. It contains all the possible histories of Xenophon. The root of this game tree is the initial blank tic-tac-toe board. Since Xenophon opens the same way in every universe, this blank board is necessarily followed by a board with an X in the center. But Xenophon will encounter partners who respond in all possible ways. Since we can treat symmetrical cases the same, there are two types of responses: an O in a corner or an O on the side. Xenophon responds in exactly one characteristic (that is, essential) way to each of those O moves – and thus Xenophon marks a square with an X. Now there are quite a few possible moves for the O player. The game tree unfolds. Figure 1 illustrates part of the game tree for Xenophon. It is a formal structure abstracted from a plurality of possible universes. It encodes all the possible histories of Xenophon. Some of these turn out well (she wins), some turn out indifferently (she draws), while others turn out badly (she loses). Thus these histories can be ranked in terms of value. Some are better or worse than others.

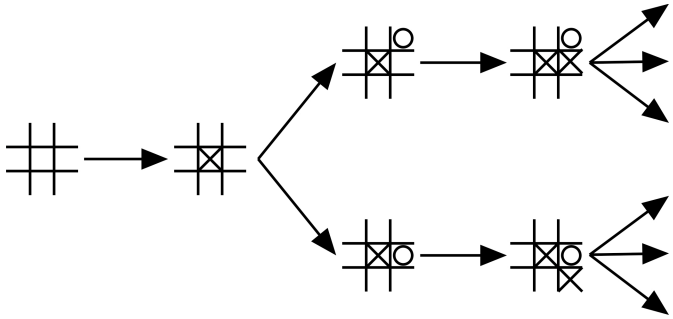


Figure 1. Part of the game tree for Xenophon.