1
Motivation

This chapter tries to explain why coalgebras are interesting structures in mathematics and computer science. It does so via several examples. The notation used for these examples will be explained informally, as we proceed. The emphasis at this stage is not so much on precision in explanation but on transfer of ideas and intuitions. Therefore, for the time being we define a coalgebra – very informally – to be a function of the form

$$S \xrightarrow{\ c\ } \boxed{\cdots \quad S \quad \cdots} \,. \tag{1.1}$$

What we mean is: a coalgebra is given by a set S and a function c with S as domain and with a 'structured' codomain (result, output, the box $\boxed{\cdots}$), in which the domain S may occur again. The precise general form of these codomain boxes is not of immediate concern.

Some terminology: We often call S the *state space* or *set of states* and say that the coalgebra *acts on* S. The function c is sometimes called the *transition function* or *transition structure*. The idea that will be developed is that coalgebras describe general 'state-based systems' provided with 'dynamics' given by the function c. For a state $x \in S$, the result $c(x)$ tells us what the successor states of x are, if any. The codomain $\boxed{\cdots}$ is often called the *type* or *interface* of the coalgebra. Later we shall see that it is a *functor*.

A simple example of a coalgebra is the function

$$\mathbb{Z} \xrightarrow{\ n \mapsto (n-1, n+1)\ } \mathbb{Z} \times \mathbb{Z}$$

with state space $\mathbb{Z}$ occurring twice on the right-hand side. Thus the box or type of this coalgebra is: $\boxed{(-) \times (-)}$. The transition function $n \mapsto (n-1, n+1)$

may also be written using λ-notation as $\lambda n.\,(n-1, n+1)$ or as $\lambda n \in \mathbb{Z}.\,(n-1, n+1)$.

Another example of a coalgebra, this time with state space the set $A^{\mathbb{N}}$ of functions from $\mathbb{N}$ to some given set A, is

$$A^{\mathbb{N}} \xrightarrow{\sigma \mapsto (\sigma(0),\, \lambda n.\, \sigma(n+1))} A \times A^{\mathbb{N}}.$$

In this case the box is $\boxed{A \times (-)}$. If we write σ as an infinite sequence $(\sigma_n)_{n\in\mathbb{N}}$ we may write this coalgebra as a pair of functions $\langle \mathsf{head}, \mathsf{tail} \rangle$ where

$$\mathsf{head}((\sigma_n)_{n\in\mathbb{N}}) = \sigma_0 \quad \text{and} \quad \mathsf{tail}((\sigma_n)_{n\in\mathbb{N}}) = (\sigma_{n+1})_{n\in\mathbb{N}}.$$

Many more examples of coalgebras will occur throughout this text.

This chapter is devoted to 'selling' and 'promoting' coalgebras. It does so by focusing on the following topics.

1. A representation as a coalgebra (1.1) is often very natural, from the perspective of state-based computation.
2. There are powerful 'coinductive' definition and proof principles for coalgebras.
3. There is a very natural (and general) temporal logic associated with coalgebras.
4. The coalgebraic notions are on a suitable level of abstraction, so that they can be recognised and used in various settings.

Full appreciation of this last point requires some familiarity with basic category theory. It will be provided in Section 1.4.

Remark 1.0.1 Readers with a mathematical background may be familiar with the notion of coalgebra as comonoid in vector spaces, dual to an algebra as a monoid. In that case one has a 'counit' map $V \to K$, from the carrier space V to the underlying field K, together with a 'comultiplication' $V \to V \otimes V$. These two maps can be combined into a single map $V \to K \times (V \otimes V)$ of the form (1.1), forming a coalgebra in the present sense. The notion of coalgebra used here is thus much more general than the purely mathematical one.

1.1 Naturalness of Coalgebraic Representations

We turn first to an area where coalgebraic representations as in (1.1) occur naturally and may be useful, namely programming languages – used for writing computer programs. What are programs, and what do they do? Well,

programs are lists of instructions telling a computer what to do. Fair enough. But what are programs from a mathematical point of view? Put differently, what do programs mean?[1] One view is that programs are certain functions that take an input and use it to compute a certain result. This view does not cover all programs: certain programs, often called processes, are meant to be running forever, like operating systems, without really producing a result. But we shall follow the view of programs as functions for now. The programs we have in mind work not only on input, but also on what is usually called a state, for example for storing intermediate results. The effect of a program on a state is not immediately visible and is therefore often called the *side effect* of the program. One may think of the state as given by the contents of the memory in the computer that is executing the program. This is not directly observable.

Our programs should thus be able to modify a state, typically via an assignment like `i = 5` in a so-called imperative programming language. Such an assignment statement is interpreted as a function that turns a state x into a new, successor state x' in which the value of the identifier `i` is equal to `5`. Statements in such languages are thus described via suitable 'state transformer' functions. In simplest form, ignoring input and output, they map a state to a successor state, as in

$$S \xrightarrow{\text{stat}} S, \tag{1.2}$$

where we have written S for the set of states. Its precise structure is not relevant. Often the set S of states is considered to be a 'black box' to which we do not have direct access, so that we can observe only certain aspects. For instance, the function $i\colon S \to \mathbb{Z}$ representing the above integer `i` allows us to observe the value of i. The value $i(x')$ should be 5 in the result state x' after evaluating the assignment `i = 5`, considered as a function $S \to S$, as in (1.2).

This description of statements as functions $S \to S$ is fine as first approximation, but one quickly realises that statements do not always terminate normally and produce a successor state. Sometimes they can 'hang' and continue to compute without ever producing a successor state. This typically happens because of an infinite loop, for example in a `while` statement or because of a recursive call without exit.

There are two obvious ways to incorporate such non-termination.

1. **Adjust the state space**. In this case one extends the state space S to a space $S_\perp \stackrel{\text{def}}{=} \{\perp\} \cup S$, where $\perp$ is a new 'bottom' element not occurring in

[1] This question comes up frequently when confronted with two programs – one possibly as a transformation from the other – which perform the same task in a different manner and which could thus be seen as the same program. But how can one make precise that they are the same?

S that is especially used to signal non-termination. Statements then become functions:

$$S_\perp \xrightarrow{\mathsf{stat}} S_\perp \quad \text{with the requirement} \quad \mathsf{stat}(\perp) = \perp.$$

The side-condition expresses the idea that once a statement hangs it will continue to hang.

The disadvantage of this approach is that the state space becomes more complicated and that we have to make sure that all statements satisfy the side-condition, namely that they preserve the bottom element $\perp$. But the advantage is that composition of statements is just function composition.

2. **Adjust the codomain**. The second approach keeps the state space S as it is but adapts the codomain of statements, as in

$$S \xrightarrow{\mathsf{stat}} S_\perp \quad \text{where, recall,} \quad S_\perp = \{\perp\} \cup S.$$

In this representation we easily see that in each state $x \in S$ the statement can either hang, when $\mathsf{stat}(x) = \perp$, or terminate normally, namely when $\mathsf{stat}(x) = x'$ for some successor state $x' \in S$. What is good is that there are no side-conditions anymore. But composition of statements cannot be defined via function composition, because the types do not match. Thus the types force us to deal explicitly with the propagation of non-termination: for these kind of statements $s_1, s_2 \colon S \to S_\perp$ the composition $s_1 ; s_2$, as a function $S \to S_\perp$, is defined via a case distinction (or pattern match) as

$$s_1 ; s_2 = \lambda x \in S. \begin{cases} \perp & \text{if } s_1(x) = \perp \\ s_2(x') & \text{if } s_1(x) = x'. \end{cases}$$

This definition is more difficult than function composition (as used in point 1 above), but it explicitly deals with the case distinction that is of interest, namely between non-termination and normal termination. Hence being forced to make these distinctions explicitly is maybe not so bad at all.

We push these same ideas a bit further. In many programming languages (such as Java [43]) programs may not only hang, but also terminate 'abruptly' because of an exception. An exception arises when some constraint is violated, such as a division by zero or an access `a[i]` in an array `a` which is a null-reference. Abrupt termination is fundamentally different from non-termination: non-termination is definitive and irrevocable, whereas a program can recover from abrupt termination via a suitable exception handler that restores normal termination. In Java this is done via a `try-catch` statement; see for instance [43, 173, 240].

Let us write E for the set of exceptions that can be thrown. Then there are again two obvious representations of statements that can terminate normally or abruptly or can hang.

1. **Adjust the state space**. Statements then remain endofunctions[2] on an extended state space:

$$\big(\{\bot\} \cup S \cup (S \times E)\big) \xrightarrow{\quad\mathsf{stat}\quad} \big(\{\bot\} \cup S \cup (S \times E)\big).$$

   The entire state space clearly becomes complicated now. But also the side-conditions are becoming non-trivial: we still want $\mathsf{stat}(\bot) = \bot$, and also $\mathsf{stat}(x, e) = (x, e)$, for $x \in S$ and $e \in E$, but the latter only for non-catch statements. Keeping track of such side-conditions may easily lead to mistakes. But on the positive side, composition of statements is still function composition in this representation.
2. **Adjust the codomain**. The alternative approach is again to keep the state space S as it is, but to adapt the codomain type of statements, namely as

$$S \xrightarrow{\quad\mathsf{stat}\quad} \big(\{\bot\} \cup S \cup (S \times E)\big). \tag{1.3}$$

   Now we do not have side-conditions and we can clearly distinguish the three possible termination modes of statements. This structured output type in fact forces us to make these distinctions in the definition of the composition $s_1 \,;\, s_2$ of two such statements $s_1, s_2 \colon S \to \{\bot\} \cup S \cup (S \times E)$, as in

$$s_1 \,;\, s_2 \;=\; \lambda x \in S.\, \begin{cases} \bot & \text{if } s_1(x) = \bot \\ s_2(x') & \text{if } s_1(x) = x' \\ (x', e) & \text{if } s_1(x) = (x', e). \end{cases}$$

   Thus, if s_1 hangs or terminates abruptly, then the subsequent statement s_2 is not executed. This is very clear in this second *coalgebraic* representation.

   When such a coalgebraic representation is formalised within the typed language of a theorem prover (as in [265]), the type checker of the theorem prover will make sure that appropriate case distinctions are made, according to the output type as in (1.3). See also [240] where Java's exception mechanism is described via such case distinctions, closely following the official language definition [173].

These examples illustrate that coalgebras as functions with structured codomains $\boxed{\cdots}$, as in (1.1), arise naturally and that the structure of the

[2] An endofunction is a function $A \to A$ from a set A to itself.

codomain indicates the kind of computations that can be performed. This idea will be developed further and applied to various forms of computation. For instance, non-deterministic statements may be represented via the powerset $\mathcal{P}$ as coalgebraic state transformers $S \to \mathcal{P}(S)$ with multiple result states. But there are many more such examples, involving for instance probability distributions on states.

(Readers familiar with computational monads [357] may recognise similarities. Indeed, in a computational setting there is a close connection between coalgebraic and monadic representations. Briefly, the monad introduces the computational structure, like composition and extension, whereas the coalgebraic view leads to an appropriate program logic. This is elaborated for Java in [264].)

Exercises

1.1.1 1. Prove that the composition operation ; as defined for coalgebras $S \to \{\bot\} \cup S$ is associative, i.e. satisfies $s_1 ; (s_2 ; s_3) = (s_1 ; s_2) ; s_3$, for all statements $s_1, s_2, s_3 \colon S \to \{\bot\} \cup S$.

Define a statement $\mathsf{skip} \colon S \to \{\bot\} \cup S$ which is a unit for composition ; i.e. which satisfies $(\mathsf{skip} ; s) = s = (s ; \mathsf{skip})$, for all $s \colon S \to \{\bot\} \cup S$.

2. Do the same for ; defined on coalgebras $S \to \{\bot\} \cup S \cup (S \times E)$. (In both cases, statements with an associative composition operation and a unit element form a monoid.)

1.1.2 Define also a composition monoid $(\mathsf{skip}, ;)$ for coalgebras $S \to \mathcal{P}(S)$.

1.2 The Power of the Coinduction

In this section we shall look at sequences – or lists or words, as they are also called. Sequences are basic data structures, both in mathematics and in computer science. One can distinguish finite sequences $\langle a_1, \ldots, a_n \rangle$ and infinite $\langle a_1, a_2, \ldots \rangle$ ones. The mathematical theory of finite sequences is well understood and a fundamental part of computer science, used in many programs (notably in the language Lisp). Definition and reasoning with finite lists is commonly done with induction. As we shall see, infinite lists require *coinduction*. Infinite sequences can arise in computing as the observable outcomes of a program that runs forever. Also, in functional programming, they can occur as so-called lazy lists, as in the languages Haskell [71] or Clean [381]. Modern extensions of logical programming languages have support for infinite

sequences [431, 189]. (Logic programming languages themselves can also be described coalgebraically; see [78, 107, 305, 306, 80, 81].)

In the remainder of this section we shall use an arbitrary but fixed set A and wish to look at both finite $\langle a_1, \ldots, a_n \rangle$ and infinite $\langle a_1, a_2, \ldots \rangle$ sequences of elements a_i of A. The set A may be understood as a parameter, and our sequences are thus parametrised by A, or, put differently, are polymorphic in A.

We shall develop a slightly unusual and abstract perspective on sequences. It treats sequences not as completely given at once but as arising in a local, step-by-step manner. This coalgebraic approach relies on the following basic fact. It turns out that the set of both finite and infinite sequences enjoys a certain 'universal' property, namely that it is a *final* coalgebra (of suitable type). We shall explain what this means and how this special property can be exploited to define various operations on sequences and to prove properties about them. A special feature of this universality of the final coalgebra of sequences is that it avoids making the (global) distinction between finiteness and infiniteness for sequences.

First some notation. We write $A^\star$ for the set of *finite* sequences $\langle a_1, \ldots, a_n \rangle$ (or lists or words) of elements $a_i \in A$, and $A^{\mathbb{N}}$ for the set of infinite ones: $\langle a_1, a_2, \ldots \rangle$. The latter may also be described as functions $a_{(-)} \colon \mathbb{N} \to A$, which explains the exponent notation in $A^{\mathbb{N}}$. Sometimes, the infinite sequences in $A^{\mathbb{N}}$ are called *streams*. Finally, the set of both finite and infinite sequences A^∞ is then the (disjoint) union $A^\star \cup A^{\mathbb{N}}$.

The set of sequences A^∞ carries a coalgebra or transition structure, which we simply call next. It tries to decompose a sequence into its head and tail, if any. Hence one may understand next as a partial function. But we describe it as a total function which possibly outputs a special element $\perp$ for undefined:

$$\begin{array}{rcl} A^\infty & \xrightarrow{\qquad\qquad \mathsf{next} \qquad\qquad} & \{\perp\} \cup (A \times A^\infty) \\ \sigma & \longmapsto & \begin{cases} \perp & \text{if } \sigma \text{ is the empty sequence } \langle\rangle \\ (a, \sigma') & \text{if } \sigma = a \cdot \sigma' \text{ with head } a \in A \text{ and tail } \sigma' \in A^\infty. \end{cases} \end{array} \tag{1.4}$$

The type of the coalgebra is thus $\boxed{\{\perp\} \cup (A \times (-))}$, as in (1.1), with A^∞ as state space that is plugged in the hole $(-)$ in the box. The successor of a state $\sigma \in A^\infty$, if any, is its tail sequence, obtained by removing the head.

The function next captures the external view on sequences: it tells what can be *observed* about a sequence σ, namely whether or not it is empty, and if not, what its head is. By repeated application of the function next all observable elements of the sequence appear. This 'observational' approach is fundamental in coalgebra.

A first point to note is that this function next is an isomorphism: its inverse next^{-1} sends $\bot$ to the empty sequence $\langle\rangle$ and a pair $(a, \tau) \in A \times A^\infty$ to the sequence $a \cdot \tau$ obtained by prefixing a to τ.

The following result describes a crucial 'finality' property of sequences that can be used to characterise the set A^∞. Indeed, as we shall see later in Lemma 2.3.3, final coalgebras are unique, up-to-isomorphism.

Proposition 1.2.1 (Finality of sequences) *The coalgebra* $\mathsf{next}\colon A^\infty \to \{\bot\} \cup A \times A^\infty$ *from (1.4) is final among coalgebras of this type: for an arbitrary coalgebra* $c\colon S \to \{\bot\} \cup (A \times S)$ *on a set* S *there is a unique 'behaviour' function* $\mathrm{beh}_c\colon S \to A^\infty$ *which is a homomorphism of coalgebras. That is, for each* $x \in S$*, both*

- *if* $c(x) = \bot$*, then* $\mathsf{next}(\mathrm{beh}_c(x)) = \bot$.
- *if* $c(x) = (a, x')$*, then* $\mathsf{next}(\mathrm{beh}_c(x)) = (a, \mathrm{beh}_c(x'))$.

Both these two points can be combined in a commuting diagram, namely as

$$\begin{array}{ccc} \{\bot\} \cup (A \times S) & \xrightarrow{\ \mathrm{id} \cup (\mathrm{id} \times \mathrm{beh}_c)\ } & \{\bot\} \cup (A \times A^\infty) \\ {\scriptstyle c}\uparrow & & \cong \uparrow{\scriptstyle \mathsf{next}} \\ S & \xrightarrow[\ \mathrm{beh}_c\ ]{} & A^\infty \end{array}$$

where the function $\mathrm{id} \cup (\mathrm{id} \times \mathrm{beh}_c)$ *on top maps* $\bot$ *to* $\bot$ *and* (a, x) *to* $(a, \mathrm{beh}_c(x))$.

In general, we shall write dashed arrows, like above, for maps that are uniquely determined. In the course of this chapter we shall see that a general notion of homomorphism between coalgebras (of the same type) can be defined by such commuting diagrams.

Proof The idea is to obtain the required behaviour function $\mathrm{beh}_c\colon S \to A^\infty$ via repeated application of the given coalgebra c as follows.

$$\mathrm{beh}_c(x) = \begin{cases} \langle\rangle & \text{if } c(x) = \bot \\ \langle a \rangle & \text{if } c(x) = (a, x') \wedge c(x') = \bot \\ \langle a, a' \rangle & \text{if } c(x) = (a, x') \wedge c(x') = (a', x'') \wedge c(x'') = \bot \\ \vdots & \end{cases}$$

Doing this formally requires some care. We define for $n \in \mathbb{N}$ an iterated version $c^n\colon S \to \{\bot\} \cup A \times S$ of c as

$$\begin{array}{rcl} c^0(x) & = & c(x) \\ c^{n+1}(x) & = & \begin{cases} \bot & \text{if } c^n(x) = \bot \\ c(y) & \text{if } c^n(x) = (a, y). \end{cases} \end{array}$$

Obviously, $c^n(x) \neq \bot$ implies $c^m(x) \neq \bot$, for $m < n$. Thus we can define

$$\mathrm{beh}_c(x) = \begin{cases} \langle a_0, a_1, a_2, \ldots \rangle & \text{if } \forall n \in \mathbb{N}.\, c^n(x) \neq \bot, \text{ and } c^i(x) = (a_i, x_i) \\ \langle a_0, \ldots, a_{m-1} \rangle & \text{if } m \in \mathbb{N} \text{ is the least number with } c^m(x) = \bot, \\ & \text{and } c^i(x) = (a_i, x_i), \text{ for } i < m. \end{cases}$$

We check the two conditions for homomorphism from the proposition above.

- If $c(x) = \bot$, then the least m with $c^m(x) = \bot$ is 0, so that $\mathrm{beh}_c(x) = \langle\rangle$, and thus also $\mathsf{next}(\mathrm{beh}_c(x)) = \bot$.
- If $c(x) = (a, x')$, then we distinguish two cases:
  - If $\forall n \in \mathbb{N}.\, c^n(x) \neq \bot$, then $\forall n \in \mathbb{N}.\, c^n(x') \neq \bot$, and $c^{i+1}(x) = c^i(x')$. Let $c^i(x') = (a_i, x_i)$, then

$$\begin{aligned} \mathsf{next}(\mathrm{beh}_c(x)) &= \mathsf{next}(\langle a, a_0, a_1, \ldots \rangle) \\ &= (a, \langle a_0, a_1, \ldots \rangle) \\ &= (a, \mathrm{beh}_c(x')). \end{aligned}$$

  - If m is least with $c^m(x) = \bot$, then $m > 0$ and $m - 1$ is the least k with $c^k(x') = \bot$. For $i < m - 1$ we have $c^{i+1}(x) = c^i(x')$, and thus by writing $c^i(x') = (a_i, x_i)$, we get as before:

$$\begin{aligned} \mathsf{next}(\mathrm{beh}_c(x)) &= \mathsf{next}(\langle a, a_0, a_1, \ldots, a_{m-2} \rangle) \\ &= (a, \langle a_0, a_1, \ldots, a_{m-2} \rangle) \\ &= (a, \mathrm{beh}_c(x')). \end{aligned}$$

Finally, we still need to prove that this behaviour function beh_c is the unique homomorphism from c to next. Thus, assume also $g\colon S \to A^\infty$ is such that $c(x) = \bot \Rightarrow \mathsf{next}(g(x)) = \bot$ and $c(x) = (a, x') \Rightarrow \mathsf{next}(g(x)) = (a, g(x'))$. We then distinguish:

- $g(x)$ is infinite, say $\langle a_0, a_1, \ldots \rangle$. Then one shows by induction that for all $n \in \mathbb{N}$, $c^n(x) = (a_n, x_n)$, for some x_n. This yields $\mathrm{beh}_c(x) = \langle a_0, a_1, \ldots \rangle = g(x)$.
- $g(x)$ is finite, say $\langle a_0, \ldots, a_{m-1} \rangle$. Then one proves that for all $n < m$, $c^n(x) = (a_n, x_n)$, for some x_n, and $c^m(x) = \bot$. So also now, $\mathrm{beh}_c(x) = \langle a_0, \ldots, a_{m-1} \rangle = g(x)$. ❑

Before exploiting this finality result we illustrate the behaviour function.

Example 1.2.2 (Decimal representations as behaviour) So far we have considered sequence coalgebras parametrised by an arbitrary set A. In this example we take a special choice, namely $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, the set of decimal digits. We wish to define a coalgebra (or machine) which generates

decimal representations of real numbers in the unit interval $[0, 1) \subseteq \mathbb{R}$. Notice that this may give rise to both finite sequences ($\frac{1}{8}$ should yield the sequence $\langle 1, 2, 5\rangle$, for 0.125) and infinite ones ($\frac{1}{3}$ should give $\langle 3, 3, 3, \ldots\rangle$ for $0.333\ldots$).

The coalgebra we are looking for computes the first decimal of a real number $r \in [0, 1)$. Hence it should be of the form

$$[0, 1) \xrightarrow{\mathsf{nextdec}} \{\bot\} \cup \Big(\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \times [0, 1)\Big)$$

with state space $[0, 1)$. How to define nextdec? Especially, when does it stop (i.e. return $\bot$), so that a finite sequence is generated? Well, the decimal representation 0.125 may be identified with $0.12500000\ldots$ with a tail of infinitely many zeros. Clearly, we wish to map such infinitely many zeros to $\bot$. Fair enough, but it does have as consequence that the real number $0 \in [0, 1)$ gets represented as the empty sequence.

A little thought brings us to the following:

$$\mathsf{nextdec}(r) = \begin{cases} \bot & \text{if } r = 0 \\ (d, 10r - d) & \text{otherwise, with } d \leq 10r < d + 1 \text{ for } d \in A. \end{cases}$$

Notice that this function is well defined, because in the second case the successor state $10r - d$ is within the interval $[0, 1)$.

According to the previous proposition, this nextdec coalgebra gives rise to a behaviour function:

$$[0, 1) \xrightarrow{\mathsf{beh}_{\mathsf{nextdec}}} \Big(\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}\Big)^{\infty}.$$

In order to understand what it does, i.e. which sequences are generated by nextdec, we consider two examples.

Starting from $\frac{1}{8} \in [0, 1)$ we get

$$\begin{aligned} \mathsf{nextdec}(\tfrac{1}{8}) &= (1, \tfrac{1}{4}) \text{ because } 1 \leq \tfrac{10}{8} < 2 \text{ and } \tfrac{10}{8} - 1 = \tfrac{1}{4} \\ \mathsf{nextdec}(\tfrac{1}{4}) &= (2, \tfrac{1}{2}) \text{ because } 2 \leq \tfrac{10}{4} < 3 \text{ and } \tfrac{10}{4} - 2 = \tfrac{1}{2} \\ \mathsf{nextdec}(\tfrac{1}{2}) &= (5, 0) \text{ because } 5 \leq \tfrac{10}{2} < 6 \text{ and } \tfrac{10}{2} - 5 = 0 \\ \mathsf{nextdec}(0) &= \bot. \end{aligned}$$

Thus the resulting nextdec-behaviour on $\frac{1}{8}$ is $\langle 1, 2, 5\rangle$, i.e. $\mathsf{beh}_{\mathsf{nextdec}}(\frac{1}{8}) = \langle 1, 2, 5\rangle$. Indeed, in decimal notation we write $\frac{1}{8} = 0.125$.

Next, when we run nextdec on $\frac{1}{9} \in [0, 1)$ we see that

$$\mathsf{nextdec}(\tfrac{1}{9}) = (1, \tfrac{1}{9}) \text{ because } 1 \leq \tfrac{10}{9} < 2 \text{ and } \tfrac{10}{9} - 1 = \tfrac{1}{9}.$$

The function nextdec thus immediately loops on $\frac{1}{9}$, giving an infinite sequence $\langle 1, 1, 1, \ldots\rangle$ as behaviour. This corresponds to the fact that we can identify $\frac{1}{9}$ with the infinite decimal representation $0.11111\ldots$.